To: Distribution

From: Robert S. Coren

Date: 02/25/76

Subject: New Strategy for Conversion of Terminal Input

INTRODUCTION

MTB 234 described a new method for processing terminal output in ring zero making extensive use of EIS. The design described has since been implemented in MIT system 27-6, and will be part of Multics Release 3.1. The new implementation shows approximately a threefold improvement in the efficiency of tty_write, measured in terms of the virtual CPU time spent in tty_write for each character sent to the 355; on MIT, about 1% of total time charged is now spent in tty_write, as compared to about 2.5% in pre-27-6 systems.

The current implementation of the ring-zero input-processing module, tty_read, has essentially the same problems as those described in MTB 234 for the old tty_write: characters are processed one at a time, even in "rawi" mode; translation, canonicalization, and escape processing are handled simultaneously and driven by a single table; fixed tables in ring zero are used, pointers to which are constructed on every call. In addition, canonicalization is mishandled in some cases, as indicated in MTB 251; and the "prescan" function, which is intended to examine input for case-shift characters and to update the current column position for use by tty_write, is invoked at the wrong time and is therefore unreliable.

This MTB proposes a redesign of tty_read along the same lines as the recently-completed redesign of tty_write. Character-by-character processing is abandoned in favor of separate phases using PL/I builtin functions and ALM subroutines coded with EIS; canonicalization is reimplemented so as to conform to the rules set forth in MTB 251; the "prescan" function is removed from tty_read altogether, and its equivalent added to the 355 software (as described in a separate MTB). A version of tty_read implementing this design is intended for Multics Release

---

4.0.

One incompatible change that is being proposed is to discard all "invisible" characters (i. e., control characters that do not involve carriage or paper motion) whenever the channel is in "can" or "erkl" mode. The motivation for this proposal arises from these characters' invisibility: they do not show up on most terminals, and their retention violates the principle of canonicalization, that the contents of a line of input depend on its physical appearance. In other words, there is no way to distinguish visually between a#b and a<ETX>#b; what does the # erase? What column position does the ETX occupy?

The ability to input such characters directly (i. e., rather than by using octal escape sequences) seems to be of limited utility. The one exception might be the desire to use such a character as a kill or erase character; there are systems in existence which use CAN (octal 030, input by typing <CTL>x), as a kill character. User-replaceable kill and erase characters are planned for the future; it would not be too difficult to arrange not to throw away control characters which were being used for a Multics-defined purpose. For the present, a user employing special characters for erase and kill must process them in the user ring, and accordingly would not be in "can" or "erkl" mode. In addition, since the elimination of control characters would be a translation function (see below), user-substitutable translation tables (also a planned future improvement) would allow a user to admit selected control characters at will. In any case, all possible 9-bit patterns can be input as octal escapes.

One implication of this change is that the special meaning of the ESC character (octal 033) is eliminated for input. This character has been used primarily to insert ribbon-shift characters; this can be done by using the octal escape sequences \016 and \017.

PROPOSED NEW DESIGN

Overview

The obligation of tty_read, when called through hcs_, is to return in a caller-supplied buffer either 1) as many characters as the caller specified; 2) all characters up to and including the first "break" character present in ring-zero buffers for the specified channel; or 3) all characters remaining in the buffers for the specified channel, whichever is fewest. The "break" character is by default a newline character; there is currently no way to change this, but future modifications may permit it.

Certain transformations may be performed on the characters typed by the user, such as reduction to canonical form, removal of "erased" and "killed" characters, and the interpretation of escape sequences. The application of these transformations depends on both the modes associated with the channel and the contents of certain tables which are available to tty_read.

The functions of tty_read may be divided into the following phases:

1. Copying raw input data from tty_buf, and freeing the ring-zero buffers;

2. Translation to ASCII

3. Canonicalization of the contents of column positions

4. Erase and kill processing

5. Escape sequence processing

Clearly, these five phases are not always necessary. Phases 3, 4, and 5 depend on "can", "erkl", and "esc" modes, respectively; in "rawi" mode, only phase 1 is required.

For convenience and to ensure consistency, conversion (the generic term used here for the relevant subset of phases 2 through 5) is done on all characters up to and including the first break character in the input buffers, whether or not the break character is found within the limit specified by the caller. This avoids the possibility of terminating conversion in the middle of an escape sequence or of a line that is subsequently killed, and also allows for the possible shrinkage of the input string (through the deletion of extraneous white space and the condensation of escape sequences, for example). "Extra" characters thus converted (i. e., those that cannot be returned because the caller has not provided sufficient space) are saved in reallocated buffers in tty_buf; these buffers are marked with a "converted" flag and chained to the head of the channel's input chain so that they can be picked up by the next call to tty_read. In two exceptional cases, conversion cannot proceed to the first break character: the first is, obviously, when no break character is present; the other is when the size of tty_read's internal automatic buffers is exceeded. For reasons that will be explained later in this document, both these cases are expected to be very rare.

