To: Distribution

From: Robert S. Coren

Date: 11/04/75

Subject: New Strategy for Conversion of Terminal Output

INTRODUCTION

The parts of the ring zero typewriter DIM concerned with
character conversion -- i.e., the subroutines tty_read and
tty_write -- have remained largely unchanged in design for a long
time. The process of character conversion on Multics is currently
very slow and inefficient, in particular taking no advantage of
EIS. The problem is especially acute with respect to output,
since there is in general about 8 times as much terminal output
as terminal input; accordingly tty_write is a major bottleneck in
ring zero. This MTB describes a proposed redesign of tty_write
which will speed it up considerably without any loss of function.
Similar changes are planned for tty_read at a later date.

THE CURRENT METHOD OF OUTPUT CONVERSION

In tty_write as currently implemented, each character of
user-supplied data is individually examined and looked up in
various tables to determine what should be placed in output
buffers to be sent to the 355 and thence to the terminal. Even in
"rawo" mode, where the user's data is passed on with no
conversion, each character is nonetheless copied individually,
with the count of characters being incremented one at a time.
When either the end of the user's data is reached or the maximum
number of ring-zero buffers the user is allowed to have is
filled, conversion stops and, if appropriate, the data so far
converted is shipped to the 355.

This mechanism has the obvious advantage of simplicity: it
is particularly easy to keep track of how many of the user's
characters have been transmitted and how much buffer space is
being used. However, this advantage is more than offset by the
loss of efficiency in processing characters one at a time. In
addition, the tables used for the conversion are kept, by
terminal type, in a ring zero data base (tty_ctl), and pointers

---

to them are derived by tty_write every time it is called. In this
setup, no method is available for the user to substitute his/her
own translation tables. Still worse, the same table is used both
for determining whether a character is "special" (requires
escaping or the addition of delays) and for converting from ASCII
to some "foreign" code (such as EBCDIC); this situation makes it
virtually impossible to avoid looking up and doing something
about every character input to tty_write.

PROPOSED_NEW_METHOD

    The new design is predicated on the assumption that the vast
majority of characters sent to the user's terminal are
"uninteresting" -- i. e., they are to be shipped as they are,
they do not require delays, and each one advances the carriage by
one position. A block of such characters can clearly be copied
into tty_buf all at once with a single EIS instruction, or at
least in buffer-sized chunks. The only problem is identifying the
limits of such a block, and making the necessary additions and
substitutions when an "interesting" character is encountered.
Wholesale translation (e. g., ASCII to EBCDIC) is a separate
issue, and can also be dealt with economically using EIS.

    The functions of tty_write can be logically divided into
four phases:

    1. Preliminary_conversion (specifically the translation of
       of lowercase letters to uppercase for a Teletype model 33
       or terminals in "capo" mode;

    2. Formatting, i.e., substitution of escape sequences,
       insertion of new-line characters in long lines,
       canonicalization and optimiziation of white space, etc.;

    3. Translation, as from ASCII to EBCDIC;

    4. Buffer_allocation_and_copying of characters into buffers
       in tty_buf, whence they will be read by the 355.

In the current tty_write, these four phases are executed more or
less simultaneously on each character; in particular, phases 2
and 3 (formatting and translation) are not distinguished, and are
driven by the same table. The new design executes each phase over
the entire input string (or as much of it as will be transmitted
at once) before passing on to the next phase. In most cases, of
course, either phase 1 or phase 3 or both can be omitted; in
"rawo" mode, tty_write can and does proceed directly to phase 4.

Each phase is provided with an "input pointer" to the
location where the previous phase left the data in its latest
form. This pointer points either to the user's original input or
to either of two buffers in tty_write's automatic storage, as
described later.

The only serious disadvantage to this scheme is that the
determination of how many of the user's characters are actually
to be shipped must be made in advance of conversion, and this
determination must attempt to take into account the probability
that the final output will contain more characters than the user
supplied. There is no ideal solution to this problem, but one has
been developed which ensures that the program will behave
correctly in all cases, and in general will have the same effect
as today (in terms of the number of calls required to output a
given string, the pressure put on tty_buf, etc.). This solution
is described later in this document.

