verify_info Command & info_seg_

An expanded version of validate_info_seg (vis).

Author: Gary Dixon
Date: January 14, 2021

Abstract

This MTB discusses problems with the validate_info_seg (vis) command; then introduces a verify_info (vi) command and supporting info_seg_ routines to correct those problems.

Table 1: Revision History

Date	Revision	Author	Comment
2020-06-29	0.1	Gary Dixon	Initial draft.
2020-07-13	0.2	Gary Dixon	Revised outline for early chapters.
2020-10-31	0.3	Gary Dixon	Refocused on need for verify_info, and new support routines for info seg handling.
2020-11-26	0.4	Gary Dixon	First review draft.
2020-11-29	0.5	Gary Dixon	Review draft with typo corrections.
2020-12-27	1.0	Gary Dixon	Add Testing, Installation Plan and Guideline chapters.
2021-01-14	1.1	Gary Dixon	Fix typographical errors.

Table of Contents

List of Tables

Introduction

One important development tool was not incorporated into the recent mbuild project: a tool for verifying the structure, format, and names on info segments. The **validate_info_seg** (**vis**) command has traditionally been used for this purpose. However, vis has several shortcomings that make it poorly suited as a general info segment verification tool. These can be summarized as follows:

- vis cannot correctly identify the kind of data described in newer kinds of info segments. Having misidentified the kind of data, vis then reports incorrect or misleading errors in the data.
- vis supports multiple blocks within an info segment only as individual help data. It fails to assist the writer in managing relationships between info blocks like detecting: missing info blocks; misspelled operation or request names; or incorrect external names on the info segment.
- vis provides little guidance to assist writers in creating info segs, or correcting reported problems in an info segment.

These shortfalls became apparent only as info segment formatting rules were broadened to handle new kinds of data. vis works quite well for simple command and active function documentation, and for general information data. However, info segments are also used to document multi-operation commands (e.g., io_call, window_call, and history_comment); subroutines; and I/O modules.

A quick review of vis code found its program structure unsuited to resolving the issues above. Like the help_ subroutine, vis tries to parse an info segment as it is validating that content. Rules for validating info seg content vary depending on the kind of information it contains. But the kind of information can only be determined after parsing the entire info segment.

An alternate approach to parsing an information segment would apply both bottom-up and top-down analysis strategies.

- [bottom-up]: Parse the entire info segment file into lines.
- [top-down]: Examine file lines to locate block dividers, thereby defining info blocks (highest-level units of the info seg).
- [top-down]: Examine pattern of block dividers to determine overall info segment structure.
- [bottom-up]: Parse block lines into a block heading and paragraphs.
- [top-down]: Examine paragraphs to locate those beginning with a section title, thereby grouping paragraphs into titled sections.
- [top-down]: Examine sequence of sections to identify kind of block, thereby determining content rules for the block.

Once the info segment is parsed into known block kinds, the parsed content can be compared with rules to validate expected block content and inter-block relationships.

A new info_seg_ subroutine implements the approach above. Low-level errors (lines too long, lines ending in whitespace, too many lines between paragraphs, etc.) are detected while parsing. These errors are recorded in data structures identifying each info seg component. Then a new info_seg_verify_iFile_ routine looks for errors between info structures, records those additional errors, and then reports both low-level and inter-structure errors to the user.

Components of an Info Block

An info segment is comprised of one or more blocks. If several blocks are present, each block begins with a divider token specifying the type of block. The divider token is the first element in a divider line. Divider items that follow are: one or more names for the info block created by that divider line. The heading line which begins the actual text of an info block may appear after the last name on the divider line; or it may appear on the next line after the divider.

Text within an info block includes the following components.

lines: a sequence of printable characters ending with a newline character.

heading line:
first line of an info block identifies the information content of the lines that follow.

non-heading lines:
second through final lines in the info block.

paragraph:
group of consecutive lines in the info block containing related information. help displays each paragraph as a unit. A paragraph ends with two consecutive 0-length lines (also known as *blank lines*), or with the final non-blank line in the info block. Format of lines in a paragraph differs depending upon the type of information on the line.

paragraph title, *section title*:
portion of the first line of a paragraph preceding the first colon (**:**) character on that line. Many paragraphs have no title. Each paragraph that begins with a title also starts a new section of the info block. Therefore, the *paragraph title* is also known as a *section title* for all paragraphs in that new section.

Sample titles: Function, Syntax, Arguments, Control arguments, Notes, Examples.

section:
group of adjacent paragraphs starting with a titled paragraph, and including any untitled paragraphs that follow. The paragraphs have related content. That relationship is usually characterized by the section title.

Note: any untitled paragraph(s) immediately following the heading line and preceding the first titled paragraph are treated as an untitled first section of the info block. Some general info data consist only of a heading line and one untitled section.

lists: Certain sections are formatted as item names, followed optionally by a description of that item. The help -brief command displays such lists (without the item descriptions) to summarize use of a command, subroutine, subsystem request, etc.

Command: verify_info

A new verify_info (vi) command:

- Initializes the info_seg_ subroutine.
- Gathers a list of info segment pathnames to be verified. info_seg_$append_iFile records these pathnames.
- Calls info_seg_verify_$iFiles which verifies each info seg, by:
  - Calling info_seg_parse_ to parse each info seg into structures with low-level errors recorded in those structures.
  - Calling info_seg_verify_iFile_ to look for high-level errors and report all errors to the user.

In addition, the vi command can display the parsing rules used to:

- Analyze good and bad info seg block structures.
- Determine the kind of an info block.
- Show the list of preferred or commonly-used section titles.
- Show expected order of sections in each kind of block.

This information comes from a new info_seg_specifications_.cds data segment, and is displayed by info_seg_verify_$display_specifications.

Typical output from the verify_info command is shown below.

```
FILE: >doc>info>apl.info
        structure:  Info Segment - no block dividers                    lines:   73
            names:                          recommended name order:
              apl.info                        apl.info
                                              v2apl.info

       Severity 2.  File names do not follow recommended name order (see list above).
       Severity 4.  Expected names not found on the info segment:
           v2apl.info

     Line    Kind                                Block Names
 ---    1 Command                             apl  v2apl
           header:  03/09/84 apl, v2apl
        recommend:  1984-03-09  apl, v2apl
       Severity 4.  Use an iso_date in the header.
       Severity 3.  Command/active function header words should each be a reference_name
                     (begin with a letter, followed by letter/digit/underscore characters) and
                     appear in a comma-separated list of words ordered by descending lengths.
       Severity 2.  Recommend sections be re-ordered in ascending sequence--v
        4   Syntax                                                      100
       Severity 4.  Section title not used for a Command block.
        recommend:  Syntax as a command:
        7   Function                                                    120
       11   Arguments                                                   140
       18   Control arguments                                           200
       48   Control arguments for debugging                             201
       62   Compatibility with Version 1 APL                            900
       68   Notes                                                       250
       Severity 1.  1 section has a nonstandard title, beginning on line:
                     62

r 11:32 0.552 24
```

The proposed user interface is documented by verify_info.info:

```
>udd>m>gd>w>vi>verify_info.info   (168 lines in info)

2020-06-21  verify_info, vi

Syntax as a command:  vi {INFO_PATHs} {-control_args}

Syntax as an active function:  [vi INFO_PATHs]

Function: verifies info segment format, section titles, and names
against guidelines for Multics info segments.  The active function
returns the number of the highest severity error that occurs.

Also, control arguments can display important guidelines for info
segments.

Arguments:
INFO_PATHs
  relative or absolute pathnames of the info segments to be verified.
  A suffix of .info is assumed if not given.  The star convention is
  supported to match several segments in the given directory.

Control arguments as a command:
-names, -nm
  add names required by guidelines if missing from an info segment.
  Reorder names following guidelines.  If names exist that conflict
  with guidelines, ask user for permission to remove those names.
  If removal not granted, move conflicting names to end of name list.
  Name changes are listed in informational messages.
-no_names, -nnm
  suppress name additions, removals, and reordering.  Error messages
  report names that are missing, conflict with guidelines, or need to
  be reordered.  (default)
-force_names, -fnm
  change names to follow all guidelines without user queries.

-totals, -tt
  displays only a count of errors found, with level of the highest
  severity error.  The active function assumes -totals and returns
  only the level of the highest severity error.  All other output is
  suppressed.
-long, -lg
  displays long error messages and warnings. (default)
-brief, -bf
  displays only error message keywords at end of block or section
  plus summary messages for paragraph and line errors.
-rules AREA
  displays verify_info guidelines and rules for info segment structure
  and format.  See "List of rule areas" below.
```

```
Control arguments (displaying info lines):
   It is sometimes useful to display lines of an info segment or block
   to understand errors reported by verify_info.  Line-related errors
   are reported by keyword(s) at the end of each displayed line.
-lines,
-ln COUNT,
-ln START:COUNT
  displays lines of the info segment.  The optional START line number
  and COUNT of lines refer to lines within the entire info segment.
  If START is a negative integer, it refers to a line number
  counting back from the end of the segment: -lines -3: displays the
  final 3 lines of the segment.

-block,
-bk COUNT,
-bk START:COUNT
  displays lines of each block of the info segment.  The START and
  COUNT values refer to lines within the block.  If START is a
  negative integer, it refers to a line counting back from the end of
  the block.  -block -3:  displays the final 3 lines of the info block.

Control arguments (debugging verify_info):
  These control arguments display additional data used to check info
  segment format against guidelines.
-debug SELECTOR,
-db SELECTOR
   selects a debugging function.  -debug may be given more than once
   to select several functions; or once with several SELECTORs
   separated by commas.

List of rule areas:
  The -rules AREA operand may be any of the following.
all, a
  displays all rules and guidelines used by verify_info.
file, f
  displays guidelines for supported file structures (allowed order of
  :Info: :Entry: and :hcom: blocks within an info segment).
block, bk
  displays rules used to determine the kind of each info block
  (e.g., command or active function, subroutine, subsystem request,
  etc.).

section, scn
  displays Multics standard info segment section titles, and a list of
  obsolete or deprecated titles with a preferred replacement.
KIND_OF_INFO_BLOCK
  displays section titles typical for a particular kind of info block
  including the usual order of appearance within the block of each
  title.  See "List of info block kinds" below.
```

```
List of info block kinds:
  The -rule KIND_OF_INFO_BLOCK value may be any of the following.
all_kinds, ak
  display section titles typical of all of the following block kinds.
general_info, gi
  a block describing a general info topic.

command, cmd
  a block describing a program that operates as a Multics command.
active_function, af
  a block describing a program that operates as a Multics active
  function.
command/af, cmd/af
  a block describing a program that operates both as a Multics command
  and as an active function.
commands, cmds
  display section titles typical of all three of the kinds above.

request, req
  a block describing a program that operations as a subsystem request.
active_request, ar
  a block describing a program that operations as a subsystem
  active request.
request/ar, req/ar
  a block describing a program that operations both as a subsystem
  request and an active request.
topics, topic
  a block giving information about a subsystem topic.
subsystem, ss
  display section titles typical of all four of the kinds above.

subroutine, subr
  a block describing a subroutine or function.  This includes a
  subroutine introduction block, and blocks describing a subroutine
  entry point.
io_module, io
  a block describing an I/O module; or one of its setup operations
  (open_file, close_file, or detach); or one of its control orders.

List of debug selectors:
file, f
  display names, plus a specification line containing:
    - file structure case number
    - specification string giving block cardinality and block
      type constraints
    - count of each type of info block divider:
       N:  first info block of segment, having no :Info: divider.
       E:  a subroutine :Entry: divider.
       I:  an :Info: divider.
      [I]: an :[Info]: divider.
       H:  an :hcom: history comment divider.

block, bk
  displays block divider and header details.
```

```
section, scn
  displays items found in Syntax... sections.
list, ls
  displays item names found in Arguments, Control arguments, and
  List of... sections.

Notes on verification:
The verify_info (vi) command verifies the format, section titles, and
names of an info segment.  For more details on info segment format,
type:  help info_seg.gi

List of error message severities:
  Error messages printed by vis are of five severity levels:
Severity 5
  structural errors preventing parsing of the info segment into
  blocks, sections, paragraphs, and lines.
Severity 4
  errors that should be corrected before installing the info segment.
Severity 3
  errors in following info segment guidelines, a few of which may
  qualify as valid exceptions.
Severity 2
  even less important errors.
Severity 1
  warnings, such as the existence of nonstandard section titles.
```

Subroutine: info_seg_

The info_seg_ subroutine:

- Parses an info segment into a hierarchy of components:
  - **info blocks**, each beginning with a heading, and sections of information describing: a command, active function, general information topic; or introducing a subroutine, or describing one of its entry points.
  - **titled sections** within a block describing: related, adjacent paragraphs of information.
  - **paragraphs** formed from adjacent lines.
  - information **lines**.
- Paragraphs and sections only have meaning as components of an info block. Each line is a component of its containing info segment, as well as any containing info block, section and paragraph. Note that some lines are not part of any paragraph or section or block; a blank line may reside between paragraphs or between blocks.

A typical caller parses one or more info segments using the following info_seg_ entry points. Details of each entry point are given later in this subroutine description.

call info_seg_$initialize(…);
: Called once to initialize the parsing environment.

call info_seg_$append_iFiles(…);
: Called to add each info segment path to the list of files to be parsed. An iFile structure represents each selected file. Several files may be selected by a path ending in an entryname that follows the star convention.

call info_seg_$parse_iFile(…);
: Called for each appended iFile structure to parse the described info segment into components. The caller then uses the component info structures to access specific information in the described info segment.

call info_seg_$terminate(…);
: Called when finished with the parsed info segments to terminate the info segments, and remove storage used by the parsing environment.

info_seg_ Structure Hierarchy

Output from parsing is a set of PL/I structures describing each component of the info segment. Pointers between structures identify the location of that structure in the hierarchy of components listed above.

Structures of a given type are linked in a sibling thread showing their order of appearance within the info seg. Each parent structure declares an **iChild substructure** locating the threaded list of contained items. From info_seg_dcls_.incl.pl1:

```
dcl 1 iChild aligned based(iChildP),
    2 firstP ptr,    /*  - points to first child structure in list.    */
    2 lastP ptr;     /*  - points to final child structure in list.    */
```

Top-most parent structure in the hierarchy outlined above is the following.

- The **info_seg_data** structure (an input to info_seg_$initialize) contains the threaded list of info segments (files) to be parsed. info_seg_data.relatives.files is an iChild substructure; the firstP and lastP elements point to an iFile structure.

```
dcl 1 info_seg_data aligned based(info_seg_dataP),
    2 version char(11),
    2 standalone_invocationS bit(1) aligned,
    2 ptrs,
      3 sciP ptr,
      3 areaP ptr,
      3 std_areaP ptr,
    2 relatives,
      3 files like iChild;
```

Other structures in the hierarchy are described from higher to lowest member. Each element below is declared as an iChild substructure.

- Each info segment is represented by an **iFile** structure that contains separate threaded lists for its info blocks, and for lines of the info segment: iFile.relatives.bloks and iFile.relatives.lines
- Each info block is represented by an **iBlok** structure that contains threaded lists for its sections, paragraphs and lines: iBlok.relatives.sects, iBlok.relatives.pghs, and iBlok.relatives.lines
- Each section of an info block is represented by an **iSect** structure that contains lists for its paragraphs and lines: iSect.relatives.pghs and iSect.relatives.lines
- Each paragraph of an info block is represented by an **iPgh** structure that contains a list for its lines: iPgh.relatives.lines
- Each line is represented by an **iLine** structure that is the lowest-level structure type in the hierarchy which therefore includes no list of "contained" items.

Each structure in a threaded list includes an **iSib substructure** linking it with previous and next structures in that thread. All members of a thread are declared identically. From info_seg_dcls_.incl.pl1:

```
dcl 1 iSib aligned based(iSibP),
    2 nextP ptr,     /*  - points to next structure in list.          */
                     /*     - null if structure is last in the list.  */
    2 prevP ptr;     /*  - points to prior structure in list.         */
                     /*     - null if structure is first in the list. */
```

The threaded item includes one iSib substructure for each thread on which that item can appear.

- A given line in an info seg may be a part of a paragraph, which is part of a section, which is part of an info block, which is part of the entire info segment. The iLine structure declares a sibling thread for each of these containers:
  - iLine.sibs.pgh, iLine.sibs.sect, iLine.sibs.blok and iLine.sibs.file are iSib substructures for the thread of adjacent lines within its containing paragraph, within its section, within the info block, and within the entire info segment. iLine.sibs.pgh and iLine.sibs.blok may be null if the line is a blank line between paragraphs. iLine.sibs.blok is null if the line is a blank line separating blocks.

- A given paragraph in an info block is part of a section, which is part of the info block. So an iPgh structure contains an adjacent sibling thread for each of these container levels:
  - iPgh.sibs.sect and iPgh.sibs.blok are iSib substructures for the thread of adjacent paragraphs in its containing section or in the containing info block.

- A given section in an info block has only one sibling thread:
  - iSect.sib is an iSib substructure for the thread of adjacent sections in its containing info block.

- A given info block has only one sibling thread:
  - iBlok.sib is an iSib substructure for the thread of adjacent info blocks in its containing info segment.

The following declaration shows the **iPgh** structure that includes both iChild and iSib substructures. The iChild substructure forms a thread of iLine structures contained in the paragraph. The two iSib substructures connect this paragraph to its containing iBlok and iSect structures. iPgh.relatives.sectP points to the particular iSect structure for the section containing that paragraph.

```
dcl 1 iPgh aligned based (iPghP),
    2 common like iCommon.content,
    2 sibs,
      3 blok like iSib,    /* thread of adjacent pghs of a block.     */
      3 sect like iSib,    /* thread of adjacent pghs of a section.   */
    2 relatives,
      3 sectP ptr,         /* points to iSect containing this pgh.    */
      3 lines like iChild, /* thread of lines in this paragraph.      */
                           /*  - follows the iLine.sibs.pghs thread.  */
    2 line_in_file fixed bin,
    2 seenS bit(1) aligned,
    2 errors aligned,
      3 (long_paragraphS,
         few_blank_lines_before_paragraphS,
         many_blank_lines_before_paragraphS
         ) bit(1) unal,
       3 pad_errors bit(33) unal;
```

See the info_seg_dcls_.incl.pl1 include file for details on all of the structures that define the info_seg_ hierarchy of components. See “Structure: COMPONENT_NAME” sections that follow for a discussion of each of the major structures describing an info seg component.

Entry Point: info_seg_$initialize

Initializes an execution environment in which one or more info segments can be parsed into info blocks, sections, paragraphs and lines. The environment consists of an ssu_ standalone invocation that manages a chain of extensible areas in which structures describing each component are allocated, alongside the initiated info segments.

The caller may use this ssu_ standalone invocation to parse command input arguments and display error messages. The caller must also call the info_seg_$terminate entry point to cleanup and destroy the ssu_ environment before returning.