Reference is made in the course of this document to entries in the subroutine tty_util_, which is described in MTB 234. A new entry, tty_util_$tct, has been added; it performs the same function as tty_util_$find_char, except that it checks neither for characters with their high-order bits on nor for combinations of white-space characters.

The remainder of this document consists of the following:

1. A few remarks on the management of tty_read's internal buffer space;

2. A more detailed description of the five conversion phases mentioned above;

3. A description of the modifications required to the data structures described in MTB 234;

4. Module descriptions of the new column canonicalization routine, tty_canon (which replaces the old tty_con), and the new entry tty_util_$tct.

Familiarity with the material in MTBs 234 and 251 is assumed throughout.

Space Management

During conversion, intermediate forms of the input string result from each conversion phase; for the storage of these intermediate strings, two buffers are maintained in tty_read's automatic storage. Clearly this sets an upper limit on the allowable length of the input string. The normal limiting factor, of course, is the presence of a break character, and input lines longer than 100 characters are rare; a further limitation is imposed by the 355 software, which takes a channel out of receive mode if more than 600 characters are input without a break character. The input string can grow during canonicalization through the replacement of carriage returns by multiple backspaces, but this occurrence too is rare. All in all, a buffer size of 720 is very unlikely to be exceeded.

Consequently, no more than 720 characters are copied into the internal buffer from tty_buf. If the canonicalization phase attempts to increase the length of the string past 720, tty_read will start again from the beginning with a limit of 480 characters to be copied. This limit is entirely safe, since canonicalization cannot increase the length of the string by more than 50%. Because of the remote possibility that this restart may be necessary, buffers in tty_buf from which input characters have been copied cannot be freed until after the canonicalization phase is completed.

Since conversion is, if possible, carried out on all characters up to and including the first break character, the final converted string may be larger than the buffer provided by the caller. If this is the case, enough characters to fill the caller's buffer are returned; the remainder of the converted characters, as indicated above, are saved in buffers in tty_buf in each of which a "converted" flag is set. In addition, if one of these buffers contains a break character (the last one generally will), a "break" flag is set in that buffer. These buffers are added to the head of the chain of unconverted input buffers (the "read chain"), and the input pointer in the control block associated with the channel is set to point to the first "converted" buffer.

Copying

IN "rawi" MODE

The copying phase in "rawi" mode is very simple. Characters are copied from tty_buf, starting at the head of the read chain, directly into the caller's buffer, until either the caller's buffer is filled or the read chain is exhausted. Any buffer from which all the characters are thus copied is freed.

NOT IN "rawi" MODE

If there are any "converted" buffers at the head of the read chain, characters are copied from these buffers directly into the caller's buffer until either the caller's buffer is full, a break character has been copied, or the chain of converted buffers is exhausted. (In general, the last converted buffer contains a break character, and non-last converted buffers do not.) Any converted buffer from which all the characters are copied is freed.

If there are no converted buffers, or the converted buffer chain is exhausted without encountering a break character or filling the caller's buffer, characters are copied from the unconverted read chain (if present) into the first of tty_read's automatic buffers, until either a break character is encountered, the read chain is exhausted, or the internal buffer is filled. Buffers are not freed at this time, for the reason given above under "Space Management."

Because the 355 does not normally send input to the 6180 until a break character is typed, the read chain almost always ends with a break character. (Consequently, the converted chain usually does, too.) It might not if there was a quit on a channel not in "hndlquit" mode (in "hndlquit" mode the read chain is discarded on a quit), or if the channel exceeded the 355 software's 600-character limit.

If any characters were copied from unconverted buffers, conversion of the contents of tty_read's automatic buffer begins.

Translation

If a translation table exists for the terminal type associated with the channel, it is used in a call to tty_util_$mvt to copy the characters from one internal buffer to the other, simultaneously translating it to ASCII. Translation is required for IBM-type terminals using either EBCDIC or Correspondence character codes; it is also used to translate capital letters to lowercase for uppercase-only terminals such as a Teletype Model 33. (Escaped letters will be changed back to uppercase by the escape-processing phase.)

The translation phase does not have to deal with case-shift characters. Under the new design, the 355 is responsible for recognizing case shifts, and for turning on the 100(8) bit in all uppercase characters (characters on shifting terminals are only six bits). All that is necessary on the 6180 side is a translation table that includes characters with the "100" bit on and translates case-shift characters to ASCII NUL characters.