Extensive use has been made in this design of three EIS
instructions: move with translation (mvt), test character and
translate (tct), and scan with mask (scm). PL/I builtin functions
such as translate do not completely meet our requirements;
therefore an ALM subroutine, tty_util_, is supplied, containing
entry points to perform the necessary functions.

The remaining sections of this MTB contain the following:

1. A more detailed description of the four phases of
   conversion mentioned above;

2. A discussion of space allocation and character counting;

3. A description of the data structures used for conversion
   and translation, as well as an indication of the proposed
   method for allowing the user to substitute his own
   versions of the relevant tables;

4. A module description of tty_util_.

Preliminary_Conversion

Certain terminals require uppercase-only output; similarly,
a user can specify (by entering "capo" mode) that all lowercase
letters are to be converted to uppercase for output. These cases
are treated identically by tty_write: an mvt (move with
translation) instruction is used to copy the user's data into an

automatic buffer, using a translation table which substitutes
uppercase ASCII for lowercase. If the user is in "edited" mode,
this is all that needs to be done for this phase; if not,
however, each letter which was originally uppercase must be
preceded by an escape character ("\"). Therefore, in "¯edited"
mode, the translation table also replaces each uppercase letter
with the same character with its high-order bit (the "400(8)"
bit) turned on. After the mvt is completed, an scm (scan with
mask) instruction is executed to find the first character with
the "400" bit on; if one is found, all characters to the left of
it are copied to a second internal buffer, an escape is inserted
after the copied characters, and the high-order bit of the found
character is turned off. The scm is repeated on the remainder of
the characters in the first buffer until all characters have been
copied to the second buffer with escapes inserted as needed. If
no characters with the high-order bit on are found in the entire
string, no copying is done.

Formatting

     The search for, and correct handling of, "interesting"
characters is the most crucial of tty_write's functions, and the
one to which most of the time spent in tty_write is devoted. The
identification of "interesting" characters is facilitated by the
use of the tct (test character and translate) instruction under
control of a table containing zero entries for all
"uninteresting" characters and various indicators identifying the
different kinds of "interesting" ones: carriage movement
characters, ribbon shifts, and characters requiring the
substitution of escape sequences.

     The formatting phase of tty_write calls tty_util_$find_char
to find the first "interesting" character in the string;
tty_util_$find_char returns a tally of "uninteresting" characters
skipped over, the indicator value for the character it stopped
at, and an updated pointer to the character at which to start the
next scan. (See the module description of tty_util_ later in
this document.) tty_write copies the uninteresting characters
into an internal buffer (whichever one does not contain the
source string) and examines the indicator. If it designates an
escape sequence, the sequence is inserted in the buffer. For a
new-line, vertical tab, or form-feed character, a special table
is indexed to find the appropriate representation of the
character, and another table is searched to find the correct
number of delays to be inserted depending on column position,
terminal type, and baud rate. For "white space" (horizontal tab,
backspace, carriage return, or two or more blanks) tty_write
simply calculates and remembers what column position to end up
in; this information will either be used to insert appropriate

carriage motion characters before the next graphic to be inserted, or discarded if the next character involves vertical carriage motion. This process is repeated until all the source characters are used up. If it happens that the first call to tty_util_$find_char returns an indicator of zero and has used up the entire source string, no characters are moved by this phase.

The subroutine tty_util_$find_char uses a tct instruction to find interesting characters, but it must do other things as well. In the first place, for the instruction to notice that a character has either or both of its high-order bits on, a table of 512 entries would be needed, of which 384 would be identical; secondly, a single blank between two printing graphics is not interesting to tty_write, but two or more consecutive blanks are considered "white space," as is any combination of blank and one or more other carriage movement characters. To cover the first case, tty_util_$find_char performs two scm instructions to find the earliest character (if any) which does not fit in seven bits. For the case of multiple blanks, it is clearly undesirable to have a non-zero indicator in the tct table for blank, and thus force the tct to stop on all blanks, test to see if the next character is a blank, and then proceed if it is not. Instead, the tct is preceded by an scd (scan character double) instruction which looks for two successive blanks. The tally and pointer returned to tty_write reflect the earliest point in the source string at which either the tct, the scd, or either of the two scm's found anything interesting.