The info_seg_.info segment provides the following details for this entry point.

```
Syntax:
   declare info_seg_$initialize entry (ptr, char(*), char(*), ptr,
     entry variable, fixed bin(35);
   call info_seg_$initialize (info_seg_dataP, caller_name,
     caller_version, arg_list_ptr, ssu_abort_routine, code);

Arguments:
info_seg_dataP
   points to the info_seg_data structure (see info_seg_dcls_.incl.pl1)
   which caller provides as input.
     A) Caller sets info_seg_data.version to indicate which structure
        version is being passed.
     B) Caller sets info_seg_data.ptrs, info_seg_data.files = null()
        to prepare for a later call to info_seg_$terminate.
     C) Caller optionally sets info_seg_data.sciP to point to an
        existing ssu_ invocation.  If this pointer is null(), then
        initialization also creates its own ssu_ standalone
        invocation, using the other parameters.
   See the "Notes on cleanup" section for more information.

   If info_seg_data.sciP is null, the following arguments are passed
   directly to ssu_$standalone_invocation.  See the MPM Subroutines
   description of that entry point for more details.

caller_name
   gives name of calling command.  Can be accessed by
   ssu_$get_subsystem_name.

caller_version
   gives caller's version number.  Can be accessed by
   ssu_$get_subsystem_version.

arg_list_ptr
   points to calling command's input argument list.  Use ssu_$arg_ptr
   to access arguments.  The caller may use ssu_ calls to process its
   input arguments.  The info_seg_ subroutine does not access this
   arg_list_ptr.    To simply argument handling, see:
      ssu_standalone_command_.incl.pl1
```

```
ssu_abort_routine
   subroutine to be called if an info_seg_$XXX routine encounters a
   fatal error and needs to abort execution of the calling command.
   This subroutine should go to a label that exits the command (call's
   its cleanup routine, then returns).
code
   is an ssu_ status code.  It is non-zero when a fatal error occurs.
   Cases include:
    - info_seg_data.version has a value not supported by info_seg_; or
    - ssu_$standalone_invocation creation failed.
   ssu_$print_message (and other ssu_ routines) may not be called to
   report such errors.

Notes on cleanup:
When initialization completes, info_seg_data.ptrs point to data items
used in subsequent calls to info_seg_ subroutines.  The caller must
create a cleanup on-unit that calls info_seg_$terminate to terminate
info segments, and release storage for structures describing those
info segments.
```

Structure: info_seg_data

The info_seg_data structure is the highest-level member of the *info_seg_ Structure Hierarchy* (introduced above). This structure provides input for info_seg_$initialize, and returns results from subsequent info_seg_ entry points.

```
dcl 1 info_seg_data aligned based(info_seg_dataP),       /* Structure to hold info_seg_ parser information.        */
    2 version char(11),                                  /*  - version of this structure: info_seg_data_version_01 */
    2 standalone_invocationS bit(1) aligned,             /*  - T: info_seg_ provided the .sciP value.              */
                                                         /*    F: Caller provided the .sciP value.                 */
                                                         /*     - info_seg_$initialize sets bit = (.sciP = null)   */
                                                         /*       so info_seg_$terminate knows whether to cleanup  */
                                                         /*       the ssu_ standalone invocation.                  */
    2 ptrs,
      3 sciP ptr,                                        /*  - ssu_ invocation pointer.                            */
      3 areaP ptr,                                       /*  - translator_temp_ extendable no-freeing area pointer.*/
      3 std_areaP ptr,                                   /*  - standard Multics area (system free area).           */
    2 relatives,
      3 files like iChild,                               /*  - threaded list of iFile structures.                  */
     info_seg_dataP ptr;

dcl  info_seg_data_version_01 char(11) int static options(constant) init("info_seg_01");
                                                         /* Currently supported version of info_seg_data structure */
```

Entry Point: info_seg_$append_iFiles

Creates an iFile structure for each info segment matching an input pathname. Since the pathname may use the Multics star convention, one or more iFile structures may result from a single call. Each iFile structure is attached to the info_seg_data.relatives.files threaded list, ordered alphabetically by matching entryname on the info segment.

This entry point may be called several times to create a longer threaded list of iFile structures from different input pathnames.

The info_seg_.info segment provides the following details for this entry point.

```
Syntax:
   declare info_seg_$append_iFiles entry (ptr, char(*));
   call info_seg_$append_iFiles (info_seg_dataP, path);

Arguments:
info_seg_dataP
   points to the info_seg_data structure.  Refer to
   info_seg_dcls_.incl.pl1.
path
   a relative or absolute pathname identifying one or more info
   segments.  A suffix of .info is assumed if not present.  The star
   convention is supported.

Notes on the structure:
Each iFile structure created by info_seg_ is added to the
info_seg_data.files threaded list.  Use the following code to set
iFileP to point to structures on this list:

     do iFileP = info_seg_data.relatives.files.firstP
          repeat iFile.sib.nextP  while (iFileP ^= null() );
             ... <code to operate on each iFile> ...
          end;

The code for each iFile would include a call to parse the info segment,
followed by code to process the structure hierarchy returned by parsing.
The iFile and info_seg_data structures are declared in
info_seg_dcls_.incl.pl1.
```

Structure: iCommon

Each of the component structures in the info_seg_ Structure Hierarchy begins with a substructure that: identifies the content of the component (substring of the info segment that forms the component); declares one or more iSib structures forming threaded lists of that component. The iCommon structure is declared in info_seg_dcls_.incl.pl1 as follows.

```
dcl 1 iCommon aligned based(iCommonP),
    2 content,           /* Content described by the major structure. */
      3 ID char(4),      /*  - content identifier                     */
      3 L fixed bin(21), /*  - content length (in characters)         */
      3 P ptr,           /*  - content pointer                        */
    2 sib like iSib,     /*  Pointers to next/prev items in list of   */
     iCommonP ptr;       /*   these structures.                       */
```

iCommon.content.P points to a substring in the initiated info segment defining the content described by the component structure.

Structure: iNames10 & iNames100

Structures describing the info segment and its info blocks include substructures holding file names or info block divider names. Declarations for these substructures follow.

```
dcl  entrynameL fixed bin int static options(constant) init(32);

dcl 1 iName10 aligned,              /* A 10-element name array.       */
    2 N fixed bin,                  /*  - number in use.              */
    2 nm (10) char(entrynameL) var; /*  - array holding names         */

dcl 1 iName100 aligned based,       /* A 100-element name array.      */
    2 N fixed bin,                  /*  - number in use.              */
    2 nm (100) char(entrynameL) var;/*  - array holding names         */
```

Structure: iFile

Each iFile structure describes an entire info segment: its location in the Multics file system; its contained info blocks and lines; its attributes as determined when parsing; and any low-level errors detected by the parser. The structure also includes switches to record file naming errors detected by the verify_info command and its info_seg_verify_ helper.

The iFile structure is declared in info_seg_dcls_.incl.pl1 as follows.

```
dcl 1 iFile aligned based (iFileP),
    2 common like iCommon,
    2 location,
      3 dir char(168) unal,          /* containing directory pathname   */
      3 ent char(entrynameL) unal,   /* entryname within this directory */
      3 uid bit (36),                /* unique ID for info segment      */
    2 names like iName100,           /* array of all names on info_seg  */
    2 relatives,                     /* child components:               */
      3 bloks like iChild,           /*  - list of blocks contained     */
      3 lines like iChild,           /*  - list of lines contained      */
    2 caseI fixed bin,               /* pattern of block kinds          */
    2 structure fixed bin,           /* iFile_structure_XXX value       */

    2 errors aligned,                /* ERROR possibilities found       */
                                     /*  those with - are by parsing    */
                                     /*  those with + are by verify_info*/
      3 (all_nulS,                   /*  - File contains only NUL (\000)*/
         all_white_nulS,             /*  -  only HT SP NUL and NL chars.*/
         ends_nulS,                  /*  - File ends with NUL chars.    */
         zero_lengthS,               /*  - File contains no characters. */

         names_missing_info_suffixS,
                                     /*  +  name does not end with .info*/
         names_extraS,               /*  +  name not expected           */
         names_missingS,             /*  +  missing name(s)             */
         name_order_warningS         /*  +  name order differs from     */
                                     /*     block divider/heading names */
         ) bit(1) unal,
     iFileP ptr;

dcl  File char(iFileP->iFile.L) unal based(iFileP->iFile.P);
            /* Character contents of entire info segment.              */

dcl  File_names (iFileP->iFile.names.N) char(entrynameL) var
                  based(addr(iFileP->iFile.names.nm(1)));
            /* Array of names on current info segment.                 */
```

The iFile.structure element may have one of the following values.

```
dcl (                                          /* Values for the iFile.structure element:              */
                                               /*  supported patterns of blocks within info segment.   */

    iFile_structure_NO_DIVIDERS          init( 1),  /*  - 1 iBlok                [ A ]                    */
                                                    /*    A)      only iBlok.divider = iBlok_divider_None  */


    iFile_structure_SUBROUTINE           init( 2),  /*  - 2 or more iBloks:      [ A  B... ]              */
                                                    /*    A)     first iBlok.divider = iBlok_divider_None  */
                                                    /*              <subroutine intro description>         */
                                                    /*    B) all other iBlok.divider = iBlok_divider_Entry */
                                                    /*              <subroutine entrypoint description>    */

    iFile_structure_INFO_SUBROUTINE_HCOM init( 3),  /*  - 3 or more iBloks:      [ A  B... C ]            */
                                                    /*    A)     first iBlok.divider = iBlok_divider_Info  */
                                                    /*              <subroutine intro description>         */
                                                    /*    B)    middle iBlok.divider = iBlok_divider_Entry */
                                                    /*              <subroutine entrypoint description>    */
                                                    /*    C)      last iBlok.divider = iBlok_divider_hcom  */
                                                    /*              <history comments>                     */
```

```
iFile_structure_INFO_SUBROUTINE          init( 4),  /*  - 2 or more iBloks:       [ A  B... ]                */

                                                    /*    A)     first iBlok.divider = iBlok_divider_Info     */
                                                    /*              <subroutine intro description>           */
                                                    /*    B) all other iBlok.divider = iBlok_divider_Entry    */
                                                    /*              <subroutine entrypoint description>      */

iFile_structure_INFO_HCOM                init( 5),  /*  - 2 or more iBloks:       [ A... B ]                 */
                                                    /*    A)   leading iBlok.divider = iBlok_divider_Info     */
                                                    /*              <info contents>                          */
                                                    /*    B)      last iBlok.divider = iBlok_divider_hcom     */
                                                    /*              <history comments>                       */

iFile_structure_INFO                     init( 6),  /*  - 1 or more iBloks:       [ A... ]                   */
                                                    /*    A)       all iBlok.divider = iBlok_divider_Info     */
                                                    /*              <info contents>                          */

                                                    /* ERROR values for iFile.structure                      */
iFile_struc_err_UNSET                    init( 0),  /*  - structure not yet determined.                       */
iFile_struc_err_EMPTY_INFO               init(-1),  /*  - 0 iBlok structures; or 1 iBlok.emptyS               */
iFile_struc_err_HCOM_NOT_LAST            init(-2),  /*  - History Comment block not last in info seg.         */
iFile_struc_err_INFO_SUBROUTINE_MIX      init(-3),  /*  - Unknown mixture of :Info: and :Entry: divider types.*/
iFile_struc_err_1st_INFO_no_ext_names    init(-4),  /*  - 1st block :[Info]: -- it must be :Info:             */
iFile_struc_err_MISSING_1st_INFO         init(-5),  /*  - 1st block no divider; other blocks :Info:           */
iFile_struc_err_MISSING_SUBROUTINE_INTRO init(-6),  /*  - all blocks :Entry: (1st block must have no divider) */
iFile_struc_err_MULTIPLE_HCOM            init(-7),  /*  - 2 or more History Comments in info seg.             */
iFile_struc_err_SUBROUTINE_INFO_MIX      init(-8),  /*  - :Entry: followed by :Info: blocks                   */
iFile_struc_err_STRUCTURE_UNKNOWN        init(-9)   /*  - Unsupported pattern of block kinds.                 */
) fixed bin int static options(constant);
```

Entry Point: info_seg_$parse_iFile

Initiates the info segment identified by an iFile structure, and parses that segment into a hierarchy of structures describing components of that info segment. These components are summarized in the *info_seg_ Structure Hierarchy* section above.

```
Syntax:
   declare info_seg_$parse_iFile entry (ptr, ptr);
   call info_seg_$parse_iFile (info_seg_dataP, iFileP);


Arguments:
info_seg_dataP
   points to the info_seg_data structure (see info_seg_dcls_.incl.pl1).
iFileP
   points to an iFile input structure for the file to be parsed.


Notes:
Before calling info_seg_$parse_iFile, the iFile structure contains
only the location of an info segment, and threads to other iFile
structures.


After parsing, the following information has been added to the iFile
structure...
 - A pointer to, and length of, the characters in the info segment.
 - An array of all names on the info segment.
 - A list of iLine structures, each describing a line of the info
   segment.
 - A list of iBlok structures, each describing a block of the info
   segment.
 - Integers classifying the block organization within the info segment.


In addition, each iBlok structure describes...
 - A pointer to, and length of, the characters in the block.
 - An array of names in the block divider (if the block began with a
   divider).
 - Block heading line information (date and header string).
 - A list of iSect structures, each item describing sections within the
   block.
 - A list of iPgh structures, each item describing paragraphs within the
   block.
 - A list of iLine structures, each item describing lines within the
   block.
```

Entry Point: info_seg_$terminate

Terminates info segment files that were parsed, and releases temporary storage used for info segment data structures. If info_seg_$initiate created its own ssu_ standalone invocation, that invocation is destroyed.

```
Syntax:
   declare info_seg_$terminate entry (ptr);
   call info_seg_$terminate (info_seg_dataP);

Arguments:
info_seg_dataP
   points to the info_seg_data structure (see info_seg_dcls_.incl.pl1).

Notes on cleanup:
When initialization completes, info_seg_data.ptrs point to data items
used in subsequent calls to info_seg_ subroutines.  The caller must
create a cleanup on-unit that calls a routine to release storage for
these data items:
   - Calls info_seg_$terminate to terminate info segments, and
     release storage for structures describing those info segments.
   - Calls ssu_$destroy_invocation to release the ssu_ standalone
     invocation identified by the info_seg_data.sciP pointer.

In addition, the caller must call the cleanup routine described
above before returning to its caller.
```

Structure: iBlok

An iBlok structure is created for each info block within the info segment. Each block usually begins with a *block divider line.*

- A 1-block info segment is not required to begin with a block divider line.
- The first block of a multi-block subroutine description (the block introducing the subroutine) may or may not begin with a block divider line.

The block contains several lines of information: a heading line; followed by one or more paragraphs of information.

In most cases, the content of an info block:

- describes function and syntax for a **command** or **active function**; or
- gives definition information about a **general information topic**.

For an info segment describing a **multi-entry-point subroutine**, content of an info block:

- introduces the subroutine; or
- gives function and syntax information for a particular subroutine entry point.

Some commands perform multiple operations as specified by a positional argument in the command syntax. Examples are: history_comment, io_call and window_call, library_descriptor, mbuild_type, manage_volume_pool, transaction, bj_mgr_call, update_seg.

For **multi-operation commands**:

- The first info block introduces the command and gives a syntax line showing location of the operation name in the command line. A "List of operations" section in that block gives names or names with brief description for each supported operation.
- Subsequent blocks describe in detail each operation, with function and full syntax for that operation. Block divider for such per-operation blocks includes several names:
  `:Info: SHORT_CMD_NAME.LONG_OP_NAME: SHORT_CMD_NAME.SHORT_OP_NAME:`
  and those divider names also appear as external names (with .info suffix) on the info segment.

Some subsystems find it beneficial to document all aspects of the subsystem in a single info segment. Examples include: probe, mbuild.

For **single-info subsystems**, blocks may include the following.

- Description of the command that invokes the subsystem.
- Summary of all the requests implemented by the subsystem.
- Detailed description of each subsystem request or active request.
- General information topics describing one or more concepts employed by the subsystem.

Names for blocks describing subsystem requests and subsystem information topics are usually not added as external names on the info segment. The subsystem's help request accepts a topic argument, then calls help_ with: the name of subsystem's single-info segment; and a separate *info-name parameter* that selects the desired info topic.

At Multics command level, the help command can find and display the subsystem-invoking command description using the external names (e.g., probe.info and pb.info) on the info segment. At subsystem request level, its help request can find and display information about a particular subsystem request or topic using the info block divider names.

Specialized subroutines known as I/O modules are programs: attached to an I/O switch using a flexible attach description; supporting several opening modes; implementing various I/O operations, modes, and a variety of control orders. Newer I/O modules (e.g., mtape_) support flexible setup operations, each giving an opening, closing or detach description. Consolidating descriptions of these various modes, operation arguments, and control orders in blocks of a single info segment makes such online documentation easier to use and maintain.

For a **single-info I/O module**, info blocks include most of the following:

- An initial info block giving:
  - Syntax of the attach description (for use in an iox_$attach call).
  - List of opening modes supported by the I/O module.
  - List of iox_ setup operations requiring a description string from the list: iox_$open_file, iox_$close_file, and iox_$detach.
  - List of other iox_ operations supported (e.g., read_record, write_record, position, control, etc.).
  - List of iox_$control orders supported.
    - Control orders that require no input data (iox_$control info_ptr argument is null) are listed in a "List of controls" section of the main I/O module info block.
    - Control orders that require input data describe that data in an I/O Control info block. Such blocks are referenced in a "List of control operations" section of the initial I/O module info block.
- Subsequent I/O Operation info blocks document one of the iox_$open_file, iox_$close_file, and/or iox_$detach setup operations if any of these are supported by the I/O module.
- Subsequent I/O Control info blocks give details for one of the iox_$control orders that requires input data (the data pointed to by the iox_$control info_ptr argument).

The iBlok structure records the attributes of many block kinds. These include:

- Divider identifier (:Info: or :Entry: or :[Info]: or :hcom:) that began the block divider line is in the iBlok.divider element.
- Names found in the block divider line are in the iBlok.names substructure.
- Nature of the block is in the iBlok.kind element. Table 2 shows the supported block kinds.
- Heading line contents (original string, reformatted string, and individual words found after the heading date) are in the iBlok.header substructure.

