MTB- 418

Date: 7/11/79

To: Distribution

From: Bernard S. Greenberg

Subject: The Echo Negotiation Papers

This MTB consists of the two design documents for the "Echo Negotiation" schemes, Ring Zero and FNP. Echo negotiation is a general scheme for allowing synchronization of output with echoing of input in a multilevel communications processing hierarchy. Such synchronization is necessary in a managed-video environment, for echoing of input is conditional upon processing of previous input, and must be performed at places on the screen determined by output from previous input.

The need for these schemes in Multics is motivated by the need for performance amelioration of the experimental Emacs editor, which maintains such a managed-video environment, and has need for such synchronization. In its first implementation (March, 1978), Multics Emacs achieved this synchronization by receiving all characters in ring 4 as they were typed. This was inordinately expensive, requiring a wakeup and process loading, as well as the running of a user process and multiple calls through the TTY Dim, for each character typed.

In January, 1979, the scheme described by the first of the two papers contained herein was implemented, shifting the principal responsibility for echoing echoable typed input from the ring 4 code (Emacs) to the ring zero interrupt side of the Multics TTY software. This reduced the number of wakeups and loadings for the inputting of text from one per character (50 or so per line) to two per line. This implementation was shipped as part of Multics Release 7.0a.

As some experience with this scheme was gained, it was predicted (given 6 milliseconds, worst case, for MCS interrupts to the mainframe) that 100 Emacs users would entirely saturate one CPU with interrupts (assuming text type-in at 5 characters a second). This appeared to be the most immediate and most crushing bottleneck in Emacs performance. Thus, a scheme was devised to eliminate per-character mainframe transactions where possible, by having the Front End Processor (FNP, FEP, 355, 66/32, 66/78, etc.) echo characters for typed input. Correct implementation of this facility requires synchronization protocols to carefully control the circumstances under which echoing is performed: the circumstances under which it starts, and stops. A now-operative (July, 1979) implementation of this idea is described in the second paper.

As implemented now, a four level hierarchy of echoing software exists:

I	Emacs Lisp Code
II	Emacs MCS interface (e_pl1_)
III	Ring Zero MCS (TTY Dim)
IV	FNP (potentially, other "multiplexers")

The top level (the Emacs editor) asks for and receives non-echoed characters from the lower (higher-numbered) levels, until it enters the "negotiated echo state", by being at "rest" at the end of a line (and certain other conditions met: see the first paper). At this point, Level I calls level II, passing as a parameter the length of the line left on the screen to the right of the cursor. Ultimately, Level II returns to Level I a collection of N characters, the first E (E possibly zero) of which were echoed.

Level I does not care which level echoed the E characters, all it cares is that they were (or perhaps were not echoed). Level I will process all the returned characters that were not indicated as echoed, processing them as Emacs "commands", including echoing those that should be echoed but were not indicated as already echoed.

The interface between Level II and Level III, and between Level II and Level IV, is virtually identical. Each returns what characters it has buffered if it has characters buffered, telling how many were known to have been echoed, and in this case calling no lower level. If a level has no characters, the next lower level is called, passing the screen-length left. That lower level returns characters, some of which it indicates as having been echoed, and the rest not. The upper level considers the lower level's echoage as part of its own, and processes the remainder to attempt to echo the first E' characters after the returned echoed string. In this way, any lower level can "punt" echoing for arbitrary or internal reasons, being assured that echoable characters will be echoed by some level, and optimally, one as far down (and thus, efficient) as possible.

The "multilevel echo hierarchy" also provides for lower levels that do not support this echo protocol: the upper level will simply echo, as the lower level has returned no echoed characters. Similarly, lower levels yet (i.e., programmably-echoing terminals or concentrators) can be added in an integrated and consistent fashion.

Levels also ship "break tables" down the line to each other. It can be shown that these tables can only be requested to be shipped when the entire system is in the non-echoing state.

The current FNP echoing is not more consumptive of FNP time than echoplex: no number of asynchronous lines running echoplex has ever caused a shortage of FNP time; thus, there

appears to be little motivation (for the FNP, at least) of devising schemes to have "echo-negotiating terminals". Attempting to do so even introduces new problems (see the second paper). For common-carrier net (e.g., TYMNET) access to FNP channels, this is still an interesting problem.