It will be seen from the module description of tty_util_$find_char later in this MTB that a "white space" indicator implies that the pointer points to the beginning of a block of white space, which tty_write then examines until it finds the end of the block. Therefore if the first interesting character found by tty_util_$find_char is a carriage movement character, it must check to see if the immediately preceding character is a blank, in which case it returns a pointer to the blank rather than the character following it.

Another responsibility of the formatting phase is the counting of output lines and watching for full pages. In the old tty_write, page length is respected only for ARDS-like screen terminals; when the maximum line count is reached, tty_write stops processing characters and sets a flag in the fixed control block (fctl) associated with the terminal. This flag gets transmitted to the 355, which then understands that, when the output is completed, it must not ask for more output for that channel until it receives a form-feed character as input. The new design extends the concept of page length to all terminals capable of receiving or transmitting a form-feed, and removes all knowledge of the end-of-page condition from the 355. In addition,

tty_write no longer stops processing characters when the line
count reaches maximum; instead, the formatting phase inserts a
warning string (such as "EOP") and a sentinel character at the
end of the page, and the copying phase (see below) later removes
each sentinel and turns on a flag in the buffer that ends the
page. When dn355 (the program that actually sends the buffers to
the 355) sees this flag, it ceases transmission, and now sets the
flag in the fctl block. When it receives input for a channel with
the end-of-page flag on, it scans this input for a form-feed; if
it finds one, it replaces it with a PAD character (177(8)), turns
off the fctl flag, and starts up output for the channel again.
(1)

Translation

The translation phase is very similar to the preliminary
conversion phase described earlier. An mvt instruction is used to
copy the entire string from wherever it was left by the preceding
phase to an automatic buffer, translating it from ASCII to the
appropriate output code in the process. (At present the only
output codes other than ASCII known to Multics are EBCDIC and IBM
Correspondence.) This does not complete the process for a
terminal which requires case-shift characters (which currently
includes all terminals for which translation is done); the
insertion of case-shift characters is done in a similar manner to
the insertion of escapes before capital letters as described
under "Preliminary Conversion." The translation table causes the
high-order bit of each uppercase character to be turned on (in
this context the term uppercase refers not only to capital
letters but to all characters for which the shift key must be
depressed while typing) and the "200(8)" bit of each lowercase
character to be turned on; characters which may be in either case
(such as space) contain no extra bits. After translation, an scm
is done to find the first character in the opposite case to the
one in which the terminal was at the start of the output; all
characters to the left of it are copied, an appropriate shift
character is inserted after the copied characters, and another
scm is used to find the next change of case. If all the output
characters are in the same case, no copying is done. Note that it
is not necessary to turn off the high-order bits of the uppercase
characters, since these bits will be ignored by the remainder of
the tty DIM and ultimately thrown away by the 355.

----

(1) A mode may be added in future which would allow a user to
specify that when a page is full the tty DIM should automatically
output a form-feed rather than waiting for one to be input. On a
hard-copy terminal, this mode would probably make more sense than
the current method.

Buffer_Allocation_and_Copying

The final phase of tty_write consists of allocating buffers
in tty_buf and copying the final output into these buffers. A
buffer in tty_buf is 16 words long, of which the first contains a
forward pointer, flags, and a tally; each buffer therefore holds
up to 60 characters. Thus one buffer is allocated by tty_write
for every 60 characters of final output, and the characters are
copied in 60-character chunks. If an end-of-page sentinel is
encountered, the end-of-page flag is turned on in the current
buffer, and the buffer is not filled past the sentinel. If output
already processed for the particular channel has not yet been
sent, a chain of buffers for that channel will already exist; if
the last buffer in this chain is not full, and does not have its
end-of-page flag on, it will be filled before further buffers are
allocated. The newly-allocated buffers will be threaded onto the
old chain. Finally, if the "send_output" flag in the fctl block
is on, indicating that dn355 and the 355 itself are prepared to
handle output for the channel, tty_write calls dn355$io_command
to cause a mailbox to be sent to the 355 telling it that output
is on the way.