Table 2: Kinds of help Data (iBlok.kind value)

1	Command
2	Active Function
3	Command/Active Function
4	General Info
5	Subroutine Introduction
6	Subroutine Brief Intro
7	Subroutine Entry point
8	Subsystem Request
9	Subsystem Active Request
10	Subsystem Request/Active Request
11	Subsystem Summary
12	Subsystem Topic
13	I/O Module
14	I/O Operation
15	I/O Control

```
dcl 1 iBlok aligned based (iBlokP),                 /* Structure describing one block of an info segment.     */
    2 common like iCommon,                          /*  - ID; pointer to, and length of chars in block;       */
                                                    /*    pointers to next/prev blocks within info segment.   */
    2 line_in_file fixed bin,                       /*  - line number (in info seg) on which block starts.    */
    2 line_count fixed bin,                         /*  - number of iLine structs in .relatives.sects list,   */
                                                    /*     + 1 header line & 1 blank line before each section */
    2 seenS bit(1) aligned,                         /*  * some block paragraphs have been displayed to user . */
                                                    /*     (* - set by help_ when paragraph printed.)         */
    2 divider fixed bin,                            /*  - records divider line starter string for block       */
                                                    /*    (one of the iBlok_divider_XXX values below).        */
    2 names like iName10,                           /*  - names in divider part of the block divider line.    */
                                                    /*     :Info: probe: pb:  2020-02-27  probe, pb           */
                                                    /*            ^^^^^  ^^                                   */
    2 kind fixed bin,                               /*  - classification of block by nature of its contents   */
                                                    /*    (one of the iBlok_kind_XXX values below).           */

    2 header aligned,                               /* Every iBlok begins with a header.                      */
      3 str char(100) var,                          /*  - full header string as it appears in info segment.   */
                                                    /*       <date> <rest-of-header>                         */
                                                    /*    length(str) must be >INFO_CHARS_PER_HEADER to show  */
                                                    /*    over-long header string.                            */
      3 reformatted,                                /*  - reformatted header with:                            */
        4 iso_date char(12) unal,                   /*     - <date> portion formatted in iso_date format.     */
        4 rest char(88) unal,                       /*     - <rest-of-header>                                 */
      3 wordN fixed bin,                            /*  - count of initial words found in <rest-of-header>    */
      3 word (5) char(72) var,                      /*  - comma-delimited tokens in <rest-of-header>.         */
                                                    /*       2020-02-27  probe, pb                          */
                                                    /*                   ^^^^^  ^^                          */
                                                    /*    For command/AF and request info blocks:             */
                                                    /*     - these words are the command/AF/request names.    */
                                                    /*     - All of these words should be in iBlok.names      */
                                                    /*       for such block kinds.                            */
      3 after_header_iLineP ptr unal,               /*  - points to first non-blank iLine that ended header   */
    2 relatives,                                    /*  Block may be child of higher-level structures.        */
      3 fileP ptr,                                  /*  - points to iFile containing this block.              */
      3 multi_operation_listP ptr,                  /*  - points to List describing operations performed by   */
                                                    /*    command/AF/request/AR described in this blok.       */
                                                    /*    Should be non-null if syntax.multi_operationsS or   */
                                                    /*    syntax.is_operationS is T.                          */
      3 multi_control_listP ptr,                    /*  - points to List describing controls performed by     */
                                                    /*    I/O Module described in this blok.                  */
                                                    /*    Should be non-null if syntax.multi_controlsS or     */
                                                    /*    syntax.is_controlS is T.                            */
      3 sects like iChild,                          /*  - list of sections contained in this block.           */
      3 pghs  like iChild,                          /*  - list of paragraphs contained in this block.         */
      3 lines like iChild,                          /*  - list of lines contained in this block.              */

    2 syntax aligned,
      3 (is_function_procedureS,                    /* SUBROUTINE ENTRYPOINT: function rather than subroutine */
                                                    /*    requires "Arguments" section even if has no parms   */
         is_command_requestS,                       /* COMMAND or REQUEST                                     */
         is_activeFunction_activeRequestS,          /* ACTIVE FUNCTION (AF) or ACTIVE REQUEST (AR)            */
         has_argumentsS,                            /* SUBROUTINE ENTRYPOINT, COMMAND or AF, REQUEST or AR:   */
                                                    /*    requires 1 or more "Arguments" sections.            */
         has_control_argsS,                         /* COMMAND or AF, REQUEST or AR:                          */
                                                    /*    requires 1 or more "Control arguments" sections.    */
         has_ellipsisS,                             /* SUBROUTINE ENTRYPOINT, COMMAND or AF, REQUEST or AR:   */
                                                    /*    ... requires 1 or more "Arguments" sections.        */
         multi_operationsS,                         /* COMMAND or AF, REQUEST or AR, or IO_MODULE             */
                                                    /*    (with associated IO_OPERATION) does several ops.    */
         multi_controlsS,                           /* IO_MODULE does several controls doc'd as :Info: blocks */
         request_summaryS,                          /* SUBSYSTEM_SUMMARY documents a list of subsystem        */
                                                    /*    request names (the operation_names).  Look for      */
                                                    /*    per-request blocks with header words same as        */
                                                    /*    those operation_names.                              */
         is_operationS,                             /* COMMAND or AF, REQUEST or AR, or IO_OPERATION          */
                                                    /*    documents one such op.                              */
                                                    /*    Its header line has the format:                     */
                                                    /*     SHORTEST_COMMAND_NAME LONG_OPERATION_NAME operation*/
                                                    /*    Get SHORTEST_OPERATION_NAME from divider (or from   */
                                                    /*    "List of operations" List of multi_operationsS blok)*/
                                                    /*    and look in each Syntax line for:                   */
                                                    /*      SHORTEST_COMMAND_NAME SHORTEST_OPERATION_NAME ... */
         is_controlS,                               /* IO_CONTROL documents one I/O control operation.        */
                                                    /*    Its header line has the format:                     */
                                                    /*     SHORTEST_COMMAND_NAME LONG_CONTROL_NAME control    */
                                                    /*    Get SHORTEST_OPERATION_NAME from divider (or from   */
                                                    /*    "List of control operations" List of                */
                                                    /*    multi_controlsS blok.  Look for "Control order:"    */
                                                    /*    section giving just name(s) of control order.       */
         is_requestS,                               /* REQUEST or AR that documents one of the requests in    */
                                                    /*    a "List of requests" section of request_summaryS    */
                                                    /*    blok.  header words must equal operation_names.     */
         has_subsystem_pathS                        /* REQUEST or AR residing in segment below substree with  */
                                                    /*    path containing entryname: subsystem or ss          */
```

```
        ) bit(1) unal,
      3 syntax_pad bit(23) unal,

    2 errors aligned,                                /* ERROR possibilities found while parsing for blocks.    */
                                                     /*     Error comments starting with - are diagnosed by    */
                                                     /*       info_seg_ and its info_seg_parse_ helper.        */
                                                     /*     Error comments starting with + are diagnosed by    */
                                                     /*       verify_info and its info_seg_verify_ helper.     */
    3 (few_blank_lines_before_blockS,                /*  - Blok preceded by <INFO_BLANK_LINES_BEFORE_PARAGRAPH */
                                                     /*    blank or all-white lines.                          */
        many_blank_lines_before_blockS,              /*  - Blok preceded by >INFO_BLANK_LINES_BEFORE_PARAGRAPH */
                                                     /*    blank or all-white lines.                          */

        long_namesS,                                 /*  - Blok divider name(s) longer than 32 chars.          */
        bad_namesS,                                  /*  - Blok divider name(s) quoted incorrectly.            */
        no_namesS,                                   /*  - Blok divider contains no names.                     */
        many_namesS,                                 /*  - Blok divider contains too many names.               */
        missing_headerS,                             /*  - Blok header is missing (zero-length)                */
        big_headerS,                                 /*  - Blok divider/header >INFO_LINES_PER_HEADER long.    */
        long_headerS,                                /*  - Blok header >INFO_CHARS_PER_HEADER in length.       */
        missing_dateS,                               /*  - Blok header does not begin with a date field.       */
        bad_dateS,                                   /*  - Blok header contains token in date position with    */
                                                     /*    DATE_CHARS rejected by convert_date_to_binary_      */
        non_iso_dateS,                               /*  - Blok header date is not in iso_date format.         */

        older_subr_intro_dateS,                      /*  + Blok header date too old; violates rule:            */
                                                     /*     subroutine_intro_date >= all entrypoint_dates      */
        bad_subr_intro_headerS,                      /*  + Blok header line for subroutine intro ^= REF_NAME_  */
        bad_subr_headerS,                            /*  + Blok header line for subroutine entrypoint name(s)  */
        bad_subr_epS,                                /*  + Blok header subroutine entrypoint not found by      */
                                                     /*     cv_entry_.                                         */

        bad_cmd_af_req_headerS,                      /*  + Blok header line for command/AF/request/AR is not a */
                                                     /*     comma-separated list of reference names            */
        bad_io_module_headerS,                       /*  + Blok header line for IO_Module is not:              */
                                                     /*     IO_MODULE_NAME I/O Module                          */
        bad_request_headerS,                         /*  - Header words not identical to operation_names in    */
                                                     /*    "List of requests" item for per-request blok.       */

        bad_dividerS,                                /*  + Blok :[Info]: divider wrong for block kind          */
        bad_divider_namesS,                          /*  + Blok_words disagree with Blok_names                 */
        obsolete_dividerS,                           /*  + Blok divider :Internal: obsolete; use :hcom:        */
        order_divider_namesS,                        /*  + Blok_words order differs from Blok_names order      */

        no_paragraphsS,                              /*  - Blok has header line, but no paragraphs.            */
        no_Syntax_sectionS,                          /*  - Blok is missing its "Syntax..." section.            */
        many_Syntax_sectionsS,                       /*  - Blok contains too many "Syntax..." sections.        */
        needs_ArgumentsS,                            /*  - Blok needs an "Arguments" section of some type.     */
        needs_Control_argumentsS,                    /*  - Blok needs "Control arguments" section of some type */

                                                     /* Operation Format errors                                */
        multi_op_Arguments_missing_op_namesS,        /*  + Blok needs "Arguments" section that lists all       */
                                                     /*    operation names for the syntax.is_operationS block  */
                                                     /*    ordered same as they appear in "List of operations" */
                                                     /*    of associated syntax.multi_operationsS block.       */
        multi_op_header_bad_cmdReqIOmod_nameS,       /*  + Header word(1) is not short name of cmd/req/IOmod   */
        multi_op_header_bad_op_nameS,                /*  + Header word(2) is not 1st operation/control name    */
                                                     /*    (in "List of ... operations" item of associated     */
                                                     /*     syntax.multi_operationsS or .multi_controlsS blok).*/
        multi_op_name_not_listedS,                   /*  + Header word(2) not in "List of ... operations".     */

        bad_section_orderS                           /*  - Blok section titles not in preferred order.         */
        ) bit(1) unal,
     3 error_pad bit(3) unal,
    iBlokP ptr;

dcl  Blok char(iBlokP->iBlok.L) based(iBlokP->iBlok.P); /* String of characters described by iBlokP->iBlok        */
                                                     /*  (the current block).                                  */

dcl  Blok_names (iBlokP->iBlok.names.N) char(entrynameL) var based(addr(iBlokP->iBlok.names.nm(1)));
                                                     /* Array of names found in current block's divider.       */
dcl  Blok_words (iBlokP->iBlok.header.wordN) char(72) var based(addr(iBlokP->iBlok.header.word(1)));
                                                     /* Array of words found in current block's header.        */
```

Structure: iLine

An iLine structure describes content of one line within an info segment, not including its ending NL delimiter (character separating the line from data beyond end of that line).

An iLine is a child of several higher-level members of the info_seg Structure Hierarchy: a line of the info segment; a line in one block of the info segment; often a line in one section or paragraph of the info block. The iLine.sibs threads link adjacent lines: of the info seg; of its parent info block; of its containing section and/or paragraph.

The iLine.relatives point to the structure describing: any containing section (an iSect structure) of that iLine; or any containing paragraph (an iPgh structure) of that iLine.

```
dcl 1 iLine aligned based (iLineP),                  /* Structure describing one line of an info segment.      */
    2 common like iCommon.content,                   /*  - ID; pointer to, and length of chars in the line.    */
    2 sibs,                                          /*  Line may be child of several higher-level structures. */
      3 file like iSib,                              /*  - thread between adjacent lines of info seg iFile.    */
      3 blok like iSib,                              /*  - thread between adjacent lines of info seg iBlok.    */
      3 sect like iSib,                              /*  - thread between adjacent lines of info seg iSect.    */
      3 pgh  like iSib,                              /*  - thread between adjacent lines of info seg paragraph.*/
    2 relatives,
      3 sectP ptr,                                   /*  - Pointer to iSect containing this line.              */
      3 pghP ptr,                                    /*  - Pointer to iPgh containing this line.               */
    2 line_in_file fixed bin,                        /*  - Line's position (sequence number) within info seg.  */

    2 errors aligned,                                /*  ERRORS detected while determining line content.       */
      3 (NL_missingS,                                /*  - Last line of info seg did not end with NL.          */

        all_whitespaceS,                             /*  - Line contains only whitespace chars.                */
        ends_whitespaceS,                            /*  - Line contains non-whitespace chars, but ends with
                                                     /*    whitespace chars.                                   */

        overlengthS,                                 /*  - Line length exceeds INFO_CHARS_PER_LINE.            */
        backspaceS,                                  /*  - Line contains BS (backspace) characters.            */
        unprintableS                                 /*  - Line contains characters not in INFO_PRINTABLE.     */
        ) bit(1) unal,
     iLineP ptr;                                     /* Points to "current" line's iLine structure.            */

dcl  Line char(iLineP->iLine.L) based(iLineP->iLine.P);  /* Character content of line described by iLineP->iLine   */
                                                     /*  (the "current" line).  Does not include NL delimiter. */
```

Structure: iPgh

An iPgh structure describes content of one paragraph. A paragraph of an info segment is 1 or more consecutive lines within an info segment, ending with either: two blank lines; or the end of the info block containing that paragraph. The two blank lines ending the paragraph are not included in the Pgh string (see the declaration below).

```
dcl 1 iPgh aligned based (iPghP),                    /* Structure describing one paragraph (group of lines) in */
                                                     /*  an info segment block.                                */
    2 common like iCommon.content,                   /*  - ID; pointer to, and length of chars in paragraph    */
    2 sibs,                                          /*  Paragraph may be child of higher-level structures.    */
      3 blok like iSib,                              /*  - thread between adjacent paragraphs of a block       */
      3 sect like iSib,                              /*  - thread between adjacent paragraphs of a section     */
    2 relatives,
      3 sectP ptr,                                   /*  - points to iSect containing this paragraph.          */
      3 lines like iChild,                           /*  - list of lines in this paragraph.                    */
                                                     /*     - follows the iLine.sibs.pghs thread.              */
    2 line_in_file fixed bin,                        /*  - paragraphs's position (starting line number) in seg */
    2 line_count fixed bin,                          /*  - number of lines within this paragraph.              */
    2 seenS bit(1) aligned,                          /*  * paragraph has been displayed to user.               */
                                                     /*     (* - set by help_ when paragraph printed.)         */
```

```
      2 errors aligned,                                 /* ERRORS detected while locating paragraph content.      */
        3 (long_paragraphS,                             /*  - iPgh.line_count >INFO_LINES_PER_PARAGRAPH           */
           few_blank_lines_before_paragraphS,           /*  - pgh preceded by <INFO_BLANK_LINES_BEFORE_PARAGRAPH  */
                                                        /*    blank or all-white lines.                           */
           many_blank_lines_before_paragraphS           /*  - pgh preceded by >INFO_BLANK_LINES_BEFORE_PARAGRAPH  */
                                                        /*    blank or all-white lines.                           */
           ) bit(1) unal,
         3 pad_errors bit(33) unal,
       iPghP ptr;

  dcl  Pgh char(iPghP->iPgh.L) based(iPghP->iPgh.P);    /* Character content of iPghP->iPgh (current paragraph).  */
```

Structure: iSect

An iSect structure describes content of one section of an info block. A section is a titled set of adjacent paragraphs in an info block.

- If the first line of any paragraph contains a colon (:) character in any but the first column, the characters preceding that colon form a section title; and that paragraph begins a section. The section includes all subsequent paragraphs until a new section title is found.
- The first paragraph following the info block heading line begins a new section, even if that paragraph does not begin with a section title.

```
  dcl  Syntax_MAX_PIECES fixed bin int static options(constant) init(10);

  dcl 1 iSect aligned based (iSectP),                   /* Structure describing a titled group of paragraphs.     */
      2 common like iCommon,                            /*  - ID; pointer to, and length of chars in section;     */
                                                        /*    pointers to next/prev sections within block.        */
      2 relatives,                                      /*  Section may be child of higher-level structures.      */
        3 blokP ptr,                                    /*  - points to iBlok containing this section.            */
        3 listP ptr,                                    /*  - points to ListP (if any) summarizing Arguments,     */
                                                        /*    Control_args, or Operation names defined in this    */
                                                        /*    section.                                            */
        3 help_listP ptr,                               /*  * points to help_'s LIST (displayable form of listP)  */
                                                        /*     (* - defaults to null; set by help_)               */
        3 pghs like iChild,                             /*  - list of paragraphs contained in this section.       */
        3 lines like iChild,                            /*  - list of lines contained in this section.            */

      2 line_in_file fixed bin,                         /*  - line number (in info seg) on which section begins.  */
      2 line_count fixed bin,                           /*  - number of iLine structs in .relatives.lines list.   */
      2 type fixed bin,                                 /*  - one of the iSect_XXX values below.  Identifies      */
                                                        /*    the section as one having known type and format     */
                                                        /*    of information.                                     */
      2 sequence fixed bin,                             /*  - verify_info sequence in which section should appear.*/

      2 title,                                          /*  - section title for this section.                     */
        3 (in_file,                                     /*     - exactly as it appears in the block (w/o COLON)   */
           case_adjusted,                               /*     - with 1st word having initial-capital, and        */
                                                        /*       subsequent words in lower case.                  */
           should_be                                    /*     - possibly restated with preferred wording or      */
                                                        /*       capitalization.                                  */
           ) char(71) var,                              /*       This length should be: >INFO_CHARS_PER_TITLE to */
                                                        /*       allow detection of long titles.                  */
      2 syntax,                                         /*  - items found in a Syntax... section.                 */
        3 N fixed bin,                                  /*     - number of items found.                           */
        3 str (Syntax_MAX_PIECES) char(1000) var,       /*     - each item gives syntax for a command/request;    */
                                                        /*       active function/request; or declare and call for */
                                                        /*       a subroutine, or declare and return value assign */
                                                        /*       for a function procedure.                        */

      2 errors aligned,                                 /* ERRORS detected which locating section contents.       */
                                                        /*     Error comments starting with - are diagnosed by    */
                                                        /*       info_seg_ and its info_seg_parse_ helper.        */
                                                        /*     Error comments starting with + are diagnosed by    */
                                                        /*       verify_info and its info_seg_verify_ helper.     */
        3 (bad_titleS,                                  /*  - Section title marked BAD (not preferred)            */
           blank_titleS,                                /*  - Section title is all whitespace.                    */
           long_titleS,                                 /*  - Section title length >INFO_CHARS_PER_TITLE          */
           unrecognized_titleS,                         /*  - Section title not known to info_seg_.               */
           untitledS,                                   /*  - Section title not found in first section of block.  */
                                                        /*     This is not an error, just an unusual circumstance */
```

```
          syntax_title_wrong_for_kindS,               /*  - Syntax... title wrong for block kind.              */
          syntax_for_subroutine_invalidS,             /*  - Syntax: missing declare/dcl or call or = (assign)  */
          syntax_missing_left_bracketS,               /*  - Syntax as an active... missing [ in line           */
          syntax_missing_right_bracketS,              /*  - Syntax as an active... missing ] in line           */
          syntax_missing_namesS,                      /*  - Syntax... has none of words in block header.       */
          syntax_missing_short_nameS,                 /*  + Syntax... does not use shortest word in block header*/
          syntax_uses_control_argumentsS,             /*  - Syntax... uses -control_arguments                  */
          syntax_uses_control_argumentS,              /*  - Syntax... uses -control_argument                   */

          list_of_ops_bad_namesS,                     /*  - List of ... operations: names non-unique or names  */
                                                      /*                            have non-descending length */
          list_of_ops_long_nameS,                     /*  - List of ... operations: length(name) > 32          */
          list_of_ops_nonunique_namesS,               /*  - List of ... operations: all item names not unique  */
          list_of_ops_op_not_documentedS,             /*  - List of ... operations: some op items have no op   */
                                                      /*                            documentation block        */
          control_order_bad_listS,                    /*  + Control order: line constructed badly.  More than  */
                                                      /*                   2 names, or nothing before comma.  */
          control_order_bad_name_orderS,              /*  + Control order: name order differs from entry in    */
                                                      /*                   "List of control operations" section.*/

                                                      /* error checks for Subroutine Introduction:            */
          bad_entry_points_inS                        /*  + "Entry points in" section formatted incorrectly.  */
          ) bit(1) unal,
        3 pad_errors bit(15) unal,
        3 display_listS bit(1) unal,                  /*  - verify_info displays iList to aid error msgs      */
      iSectP ptr;

dcl   Sect char(iSectP->iSect.L) based(iSectP->iSect.P);  /* Character content of iSectP->iSect (current section)  */
                                                          /*  as it appears in the info segment.                   */
```