(END)

Interrupt-time Echo Negotiation

BSG 1/21/79
revised 1/27/79
ESC-Q'ed to ^XF = 72 6/30/79

Interrupt-time echo negotiation is a proposal to modify the interrupt side of ring-0 MCS to conditionally "echo", or send back to the user terminal, characters input from the keyboard. The goal is performance improvement of Multics Emacs (or any other conceivable real-time editor implementation on Multics), and increasing the apparent responsiveness thereof many fold. Complete transparency to the Emacs user as well as the Emacs extension-writer is a design constraint.

Programs which attempt to manage a display terminal intelligently cannot allow the terminal itself to unconditionally print characters as they are typed. As long as type-ahead is allowed, characters must be placed on the screen at positions determined by the central system's interpretation of previous characters. Until all previous characters have been processed by the central system, and the terminal's cursor repositioned appropriately, characters may not be typed out as they are typed in ("echoed"), whether it be by the central system, the FNP, or the terminal itself. Thus, the use of "local copy" (unconditional terminal echoing) is ruled out in a display-managed environment.

The current Emacs implementation receives each character from the terminal via calls to the Ring 0 tty dim; several may be read at once. If Emacs was blocked waiting for input, only one wakeup will be sent. If Emacs was not blocked, no wakeups will be sent (this is an old feature of the tty dim). If Emacs can process characters as fast as they are typed, one wakeup, one process loading, one read (get_chars) call on ring zero, and one write (put_chars) call on ring zero will be made for each character. This shows up as a tremendous CPU and paging consumption in the user ring (and user-initiated calls). If Emacs falls behind, fewer calls are made and more characters are read and written per each call to ring zero, and no wakeups occur until Emacs actually catches up. Strangely enough, it is thus much more efficient when it falls behind.

The case of "just typing in text" is, statistically, a very important one, one of the most frustrating when the system is slow, the greatest cause of expense (largest number of characters per second) and the one most amenable to optimization. If no typing errors are made, it is the case where Emacs is doing the least for the user for the greatest expense. But when typing errors are made, Emacs excels here. "Just typing in text" means adding "trivial" printing characters (i.e., "a", "$", "/", " ", but NOT backspace, rubout, control characters, carriage return, etc.) at the end of a line. Included in this case is the typing of "minibuffers" (responses to Emacs-generated prompts). This case is what is dealt with by "input mode" in QEDX or EDM. It is one of the things that Emacs does best, but one of the most expensive.

The proposal is to optimize the "just typing in text", known in Emacs jargon as "EOL (end of line) self-insert" by having the

nterrupt side of the tty dim echo characters when Emacs deems it
ppropriate. The effect of this will be that, once Emacs is caught up,
haracters will be echoed (and thus appear on the screen) as soon as
hey are typed. Emacs will not be woken up (or loaded, or run) until
ome "break condition" occurs, as defined below. The response time for
OL self-insert echo will only be a function of interrupt response
milliseconds even on a heavily loaded system) instead of
rocess-loading response (which can vary from milliseconds to seconds).

A break condition, in this context, means a situation where
ing 0 can no longer echo characters. Such break conditions are:

1) Enough characters have been echoed to fill the screen (terminal) line. We cannot know where to put the next character; although it may be simply at the start of the next line (this is an avenue for more future optimization), a complex redisplay involving moving the entire screen around may need to be done before the \c and continuation line can be displayed.

2) A non-echoable character is received. The ring zero dim will be supplied with a bit-table by Emacs of echoable characters. All control characters are non-echoable. They are editor commands, not characters to be printed on the screen. Ditto #, @, etc. Similarly, tabs and backspaces require special redisplay action.

3) The interrupt side can no longer echo, due to inability to allocate a buffer bound for the FNP, or other internal problem.

4) The person at the terminal hits QUIT (which also causes a resetread).

When a break condition occurs, Emacs (if blocked) will be
oken up, and echoing will cease. Characters will continue to
ccumulate in the normal breakall-mode fashion until Emacs calls in and
eads them. A return parameter of the read call is the count of echoed
haracters; the total count of input characters has always been a
eturn parameter.

Emacs will process all characters read, as usual, inserting
trivial" characters in its buffer or minibuffer as it has always done.
owever, the echoing of "trivial" characters processed as EOL
elf-inserts by ring zero will be suppressed.

This implementation is believed to be race-free. Some of the
deas in it are due to J. Spencer Love and Jerry Stern. Earl Killian
nd Dave Moon contributed a few improvements, too.

A new entry point to the ring zero tty dim will be defined,
cs_$echo_negotiate_get_chars (see the include file at the end of this
ull). It is like the normal get_chars entry of the ring 0 dim
hcs_$tty_read), plus extra. Before describing the differences, it is
ritical to note some features of both these entrypoints:

1) The entrypoints are called with a device index, and a buffer and its size, and read characters into the buffer. The count of returned characters is returned.