SPACE_ALLOCATION_AND_CHARACTER_COUNTING

Because the input string undergoes wholesale modification at
several points, it is necessary to decide how many of the user's
characters to process before actually doing anything. Certain
constraints which exist in the present implementation will be
retained: no more than a certain fraction of available buffers in
tty_buf are to be assigned to a single channel at any time; and
no output chain of more than a certain number of buffers will be
built. The particular numbers involved are, for the sake of
convenience and simplicity, preset system-wide constants. The
current values, which appear reasonable, are 1/4 and 16
respectively; i.e., no channel is ever assigned more than 1/4 as
many buffers as are free at the time of assignment, and a maximum
of 16*60 = 960 characters will be processed by a single call to
tty_write.

The first determination made by tty_write, then, is the
maximum number of buffers the caller is allowed to have, which
is:

        maxbuf = min(16, (buffers_left/4) - buffers_assigned)

The number of characters to process may then be expressed as

nchars = min(chars_supplied, maxbuf*chars_per_buffer)

If the terminal is in "rawo" mode, this is the number of
characters that will actually be shipped, and nothing further
need be done. In general, however, the number of characters
actually output is somewhat larger than the number supplied;
meters done at various times show an average growth ratio of
about 6:5. Accordingly, for non-raw output, tty_write will
multiply nchars as calculated above by 0.8 to allow for growth
(this actually allows for a growth ratio of 5:4, which gives us
some leeway). As a result, the size of the output string can
grow by as much as 25% without requiring more buffers than one
line is "supposed" to have; however, the restriction to 1/4 of
the available buffers is a very conservative one, so if it
occasionally proves necessary to allocate an "extra" buffer the
overall effect on available buffer space should not be
noticeable.

     An additional consideration arises from the use of internal
buffers in tty_write. Because of the possibility of more than one
intermediate copy, two such buffers are needed, and rather than
create two segments so as to allow each buffer to grow
essentially without limit, it was decided to set aside fixed-size
buffers in tty_write's stack frame. The size chosen for each of
these buffers is the maximum allowable output chain size, i. e.,
960 characters.

     Clearly growth ratios greater than 5:4 can and will occur;
there are pathological cases such as an object or other non-ASCII
segment being printed on a 2741 terminal, which involves a growth
ratio of more than 5:1 (<upper_shift> ¢ <lower_shift> nnn for
each input character, plus added new-lines and ¢c markers). Thus
despite precautions we must be prepared for the possibility that
in the course of translation or formatting we will run out of
space in the internal buffer. When this happens, the number of
input characters to be handled is cut in half, and character
processing is started over from phase 1. This solution is
admittedly crude, but the alternative is to keep track at all
times of the number of the user's characters which have been
processed, which in some cases (particularly the transformation
of white space) is non-trivial in the new scheme; it seems
inadvisable to incur this overhead on every call to tty_write in
order to avoid expense in a rare case. The problem will only
arise when attempting to process 768 user characters of which an
unusually large number have to be escaped; considering that the
average output message is around 50 characters, the overall
expense of double processing in such a case is not likely to be
significant.

If space in tty_buf is unusually tight, then an abnormal
character string which is not large enough to overflow the
internal buffer space might nonetheless require the allocation of
more buffers than are available. If tty_write finds that it is
about to allocate the last buffer, it will take the same action
as if it were about to overflow one of its internal buffers, i.
e., divide the number of input characters in half and start over.
This circumstance is considered even less probable than the
overflowing of an internal buffer; if it happens often it is
probably an indication that tty_buf is too small.