If the channel is in "can" or "erkl" mode, a further translation is done using a general table which translates "invisible" characters (see above) to NUL (all zero) characters. NUL characters are subsequently discarded by the canonicalization phase.

Canonicalization

Column-position canonicalization takes care of itself unless the input string contains leftward carriage motion, i. e., backspace and/or carriage return characters. In addition, backspaces and carriage returns at the left margin or immediately preceding a newline are discarded. In other cases, canonicalization must be performed in accordance with the rules given in MTB 251.

The canonicalization phase therefore begins by searching the internal buffer (using the PL/I "search" builtin) for a left-motion character (carriage return or backspace). If the first character is a left-motion character, the buffer pointer is advanced by one character, the string length is decremented by one, and the new string is searched as before. If a left-motion character is found, a verify builtin is used to discover if the rest of the line consists of white space (backspaces, carriage returns, spaces, horizontal tabs, or NULs) followed by a newline. If this is the case, the string length is reduced to the result of the search, and the newline is copied to the new end of the string. If a left-motion character is discovered in any other position, tty_canon is called to perform column canonicalization.

The subroutine tty_canon is a revised version of the old tty_con, and uses the same basic algorithm: store each printing graphic from the input string in an array along with its correct column position; sort the array by column position, and by character within each column position; restore the characters to the input string location in the resulting order, inserting backspaces and spaces as appropriate. Tabs must be treated as a slightly special case of printing graphic, so that tabs which are in no way overstruck are preserved but others are replaced by spaces.

A module description of tty_canon appears at the end of this document; the calling sequence has been modified so that the module could theoretically be called with an arbitrary string in other environments than that of the ring-zero typewriter DIM. The resulting calling sequence is still not ideal, as it contains arguments that are both input and output; this approach is retained for reasons of efficiency. Eventually, an essentially equivalent module can be implemented in the user ring.

The structure used for the elements of the sorting array makes the sort very easy, thus:

```
dcl 1 column_array (max_size) aligned,
      2 column fixed bin (17) unaligned,
      2 erase bit (1) unaligned,
      2 kill bit (1) unaligned,
      2 vertical bit (1) unaligned,
      2 pad bit (5) unaligned,
      2 not_tab bit (1) unaligned,
      2 char char (1) unaligned;
```

The "erase" bit indicates an erase character; the "kill" bit indicates a kill character; the "vertical" bit indicates a non-newline character requiring vertical carriage motion (i. e., vertical tab or form-feed); the "not_tab" bit is on for any character except a horizontal tab. It can be seen that by treating each element of the array as a single value for the purpose of sorting, the characters automatically come out in column order and in character order in each column, except that: 1) an erase character will always be the last character in its column position; 2) a kill character will be last in its column position unless overstruck with an erase character; 3) a horizontal tab will always be the first character in its column position; and 4) a vertical-motion character will follow all characters other than an erase or kill character. Since during the initial scan, a vertical-motion character causes both the "current" column and the "starting" column to be set to the next highest multiple of 1000 (the "starting" column is the column assigned to the left margin, initially 0), a vertical-motion character cannot share a column position unless 1000 or more column positions are actually typed. A newline is assigned a column position of 2**17 - 1 so that it will always sort to the end of the line.

Kill processing is not done by tty_canon; kill characters are sorted to the end of the column position to make things easier for the kill-processing phase of tty_read. Erase characters are only interesting to tty_canon if they are overstruck; since an overstruck erase character sorts to the end of its column position, the rescan step, when it finds an erase character that is not first in its column position, deletes it and all preceding characters with the same column position.

Since a tab sorts to the beginning of its starting column position, it is sufficient to check whether the graphic following the tab has a column position less than the next tab stop; if it does, the tab is dropped, and spaces are inserted as they are whenever there is gap between two graphics. Otherwise the tab is inserted in the final string.

NUL characters are not stored in the column_array; thus tty_canon completes the elimination of "invisible" characters.

The maximum length of the input string is passed as an argument to tty_canon; if the final string exceeds this length, only max_length characters are returned, and a status code of error_table_$long_record is returned.

Upon return from tty_canon, if the status code is zero, tty_read frees the ring-zero buffers from which characters were copied, as explained above; otherwise it resets its internal buffer size limit to 480 and starts again from the copying phase.

If the canonicalization phase completes without calling tty_canon, the string may still contain NUL characters; therefore if tty_canon has not been called, tty_read indexes the string for NUL characters, and copies the characters preceding and following each NUL into the other internal buffer, decrementing the string length by one for each NUL it finds.