2) If no characters are returned, a flag is set in ring zero, such that a wakeup will be sent when characters "are available". If characters are returned, this flag will not be set, and thus no wakeup will be sent, even when characters become "available", unless another call is made in the interim which finds no characters available.

   For normal tty_dim get_line calls, "available" means a complete line has arrived in ring zero. For get_chars calls in breakall mode, and echo_negotiate_get_chars calls in breakall mode, this means any characters having arrived (from the FNP) in ring zero at all.

   There is no way the input side of the tty dim ever sends wakeups (other than QUIT) other than this mechanism. The user ring is allowed to go blocked (i.e., expect a wakeup), if and only if one of these calls returned no characters.

The new entrypoint will, if no characters are available, return this fact, set this "wakeup flag", and set a bit in the wired terminal control block (WTCB) placing the terminal in "ring zero echo state". This means that ring zero's interrupt side will echo characters until a break condition occurs. This call-side operation will be locked against the interrupt side via extant tty dim mechanisms. Thus, the determination that no characters are available is a unitary operation.

If characters are available, all available characters will be returned. A count of characters that have been echoed before the break condition was encountered will be returned: call that count <n>. The wakeup-flag will not be set. The dim will not go into ring zero echo state. Emacs will be expected to process the characters, generating any redisplay necessary. The first <n> of these characters have already been echoed by ring zero; Emacs is not to echo them. Emacs is expected to call back when these characters have been processed, and all redisplay output from the the interpretation of these characters queued to the FNP in ring zero (via having made sufficient calls to output all redisplay output). At this time the terminal is NOT in the ring zero echo state; characters are not being echoed by ring zero.

When Emacs calls back, it will either be via the normal get_chars or the echo_negotiate_get_chars entries. If characters are available when Emacs calls back, thus, <n> will be zero for the new call, and Emacs is expected to echo all characters as necessary. Only when Emacs "catches up" will the new call enable the ring zero echo state.

Emacs will call the new entrypoint only when it describes itself as being "at the end of a line", i.e., amenable to EOL

elf-insert echo. Otherwise, the regular get_chars (hcs_$tty_read) ntry will be called. Thus, inserting text in the middle of a line which may involve "hard" redisplay) will not benefit by this ptimization. Similarly, when ring zero echoing has been stopped due o typing a "prefix character" (say, ESCAPE or \), the next call will e to the regular get_chars (the motivation here is that ESC-A is an ditor command, the A should not be echoed). This simply falls out of he implementation as so far described. Thus, only when Emacs states I am at top-level command-level at the end of a line" is ring zero choing enabled.

If the user can type characters as fast or faster than Emacs an process them, all echoing will be done by Emacs in ring four. By ypothesis, Emacs will never go blocked, and thus never be woken up, or eed to be loaded. It will be processing characters in batch mode, eading and outputting many at a time. If the system is so heavily oaded that this is the case, that we cannot even get the sheer cpu ime to process these characters, we are in deep trouble, which is to ay, no worse than today. Unless this is the case, by hypothesis, macs will eventually catch up. Once it is caught up, ring zero will cho characters until a break condition occurs, and then we are waiting n Emacs to be loaded, process the characters, and ultimately revert ing zero echoing. Compare this to today where typing ANY character ill wake up and load Emacs to echo it; the gain should be clear. I am leased and proud that even under these adverse circumstances today, macs responds elegantly and rapidly most of the time.

A new order to the ring zero tty dim will be provided to upply a table of which characters will cause a break condition. This able will be set in wired ring zero storage by explicit call any time macs changes character bindings such that the elf-insert/non-self-insert status of characters changes. This call ill be made right before the next call to echo_negotiate_get_chars, nce all such binding changes have been determined. There is no race ere; this table can only affect characters that have not yet been rocessed by the ring zero interrupt side. Characters already in ring ero, processed, will be returned. None of them could have been echoed perhaps falsely) by ring zero, because the previous call to ring zero ust have returned characters. This must be the case because Emacs did omething other than a self-insert, namely, something which involved at he LEAST switching some character bindings. Thus, the previous call o ring zero could not have been a zero-character-returning cho_negotiate_get_chars call, because it returned the commands that aused Emacs to "take non-trivial action". Thus, ring zero cannot, at he time of the table-set call, be echoing. Characters not yet typed, r queued on the channel lock in ring zero, in the FNP, or in transit o the Multics interrupt side are equivalent; they will be echoed ccording to the new table.