Certain section titles are expected, or are optional but recommended, in particular kinds of info blocks. These titles are categorized by the following iSect.type values:

```
dcl (iSect_ACCESS_REQUIRED                      init( 1), /* Access required:                                  */
     iSect_ARGUMENTS                            init( 2), /* Arguments:                                        */
     iSect_ARGUMENTS_FOR_IO_CALL                init( 3), /* Arguments for io_call:                            */
     iSect_ARGUMENTS_FOR_IOX_CONTROL            init( 4), /* Arguments for iox_$control:                       */
     iSect_ARGUMENTS_FOR                        init( 5), /* Arguments for ...:                                */
     iSect_ARGUMENTS_SUBSET                     init( 6), /* Arguments (...):                                  */
     iSect_CONTROL_ARGUMENTS                    init( 7), /* Control arguments:                                */
     iSect_CONTROL_ARGUMENTS_AS_A_COMMAND       init( 8), /* Control arguments as a command:                   */
     iSect_CONTROL_ARGUMENTS_AS_A_REQUEST       init( 9), /* Control arguments as a request:                   */
     iSect_CONTROL_ARGUMENTS_AS_AN_ACTIVE_FUNCTION  init(10), /* Control arguments as an active function:      */
     iSect_CONTROL_ARGUMENTS_AS_AN_ACTIVE_REQUEST   init(11), /* Control arguments as an active request:       */
     iSect_CONTROL_ARGUMENTS_FOR_ATTACH_DESCRIPTION init(12), /* Control arguments for attach description:     */
     iSect_CONTROL_ARGUMENTS_FOR_OPEN_DESCRIPTION   init(13), /* Control arguments for open description:       */
     iSect_CONTROL_ARGUMENTS_FOR_CLOSE_DESCRIPTION  init(14), /* Control arguments for close description:      */
     iSect_CONTROL_ARGUMENTS_FOR_DETACH_DESCRIPTION init(15), /* Control arguments for detach description:     */
     iSect_CONTROL_ARGUMENTS_FOR                init(16), /* Control arguments for ...:                        */
     iSect_CONTROL_ARGUMENTS_SUBSET             init(17), /* Control arguments (...):                          */
     iSect_CONTROL_ORDER                        init(18), /* Control order:                                    */
     iSect_ENTRY_POINTS_IN                      init(19), /* Entry points in SUBROUTINE_NAME:                  */
                                                          /*    (List is generated by the help command)        */
     iSect_EXAMPLES                             init(20), /* Examples:                                         */
     iSect_FUNCTION                             init(21), /* Function:                                         */
     iSect_LIST_OF_CONTROL_OPERATIONS           init(22), /* List of control operations:                       */
     iSect_LIST_OF_CONTROLS                     init(23), /* List of controls:                                 */
     iSect_LIST_OF_ELEMENTS                     init(24), /* List of elements:                                 */
                                                          /*  [after section "Arguments for iox_$control"]     */
     iSect_LIST_OF_IO_OPERATIONS                init(25), /* List of i/o operations:                           */
     iSect_LIST_OF_MODE_STRINGS                 init(26), /* List of mode strings:                             */
     iSect_LIST_OF_OPENING_MODES                init(27), /* List of opening modes:                            */
     iSect_LIST_OF_OPERATIONS                   init(28), /* List of operations:                               */
     iSect_LIST_OF_REQUESTS                     init(29), /* List of requests:                                 */
     iSect_LIST_OF                              init(30), /* List of ...:                                      */
     iSect_NOTES_ON_THE_INFO_PTR                init(31), /* Notes on the info ptr:                            */
     iSect_NOTES                                init(32), /* Notes:                                            */
     iSect_NOTES_ON                             init(33), /* Notes on ...:                                     */
     iSect_SYNTAX                               init(34), /* Syntax:                                           */
                                                          /*  [first section in SUBSYSTEM REQUEST INFO]        */
     iSect_SYNTAX_AS_A_COMMAND                  init(35), /* Syntax as a command:                              */
     iSect_SYNTAX_AS_AN_ACTIVE_FUNCTION         init(36), /* Syntax as an active function:                     */
     iSect_SYNTAX_AS_AN_ACTIVE_REQUEST          init(37), /* Syntax as an active request:                      */
     iSect_SYNTAX_OF_ATTACH_DESCRIPTION         init(38), /* Syntax of attach description:                     */
     iSect_SYNTAX_OF_OPEN_DESCRIPTION           init(39), /* Syntax of open description:                       */
     iSect_SYNTAX_OF_CLOSE_DESCRIPTION          init(40), /* Syntax of close description:                      */
     iSect_SYNTAX_OF_DETACH_DESCRIPTION         init(41), /* Syntax of detach description:                     */
     iSect_Untitled                             init(42), /*  [used when there is no section title.]           */
     iSect_Another_Title                        init(43)  /*  [used if title other than one of those above     */
                                                          /*   is present.]                                    */
     ) fixed bin int static options (constant);
```

Subroutine: info_seg_verify_

The verify_info command uses the info_seg_verify_ subroutine to: parse and verify info segments; and to display rules and guidelines for info segment which are maintained in the info_seg_specifications_ data structures.

Entry Point: info_seg_verify_$iFiles

For each iFile structure in the verify_info_data.info_seg_data.files threaded list:

- Call info_seg_$parse_iFile to get a hierarchy of structures describing components of the info segment.
- Examine those structures, reporting failures to meet info segment organization and formatting guidelines.

```
Syntax:
   declare info_seg_verify_$iFiles entry (ptr);
   call info_seg_verify_$iFiles (verify_info_dataP);


Arguments:
verify_info_dataP
   points to the verify_info_data structure (see
   verify_info_data.incl.pl1).
```

Structure: verify_info_data

The input pathnames and verification control arguments for the verify_info command are organized into the verify_info_data structure, shown below.

```
dcl 1 verify_info_data aligned based(verify_info_dataP),
    2 vid_version char(16),                          /*  - version of this struct: verify_info_data_version_01 */

    2 inputs,
                                                     /* Inputs for: info_seg_verify_$display_specifications:   */
      3 rules_area fixed bin,                        /* -rules OPERAND  select which rules to display:         */
                                                     /*                 a VI_RULE_xxx value from list below.   */



                                                     /* Inputs for: info_seg_verify_$iFiles                    */
                                                     /*  (includes all remaining verify_info_data elements)    */
      3 switches,
        4 (totalsS,                                  /* -totals: (T) only set error_count, highest_severity.   */
                                                     /*          (F) display individual error messages.        */
           briefS,                                   /* -brief:  (T) block/section errors displayed as keys at */
                                                     /*              end of vi block/section heading output.   */
                                                     /*          (F) full error messages are displayed.        */

           fileS,                                    /* -debug file  display details of file structure.        */
           blockS,                                   /* -db block    display block divider and header details. */
           sectionS,                                 /* -db section  display Syntax section components.        */
           listS                                     /* -db list     display section items in "Arguments",     */
                                                     /*              "Control arguments" and "List of ..."     */
           ) bit(1) aligned,
      3 naming fixed bin(2),                         /* -names, -no_names, -force_names setting:               */
                                                     /*                 a VI_NAMING_xxx value                  */

      3 lines,                                       /* display info seg lines, for debugging purposes         */
        4 file,                                      /* -lines {count}   OR   -lines start:{count}             */
          5 (start, count) fixed bin,
        4 blok,                                      /* -block {count}   OR   -block start:{count}             */
          5 (start, count) fixed bin,
```

```
    2 results,
      3 segs_processed fixed bin,                     /*  - count of info segs found and parsed.               */
      3 error_count fixed bin,                        /*  - count of errors encountered                        */
      3 highest_severity fixed bin,                   /*  - highest severity error encountered verifying infos  */

    2 isd aligned like info_seg_data,                 /*  - info_seg_data structure used to parse segs being    */
                                                      /*    verified.  See info_seg_dcls_.incl.pl1             */

     verify_info_dataP ptr;                           /* Pointer to verify_info_data structure.               */

dcl  verify_info_data_version_01 char(14) int static options(constant) init("verify_info_01");
                                                      /* Currently supported version of verify_info_data struct */

                                                      /* Should names on info segment be changed?             */
dcl (VI_NAMING_off          init(0),                  /*  - no, issue errors for names outside guidelines.    */
     VI_NAMING_query        init(1),                  /*  - ask user about removals; add/reorder without asking */
     VI_NAMING_force        init(2)                   /*  - make all changes per guidelines without asking, but */
                                                      /*    all changes are reported via error messages.       */
     ) fixed bin(2) int static options(constant);

dcl  VI_RULES_AREA (12,2) char(12) var int static options(constant) init(
                                                      /* Keep items in VI_RULES_AREA synchronized with index    */
                                                      /*  values VI_RULE_xxx defined below.                     */
                           "all",              "a",   /*   - VI_RULE_all_areas                                  */
                           "file",             "f",   /*   - VI_RULE_file_structure                             */
                           "block",            "bk",  /*   - VI_RULE_block_kind                                 */
                           "section",          "scn", /*   - VI_RULE_section_titles                             */

                           "all_kinds",        "ak",  /*   - VI_RULE_all_kinds                                  */
                           "command",          "cmd", /*   - VI_RULE_command_titles                             */

                           "subsystem",        "ss",  /*   - VI_RULE_subsystem_titles                           */
                           "request",          "req", /*   - VI_RULE_request_titles                             */

                           "topics",           "topic", /*   - VI_RULE_topic_titles                             */
                           "general_info",     "gi",  /*   - VI_RULE_gi_titles                                  */

                           "subroutine",       "subr", /*   - VI_RULE_subroutine_titles                         */
                           "io_module",        "io"   /*   - VI_RULE_io_titles                                  */
                           );

dcl (VI_RULE_unset             init( 0),              /* The only value not represented in VI_RULES_AREA above. */
     VI_RULE_all_areas         init( 1),
     VI_RULE_file_structure    init( 2),
     VI_RULE_block_kind        init( 3),
     VI_RULE_section_titles    init( 4),

     VI_RULE_all_kinds         init( 5),
     VI_RULE_command_titles    init( 6),

     VI_RULE_subsystem_titles  init( 7),
     VI_RULE_request_titles    init( 8),

     VI_RULE_topic_titles      init( 9),
     VI_RULE_gi_titles         init(10),

     VI_RULE_subroutine_titles init(11),
     VI_RULE_io_titles         init(12)
     ) fixed bin int static options(constant);

dcl  VI_DEBUG_OPERAND (4,2) char(16) var int static options(constant) init(
                           "file",             "f",
                           "block",            "bk",
                           "section",          "scn",
                           "list",             "ls"
                           );
dcl (VI_DEBUG_file             init( 1),
     VI_DEBUG_block            init( 2),
     VI_DEBUG_section          init( 3),
     VI_DEBUG_list             init( 4)
     ) fixed bin int static options(constant);
```

Entry Point: info_seg_verify_$display_specifications

Display guidelines used by verify_info in its checking of info segment format and content. Areas covered by these guidelines include:

- Supported info segment file structures (ordering of info blocks).
- Rules for determining an *info block's kind*: a value asserting that the info block describes a command or active function; a subroutine; a subsystem request or active request; another subsystem topic; an I/O module; or some other general information.
- A list of preferred section titles used in info blocks, with recommendations for mapping each deprecated (obsolete) title to a preferred title.
- Section titles used in each kind of block, indicating their preferred order of appearance and whether they are required or optional.

Appendix B: verify_info -rules shows all of these guidelines.

```
Syntax:
   declare info_seg_verify_$display_specifications entry (ptr);
   call info_seg_verify_$display_specifications (verify_info_dataP);

Arguments:
verify_info_dataP
   points to the info_seg_data structure.  Refer to
   verify_info_data.incl.pl1.

List of selectors:
   The verify_info_data.rules_area element selects the info seg
   guidelines to display.  This element may have one of the following
   constants (also declared in verify_info_data.incl.pl1).
VI_RULE_all_areas
  displays all rules and guidelines used by verify_info.
VI_RULE_file_structure
  displays guidelines for supported file structures (allowed order of
  :Info: :[Info]: :Entry: and :hcom: block divider lines within an
  info segment).
VI_RULE_block_kind
  displays rules used to determine the kind of each info block
  (e.g., command or active function, subroutine, subsystem request,
  etc.).

VI_RULE_section_titles
  displays Multics standard info segment section titles, and a list of
  obsolete or deprecated titles with a preferred replacement.
VI_RULE_all_kinds
  displays section titles typical for each kind of info block,
  including the usual order of appearance within the block of each
  title.  Use a selector below to display section titles for
  particular kinds of info blocks.

VI_RULE_command_titles
  displays section titles typical for an info block describing a
  command or active function.
```

```
VI_RULE_gi_titles
  displays section titles typical for a generic info block (.gi.info
  segment).

VI_RULE_subsystem_titles
  displays section titles typical for an info block describing a
  subsystem request or active request; a subsystem request summary; or
  other subsystem topic.
VI_RULE_request_titles
  displays section titles typical for an info block describing a
  subsystem request or active request.
VI_RULE_topic_titles
  displays section titles typical for an info block describing a
  subsystem request summary; or other subsystem topic.

VI_RULE_subroutine_titles
  displays section titles typical for an info block describing a
  subroutine or function.  This includes a subroutine introduction
  block, and blocks describing each subroutine entry point.
VI_RULE_io_titles
  displays section titles typical for an info block describing an
  I/O module; or one of its setup operations (open_file, close_file,
  or detach); or one of its control orders.
```

Testing verify_info

The new verify_info command was tested using several test strategies.

- verify_info displays location/title for each block, section, and line to ensure that info_seg_ properly parses the info segment into the component structures of the info_seg_ Structure Hierarchy; and correctly diagnoses low-level errors. Block and section boundaries are usually displayed; line contents and per-line errors are displayed under control of debugging arguments.
- A library of test info segments verify that each error is properly detected and reported. This includes both low-level errors diagnosed by info_seg_parse_ and higher-level errors diagnosed by info_seg_verify_iFile_.
- Output of verify_info was compared with output from the installed validate_info_seg (vis) command to ensure that all the errors were caught and reported by both commands. In many cases, verify_info reports errors missed (or incorrectly diagnosed) by validate_info_seg. In some cases, it reports errors more clearly than the messages coming from vis.
- The verify_info command was run against all the info segments in the >doc>info directory: to ensure that all detected errors are reported properly; to look for errors incorrectly diagnosed by verify_info; and to identify any category of errors that are poorly handled by verify_info and validate_info_seg.
- verify_info was enhanced to better identify and check significant categories of info segments that were poorly handled by validate_info_seg. For these new checks, test case info segs were added to check for correct diagnosis by the enhanced code.

Results of these testing strategies are described in the sections that follow.

verify_info Display Information

A new mechanism to parse an entire info segment into lines, blocks, paragraphs and sections can succeed only if verify_info displays enough information to assure the user that info seg text has been correctly parsed into the intended component structures of the info_seg_ Structure Hierarchy.

The usual verify_info output includes: information from the iFile structure; names and starting line number for each iBlok structure; and section title and line number for each iSect structure.

The -lines and -block control arguments display a subset of lines found in the info seg, or in each block of the info seg. Output includes line text, line number within the file, and an interpretation of any low-level errors or indicators diagnosed for each iLine structure.

This output data was examined while performing each of the testing strategies described below.

Info Seg Test Cases

Each low-level error diagnosed by info_seg_parse_ and each higher-level error diagnosed by info_seg_verify_iFile_ is recorded as an error flag in the info_seg_ structure to which the error pertains. As error reporting code was added to info_seg_verify_iFile_ to report each flag, a numbered info segment case file was created to trigger that error. These test cases were run: individually as the reporting code was added to the software; and at the end of the development cycle to ensure error detection and reporting remains effective.

Comparison with validate_info_seg Errors

The list of errors reported by the validate_info_seg (vis) command was used as the starting point for errors that should be detected by the new verify_info (vi) command. In most cases, validate_info_seg code looking for a particular error works correctly. validate_info_seg main flaws are: mis-identifying the kind of info block being examined, causing it to mis-diagnose section title contents or ordering; and poor code structuring, making it difficult to enhance vis for newer info block types (complex subroutines like I/O modules, multi-operation commands, info segments describing compose macros for generating manuals, etc.).

Each of the validate_info_seg known errors is listed in an error message array. That array was copied into the new info_seg_verify_iFile_.pl1 source as a commented checklist. Error messages were then added to the new code for each low-level error (diagnosed during parsing of the info seg). Test case info segs were found/created containing each of these low-level errors. Then each test case was checked with validate_info_seg and verify_info to ensure that both detected the same error at the same location in the test info seg; and that verify_info reported the error in a similar or better manner than validate_info_seg.

This first comparison phase ended when all the low-level errors were known to be properly handled by verify_info, and were marked COMPLETE in the checklist.

In the second comparison phase, code was added in info_seg_verify_iFile_.pl1 to detect and report each of the incomplete errors in the checklist (errors handled by validate_info_seg but not yet handled by verify_info). As each error was handled, a test case info seg was added to the test suite; and that test was checked by validate_info_seg and verify_info. When the verify_info code correctly detected and reported the error, that entry in the checklist was marked as COMPLETE. This process was repeated for each of the remaining errors detected by validate_info_seg.

The second comparison phase ended when all errors in the checklist were marked COMPLETE. At this point, verify_info was diagnosing all the errors detectable by validate_info_seg, plus several other errors.

Checking Info Segments in >doc>info

Phase three of testing verify_info called for checks of all the info segments in >doc>info: a directory containing info segments documenting commands, general information, and subroutines targeted to the general Multics user community. Goals of this testing: to ensure that all detected errors are reported properly; to look for errors incorrectly diagnosed by verify_info; and to identify any category of errors that are poorly handled by verify_info and validate_info_seg.

To facilitate this testing, a special vi_test.ec exec_com was written.