Erase and Kill Processing

Erase and kill processing is really done in two passes, kill and then erase. The string resulting from the canonicalization phase is indexed from the right for a kill character; if one is found, and the immediately preceding character is not a non-overstruck escape character, the pointer to the beginning of the string is incremented to point to the character following the kill character, and the length of the string is decremented accordingly. If the kill character is preceded by an escape character that is <u>not</u> preceded by a backspace, the pointer and the length are not changed, and the remainder of the string (if any) is scanned for further kill characters.

The string resulting from the kill pass is now indexed for an erase character. If one is found anywhere but at the beginning of the string, the characters before and after the erased character(s) must be copied to the other internal buffer. The basic mechanism is to copy the characters to the left of the erased characters, decrement the count of total input characters by the number of erased characters plus one for the erase itself, and resume the scan starting with the character after the erase character. (If the erase character is preceded by an escape character not preceded by a backspace, the escape and erase characters are copied along with the preceding characters.) When the end of the string is reached, provided any copying has been done, all characters to the right of the last erase character are copied.

The number of characters to be erased (i. e., not copied) is determined as follows: if the character preceding the erase is "white space" (space or horizontal tab) the source string is searched backward for a non-white character, and all characters to the right of it are erased; if the character preceding the erase is a printing graphic, then the source string is searched backward until two non-backspace characters are found in succession, whereupon all characters from the one to the left of the leftmost backspace on are erased. Note that the character immediately preceding the erase character cannot be a backspace, since all overstruck erase characters are processed by tty_canon.

If the second or subsequent scan turns up an erase character as the first character in the string (as would happen if two erase characters were typed in succession), the determination of the number of erased characters is made in the same fashion as that described above, except that the characters at the end of the *target* string are examined; the erasing is carried out by decrementing the target pointer so that the erased characters will be overwritten, and decrementing the overall length accordingly.

Escape Sequence Processing

This phase, which is implemented in a similar manner to the formatting phase of tty_write (as described in MTB 234), actually deals not only with escape sequences, but with the elimination of white space before break characters and of characters designated as being "thrown away" for the current terminal type. It uses test character and translate (tct) instructions under control of a table containing zero entries for ordinary characters, and indicators identifying four types of "interesting" characters: break character, escape character, form-feed, and "throw-away" character.

This phase uses tty_util_$tct, which scans for "interesting" characters and returns a tally of characters skipped over, the indicator value for the character stopped at, and an updated pointer to the character stopped at. If the tally is non-zero, tty_read copies the skipped characters into whichever internal buffer does not contain the source string; then it examines the indicator. For a break character, it scans the copied characters (if any) from the right for the last printing graphic; the break character is copied immediately to the right of it. If any intervening white space was found, the length of the final string is decremented by the number of white-space characters. Finally, a flag is set to indicate that a break was found.

If the scan finds a form-feed, and the terminal has a non-zero page length, the form-feed is thrown away, on the assumption that the user typed it for the purpose of starting a new page. Otherwise it is stored as a normal character. The interrupt handler, dn355, is responsible for adjusting the current line count on the page when a form-feed or newline is input.

If the indicator shows an escape character, tty_read must find out if it is in fact the start of an escape sequence. If the channel is not in "esc" mode, or if the character immediately preceding or either of the two characters immediately following the escape character is a backspace, the escape is copied as a normal character and the scan continues. (The backspace test is to ensure that neither the escape nor the column position to its immediate right is overstruck.) If the following character is an escape, erase, or kill character, it is copied to the target string; if it is an octal digit, the character whose value is represented by the one to three non-overstruck octal digits following the escape character is inserted in the target string; if the escape is followed by zero or more white-space characters followed by a newline, all characters from the break through the newline are skipped (the newline is not treated as a break in this case); otherwise the character following the escape is looked up in the input_escapes string in the appropriate special_chars structure (described under "Data Structures" later in this document). If it is found, the corresponding character from the input_results string is inserted in the target string. If the character is not found, then there is no escape sequence, and the escape character is copied as above. If an escape sequence is identified, the pointer used for the next call to tty_util_$tct is updated to point past the end of the escape sequence.

If the indicator shows that the character is to be thrown away, it is not counted in the length of the final string, and the scan continues starting with the following character. Note that "invisible" characters (see above) have already been thrown away by the time this phase is reached. The present default tables do not include any other characters to be thrown away; however, a user-supplied table might specify some other character which the user wishes the typewriter DIM to discard rather than returning it to the user ring.

If the first call to tty_util_$tct returns an indicator of zero and uses up the entire source string, no characters at all are copied by this phase.