The info structure for the new order includes information for uture implementation of rubouts (delete the last character from the creen, and ultimately the Emacs buffer) by ring zero. Thus, the most ommon typing correction will be processed without waking up ring four. he preferred sequence for removing the last character from the screen,

which must be a self-inserting printable character (viz., not TAB), will be supplied by ring four, along with a count of necessary pad characters for this operation. On some terminals, this might be the sequence for "backspace, space, backspace", but on more sophisticated terminals, "backspace, kill-to-end-of-line" is preferred, which is a terminal-dependent sequence. A flag is provided as to whether characters deleted by the interrupt side in this fashion should simply be removed from the input buffer transparently or placed in the buffer with the "rubout" character; this latter alternative allows the ring four program to display to the user the "last 50 characters typed", often a useful ability. Two rubout characters are allowed, normally supplied by ring four. Pound-sign (#) and ASCII rubout would be the usual choice, but this is completely dependent on user action setting key-bindings, etc.

Note that the ring zero interrupt side, if doing "rubout" processing, can never delete a TAB or any (other) character it itself did not echo. If the dim is not in the echoing state upon the receipt of a rubout, or all accumulated, echoed characters have been rubbed out, the dim must wake up Emacs and suspend echoing, if on.

Any call to the normal get_chars entry disables ring zero echo (until the next call the echo_negotiate_get_chars that enables it). Within Emacs, as shown above, it should never be the case that it should need to. Yet, unexpected Multics action (i.e., a fault) causing ultimate return to command level, should turn it off. The first get_chars or get_line call that is made will thus do this. There is also an esoteric case involving the Emacs interrupt system (!), which is used for console messages and mode-line time displays, etc., which requires explicit disabling of ring zero echo, as cursor position will be changed even though no characters other than EOL self-inserts were entered from the keyboard. This problem is fairly hirsute. For detail on it, see

```
>udd>m>bsg>r0e.inter.text (MIT-Multics)
AI:DLW2;BSGR0I >    (MIT-AI ITS)
```

The scheme above has been implemented and tested (other than the interrupt interaction), and works as expected. It has not yet been metered.

There is a negative effect of moving any echoing out of the body of Emacs: by virtue of the same performance improvements that reduce the mean page residency time, the likelihood of the needed pages of Emacs being in main memory when needed, i.e., echoing stops, is reduced. Although this reduces the load on the system, and the cost to the user, the response time degradation when the bulk of the editor has to be paged in became quite noticeable the last time a paging/CPU performance improvement was implemented for EOL echoing, and this effect will surely be significant here too. On a very lightly loaded system, or hopefully, one where many processes are using Emacs, this effect will hopefully be ameliorated.

It has been suggested that inserting characters into the middle a line could also be optimized, placing a terminal in "insert mode" while ring zero is echoing. This is only useful, however, for terminals that support this feature, and requires management of the terminal's firmware state. Although common, this operation is nowhere as statistically significant as EOL echo. No such facility is planned for the near future.

One is tempted to ask as to why this echoing is not best done in the FNP; in truth, it would be much more efficient to do it in the FNP. Yet, the critical timing races which are solved by interrupt side versus call side locking in the above implementation are not amenable to such solution were the FNP doing the echoing. In fact, it is extremely non-trivial to devise a set of protocols such that all this works in that case. Lee Parks has proposed a set of protocols similar to the window managment and packet numbering of the CHAOS net which seem to offer some hope; Jerry Stern has devised similar, and these are under investigation, but implementation does not look imminent. If this sort of thing can be made to work, the door is open to pushing echo further down the line, to customized terminals. However, I cannot imagine that given any reasonable amount of FNP power, it would be necessary to resort to this for optimization of EOL self-insert redisplay. Moving editing functions involves extreme levels of complexity which dwarf any of the issues raised above.

But that's another ball game.