We could, of course, reduce the frequency of overflow events
still further by decreasing the percentage of the theoretical
maximum number of characters that will actually be processed at
once; however, this would increase the probability that the
user's characters could not be handled in a single call, thereby
requiring users to go blocked for output more often and
increasing the number of calls to tty_write. The figures used in
this MTB are a preliminary estimate based on what seems
reasonable; they can easily be adjusted if metering shows either
a high frequency of double processing or an excessive (i.e.,
greatly increased) number of calls to tty_write.

DATA_STRUCTURES

This section describes the tables to be used by tty_write
for translation and formatting. Packed pointers to these tables
will be kept in the control block (ctl) allocated for each line
when it dials up; the default tables are in tty_ctl on a
per-terminal-type basis as at present, and pointers to these
tables are copied from tty_ctl into the ctl block the first time
tty_write is called for any one dialup.

In a future modification, control operations will be
provided to allow a user to substitute his/her own version of one
or more of these tables. Macros (in mexp) may also be provided to
facilitate the construction of such tables. This capability,
however, introduces problems as long as the Answering Service
does not use the secure (ring 1) message facility rather than
calling hcs_$tty_write directly. Write calls from the Initializer
for a terminal using user-supplied translation tables would
reference pointers in the user's address space (not the
Initializer's), which at best would result in garbage being
printed on the user's terminal. (A possible alternative to using
the message facility is to have the Answering Service call a
special entry which uses the default tables for the terminal type
whether the user has supplied tables or not; the output might be
garbled, but at least the tables would be accessible to the

Initializer.)

Default_Table

The header of tty_ctl contains an array, indexed by terminal
type, of relative offsets of default tables. The default table
contains relative pointers to the conversion tables to be used by
default for the given terminal type. The format of the default
table is as follows:


```
dcl 1 device_defaults aligned based,
        2 flags unal,
          3 shifter bit (1) unal,
          3 upper_case_only bit (1) unal,
          3 pad bit (7) unal,
        2 delay_char char (1) unal,
        2 upper_case char (1) unal,
        2 lower_case char (1) unal,
        2 tct_offset fixed bin (18),
        2 mvt_offset fixed bin (18),
        2 special_offset fixed bin (18),
        2 delay_offset (4) fixed bin (18);
```


shifter   is "1"b if the terminal requires case
                         shift characters.

upper_case_only   is "1"b if the terminal handles only
                         capital letters.

delay_char   is the ASCII form of the character used
                         for carriage movement delays.

upper_case   is the uppercase shift character.

lower_case   is the lowercase shift character.

tct_offset   is the relative offset (in tty_ctl) of
                         the default table used by
                         tty_util_$find_char for identifying
                         "special" characters.

mvt_offset   is the relative offset of the table used
                         by tty_util_$mvt for translation, or 0
                         if translation is not required for the
                         particular terminal type.

special_offset   is the relative offset of the default
                        version of the special_chars table
                        described below.

delay_offset   is an array of offsets of the
                        delay_tables (described below) to be
                        used for this terminal type at 110, 150,
                        300, and 1200 bps respectively.

Special_Characters_Table

The special characters table is used by the formatting phase
of tty_write. It has the following format:


```
        dcl 1 special_chars aligned based,
              2 cmt (6) aligned,
                3 count fixed bin (8) unal,
                3 chars (3) char (1) unal,
              2 printer_on aligned,
                3 count fixed bin (8) unal,
                3 chars (3) char (1) unal,
              2 printer_off aligned,
                3 count fixed bin (8) unal,
                3 chars (3) char (1) unal,
              2 red_ribbon_shift aligned,
                3 count fixed bin (8) unal,
                3 chars (3) char (1) unal,
              2 black_ribbon_shift aligned,
                3 count fixed bin (8) unal,
                3 chars (3) char (1) unal,
              2 end_of_page aligned,
                3 count fixed bin (8) unal,
                3 chars (3) char (1) unal,
              2 escape_length fixed bin,
              2 not_edited_escapes (10 refer (escape_length)),
                3 count fixed bin (8) unal,
                3 chars (3) char (1) unal,
              2 edited_escapes (10 refer (escape_length)),
                3 count fixed bin (8) unal,
                3 chars (3) char (1) unal;
```