- ec vi_test LETTER: generates a list of all the >doc>info segments whose first name begins with a given LETTER of the alphabet, storing that list in a subdirectory of the working dir: tests
- The exec_com then runs verify_info against the first file in the list, using default verify_info arguments. It asks the developer to review the vi output for that file.
    - If the output shows expected errors which are correctly reported by verify_info, the developer responds “ok” or “yes”; the info seg’s pathname is moved to the end of its list (beyond an ALREADY_ANALYZED marker line).
    - If the verify_info output shows unexpected errors, is missing errors, or errors are incorrectly reported, the developer responds “no”. The exec_com asks the developer for any additional verify_info control arguments to use in running the test again. The developer can give vi debugging arguments to get additional data; or can .. escape another command line to look at contents of the info segment, or to get validate_info_seg output for that segment, etc.
    - If the verify_info output seems very incorrect, the developer responds “problem”; the exec_com copies the info segment into a problems subdirectory for later detailed analysis by the developer. This might lead to changes in the verify_info code.
    - If the verify_info output correctly identifies severe errors in the info segment, the developer responds “fixes”; the exec_com copies the file to a fixes subdirectory for later corrections within the info segment itself.
    - If the verify_info output identifies no errors in the info segment, the develop responds “delete”; the exec_com removes the pathname for that file from the alphabetical list.
    - The user may also respond “quit”; the exec_com stops operation, leaving the top-most unchecked info seg pathname for checking during a later use of the exec_com.

As a result of this phase three testing, a variety of issues were identified and fixed in the info_seg_parse_ and info_seg_verify_iFile_ design and code.

Several categories of info segments had contents that was not properly identified by either validate_info_seg or verify_info as one of the known info block kinds (command, active function, subroutine, general info, etc.). For most of these categories, the design and/or rules applied by verify_info could be enhanced to properly identify those info blocks. For some categories, it was easiest to make simple changes to info seg content to make the segment identifiable under existing verify_info rules.

A few of these problematic info seg categories are used only infrequently and/or probably will not be changed in the future. These side-line info categories may therefore be noted as “not being properly handled” by verify_info.

The info segment categories poorly handled by either validation_info_seg or verify_info are briefly described in the following subsections.

Category: Command Info Segments Created Before 1985

Info seg formatting standards changed in early 1985, shortly after the ssu_ subsystem utility was installed.

Prior to 1985, info segments documenting a command were recognized by having as first two section titles: "Syntax" and "Function". Info segs documenting a subroutine entrypoint had first two section titles: "Function" and "Syntax" in that order. When ssu_ was installed, there was a desire to have subsystem requests use the "Syntax" and "Function" title ordering of commands, because requests were like mini-commands. But if that were done, how would code distinguish between info segs for commands versus subsystem requests?

Since there was already an issue in determining from an info seg whether a command could also operate as an active function, an info seg formatting guideline was created.

- All command info segments created or modified after 1984 would use the section titles: "Syntax as a command" and "Function".
- A command that also operated as an active function would begin with three section titles: "Syntax as a command", "Syntax as an active function", and "Function".
- A subsystem request would begin with titles: "Syntax" and "Function".
- A subsystem active request would begin with titles: "Syntax as an active request" and "Function".
- The very few subsystem requests that also operated as an active request would begin with: "Syntax", "Syntax as an active request", and "Function".

validate_info_seg was changed to accept info segments using the new section titles guidelines. But it was not changed to identify info segs written before 1985 that followed the old guideline. Thus, info segs for commands written before 1985 did not get upgraded to the new guidelines when they were changed in later years, unless the developer or auditor remembered to apply the new guideline without assistance from validate_info_seg.

verify_info adds new rules to deal with this issue:

- An info block with header date prior to 1985 and first two section titles of "Syntax" and "Function" is treated as a describing a command; errors are displayed to upgrade the first section title from "Syntax" to "Syntax as a command".
- An info block with header date of 1985 or later and first two section titles of "Syntax" and "Function" is treated as describing a subsystem request.

Appendix B: verify_info -rules (**verify_info -rules block** page) lists those new rules as Rules 27 and 28.

During the phase three check of >doc>info, 39 command info segs were found to have a header date later than 1984 while still using the older section title: "Syntax". Such info segs would be treated as documenting a subsystem request by verify_info's new rules. The easiest remedy was to manually change these info segments to use the newer "Syntax as a command" title. These 39 are the largest category of info segs to be fixed as part of installing verify_info.

Category: Info Segments with Severe Errors

Severe errors may occur in info segments containing multiple blocks. Such errors cause help or check_info_segs to fail to locate the info segment or info block within the info segment. An external name might be missing from a command info segment; or a block divider name could be missing or misspelled in a command or subroutine entrypoint info block; or a header date might not have been modified, causing the check_info_segs command to fail in identifying a changed command or subroutine info segment. validate_info_seg does not check for such errors.

During the phase three check of >doc>info, verify_info correctly reported such errors in 33 info segments. A corrected version of these info segments has been prepared for installation with verify_info.

Category: Missing/Unretained Subroutine Entry Points

The phase three check uncovered >doc>info segments for several subroutines that no longer exist. verify_info tries to verify subroutine entry point names by calling cv_entry_ to access each entry point. It reports a possible error for any entry point which cv_entry_ does not locate (using the user's current linker search paths).

As part of installing verify_info, the following 4 info segments should be deleted.

- **meter_gate_.info**: at one point, a meter_gate$meter_gate_ entry point existed in meter_gate.pl1. That entry point was documented in >doc>info>meter_gate_.info. However, the entry point no longer exists while the info segment is still installed.
- **parse_channel_name_.info**: says it converts an IOM tagNumber combination to an IOM tag, and an associated decimal channel number. But no source/object for parse_channel_name_ exists in Multics Libraries; nor does any library program (listed in the crossreference) call this subroutine.
- **spg_util_.info** and **spg_ring_0_info_.info**: source for these subroutines exists in bound_spg_.s.archive, but the reference names are not added to bound_spg_; nor are entry points in those subroutines retained in bound_spg_. Thus, documentation for these internal procedures should be removed from >doc>info (perhaps by deleting the two info segments).

Category: Mis-named Info Segments

2 general info segs have a name that does not follow the latest info seg naming guidelines.

- **old_fortran.bugs.info** should be named: old_fortran.errors.info
- The name **process_overseer_.info** could be confused as a subroutine description. The name process_overseer_.gi.info should be added to clarify that it is a general info description of the process overseer mechanism.

Category: Two versions of Same Info Segment

The info segment **error_table_conversion.gi.info** contains latest information on creating a program-specific error table. However, an earlier version of this segment is still in the library; it is named: **error_table_conversion.info**. The earlier version should be deleted, with its name being added to the newer info segment.

Category: Info Segment that is Out-of-date

The **multics_libraries_.info** segment contains an out-of-date description for the directories currently defined in the multics_libraries_.ld library descriptor segment. Libraries no longer present were removed from the info segment; and the remaining libraries and directories were described using "List of …" sections that can be quickly displayed by the `help multics_libraries_ –brief` command.

Category: Info Segments for Pre-Access Commands

The **dial.info** segment contains documentation for 16 pre-access requests (requests that can be typed when connecting to a Multics system by dial-up or hardwired telephone line prior to, or instead of, giving a login command). Most pre-access requests are documented as a separate info block within the dial.info segment. An **access_requests** info block lists names of all these requests.

Unfortunately, none of these blocks are accessible to users via the help command, because the only name on this info segment is dial.info. There is no Multics dial command; only a dial pre-access request. And dial.info gives only meager information about the pre-access environment in which the dial request may be used. Thus, users may be confused about when the dial request may be used.

Similarly, **enter.info**, **login.info**, **terminal_id.info** and **terminal_type.info** segments also describe pre-access requests, but in most cases do not adequately describe the pre-access (dial-in) environment in which such requests may be used.

This causes information in the dial.info segment to be hidden from users, or to be harder for users to understand.

The remedy is to combine documentation for all these pre-access requests into one multi-block info segment, adding names so each block may be accessed by the help command. In addition, **logout.info** will be added as another block in this info segment, thus combining information about all requests for entering and exiting Multics.

The access_requests block will give a one-line summary of each pre-access request (requests that setup the terminal for use on Multics), and of the access requests (requests to login to Multics, or enter Multics as an anonymous user). The logout.info documentation will be referenced as a pre-access request in the sense of returning from the Multics environment to the pre-access environment (a logout command undoes a login or enter request).

Category: Info Segments for Multi-Operation Commands

Info segments documenting multi-operation commands (those that perform several different operations) use: a variety of section titles; differing info block divider naming conventions and heading line formats for info blocks documenting each operation; and a variety of text advising the user how to get help for a particular operation. These differences make it difficult for validate_info_seg and verify_info to recognize a multi-operation command info segment, so they might assist the writer to ensure that all operations are listed in a summary, documented in individual blocks, and have identical operation names in all these references. The different formatting methods used cause problems for the readers as well.

During the phase three check of >doc>info, both verify_info and validate_info_seg reported issues with 9 info segments documenting multi-operation commands. Several of these commands are important tools for: changing setup of a process (**io_call**, **window_call**, **bj_mgr_call**); or documenting changes in program source modules (**history_comment**); or performing system maintenance and administrative operations (**update_seg**, **library_descriptor**, **manage_volume_pool**).

Because a significant number of important tools share this issue, a new info segment guideline was developed covering format of an info segment for a multi-operation command. This new guideline is described in a later chapter of this bulletin: Guidelines for Multi-Operation Info Segments.

The 9 multi-operation info segments found during the phase three check will be revised to follow the new guideline. And verify_info has been extended to recognize multi-operation info segments attempting to use this guideline, and to check for consistency in following the guidelines in all info blocks of such segments.

Corrections for the 9 multi-operation info segments have been prepared for installation with verify_info.

Category: Info Segments for I/O Modules

Info segments documenting I/O modules (subroutines following the iox_ protocols) share many of the same problems found in multi-operation command info segs. In most cases, I/O module info segs try to describe some of the aspects of the I/O module in a single info block. This can lead to a block having several different Syntax descriptions, many different Notes sections, and List of … sections having a variety of section titles. Readers are often confused by the shortened discussions of I/O module calling syntax and usage sequences in these info segments.

To help standardize format of info segments for I/O modules, a new guideline has been added. This new guideline is described in a later chapter of this bulletin: Guidelines for I/O Module Info Segments.

The info segment for one of the newest I/O modules was converted to the format referenced in this guideline. The **mtape_** I/O module supports operations on tape media in ANSI and IBM standard formats. It follows the extended iox_ protocols for the open_file, close_file and detach operations that accept description strings (similar to an I/O module attach description) to tailor the operation. It also supports a variety of control operations, some of which expect a data structure pointed to by the iox_$control info_ptr argument. This is the only Multics I/O module that can demonstrate use of all parts of the I/O module info segment guidelines.

A full revision of the mtape_.info segment has been prepared for installation with verify_info.

Category: ssu_ Subsystem Info Segments

Most subsystems written prior to creation of the Subsystem Utilities (ssu_) either did not have an internal help request, or created their own mechanism for internally documenting requests.

Subsystems that use the ssu_ mechanism also use its facility for: separate per-subsystem info seg search paths; and its ssu_-provided help and list_help requests. Each subsystem has its own directory (below >doc>subsystems) containing individual info segs, each documenting one of the subsystem's requests. Most of these individual info segs follow the guidelines for a command info segment written before 1985. Fortunately, these are the same guidelines used today for subsystem request documentation.

Both validate_info_seg and verify_info treat such files as documenting a subsystem request. However, validate_info_seg reports problems with the unusual heading line format used in these older subsystem info segs:

```
09/28/82  read_mail request:  mailbox, mbx
```

vis thinks that "read_mail" and "request:" are names on the request, and complains that those names (with .info suffix added) should be external names on the info segment.

When phase three testing began, verify_info considered info blocks with heading line date before 1985 and first section title of "Syntax" as documenting a command; a block with heading line date later than 1984 and first section title of "Syntax" was treated as documenting a subsystem request. To force verify_info to treat all info segments below >doc>subsystem as documenting a subsystem request, rules were added to verify_info: any info segment residing below a directory named "…>subsystems>…" or "…>ss>…" is documenting a subsystem request. These rules also cause verify_info to ignore heading line words preceding a colon (e.g., the "read_mail" and "request:" words in the heading line above), treating only words following the colon as names for the request. So verify_info does not issue the bogus errors about external names missing from the info segment.

However, if a developer fetches a copy of one of these subsystem request info segs and tries to modify it, the fetched file probably won't reside below a directory named "…>subsystems>…" or "…>ss>…". If the fetched info seg has a heading line date before 1985, then verify_info treats this file as being in the older command format, and issues a bogus complaint that its "Syntax" section should now be called "Syntax as a command". However, if the user remembers to modify the heading line date to the current date (to reflect that the info segment is being modified), then verify_info rules will continue treating the info segment as documenting a subsystem request.

The rules used by verify_info to identify the kind of an info block are shown in Appendix B: verify_info -rules (**verify_info -rules block** page). Rules 16 through 21 identify segs residing below a directory named "…>subsystems>…" or "…>ss>…" as documenting a subsystem request. Rule 27 identifies an info block with heading line date before 1985 and first section title of "Syntax" as documenting a command. Rule 28 identifies an info block with heading line date after 1984 as documenting a subsystem request. If an info block matches two or more rules, the lowest-numbered rule determines the info block kind.

Category: Subsystems that Document Requests in a Single Multi-Block Info Segment

The phase three testing found three subsystems that combine documentation for subsystem requests as info blocks in a single info segment. probe and mbuild place those request documentation blocks in the info segment documenting the command: **probe.info** and **mbuild.info**. The help_ subroutine places request documentation blocks in an info segment called **help_responses.**info. Each subsystem uses its own help request program to call help_ directly. The call gives the name of the containing info segment, and a TOPIC or REQUEST_NAME (called help_args.info_name) to select an info block for display from this combined info segment.

These subsystem request info blocks differ from other multi-block info segments because names on request documentation and subsystem topic info blocks should NOT be external names (with .info suffix added) on the containing info segment. They are visible only to the subsystem's help request, and are not findable with Multics help command.

validate_info_seg complained about missing names on two of the info segments mentioned above; but does not complain about missing names on mbuild.info for some reason.

verify_info complained about missing external names for all three of these request-documenting multi-block info segments.

To avoid these bogus errors from verify_info, a new info block divider token is proposed to designate an info block whose names are not to be placed as external names on the info segment. Such info blocks are only findable if help_ is called with the name of the desired info block along with a separate name (an external entry name or full pathname) for the multi-block info segment containing the desired info block. probe, mbuild, and help_ have their own help request that knows the entry or path name for the info segment containing documentation for its requests.

Prior to this change, the accepted info block divider tokens were those words shown in bold font below.

```
:Info: NAME1: ... {NAMEn:}  BLOCK_HEADING
          Normal block divider.  From 1 to 10 names may be given.
          BLOCK_HEADER may appear on line following divider token.
          NAMEs are expected to appear as names on the info segment
          with an ".info" suffix (e.g., NAME1.info).
:Entry: EP_NAME1: ... {EP_NAMEn:}  BLOCK_HEADING
          Subroutine entrypoint description block divider.  From 1 to 10
          EP_NAMEs may be given, each up to 32 characters in length.
:Internal: history_comment.gi:  BLOCK_HEADING
          Info segment history comment divider.  Must be final divider
          in info seg.  All lines following this divider are ignored by
          help_.
```

This proposal changes this list to have the following acceptable info block divider tokens.

```
:Info: NAME1: ... {NAMEn:}  BLOCK_HEADING
          Normal block divider.  From 1 to 10 names may be given.
          BLOCK_HEADER may appear on line following divider token.
          NAMEs are expected to appear as names on the info segment
          with an ".info" suffix (e.g., NAME1.info).
:[Info]: NAME1: ... {NAMEn:}  BLOCK_HEADING
          Hidden block divider.  Like :Info: divider, except NAMEs
          will not appear as names on the info segment.  Hidden blocks
          are accessed by a subsystem's help request, which calls help_
          with a specific info seg PATHNAME and separate INFO_NAME to
          locate a particular hidden block.
:Entry: EP_NAME1: ... {EP_NAMEn:}  BLOCK_HEADING
          Subroutine entrypoint description block divider.  From 1 to 10
          EP_NAMEs may be given, each up to 32 characters in length.
:hcom:
          Info segment history comment divider.  Must be final divider
          in info seg.  All lines following this divider are ignored by
          verify_info and help_.
```

The `:Internal:` token continues to be recognized by help_ as an obsolete version of the newer `:hcom:` divider. verify_info reports use of `:Internal:` as an error, recommending use of the simpler `:hcom:` divider line to resolve that error.

Corrections for the 3 subsystems documenting their requests as blocks in a single info segment have been prepared for installation with verify_info.

Category: Info Segments Poorly-Handled by verify_info

The phase three testing of verify_info identified two categories of info segments that follow an unusual format and are unlikely to be changed in the future.

- XXX.compin.info segments document WordPro compose application's macros that format input data to appear as elements of a Multics manual page.
- ted_XXX.info segments document WordPro ted editor's requests; and ted's support routines.

Neither validate_info_seg nor verify_info understand the special formats used in info blocks within these segments. Both commands fail to identify the info block kind, and therefore generate many bogus errors against these info blocks.

Because neither of these WordPro applications are frequently used, they are quite unlikely to change in the future. The cost of training verify_info to understand formats used in these info segments is not justified. No one will need to verify such info segments in the future because there is high probability they won't be changed.

Therefore, this category of info segments will remain as improperly handled by verify_info. verify_info will continue to mis-identify info block types and to report bogus errors in info segments of this category.

Guidelines for Multi-Operation Info Segments

A single command may perform several different operations. Examples include: archive, history_comment, io_call and window_call, manage_volume_pool, library_descriptor and update_seg. In such commands, all operations share a syntax that includes an **OPERATION** argument; but each operation has its own syntax for arguments and control arguments following that **OPERATION** argument. So each operation needs to be documented in a separate info block; and an initial info block is needed to briefly list the possible operations, and tell the user how to display documentation for an individual operation.

Such *multi-operation commands* form a sub-class of commands. This sub-class has its own info segment structure and format rules (known as *Operations Format* guidelines).

mbuild_type (mbt) is a small command with two operations. It will be used to demonstrate the various elements of Operations Format.

Operations Format: Overview

A multi-operation info segment begins with a Command block (known as a *multi-op block)* describing the overall function of the command and its shared syntax in a "Syntax as a command" section. (If some operations performed by the command can operate as an active function, this block may also include a "Syntax as an active function" section.) An "Arguments" section describes each of the positional arguments in the shared syntax. One of these is the OPERATION argument.

The multi-op block continues with a "List of operations" section that gives either: names of each possible operation; or names with brief description of each possible operation. This section opens with a few lines telling the user how to display documentation for an individual OPERATION in the list.

The multi-op block may also include a "Control arguments" section. This alerts the reader that some operations may accept control arguments. It may include a "Notes" section describing any details shared by all (or most) operations.

An individual operation is documented in its own Command info block (known as an *opDoc block*). The Operations Format guidelines describe:

- Format and order of names to place in the block divider line.
- Format of the info block heading line.
- Format of the info block "Syntax as …" lines.
- Content of the "Arguments" section.

verify_info checks that:

- All OPERATIONs listed in the multi-op block are described by an opDoc block.
- The same names are given for each operation in the "List of operations" section, and in the Arguments section of that operation's opDoc block.
- The same operation names (each preceded by the short command name) are given as divider names in each opDoc block.
- The opDoc block heading line contains: the short command name; the long operation name; and the word "operation".