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

```
/* BEGIN INCLUDE FILE mcs_echo_neg.incl.pl1  Bernard Greenberg 1/20/79 */

/* This include file defines the callable entrypoints and argument data
   structures for ring 0 echo negotiation */

dcl  echo_neg_datap ptr;
dcl  echo_neg_data_version_1 fixed bin static options (constant) init (1);

dcl 1 echo_neg_data based (echng_datap) aligned, /* Echo negotiation data */
    2 version fixed bin,
    2 break (0: 127) bit (1) unaligned,          /* Break table, 1 = break */
    2 rubout_trigger_chars (2) unal,/* Characters that cause rubout action */
      3 char char (1) unaligned,
    2 rubout_sequence_length fixed bin (4) unsigned unaligned,
    2 rubout_pad_count fixed bin (4) unsigned unaligned,
    2 buffer_rubouts bit (1) unaligned,
    2 rubout_sequence char (12) unaligned; /* Actual rubout sequence */

dcl  hcs_$echo_negotiate_get_chars
     entry (fixed bin (35), ptr, fixed bin, fixed bin, fixed bin,
     fixed bin, fixed bin, fixed bin, fixed bin (35));
/*
```

```
call  hcs_$echo_negotiate_get_chars
      (devx, datap, nelemt, offset, NRETURNED, NECHOED_RETURNED, screen_le
           STATE, CODE);

   END INCLUDE FILE mcs_echo_neg.incl.pl1 */
```

(END)

FNP Echo Negotiation -- 6/30/79, 7/11/79 -BSG
(>udd>m>bsg>fnp-echnego.text, MIT-MULTICS/System M)
(AI:DLW2;BSGFNE >, AI. ITS)

This paper describes the scheme devised by Jerry Stern and myself to move the responsibility for "trivial Emacs echo" into the FNP. Lee S. Parks (MIT at the time, Lawrence Livermore Labs now) and Earl A. Killian (Bolt, Beranek, and Newman) also contributed ideas related and leading up to the current proposal.

The goal of this scheme is to eliminate the need for a Multics interrupt for every character received when the only action of Multics would be to accumulate and echo (send back to the terminal) that character. (Such "conditional negotiated echoing" is necessary in any managed-video environment, it is not something Emacs-specific). This echoing is currently performed by the interrupt side of the ring zero teletype dim (MCS, Multics Communications System) when feasible. That scheme, now part of Multics (as of Release 7.0), is described in

>udd>m>bsg>r0-echnego.text
(MIT-Multics and System M, Phoenix)
DLW2;BSGR0E >
(MIT-AI PDP-10)

That document is prerequisite reading for comprehending this paper. The present scheme is built upon that scheme, is an integrated extension to it, and cooperates with it.

The current scheme fits within the architecture of "Ring Zero Demultiplexing", which is a scheme put forth in Release 7.0 whereby a hierarchy of software levels called "multiplexers" are interposed between the user's physical terminal and the common code of MCS. For example, the code which manages the communications processor is a multiplexer (known as "fnp_multiplexer"), which manages most TTY channels. Ring Zero MCS does not call fnp_multiplexer directly, but rather via a standard set of interfaces (channel_manager), specifying the device-index of the terminal under consideration. Channel_manager realizes that fnp_multiplexer is responsible for that terminal, and dispatches to the latter. Similarly, when fnp_multiplexer processes an interrupt from the communications processor, it determines which channel is involved, and calls the channel manager interrupt entry to call the interrupt side of ring zero MCS (tty_interrupt) to do "TTY interrupt-side processing", specifying one of a standard repertoire of "interrupt types" along (perhaps) with data received from the front-end. Fnp_multiplexer may be thought of as a procedure multiplexing and de-multiplexing the single interface to the communications processor.

The point of this "Multiplexer" protocol is that a device such as a terminal concentrator may be attached to Multics via one channel of the Front-End (or even some other path into Multics), and have its demultiplexing/routing mechanism managed automatically as part of this

scheme. If such a terminal concentrator exists, requests made by ring-zero MCS to control (e.g., output to) terminals on it will be forwarded by channel_manager to the concentrator-manager procedure, which will perform the necessary encodings as required by the concentrator's protocol, and call channel_manager recursively to pass the new data on to fnp_multiplexer to control the actual channel. Similarly, interrupts received by fnp_multiplexer for the concentrator's channel will be forwarded by channel_manager not to tty_interrupt, but to the concentrator-manager procedure. The latter will interpret the received data, and call channel_manager to invoke tty_interrupt for the appropriate terminal.

It is via this means that arbitrarily-nested multiplexed communications channels may be supported, as long as the multiplexing/demultiplexing procedure exists within the supervisor. The set of calls from Ring Zero MCS, or lower-level multiplexers, to each "multiplexer" consists of calls to read and write data, and peform a well-defined set of "control orders" upon the channel. Each multiplexer has the right to perform or refuse to recognize any of these control orders, indicating which via a status code. The set of calls from the interrupt side of the multiplexer back to superior multiplexers, or the interrupt side of Ring-Zero MCS, consists of the set of "interrupts", some with data, some without.