Note: In each of the level 2 substructures in this structure
declaration, count, which has a value 0 <= count <= 3, indicates
the number of characters in the sequence; the first count
elements of the chars array is the sequence itself. If count is
zero, there is no sequence for the character in question.

cmt

describes the character sequences to be used for the six carriage movement characters, in this order: new-line, carriage return, backspace, horizontal tab, vertical tab, form-feed. If count is zero, the carriage movement function in question is not available on the terminal. In this case, the following action is taken:

new-line   <invalid>

carriage return   substitute appropriate number of backspaces

backspace   substitute carriage return and appropriate number of blanks and/or horizontal tabs

horizontal tab   substitute appropriate number of blanks

vertical tab, form-feed   ignore character

The counts for carriage reurn and backspace may not both be zero.

printer_on

is the character sequence to be used to implement the "printer_on" control operation.

printer_off

is the character sequence to be used to implement the "printer_off" control operation.

red_ribbon_shift

is the character sequence to be substituted for a red ribbon-shift character.

black_ribbon_shift

is the character sequence to be substituted for a black ribbon-shift character.

end_of_page

is the character sequence to be printed to indicate that a page of output is full.

escape_length

is the number of output escape sequences in each of the two escape arrays.

not_edited_escapes   is an array of escape sequences to be
                      substituted for particular characters if
                      the terminal is in "^edited" mode. This
                      array is indexed according to the
                      indicator returned by
                      tty_util_$find_char.

edited_escapes   is an array of escape sequences to be
                      used in "edited" mode. It is·indexed in
                      the same fashion as not_edited_escapes.

Delay_Table

    The delay table provides the number of delays to be used in
conjunction with carriage movement characters. It has the
following format:


```
dcl 1 delay based aligned,
        2 vert_nl fixed bin,
        2 horz_nl fixed bin,
        2 const_tab fixed bin,
        2 var_tab fixed bin,
        2 backspace fixed bin,
        2 vt_ff fixed bin;
```


vert_nl   is the number of delay characters to be
                      output for all new-lines to allow for
                      the line-feed.

horz_nl   is a factor used to determine the number
                      of delays to be added for the carriage
                      return portion of a new-line, depending
                      on column position. The formula for
                      calculating the number of delay
                      characters to be output following a
                      new-line is:

          ndelays = vert_nl + (horz_nl*column)/512

const_tab   is the constant portion of the number of
                      delays associated with any horizontal
                      tab character.

var_tab   is a factor used to determine the number
                      of additional delays associated with a
                      horizontal tab depending on the number

of columns traversed. The formula for
calculating the number of delays to be
output following a horizontal tab is:

$$ndelays = const\_tab + (var\_tab * n\_columns)/512$$

backspace   is the number of delays to be output
following a backspace character. If it
is negative, it is the complement of the
number of delays to be output with the
first backspace of a series only (or a
single backspace). This is for terminals
such as the TermiNet 300 which need
delays to allow for hammer recovery in
case of overstrikes, but do not require
delays for the carriage motion
associated with the backspace itself.

vt_ff   is the number of delays to be output
following a vertical tab or form-feed.

MODULE_DESCRIPTION_OF tty_util_

Name: tty_util_

     The entries in this module are used for translation and
formatting of typewriter input and output. All of them run in
the caller's stack frame, and take as an argument a pointer to an
argument structure provided by the caller.

Entry: tty_util_$find_char

     This entry uses a tct (test character and   translate)
instruction to search a given string for "interesting" characters
as defined by a translation table supplied by the caller.