If the total number of characters in the now fully-converted string plus the number of previously-converted characters already copied into the caller's buffer is less than or equal to the number of characters requested by the caller, and the converted string ends in a break character, all the converted characters are copied into the caller's buffer, and tty_read returns. If the total number of converted characters exceeds the number requested by the caller, the caller's maximum is copied into the caller's buffer, and the remainder are placed in "converted" buffers in tty_buf as described above, to be picked up by a future call. If the total number of converted characters is less than the number requested by the caller, and the converted string does not end in a break character (either because a break character was escaped, or because the internal buffer size limit was reached), all available characters are copied to the caller's buffer and, if a read chain is still present, the next block of characters (up to the next break) is copied from the read chain and converted as above; any excess characters resulting from the latter conversion are saved in "converted" buffers as above.

DATA STRUCTURES

This section describes the modifications necessary to the data structures described in MTB 234 to make them useable for input conversion as well. Translation tables used by tty_util_$mvt and tty_util_$tct are similar to those used by tty_write, and, like them, are kept in ring zero by terminal type; future modifications will allow a user to specify his own version of one or more of these tables.

Default Table

The default table has been expanded and rearranged slightly, and the names of some of the items have been changed. The new format is shown below:

```
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 delay_offset (4) fixed bin (18),
      2 output_tct_offset fixed bin (18),
      2 output_mvt_offset fixed bin (18),
      2 special_offset fixed bin (18),
      2 input_tct_offset fixed bin (18),
      2 input_mvt_offset fixed bin (18),
      2 break_char char (1) unal,
      2 pad bit (27) unal;
```

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.

delay_offset — is an array of offsets of the delay_tables (described in MTB 234) to be used for this terminal type at 110, 150, 300, and 1200 bps respectively.

output_tct_offset — is the relative offset (in tty_ctl) of the default table used by tty_util_$find_char for identifying "special" characters during output processing.

output_mvt_offset
: is the relative offset of the table used by tty_util_$mvt for translation during output processing, 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.

input_tct_offset
: is the relative offset of the default table used by tty_util_$tct for identifying "special" characters during input processing.

input_mvt_offset
: is the relative offset of the table used by tty_util_$mvt for translation during input processing, or 0 if translation is not required for the particular terminal type.

break_char
: is the break character for this device.

Special Characters Table

The special characters table is as described in MTB 234, except that the following items have been added at the end of the structure:

```
2 input_escape_length fixed bin,
2 input_escapes char (1 refer (input_escape_length))
                                   unaligned,
2 input_results char (1 refer (input_escape_length))
                                   unaligned;
```

input_escape_length
: is the number of characters in each of the strings input_escapes and input_results.

input_escapes
: is a string of characters each of which forms an escape sequence when preceded

by an escape character.

input_results — is a string of characters each of which is to replace the escape sequence consisting of an escape character and the character occupying the corresponding position in input_escapes (above).

ADDITION TO MODULE DESCRIPTION OF tty_util_

Entry: tty_util_$tct

This entry uses a tct (test character and translate) instruction to search a given string for "interesting" characters in the same manner as tty_util_$find_char.

Usage

```
declare tty_util_$tct entry (ptr);

call tty_util_$tct (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;
```

All members of the structure have the same meaning as for tty_util_$find_char, except for the following:

stringp — is a pointer to the string to be tested; it is updated to point to the first "interesting" character in the string.

(Input/Output)

indicator — is the result of the search. It may have the following values: (Output)

0 -- no special characters

1 -- break character

2 -- escape character

3 -- character to be thrown away

MODULE DESCRIPTION OF tty_canon

Name: tty_canon

This subroutine is used to reduce a character string (which is expected to consist of one typed line image) to canonical form, i. e., sort the characters by column position and by ASCII value within each column position.

Usage

```
declare tty_canon entry (ptr, fixed bin (24), fixed bin (24),
                    char (1) aligned, char (1) aligned,
                    fixed bin (35));

call tty_canon (string_ptr, length, max_length,
                    erase_char, kill_char, code);
```

string_ptr — is a pointer to the string to be reduced; the result string replaces the input string. (Input)

length — is the length of the string. It is adjusted to reflect the length of the result string. (Input/Output)

max_length — is the maximum allowable length of the result string. (Input)

erase_char — is the character which is to be interpreted as an erase character, or blank if no erase processing is to be done. (Input)

kill_char
: is the character which is to be interpreted as a kill character, or blank if no kill character is to be recognized. (Input)

code
: is a standard system status code. If the canonicalization of the string requires a result string whose length exceeds max_length, code is set to error_table_$long_record; otherwise it is set to zero. (Output)