Operations Format: Multi-Op Info Block

The mbuild_type (mbt) command block demonstrates these components: two brief **"Syntax as …"** sections that summarize the syntax shared by all operations; an **"Arguments"** section describing the placeholder **OPERATION** argument; a description of the **OPERATION** argument which references a **"List of operations"** section giving name(s) and description for each operation. These components are highlighted in **bold text** below.

```
:Info: mbuild_type: mbt:  2019-08-09  mbuild_type, mbt

Syntax as a command:
   mbt OPERATION {NAME} {-control_args}


Syntax as an active function:
  [mbt OPERATION NAME {-control_args}]


Function: displays information about: segment types installed in the
Multics Libraries; and build paradigms, the procedures and policies for
building and installing a given type of segment.


Arguments:
OPERATION
   selects the operation to be performed.  See "List of operations"
   below.
NAME
   selects information to be displayed.


List of operations:
   For details about each operation, type: help mbt.OPERATION
paradigm, pdm
   displays information about a build paradigm; or summarizes known
   paradigms used in the Multics Libraries.
seg_type, seg
   summarizes known segment types found in the Multics Libraries; or
   displays information about building and installing a given segment.


Control arguments:
-control_args
   control function performed by each OPERATION.  For details, type:
     help mbt.OPERATION
```

Operations Format: OpDoc Info Block

The info segment continues with a Command block for each OPERATION named in the **"List of operations"** section. Each such block is known as an *opDoc block*. This block describes the full syntax, function, arguments, control arguments, and notes for a particular command operation. The opDoc block for the mbt paradigm operation is shown below.

The block begins with a block divider line giving two names:

:Info: CMD_SH_NAME.OP_LG_NAME: CMD_SH_NAME.OP_SH_NAME:

The short name of the command is connected by a period with each of the names for the operation, in longest-to-shortest order. Each name ends with a colon (:) character.

A 3-word heading line includes: the short command name; and the long operation name being described in this block; with "operation" as the third word:

DATE_MOD CMD_SH_NAME OP_LG_NAME operation

Both of these divider line parts are shown for the mbt paradigm operation:

```
:Info: mbt.paradigm: mbt.pdm:  2019-08-09  mbt paradigm operation
```

The **"Syntax as a command"** and **"Syntax as an active function"** sections show full syntax for the paradigm operation. The OP_SH_NAME (pdm) is always used in these syntax expressions.

```
Syntax as a command:  mbt pdm {PARADIGM_NAME} {-control_args}


Syntax as an active function:
  [mbt pdm PARADIGM_NAME {-control_args}]


Function: displays information about build paradigms, the procedures
and policies for building a given type of segment, and installing it
into the Multics Libraries.


If invoked without args, displays a 3-line description of the known
build paradigms including: name, purpose and example segments using
that paradigm.

If invoked with a paradigm_name and no control_args, displays a longer
description of that paradigm.

As an active function, specific information about a given paradigm is
returned.
```

The "**Arguments**" section is always present, and always gives **OP_LG_NAME**, **OP_SH_NAME** for the operation being described. This reference in the only place in the opDoc block where both names are visible to the user. (help never shows divider names to the user.) Any other positional arguments are described here.

```
Arguments:
paradigm, pdm
   the operation being performed.
PARADIGM_NAME
   names a paradigm to be displayed.  This is optional when invoked as
   a command, but required when used as an active function.
```

The opDoc block ends with a description of the control arguments accepted by this operation. This section is required if one of the syntax lines includes: -control_arg or -control_args or {-control_arg} or {-control_args}.

```
Control arguments:
-all, -a
   displays all information for the paradigm.
-name, -nm
   displays the paradigm name.
-purpose
   displays the purpose of the paradigm.
-example
   displays an example segment that is built or installed using the
   paradigm.
-steps
   displays steps performed as part of this build paradigm.
```

The info segment continues with an opDoc block for other operations identified in the multi-op block "**List of operations**" section. In our example, it is the seg_type, seg operation.

```
:Info: mbt.seg_type: mbt.seg:  2019-06-10  mbt seg_type operation

Syntax as a command:  mbt seg {SEG_NAME} {-control_args}
                      mbt seg -for_paradigm paradigm_name


Syntax as an active function:  [mbt seg SEG_NAME {-control_args}]


Function:  displays information about the different types of segments
installed in the Multics Libraries.

If invoked without args, summarizes known segment types installed in
the Multics Libraries.  This includes a starname selecting a group of
segments, and characteristics of segments in the group.

If invoked with a seg_name and no control_args, displays full
information about building and/or installing that segment.

As an active function, returns selected information about build a given
segment.
```

```
Arguments:
seg_type, seg
   the operation being performed.
SEG_NAME
   names a segment whose build/install information is displayed.  This
   argument is optional when invoked as a command, but required when
   used as an active function.

Control arguments:
-for_paradigm paradigm_name, -fpdm paradigm_name
   selects all segment types built using the given paradigm.  This
   control argument may not be used as an active function.
-all, -a
   displays all information for the build type.
...
```

Guidelines for I/O Module Info Segments

Multics I/O modules are an important category of subroutines. Each I/O module follows a subset of the protocols defined by the iox_ I/O system. iox_ provides a flexible framework that can handle I/O to many media types supporting many data formats, access modes, and control orders. The developer of a new I/O module must choose which subset of the possible protocols to support. The info segment describing an I/O module must therefore describe the choices made by the developer, so the reader will know which iox_ features and operations are available, and how to access them through the iox_ subroutine.

The MPM Reference Guide (AG91-04) Section 5 discusses the possible protocols available in an I/O module. A major choice is the decision to support: stream-oriented (character-at-a-time) data; or record-oriented (binary and/or character) data divided into chunks (records) of known format; or both orientations. The I/O system provides different operations for handling these two data orientations. In Table 5.1 from the MPM Reference Guide, the first three opening modes on the vertical axis are the stream-oriented operations; the remaining operations are record-oriented. The horizontal axis shows which iox_$XXX operations are allowed for each opening mode.

Table 5-1. Opening Modes and Allowed Input/Output Operations

Opening Mode No.	Name	get_line	get_chars	put_chars	read_record	rewrite_record	delete_record	read_length	position	seek_key	read_key	close or close_file	write_record	control	modes
1	stream_input	X	X						2			X		1	1
2	stream_output			X								X		1	1
3	stream_input_output	X	X	X					2			X		1	1
4	sequential_input				X			X	X			X		1	1
5	sequential_output											X	X	1	1
6	sequential_input_output				X			X	X			X	X	1	1
7	sequential_update				X	X	3	X	X			X	4	1	1
8	keyed_sequential_input				X			X	X	X	X	X		1	1
9	keyed_sequential_output									X		X	X	1	1
10	keyed_sequential_update				X	X	X	X	X	X	X	X	X	1	1
11	direct_input				X			X		X		X		1	1
12	direct_output									X		X	X	1	1
13	direct_update				X	X	X	X		X		X	X	1	1

1. Depends on the attachment.
2. Allowed if attached to a file in the storage system.
3. Allowed unless file is blocked.
4. Allowed for blocked and sequential files in the storage system.

An I/O module info segment needs to summarize the particular opening modes and chosen I/O operations supported by that module.

In addition there are four setup operations supported by every I/O module which permit I/O module-specific descriptions (arguments and control arguments) to select a device and/or media, select a particular file to be opened on that medium, specify disposition of that file when it is closed, and specify device or media disposition when the I/O module is detached from an I/O switch. These are **iox_$attach_ptr**, iox_$open or **iox_$open_file**; iox_$close or **iox_$close_file**; and iox_$detach_iocb or **iox_$detach**. The operations in **bold** font accept a description string; those in normal font do not.

Every I/O module must support iox_$attach_ptr; so the I/O module's info segment must describe syntax and meanings for elements of an attach description. For those I/O modules that support iox_$open_file, iox_$close_file and/or iox_$detach, syntax and meanings of elements in the description argument for those operations must be described.

An I/O module may optionally support iox_$position calls to move the input or output position to a different location within the input file.

The I/O module must describe which positioning types are accepted by the I/O module for each opening mode that is supported by the iox_$position subroutine.

An I/O module may optionally accept I/O mode values which tailor and adjust I/O operations supported by the module.

The I/O module info segment must list mode values accepted by the module, and describe the meaning of each value (its impact on I/O operations, etc.).

Also, all I/O modules may optionally support iox_$control operations to perform special actions (called *control orders*) relating to the I/O operations. Some control orders accept additional data (binary or character values, data structures, etc.).

The I/O module info segment must summarize which control orders are supported, and may optionally provide details about data associated with a particular control order.

In addition, the I/O module may choose to support command-level invocation of control orders that accept or return data via the io_call command. The I/O module provides command-level support by implementing the "io_call" control order, which: permits the user to submit arbitrary argument and control argument inputs to a control order; and permits the I/O module to display or interpret control order data that would normally be returned in a data structure, etc. The MPM Subsystem Writer's Guide (AK92) section titled "Performing Control Operations From Command Level" provides details for accepting the "Io_call" control order to support other control orders provided by the module via the io_call command/active function interface.

The I/O module info segment descriptions for each control order should also describe the io_call command or active function interface syntax for that control order.

This chapter gives guidelines for describing an I/O module in an info segment.

I/O Module Format: Overview

The format of an info segment describing an I/O module is an extension of the format used for multi-operation commands. An I/O Module block kind begins the info segment. This block summarizes most of the features (the developer's design choices) implemented in the I/O module. The block describes:

- syntax of the attach description;
- functions of the module;
- arguments and control arguments in the attach description string;
- supported opening modes;
- i/o operations that are implemented;
- control orders that are provided; mode strings allowed; etc.

If the position operation is supported, its description covers which of the opening modes support iox_$position operations, and which positioning types are supported.

If the I/O module supports any of the setup operations requiring a description argument (iox_$open_file, iox_$close_file, or iox_$detach), they are summarized in a "List of operations" section. Full details for each operation are then given in a separate I/O Operation info block. The first sentence of the "List of operations" section tells the reader how to use help to display these I/O Operation blocks.

Other operations are summarized in a "List of i/o operations" section. Interface for each operation is defined by the iox_$OPERATION entry point; and these operations accept only arguments well-documented in the iox_$OPERATION description. So each list item only gives the operation name, with a 1-line description of the function(s) performed by that operation.

Control orders that accept a non-null info_ptr argument are summarized in a "List of control orders" section of the I/O Module block. Each item gives name(s) for the control order, and a brief description of its function. Full details for each operation are then given in a separate I/O Control info block. The first sentence of the "List of control orders" section tells the reader how to use help to display these I/O Control blocks.

If the I/O module supports any control orders that accept a null info_ptr argument, they are summarized in a "List of controls" section. The iox_$control subroutine description describes the complete syntax for such operations. So each item in the "List of controls" gives only the control name(s) followed by a brief description of the control function.

Each I/O Operation block describes:

- syntax of the io_call command for that operation;
- syntax of the description string used in both command and iox_ subroutine forms of the setup operation;
- function performed by the setup operation;
- arguments and control arguments in the description string.

Each I/O Control block includes the following sections.

- "Control order" section that:
  - begins with long name of the control order, and any short name;
  - includes one or more paragraphs briefly describing function of the control.
- "Syntax" section for an iox_$control subroutine call invoking the control order.
- "Arguments for iox_$control" section describing data pointed to by the **info_ptr** argument of iox_$control subroutine.
- "List of elements: section describing elements of any structure pointed to by the info_ptr argument.
- "Notes on the info_ptr" section giving additional details about data pointed to by the info_ptr argument (e.g., name of any include file that declares the structure pointed to by info_ptr; details about providing input data for any structure; explanation of output elements of any structure pointed to by info_ptr; etc.).
- "Syntax as a command" section describing how to use the io_call command to invoke the control order; and/or a "Syntax as an active function" section describing how to use the io_call active function to return data provided by the control order.
- "Arguments for io_call" section describing items shown in these syntax sections. These include: the SWITCHNAME argument; long (and any short) name for the control order; and positional arguments used to supply data for input elements of any structure pointed to by the info_ptr.
- Optional "Control arguments" or "Control arguments as a command" or "Control arguments as an active function" describing any control arguments used in the io_call syntax.
- "Notes on the io_call command" describing how it interprets any output data returned by the control order.

In an I/O Module block, use of a "List of operations" section extends the concepts of *Operations Format* (described for Command info blocks above) to the I/O Module info block. That block is tagged as a multi-op block. Each of the corresponding I/O Operation blocks is tagged as an opDoc block.

In an I/O Module block, use of a "List of control operations" section also extends the concepts of Operations Format to the I/O Module info block. The I/O Module block is tagged as a multi-control block. Each of the corresponding I/O Control blocks is tagged as an controlDoc block.

When Operations Format concepts are used, verify_info checks that:

- All setup OPERATIONs given in a "List of operations" section of the multi-op block are described by an opDoc block.
- The same names are given for each operation in the "List of operations" section, and in the Arguments section of that operation's opDoc block.
- The same operation names (each preceded by the I/O module name) are given as divider names in each opDoc block.
- The opDoc block heading line contains: the I/O module name; the long operation name; and the word "operation".

- All control ORDERNAMEs given in a “List of control operations” section of the multi-control block are described by a controlDoc block.
- The same names are given for each control in the “List of control operations” section, and in the “Control order” section that begins that controlDoc block.
- The same control names (each preceded by the I/O module name) are given as divider names in each controlDoc block.
- The controlDoc block heading line contains: the I/O module name; the long name of the control order; and the word “control”.

The guidelines recommend use of the following sections in the I/O module info block. These are displayed as part of the output from the command: **vi -rules io_module**

```
Section Titles for  I/O Module:

   ORDER    CARDINALITY           TITLE
     1     1 allowed             Syntax as a command:
     2     1 required            Syntax of attach description:
     3     1 required            Function:
     4     1 allowed             Arguments:
           0 or more             Arguments for ...:
           0 or more             Arguments (...):
     5     1 allowed             Control arguments for attach description:
           0 or more             Control arguments:
           0 or more             Control arguments as a command:
           0 or more             Control arguments for ...:
           0 or more             Control arguments (...):
     6     1 required            List of opening modes:
     7     1 allowed             List of operations:
     8     1 required            List of i/o operations:
     9     1 of ORDER allowed    List of controls:
           1 of ORDER allowed    List of control operations:
    10     1 allowed             List of mode strings:
           0 or more             List of ...:
    11     1 allowed             Access required:
    12     1 allowed             Notes:
           0 or more             Notes on ...:
    13     1 allowed             Examples:

           ________________________________________________
```

The following section titles are recommended for an I/O Operation info block.

```
Section Titles for  I/O Operation:

  ORDER    CARDINALITY           TITLE
    1     1 allowed             Syntax as a command:
    2     1 of ORDER required   Syntax of open description:
          1 of ORDER required   Syntax of close description:
          1 of ORDER required   Syntax of detach description:
    3     1 required            Function:
    4     1 allowed             Arguments for io_call:
          0 or more             Arguments:
          0 or more             Arguments for ...:
          0 or more             Arguments (...):
    5     1 required            List of opening modes:
    6     1 of ORDER allowed    Control arguments for open description:
          1 of ORDER allowed    Control arguments for close description:
          1 of ORDER allowed    Control arguments for detach description:
          0 or more             Control arguments as a command:
          0 or more             Control arguments as an active function:
          0 or more             Control arguments:
          0 or more             Control arguments for ...:
          0 or more             Control arguments (...):
    7     1 allowed             Access required:
          0 or more             List of ...:
          1 allowed             Notes:
          0 or more             Notes on ...:
          1 allowed             Examples:

          ------------------------------------------------
```

The following section titles are recommended for an I/O Control info block.

```
Section Titles for  I/O Control:

  ORDER    CARDINALITY           TITLE
    1     1 required            Control order:
    2     1 required            Syntax:
    3     1 required            Arguments for iox_$control:
    4     1 allowed             List of elements:
          1 allowed             Notes on the info_ptr:
    5     1 allowed             Syntax as a command:
          1 allowed             Syntax as an active function:
    6     1 allowed             Arguments for io_call:
    7     1 allowed             Control arguments:
          1 allowed             Control arguments as a command:
          1 allowed             Control arguments as an active function:
          0 or more             Control arguments for ...:
          0 or more             Control arguments (...):
    8     1 allowed             Access required:
          0 or more             List of ...:
          1 allowed             Notes:
          0 or more             Notes on ...:
          1 allowed             Examples:

          ------------------------------------------------
```

Use of the I/O Module Format provides a nice summary of I/O module essentials displayed by **help -brief**.

```
help mtape_ -brief

>user_dir_dir>Multics>GDixon>w>vi>f>mtape_.info  (257 lines in info)

2020-12-26  mtape_ I/O module

Syntax as a command:
   io_call attach SWITCHNAME mtape_ VOLUME_SPECs {-control_args}

Syntax of attach description:
   mtape_ VOLUME_SPECs {-control_args}

Arguments:
 attach  SWITCHNAME   mtape_

Arguments for volume specification:
 VOL_NAME

Control arguments for volume specification:
 -volume VOL_NAME     -comment VNi_STR, -com VNi_STR

Control arguments (attach description):
 -default_volume_type STR,     -ring, -rg
   -dvt STR                    -no_ring, -nrg
 -density N, -den              -speed N1{, N2, ..., Nn}
 -device N, -dv N              -system, -sys
 -display, -ds                 -no_system, -nsys
 -no_display, -nds             -track N, -tk N
 -error, -err                  -volume_type STR, -vt STR
 -no_error, -nerr              -wait, -wt
 -label, -lbl                  -no_wait, -nwt
 -no_label, -no_labels, -nlbl  -wait_time N, -wtm N

List of opening modes:
 sequential_input sequential_output

List of operations:
 open_file, open   close_file, close   detach, detach_iocb

List of i/o operations:
 read_record   write_record    position  control

List of controls:
 force_end_of_volume,   ring_in, rin
   feov

List of control operations:
 file_status, fst         hardware_status, hwst   volume_set_status, vsst
 file_set_status, fsst    volume_status, vst      io_call
```

I/O Module Format: the I/O Module Block

The *I/O Module* block must include: syntax and argument descriptions for the I/O module's ATTACH_DESCRIPTION which is an argument to the iox_$attach_loud, iox_$attach_name, or iox_$attach_ptr subroutine; and a different syntax used when attaching the I/O module via the **io_call** command.

Some I/O modules also accept a DESCRIPTION argument when calling the iox_$open_file, iox_$close_file, and iox_$detach operations. Each must be described separately in an *I/O Operation* block because each DESCRIPTION argument has its own syntax when given in the iox_ subroutine call; and when used in the equivalent io_call command.

- An I/O module block that supports any of these special operations is also known as a *multi-op block*. The I/O module block has a **"List of operations"** section naming which of the three I/O OPERATIONs (open_file, close_file, or detach) is supported.
- Each such OPERATION has its own *I/O Operation* block to describe the DESCRIPTION control arguments, with any notes for that operation; this block is known as an *opDoc block* when it is linked to one of the operations listed in the multi-op block.

Besides supporting standard I/O operations defined by iox_ (such as read_record, write_record, position), an I/O module may provide a special operation known as a CONTROL_ORDER using the iox_$control subroutine. Some control orders may accept input data or return output values using either a scalar variable or a data structure pointed to by the iox_$control info_ptr argument.