Usage

          declare tty_util_$find_char entry (ptr);

          call tty_util_$find_char (argptr);

where   argptr   is a pointer to the structure   described
                              below. (Input)

          dcl 1 tct_arg_structure based aligned,
                2 stringp ptr,
                2 stringl fixed bin,
                2 tally fixed bin,
                2 tablep ptr,
                2 indicator fixed bin,
                2 workspace (3) fixed bin;

     stringp   is a pointer to the string to be tested;
                              if indicator (see below) is 3 or 7, it
                              is updated to point to   the   first
                              "interesting" character in the string;
                              otherwise, it is updated to point to the
                              character   following   the   first
                              "interesting" character. (Input/Output)

     stringl   is the length in characters of the
                              string to be tested.   If stringl is
                              greater than 2000, only the first 2000

characters are tested. stringl is decremented by the same number of characters as stringp is advanced. If the entire string is searched and indicator is 0, stringl is set to 0. (Input/Output)

tally   is the number of "uninteresting" characters passed over by the test. (Output)

tablep   is a pointer to an aligned packed array of 128 fixed bin (8) values to be used as a translation table. The elements correspond to ASCII characters in the normal collating sequence; the value of each element is zero if the corresponding character is uninteresting, or else the value of the indicator to be returned if the corresponding character is encountered. (Input)

indicator   is the result of the search. It may have the following values:

                         0 -- no special characters

                         1 -- new-line

                         2 -- carriage return

                         3 -- "white space," i. e., horizontal
                              tab, two or more consecutive
                              blanks, or a combination of one or
                              more blanks and a tab or backspace
                              character. stringp is set to point
                              to the first "white space"
                              character.

                         4 -- backspace

                         5 -- vertical tab

                         6 -- form-feed

                         7 -- character requiring octal escape

                         8 -- red ribbon shift

                         9 -- black ribbon shift

other -- a character requiring a special
escape sequence. The indicator
value is the index into the
escape table of the sequence to
be used, plus 16.

workspace   is to be used by tty_util_ for temporary
storage if necessary.

Entry: tty_util_$mvt

This entry is used to translate a character string using an
mvt (move with translation) instruction.

Usage

        declare tty_util_$mvt entry (ptr);

        call tty_util_$mvt (argptr);

where   argptr   is a pointer to the mvt_arg_structure
                         described below. (Input)

        dcl 1 mvt_arg_structure based aligned,
              2 stringp ptr,
              2 stringl fixed bin,
              2 pad fixed bin,
              2 tablep ptr,
              2 targetp ptr,
              2 workspace (2) fixed bin;

        stringp   is a pointer to the character string to
                             be translated. (Input)

        stringl   is the length in characters of the
                             string pointed to by stringp. (Input)

        tablep   is a pointer to an aligned character
                             string of length 128 to be used as a
                             translation table. (Input)

        targetp   is a pointer to the place where the
                             translated string is to be placed; it
                             must point to a character string of
                             length stringl or greater. (Input)

workspace   is as above.

Entry: tty_util_$scm

Tnis entry is used to search a character string for a
character with one of its two high-order bits on, using an scm
(scan with mask) instruction.

Usage

        declare tty_util_$scm entry (ptr);

        call tty_util_$scm (argptr);

where   argptr   points   to   the   scm_arg_structure
                           described below. (Input)

        dcl 1 scm_arg_structure based aligned,
              2 stringp ptr,
              2 stringl fixed bin,
              2 tally fixed bin,
              2 search_mask bit (2) aligned,
              2 found_flag bit (1) aligned,
              2 workspace (4) fixed bin;

    stringp   is a pointer to the string to be
                           scanned. If the scan succeeds, it is
                           updated to point to the character in
                           question. (Input/Output)

    stringl   is the length of the string pointed to
                           oy stringp. It is decremented by as many
                           characters as stringp is advanced.
                           (Input/Output)

    tally   is the number of characters passed over
                           during the scan (i. e., the number of
                           characters to the left of the character
                           found, or the length of the string if no
                           character is found). (Output)

    search_mask   is "10"b if the 400(8) bit is to be
                           searched for, or "01"b if the 200(8) bit
                           is to be searched for. (Input)

found_flag   is set to "1"b if a character with the
                            bit specified by search_mask on is
                            found; otherwise it is set to "0"b.
                            (Output)

workspace   is as above.