- An I/O module that provides a CONTROL_ORDER that does not use an input/output scalar or structure must include in its I/O Module block a **"List of controls"** section giving name(s) and a full description of that CONTROL_ORDER.
- An I/O module that provides a CONTROL_ORDER that uses an input/output scalar or structure must include in its I/O Module block a **"List of control operations"** section giving name(s) and brief description of that CONTROL_ORDER. Using this section makes the I/O Module block a *multi-control block*.
- Each CONTROL_ORDER using an input/output value has its own *I/O Control* block in the info segment which: describes the data structure in detail; and shows how that CONTROL_ORDER is called using the io_call command. Such block is also known as a *controlDoc block*.

If **"List of operations"** or **"List of control operations"** sections appear in an I/O Module block, then an **Operations Format** structure is being used in the info segment for that I/O module.

However, a simpler I/O Module block without Operations Format can describe any I/O module that:

- does not support the open_file, close_file and detach operations; and
- does not implement control orders requiring a scalar variable or data structure.

Such info segment would include only the I/O Module block. Other blocks associated with Operations Format would not be needed or included in the info segment.

The following sample shows an I/O Module block for the mtape_ I/O module. Since this I/O module supports the setup operations that accept a DESCRIPTION argument, it includes a **"List of operations"** section summarizing those operations. This makes the I/O Module block a multi-op block.

Since mtape_ provides CONTROL_ORDERs that use an input/output data structure, the I/O Module block includes a **"List of control operations"** section summarizing each such CONTROL ORDER, making that block a multi-control block.

Look in the sample below for the following sections which are often found in an I/O Module info block.

- The **"Syntax as a command"** section shows how mtape_ is attached using an io_call command.
- The **"Syntax of attach description"** section summarizes elements of the I/O module's attach description. This may include both positional arguments and control arguments.
- An "**Arguments**" section is always included in a I/O Module block. It describes the **attach** and SWITCH_NAME positional arguments used in the io_call command. The mtape_ I/O module name is a positional argument that appears at the beginning of the ATTACH_DESCRIPTION which is in both of **"Syntax** …" sections described above.
- The **"Arguments** …" and **"Control arguments** …" sections continue describing the ATTACH_DESCRIPTION positional and control arguments.
- The **"List of opening modes"** section shows which of the standard iox_ opening modes are supported by the I/O module.
- The **"List of operations"** section indicates that the mtape_ I/O module supports each of the setup iox_$open_file, iox_$close_file and iox_$detach operations that accept a DESCRIPTION string. It also supports the regular iox_$open, iox_$close and iox_$detach_iocb operations that perform without using a DESCRIPTION string by applying default values for each DESCRIPTION control argument. Both operation with and without the DESCRIPTION string are described in each I/O Operation block.
- The **"List of i/o operations"** section indicates which of the standard iox_ operations are supported by the mtape_ I/O module. Operations that work only with certain opening modes note that requirement.
- The **"List of controls"** section gives name(s) and full description for each mtape_ CONTROL_ORDER that uses no input or output data (the iox_$control info_ptr argument is set to a null value). The section begins with a few lines describing how such CONTROL_ORDER is invoked using the io_call command.
- The **"List of control operations"** section gives name(s) and brief description of each mtape_ CONTROL_ORDER that uses an input or output data structure. The section begins by telling how to display a full description of each CONTROL_ORDER. Each full description is documented in its own I/O Control block.

:Info: **mtape_:** 2020-12-26 **mtape_ I/O module**

Syntax as a command:
io_call **attach SWITCHNAME mtape_** VOLUME_SPECs {-control_args}

Syntax of attach description:
mtape_ VOLUME_SPECs {-control_args}

Function: The mtape_ I/O module supports I/O to/from tapes in
- ANSI standard format
- IBM standard format
- IBM unlabeled format
- IBM Disk Operating System (DOS) formatted

For details not described in this info segment, see the Multics Subroutines and I/O Modules manual (order no. AG93) description of the mtape_ I/O module.

Arguments:
attach
is the iox_ operation to be performed by the io_call command.
SWITCHNAME
is the name of an I/O switch to be attached to the mtape_ I/O module.
mtape_
is the I/O module name.

Arguments for volume specification:
Each VOLUME_SPEC argument requires a VOL_NAME, which may be followed by a -comment control argument.
VOL_NAME
label of a volume in the volume set to be attached.

Control arguments for volume specification:
-volume VOL_NAME
used when VOL_NAME begins with hyphen (-) character, VOL_NAME is a label of a volume in the volume set to be attached.
-comment VNi_STR, -com VNi_STR
an optional message to be displayed on the operator console at the time this VOL_NAME is to be mounted. The comment text VNi_STR may be from 1 to 64 characters in length and must be quoted if it contains embedded white space. The optional comment specification must follow its corresponding volume specification and precede the next volume specification.

Control arguments for attach description:
-default_volume_type STR, -dvt STR
   specifies the volume type (STR) to be used for Per-Format module selection (see "Notes on tape format selection" below) when an unreadable or unlabeled tape is mounted for potential output operations and no "-volume_type" control argument is given. Permissible values for this control argument are "ansi", or "ibm". (Default value is "ansi".)

-density N, -den N
   specifies the recording density for output operations in bits per inch (BPI). For input operations, the density is determined and set automatically by RCP. Permissible values are 200, 556, 800, 1600 and 6250. (Default density is 1600 BPI.)
...

List of opening modes:
   The following I/O opening modes are supported by mtape_.
sequential_input
   read data blocks from tape media.
sequential_output
   write data blocks to tape media.

List of operations:
   mtape_ implements the following I/O setup operations. For details on any operation, type: help mtape_.OPERATION

open_file, open
   opens an mtape_ I/O switch for reading or writing a file from the attached volume set.
close_file, close
   closes an mtape_ I/O switch to end operations on the current file.
detach, detach_iocb
   detaches an mtape_ I/O switch to end operations on the volume set.

List of i/o operations:
   The following standard iox_ operations are supported by mtape_.
   For details about any operation, type: help iox_$OPERATION
read_record
   read records from an mtape_ file, when opened for sequential_input.
write_record
   write records to an mtape_ file, when opened for sequential_output.
position
   accepts all types of positioning when the I/O switch is open for sequential_input except type 3 (which is for stream_input only).
control
   executes a special mtape_ operation. See "List of controls" and "List of control operations" below.

List of controls:
Controls with the following ORDERNAMEs require no additional data. The iox_$control info_ptr argument should be a null pointer. To use these controls from a command line, type:
io_call control SWITCHNAME ORDERNAME

force_end_of_volume, feov
simulates detection of the end of tape reflective foil upon the next write block operation. The PFM will then close out the volume by writing the EOV trailer labels and request a volume switch for the next volume in the volume sequence list. The I/O switch must be open for sequential_output.

ring_in, rin
requires that the I/O switch is closed and will cause all currently mounted volumes of the volume set to be demounted. When the next file opening is performed, the required volume will be re-mounted with the write ring installed.

List of control operations:
Controls with the following ORDERNAMEs accept info_ptr values. To learn more about each control order, type:
help mtape_.ORDERNAME
file_status, fst
returns a pointer to a structure that contains the status of the current file specified by the open description.
file_set_status, fsst
returns a pointer to an array of structures defining the file status for all files in the current file set.
hardware_status, hwst
returns a pointer to a structure that contains hardware status for the last I/O operation.

volume_status, vst
returns a pointer to a structure that contains the status of the current volume.
volume_set_status, vsst
returns a pointer to an array of structures defining the volume status for all volumes in the current volume set.
io_call
executes one of the other mtape_ control operations on behalf of the io_call command.

I/O Module Format: the I/O Operation Block

The following sample shows an *I/O Operation* block used to describe each operation listed in an I/O Module block's **"List of operations"** section. Appearance of that section in the I/O Module block signifies use of **Operations Format** concepts; I/O Operation blocks are then required in the info segment.

The sample below describes the mtape_ implementation of the iox_$close_file operation. Look in this sample for the following sections which usually appear in an I/O Operation block.

- The **"Syntax as a command"** section shows how an mtape_ I/O switch is closed using an io_call command. Because mtape_ supports both close_file and close operations, this section shows the syntax for each of these io_call operations.
- The **"Syntax of close description"** section summarizes elements of the I/O module's close DESCRIPTION string (used only by the iox_$close_file operation). This string consists only of control arguments and their operands.
- The **"Function"** section describes the operation being performed, referencing the particular iox_ subroutines that perform this operation, and telling how to display details about those subroutines.
- The **"Arguments for io_call"** section describes the positional arguments shown in the **"Syntax as a command"** section.
- The **"Control arguments for close description"** section describes the control arguments that may be included in the close DESCRIPTION string, and their operands.

```
:Info: mtape_.close_file: mtape_.close:
2020-06-17  mtape_ close_file operation

Syntax as a command:
   io_call close_file SWITCHNAME {-control_args}
   io_call close      SWITCHNAME


Syntax of close description:
   {-control_args}


Function:
Closing a file is accomplished by calling the iox_$close_file
entry which accepts as one of its arguments a character string
"close description".  The close description supplies close option
information to the selected Per-Format Module (PFM).

The iox_$close entry is also supported.  It passes an empty string
description to the close_file entry, thereby applying default close
description values.  For information about iox_ entrypoints, type:
   help iox_$close_file    or   help iox_$close
```

Arguments for io_call:
close_file, close
is the I/O operation to be performed. Close description control arguments may be provided when the close_file operation is used. No control arguments are permitted when the close operation is used.
SWITCHNAME
is the name of an I/O switch attached using the mtape_ I/O module.

Control arguments for close description:
-close_position STR, -cls_pos STR
specifies where to physically position the tape volume within the bounds of the file that is being closed. The values of STR are case insensitive and may be selected from "bof" (for beginning of file), "eof" (for end of file) and "leave" to leave the tape positioned where it is. (Default close position is "leave".)
-comment STR, -com STR
specifies a user comment to be displayed on the user_output I/O switch, after the file has been successfully closed. The comment text (STR) may be from 1 to 80 characters in length.

-display, -ds
specifies that the entire close description, after it has been parsed and any necessary defaults added, is to be displayed on the user_output I/O switch.
-no_display, -nds
specifies that the close description will not be displayed on the user_output I/O switch. (Default)

I/O Module Format: the I/O Control Block

The following sample shows an *I/O Control* block used to describe each CONTROL_ORDER listed in an I/O Module block's **"List of control operations"** section. Appearance of that section in the I/O Module block signifies use of **Operations Format** concepts; I/O Control blocks are then required in the info segment.

A control order provided by the I/O module may use an information structure as an input and/or output argument; and may support filling or displaying that structure when the control order is invoked using the io_call command. Such control orders must be described separately to explain elements of the data structure; and to describe io_call support for filling or displaying elements of the structure.

- An I/O module block that supports controls accepting an information structure is known as a *multi-control* block. It has a **"List of control operations"** section with name(s) and brief description for each such CONTROL_ORDER.
- Each CONTROL_ORDER is fully described in an *I/O Control* block with a section showing the data pointed to by the iox_$control **info_ptr** argument for that order. The I/O Control block is known as *controlDoc block* when it is linked to one of the CONTROL_ORDERs listed in the multi-control block.

The sample below describes the mtape_ hardware_status control order. Look in this sample for the following sections which usually appear in an I/O Control block.

- The **"Control order"** section gives names for the control order, summarizing its purpose and overall content of the data pointed to by the info_ptr argument.
- The **"Syntax"** section shows syntax of the iox_$control operation, reminding the reader about the info_ptr argument, which points to the data structure being explained by this info block.
- The **"Arguments to iox_$control"** section describes each of its arguments in more detail.
- The **"List of elements"** section describes each element of the structure pointed to by the info_ptr argument.
- The **"Notes on the info_ptr"** section describes the include file that declares the structure pointed to by the info_ptr argument; and describes how the I/O module responds if the info_ptr argument is a null pointer.
- If the I/O module supports the **io_call** control order, then a **"Syntax as a command"** section describes how to perform this control order via an io_call command.
- Any arguments or control arguments accepted by io_call for this control order are described in an **"Arguments for io_call"** section and an optional **"Control arguments for io_call"** section.
- A **"Notes on the io_call command"** section describes any special services performed when this control order is invoked by io_call. This might include supplying elements of an input data structure, or interpreting elements displayed from an output data structure.

:Info: **mtape_.hardware_status: mtape_.hwst:**
1985-02-18 **mtape_ hardware_status control**

Control order: hardware_status, hwst

The iox_$control info_ptr argument points to a structure that reports the hardware status stored by last tape I/O operation.

Syntax:

```
call iox_$control (iocb_ptr, "hardware_status", info_ptr, code);
```

Arguments for iox_$control:

iocb_ptr
points to the switch's control block. (Input)

hardware_status, hwst
is the name of the control order. (Input)

info_ptr
points to the following structure, or is a null pointer. See "Notes on the info_ptr" below. (Input)

```
dcl 1 mtape_hardware_status aligned based (info_ptr),
      2 version char(8),
      2 description char (256) varying,
      2 pad bit (36),
      2 iom_status bit (72),
      2 iom_lpw bit (72);
```

code
is an I/O system status code. (Output)

List of elements:

version
is the current structure version. If info_ptr is nonnull on input, then the caller must set the version element to hwst_version_1. If null, mtape_will set the version number in the structure it allocates.

description
is the English language description of this hardware status.

iom_status
is the raw I/O status words returned from the last I/O operation. A definition for the format of these status words can be found in the include file iom_status.incl.pll.

iom_lpw
is the I/O List Pointer Word as it appeared at the termination of the last I/O operation. A definition of the format of the LPW can be found in the include file iom_lpw.incl.pll.

Notes on the info_ptr:
The mtape_hardware_status structure declaration shown above can be found in the include file "mtape_hardware_status.incl.pl1". If the iox_$control info_ptr is null, then mtape_ will allocate the structure for the caller; the caller must free that structure.

Syntax as a command:
io_call control SWITCHNAME hwst

Arguments for io_call:
control
is the io_call operation being performed.
SWITCHNAME
is the name of an I/O switch attached to the mtape_ I/O module.
hardware_status, hwst
is the name of the control order to be performed.

Notes on the io_call command:
When the io_call command performs an mtape_ control order, any data normally returned by that control order is interpreted by mtape_ and displayed on the user's terminal.

Installation Plan

The software items described in MTB-1004 are part of a larger product change to replace the existing validate_info_seg (vis) command and help_ subroutine with:

- verify_info (vi): a new command for checking info segments
- info_seg_verify_: a new subroutine to supervise work of parsing and checking content of info segments
- info_seg_: a new set of subroutines for parsing info segments into components
- help_ Subsystem: a replacement for the existing help_ and help_rql_ subroutines

Components of this change are described below.

MTBs

MTB-1004: verify_info Command and info_seg_ (this MTB)

MTB-1005: help Command and New help_ Subsystem

MTB-????: Historical Review of help Facility on Multics

MTB-????: Changes to mbuild Subsystem for Verifying Info Segments

MCRs

MCR10080, Repair validate_info_seg handling of subsystem request infos

MCR10082, Add get_page_length_ subroutine

MCR10083, translator_temp_ $empty_all_segments

MCR100??, Copy existing help and validate_info_seg Commands to >obs>bound_old_help_

MCR100??, Replacements for help, help_, validate_info_seg in bound_info_rtns_

MCR100??, Fixes for Info Segments Required to Install verify_info

MCR100??, Add a verify_info Request to mbuild Subsystem and Other mbuild Bug Repairs

MCR10084, Revise probe.info; Enhance probe's help Request

Bugs (Multics Tickets) Not Resolved by these MTBs/MCRs

Ticket 208: validate_info_seg does not handle XXX.compin.info segments correctly

Appendix A: Supported Info Segment Structures

The following info segment block structures are supported by the help command.

Info Segment – no block dividers

An info segment containing only one block that begins without a block divider.

Info Segment – with block dividers

An info segment containing one or more blocks: the first begins with an :Info: divider; subsequent blocks must begin with either an :Info: or :[Info]: divider.

Info Segment – with History Comment

An info segment containing two or more blocks: the first begins with an :Info: divider; subsequent optional blocks must begin with either an :Info: or :[Info]: divider; a final block with an :hcom: divider contains a history comment.

Subroutine Info Segment

An info segment containing two or more blocks: the first block (with no block divider) introduces a subroutine; one or more subsequent blocks each with an :Entry: divider describe one entrypoint of that subroutine.

Subroutine Info Segment – introduction in :Info: block

An info segment containing two or more blocks: the first block with an :Info: divider introduces a subroutine; one or more subsequent blocks each with an :Entry: divider describe one entrypoint of that subroutine.

Subroutine Info Segment – with History Comment

An info segment containing three or more blocks: the first block with an :Info: divider introduces a subroutine; one or more subsequent blocks each with an :Entry: divider describe one entrypoint of that subroutine; a final block with an :hcom: divider contains a history comment.

Appendix B: verify_info -rules

The verify_info -rules control argument displays info segment guidelines used to verify info segments. These are show below.

```
vi -rules file
               Info Block Divider Lines

:Info: NAME1: ... {NAMEn:}  BLOCK_HEADING
          Normal block divider.  From 1 to 10 names may be given.
          BLOCK_HEADER may appear on line following divider token.
          NAMEs are expected to appear as names on the info segment
          with an ".info" suffix (e.g., NAME1.info).
:[Info]: NAME1: ... {NAMEn:}  BLOCK_HEADING
          Hidden block divider.  Like :Info: divider, except NAMEs
          will not appear as names on the info segment.  Hidden blocks
          are accessed by a subsystem's help request, which calls help_
          with a specific info seg PATHNAME and separate INFO_NAME to
          locate a particular hidden block.
:Entry: EP_NAME1: ... {EP_NAMEn:}  BLOCK_HEADING
          Subroutine entrypoint description block divider.  From 1 to 10
          EP_NAMEs may be given, each up to 32 characters in length.
:hcom:
          Info segment history comment divider.  Must be final divider
          in info seg.  All lines following this divider are ignored by
          verify_info and help_.
```

Supported Info Segment Structures

CASE	CARD	SPECIFICATION 1st	MIDDLE	LAST	ERROR or INFO SEG STRUCTURE
1	0				Severity 5. Info segment is empty.
2	1	N			Info Segment - no block dividers
3	1+	[I]			Severity 5. Multi-block info segment begins with :[Info]: divider
4	2+	.	H.H.	.	Severity 5. Info segment has multiple History Comment blocks.
5	2+	.	H	.	Severity 5. History Comment not last block of info segment.
6	3+	N	E.I.	.	Severity 5. Subroutine info w/ mixed :Entry: and :Info: dividers.
7	2+	N	E...		Subroutine Info
8	2+	N	I...	.	Severity 5. Multi-block info segment lacks 1st :Info: divider.
9	1+	E			Severity 5. Subroutine info is missing introduction block.
10	3+	I	E...	H	Subroutine Info - with History Comment
11	3+	I	E.I.	.	Severity 5. Info segment has mixed :Info: and :Entry: dividers.
12	2+	I	E...		Subroutine Info - introduction in :Info: block
13	2+	I	I...	H	Info Segment - with History Comment
14	1+	I	I...		Info Segment - with block dividers
15	1+				Severity 5. Info segment has an unknown block structure.

LEGEND: CARDinality

J J total blocks of any divider type.

J+ J or more total blocks of any divider type.

1st Block

N First info block has no block divider.

E First info block has :Entry: block divider.

I First info block has :Info: block divider.

[I] First info block has :[Info]: block divider.

. Don't care about first block divider.

MIDDLE Blocks

H.H. Two or more history blocks in info segment.

H History block in middle of info segment.

E... One or more :Entry: blocks in middle of segment.

E.I. Mixture of :Entry: and :Info: or :[Info]: blocks.

I... One or more :Info: or :[Info]: blocks in middle of segment.

LAST Block

H History comment is last block of segment.

. Don't care about last block of segment.

---

vi -rules block

```
        Kinds of Info Blocks

   iBlok.kind    Block Kind

            1    Command
            2    Active Function
            3    Command/Active Function
            4    General Info
            5    Subroutine Introduction
            6    Subroutine Brief Intro
            7    Subroutine Entrypoint
            8    Subsystem Request
            9    Subsystem Active Request
           10    Subsystem Request/Active Request
           11    Subsystem Summary
           12    Subsystem Topic
           13    I/O Module
           14    I/O Operation
           15    I/O Control
           16    History Comment
           17    HEADER ONLY Info
           18    UNKNOWN kind of information



        Rules for Determining Kind of Info Block

 Rule  1: File structure sets initial block kind to:  Subroutine Introduction
          &   1st section:  Function:
          =>                      block kind stays:  Subroutine Introduction

 Rule  2: File structure sets initial block kind to:  Subroutine Introduction
          &   1st section:  Entry points in ...:
          =>                 block kind changes to:  Subroutine Brief Intro

 Rule  3: File structure sets initial block kind to:  UNKNOWN kind of information
          &   :Info: block has name: summary.topic
          =>                 block kind changes to:  Subsystem Summary

 Rule  4: File structure sets initial block kind to:  UNKNOWN kind of information
          &   Block name has suffix...
                                .topic
          =>                 block kind changes to:  Subsystem Topic

 Rule  5: File structure sets initial block kind to:  UNKNOWN kind of information
          &   Block name has suffix...        or File name has suffix...
                                changes                      changes.info
                             differences                  differences.info
                                   diffs                        diffs.info
                                .errata                      .errata.info
                                 .error                       .error.info
                                .errors                      .errors.info
                                    .gi                          .gi.info
                              new_features                 new_features.info
                              old_features                 old_features.info
          =>                 block kind changes to:  General Info
```

```
Rule  6: File structure sets initial block kind to:  UNKNOWN kind of information
          &   1st section:  Function:
              2nd section:  Syntax:
               (in that order)
          =>                block kind changes to:  Subroutine Entrypoint

 Rule  7: File structure sets initial block kind to:  UNKNOWN kind of information
          &   1st section:  Syntax as a command:
              2nd section:  Syntax of attach description:
               (in either order)
          =>                block kind changes to:  I/O Module

 Rule  8: File structure sets initial block kind to:  UNKNOWN kind of information
          &   1st section:  Syntax as a command:
              2nd section:  Syntax of close description:
               (in either order)
          =>                block kind changes to:  I/O Operation

 Rule  9: File structure sets initial block kind to:  UNKNOWN kind of information
          &   1st section:  Syntax as a command:
              2nd section:  Syntax of detach description:
               (in either order)
          =>                block kind changes to:  I/O Operation

 Rule 10: File structure sets initial block kind to:  UNKNOWN kind of information
          &   1st section:  Syntax as a command:
              2nd section:  Syntax of open description:
               (in either order)
          =>                block kind changes to:  I/O Operation

 Rule 11: File structure sets initial block kind to:  UNKNOWN kind of information
          &   1st section:  Syntax of attach description:
          =>                block kind changes to:  I/O Module

 Rule 12: File structure sets initial block kind to:  UNKNOWN kind of information
          &   1st section:  Syntax of close description:
          =>                block kind changes to:  I/O Operation

 Rule 13: File structure sets initial block kind to:  UNKNOWN kind of information
          &   1st section:  Syntax of detach description:
          =>                block kind changes to:  I/O Operation

 Rule 14: File structure sets initial block kind to:  UNKNOWN kind of information
          &   1st section:  Syntax of open description:
          =>                block kind changes to:  I/O Operation

 Rule 15: File structure sets initial block kind to:  UNKNOWN kind of information
          &   1st section:  Control order:
          =>                block kind changes to:  I/O Control

 Rule 16: File structure sets initial block kind to:  UNKNOWN kind of information
          &   Info seg directory contains entryname: subsystem or ss
          &   1st section:  Syntax:
          &   2nd section:  Syntax as an active request:
          =>                block kind changes to:  Subsystem Request/Active Request

 Rule 17: File structure sets initial block kind to:  UNKNOWN kind of information
          &   Info seg directory contains entryname: subsystem or ss
          &   1st section:  Syntax as a command:
          &   2nd section:  Syntax as an active function:
          =>                block kind changes to:  Subsystem Request/Active Request
```

```
Rule 18: File structure sets initial block kind to:  UNKNOWN kind of information
          &   Info seg directory contains entryname: subsystem or ss
          &   1st section:  Syntax:
          =>                block kind changes to:  Subsystem Request

Rule 19: File structure sets initial block kind to:  UNKNOWN kind of information
          &   Info seg directory contains entryname: subsystem or ss
          &   1st section:  Syntax as a command:
          =>                block kind changes to:  Subsystem Request

Rule 20: File structure sets initial block kind to:  UNKNOWN kind of information
          &   Info seg directory contains entryname: subsystem or ss
          &   1st section:  Syntax as an active request:
          =>                block kind changes to:  Subsystem Active Request

Rule 21: File structure sets initial block kind to:  UNKNOWN kind of information
          &   Info seg directory contains entryname: subsystem or ss
          &   1st section:  Syntax as an active function:
          =>                block kind changes to:  Subsystem Active Request

Rule 22: File structure sets initial block kind to:  UNKNOWN kind of information
          &   1st section:  Syntax as a command:
              2nd section:  Syntax as an active function:
               (in either order)
          =>                block kind changes to:  Command/Active Function

Rule 23: File structure sets initial block kind to:  UNKNOWN kind of information
          &   1st section:  Syntax as a command:
          =>                block kind changes to:  Command

Rule 24: File structure sets initial block kind to:  UNKNOWN kind of information
          &   1st section:  Syntax as an active function:
          =>                block kind changes to:  Active Function

Rule 25: File structure sets initial block kind to:  UNKNOWN kind of information
          &   1st section:  Syntax:
              2nd section:  Syntax as an active function:
               (in either order)
          =>                block kind changes to:  Command/Active Function

Rule 26: File structure sets initial block kind to:  UNKNOWN kind of information
          &   1st section:  Syntax:
              2nd section:  Syntax as an active request:
               (in either order)
          =>                block kind changes to:  Subsystem Request/Active Request

Rule 27: File structure sets initial block kind to:  UNKNOWN kind of information
          &   Block header has ISO_DATE before 1985
          &   1st section:  Syntax:
          =>                block kind changes to:  Command

Rule 28: File structure sets initial block kind to:  UNKNOWN kind of information
          &   1st section:  Syntax:
          =>                block kind changes to:  Subsystem Request

Rule 29: File structure sets initial block kind to:  UNKNOWN kind of information
          &   1st section:  Syntax as an active request:
          =>                block kind changes to:  Subsystem Active Request
```

```
Rule 30: File structure sets initial block kind to:  UNKNOWN kind of information
          &   1st section:  Notes on ...:
          =>                block kind changes to:  Subsystem Topic

Rule 31: File structure sets initial block kind to:  HEADER ONLY Info
          =>                     block kind stays:  HEADER ONLY Info

Rule 32:  =>                   block kind remains:  UNKNOWN kind of information

          ________________________________________________
```

vi -rules section

```
 Preferred Section Titles (in alphabetic order, each preceded by section type)

   1    Access required:
   2    Arguments:
   3    Arguments for io_call:
   4    Arguments for iox_$control:
   5    Arguments for ...:
   6    Arguments (...):
   7    Control arguments:
   8    Control arguments as a command:
   9    Control arguments as a request:
  10    Control arguments as an active function:
  11    Control arguments as an active request:
  12    Control arguments for attach description:
  13    Control arguments for open description:
  14    Control arguments for close description:
  15    Control arguments for detach description:
  16    Control arguments for ...:
  17    Control arguments (...):
  18    Control order:
  19    Entry points in ...:
  20    Examples:
  21    Function:
  22    List of control operations:
  23    List of controls:
  24    List of elements:
  25    List of i/o operations:
  26    List of mode strings:
  27    List of opening modes:
  28    List of operations:
  29    List of requests:
  30    List of ...:
  31    Notes on the info_ptr:
  32    Notes:
  33    Notes on ...:
  34    Syntax:
  35    Syntax as a command:
  36    Syntax as an active function:
  37    Syntax as an active request:
  38    Syntax of attach description:
  39    Syntax of open description:
  40    Syntax of close description:
  41    Syntax of detach description:

 Obsolete or Deprecated Section Title              Preferred Title

       Access requirement:                          =>  Access required:
       Access requirements:                         =>  Access required:
       Active function argument:                    =>  Arguments:
       Active function arguments:                   =>  Arguments:
       Active function control argument:            =>  Control arguments as an active function:
       Active function control arguments:           =>  Control arguments as an active function:
       Active function syntax:                      =>  Syntax as an active function:
       Active function usage:                       =>  Syntax as an active function:
       Argument (...):                              =>  Arguments (...):
       Argument:                                    =>  Arguments:
       Arguments as active function:                =>  Arguments:
       Argument for io_call:                        =>  Arguments for io_call:
       Argument for iox_$control:                   =>  Arguments for iox_$control:
       Command argument:                            =>  Arguments:
       Command arguments:                           =>  Arguments:
       Command control argument:                    =>  Control arguments as a command:
```

```
Command control arguments:                      =>  Control arguments as a command:
Command syntax:                                 =>  Syntax as a command:
Command usage:                                  =>  Syntax as a command:
Control argument as command:                    =>  Control arguments as a command:
Control argument (...):                         =>  Control arguments (...):
Control argument:                               =>  Control arguments:
Control argument as active function:            =>  Control arguments as an active function:
Entry point in ...:                             =>  Entry points in ...:
Entrypoint in ...:                              =>  Entry points in ...:
Entrypoints in ...:                             =>  Entry points in ...:
Example ...:                                    =>  Examples:
List of Operations:                             =>  List of operations:
List of OPERATIONS:                             =>  List of operations:
List of Requests:                               =>  List of requests:
List of REQUESTS:                               =>  List of requests:
Note:                                           =>  Notes:
Notes for ...:                                  =>  Notes on ...:
Purpose:                                        =>  Function:
Syntax and Attach Description:                  =>  Syntax:
Syntax as active function:                      =>  Syntax as an active function:
Syntax as command:                              =>  Syntax as a command:
Syntax as request:                              =>  Syntax:
Syntax as a request:                            =>  Syntax:
Syntax of ...:                                  =>  Syntax:
Where:                                          =>  Arguments:
   ------------------------------------------------
```

vi -rules command

```
 Section Titles for  Command:

   ORDER    CARDINALITY           TITLE
     1     1 required           Syntax as a command:
     2     1 required           Function:
     3     1 allowed            Arguments:
           0 or more            Arguments for ...:
           0 or more            Arguments (...):
           1 allowed            List of operations:
     4     1 allowed            Control arguments:
           1 allowed            Control arguments as a command:
           0 or more            Control arguments for ...:
           0 or more            Control arguments (...):
     5     1 allowed            Access required:
           0 or more            List of ...:
           1 allowed            Notes:
           0 or more            Notes on ...:
     6     1 allowed            Examples:

           ------------------------------------------------

 Section Titles for  Active Function:

   ORDER    CARDINALITY           TITLE
     1     1 required           Syntax as an active function:
     2     1 required           Function:
     3     1 allowed            Arguments:
           0 or more            Arguments for ...:
           0 or more            Arguments (...):
           1 allowed            List of operations:
     4     1 allowed            Control arguments:
           1 allowed            Control arguments as an active function:
           0 or more            Control arguments for ...:
           0 or more            Control arguments (...):
     5     1 allowed            Access required:
           0 or more            List of ...:
           1 allowed            Notes:
           0 or more            Notes on ...:
     6     1 allowed            Examples:

           ------------------------------------------------

 Section Titles for  Command/Active Function:

   ORDER    CARDINALITY           TITLE
     1     1 required           Syntax as a command:
           1 required           Syntax as an active function:
     2     1 required           Function:
     3     1 allowed            Arguments:
           0 or more            Arguments for ...:
           0 or more            Arguments (...):
           1 allowed            List of operations:
     4     1 allowed            Control arguments:
           1 allowed            Control arguments as a command:
           1 allowed            Control arguments as an active function:
           0 or more            Control arguments for ...:
           0 or more            Control arguments (...):
     5     1 allowed            Access required:
           0 or more            List of ...:
           1 allowed            Notes:
           0 or more            Notes on ...:
     6     1 allowed            Examples:

           ------------------------------------------------
```

vi -rules subsystem

```
 Section Titles for  Subsystem Request:

   ORDER    CARDINALITY           TITLE
     1     1 required           Syntax:
     2     1 required           Function:
     3     1 allowed            Arguments:
           0 or more            Arguments for ...:
           0 or more            Arguments (...):
           1 allowed            List of operations:
     4     1 allowed            Control arguments:
           1 allowed            Control arguments as a request:
           0 or more            Control arguments for ...:
           0 or more            Control arguments (...):
     5     1 allowed            Access required:
           0 or more            List of ...:
           1 allowed            Notes:
           0 or more            Notes on ...:
     6     1 allowed            Examples:

           ------------------------------------------------

 Section Titles for  Subsystem Active Request:

   ORDER    CARDINALITY           TITLE
     1     1 required           Syntax as an active request:
     2     1 required           Function:
     3     1 allowed            Arguments:
           0 or more            Arguments for ...:
           0 or more            Arguments (...):
           1 allowed            List of operations:
     4     1 allowed            Control arguments:
           1 allowed            Control arguments as an active request:
           0 or more            Control arguments for ...:
           0 or more            Control arguments (...):
     5     1 allowed            Access required:
           0 or more            List of ...:
           1 allowed            Notes:
           0 or more            Notes on ...:
     6     1 allowed            Examples:

           ------------------------------------------------

 Section Titles for  Subsystem Request/Active Request:

   ORDER    CARDINALITY           TITLE
     1     1 required           Syntax:
           1 required           Syntax as an active request:
     2     1 required           Function:
     3     1 allowed            Arguments:
           0 or more            Arguments for ...:
           0 or more            Arguments (...):
           1 allowed            List of operations:
     4     1 allowed            Control arguments:
           1 allowed            Control arguments as a request:
           1 allowed            Control arguments as an active request:
           0 or more            Control arguments for ...:
           0 or more            Control arguments (...):
     5     1 allowed            Access required:
           0 or more            List of ...:
           1 allowed            Notes:
           0 or more            Notes on ...:
     6     1 allowed            Examples:

           ------------------------------------------------
```

```
Section Titles for  Subsystem Summary:

  ORDER    CARDINALITY            TITLE
    1     1 allowed             List of requests:

          ------------------------------------------------

Section Titles for  Subsystem Topic:

  ORDER    CARDINALITY            TITLE
    1     1 or more             Notes on ...:

          ------------------------------------------------
```

vi -rules subroutine

```
 Section Titles for  Subroutine Introduction:

   ORDER    CARDINALITY           TITLE
     1     1 allowed             Function:
     2     1 allowed             Access required:
           0 or more             List of ...:
           1 allowed             Notes:
           0 or more             Notes on ...:
     3     1 allowed             Examples:
     4     1 allowed             Entry points in ...:
                                    (List is generated by the help command)

           ________________________________________________

 Section Titles for  Subroutine Brief Intro:

   ORDER    CARDINALITY           TITLE
     1     1 required            Entry points in ...:
                                    (List is generated by the help command)

           ________________________________________________

 Section Titles for  Subroutine Entrypoint:

   ORDER    CARDINALITY           TITLE
     1     1 required            Function:
     2     1 required            Syntax:
     3     1 required            Arguments:
     4     1 allowed             Access required:
           0 or more             List of ...:
           1 allowed             Notes:
           0 or more             Notes on ...:

           ________________________________________________
```

vi -rules io_module

```
 Section Titles for  I/O Module:

   ORDER    CARDINALITY           TITLE
     1     1 allowed             Syntax as a command:
     2     1 required            Syntax of attach description:
     3     1 required            Function:
     4     1 allowed             Arguments:
           0 or more             Arguments for ...:
           0 or more             Arguments (...):
     5     1 allowed             Control arguments for attach description:
           0 or more             Control arguments:
           0 or more             Control arguments as a command:
           0 or more             Control arguments for ...:
           0 or more             Control arguments (...):
     6     1 required            List of opening modes:
     7     1 allowed             List of operations:
     8     1 required            List of i/o operations:
     9     1 of ORDER allowed    List of controls:
           1 of ORDER allowed    List of control operations:
    10     1 allowed             List of mode strings:
           0 or more             List of ...:
    11     1 allowed             Access required:
    12     1 allowed             Notes:
           0 or more             Notes on ...:
    13     1 allowed             Examples:

           ------------------------------------------------

 Section Titles for  I/O Operation:

   ORDER    CARDINALITY           TITLE
     1     1 allowed             Syntax as a command:
     2     1 of ORDER required   Syntax of open description:
           1 of ORDER required   Syntax of close description:
           1 of ORDER required   Syntax of detach description:
     3     1 required            Function:
     4     1 allowed             Arguments for io_call:
           0 or more             Arguments:
           0 or more             Arguments for ...:
           0 or more             Arguments (...):
     5     1 required            List of opening modes:
     6     1 of ORDER allowed    Control arguments for open description:
           1 of ORDER allowed    Control arguments for close description:
           1 of ORDER allowed    Control arguments for detach description:
           0 or more             Control arguments as a command:
           0 or more             Control arguments as an active function:
           0 or more             Control arguments:
           0 or more             Control arguments for ...:
           0 or more             Control arguments (...):
     7     1 allowed             Access required:
           0 or more             List of ...:
           1 allowed             Notes:
           0 or more             Notes on ...:
           1 allowed             Examples:

           ------------------------------------------------
```

```
Section Titles for  I/O Control:

  ORDER    CARDINALITY            TITLE
    1     1 required           Control order:
    2     1 required           Syntax:
    3     1 required           Arguments for iox_$control:
    4     1 allowed            List of elements:
          1 allowed            Notes on the info_ptr:
    5     1 allowed            Syntax as a command:
          1 allowed            Syntax as an active function:
    6     1 allowed            Arguments for io_call:
    7     1 allowed            Control arguments:
          1 allowed            Control arguments as a command:
          1 allowed            Control arguments as an active function:
          0 or more            Control arguments for ...:
          0 or more            Control arguments (...):
    8     1 allowed            Access required:
          0 or more            List of ...:
          1 allowed            Notes:
          0 or more            Notes on ...:
          1 allowed            Examples:

         ------------------------------------------------
```