It is within this architecture that the FNP Echo negotiation (or more accurately, Echo negotiation by multiplexers) is designed. It consists of a new repertoire of control orders to multiplexers and interrupts from multiplexers, via which echoing by the multiplexer is managed in an organized way.

In this scheme, a multiplexer either "knows how" to perform negotiated echo, or does not. Supporting negotiated echo means possessing the ability to accept a break-character table, and echo characters according to it, until one of the "echo negotiation break conditions" described in the previous paper are encountered, in response to the new control orders. The FNP (via fnp_multiplexer) knows how to echo (which we will take to mean "negotiated echo via this protocol"). This specifically does not mean that multiplexed channels that happen to go through the FNP "inherit" this ability: they do not. For channels whose multiplexers cannot support this protocol, ring zero MCS is prepared to perform the echoing itself as it does today. As it turns out, ring zero must be prepared to perform this echoing even for multiplexers that do support the protocol.

The multiplexer which supports negotiated echo will, when so requested, echo characters as they arrive from the terminal and not hand them to MCS until one of the echo negotiation break conditions is recognized by it. Inception of echoing by the multiplexer is started by a call from MCS, when MCS is told to begin echoing, and other conditions are ripe. Input characters are handed by multiplexers to MCS via the "ACCEPT_INPUT" interrupt ("interrupt" as defined above in the multiplexer/MCS layering). In addition to the input characters,

the ACCEPT_INPUT interrupt conveys (today) a bit, "break_character" which states whether or not a "break character" (normally CR, LF, etc.) was placed in the input by the multiplexer. In "breakall mode" (character-at-a-time processing), every such character delivery today has this bit "on". Under the new protocol, for a multiplexer which supports negotiated echo, this bit is used to indicate whether the multiplexer itself echoed the characters in this delivery. If the bit is on, the delivery consists of characters none of which were echoed by the multiplexer; the multiplexer may decide for any reason (e.g., internal buffer shortages, internal races, etc.) to stop echoing (as can ring zero vis-a-vis ring 4). Of course, it must stop echoing for the defined echo negotiation break conditions. If the "break_character" bit is off, the delivery consists of characters all of which were echoed by the multiplexer, except for perhaps the last character of the delivery. MCS must determine, for such a delivery, whether the last character of such a delivery was capable of being echoed by the multiplexer, and if so, assume that it was, otherwise not. This "wart" makes it unnecessary to make two distinct deliveries (to distinguish the echoed from non-echoed characters) every time a break condition is encountered.

When Ring Zero MCS receives an echo-break table from ring 4, it will attempt to ship the break table to the Multiplexer via a new control order ("set_echnego_break_table"). By virtue of the way negotiated echo (in general) works, MCS (Ring Zero or otherwise) cannot be in the echoing state when this table is set. Thus, both Ring Zero and the multiplexer (e.g., the FNP) will have an appropriate (and identical) break table when the the call to begin negotiated echo is made.

When Ring Zero MCS is called upon to begin negotiated echo (echo_negotiate_get_chars, as in the previous paper), and has no characters to deliver immediately (already accumulated), it calls upon the inferior multiplexer to "start negotiated echo", via a control order, also specifying the number of characters left on the line, as in the protocol of the previous paper. If this control order is refused (the multiplexer does not support the new protocol (of course, fnp_multiplexer in specific, does)), ring zero proceeds as today, with interrupt-side negotiated echo. If the multiplexer goes along with the order, the channel is marked (in the ring-zero echo data) as having a multiplexer knowledgeable about the protocol. In either case, ring zero will enter the "ring zero echo state" (of the previous paper).

After this operation has been performed, the multiplexer (e.g., the FNP) will either be echoing characters or not: by virtue of the synchronization protocol (to be described), the multiplexer will either have honored the request or not. MCS will not know whether the request has been honored or not until characters arrive from the multiplexer at the MCS interrupt side.

If the multiplexer delivers up a shipment of characters with the "break_character" bit on, either the request was not honored, or the specific multiplexer does not support echo negotiation, the multiplexer decided randomly (i.e., for internal reasons) to stop or not start

echoing, or the first character in the delivery is a break character or exceeds the length of screen left: in any case, the delivery is entirely of non-echoed characters. These cases are indistinguishable from each other and from the only case today when in the ring zero echo state, and handled identically as today. The delivery is scanned, a possible leading prefix of echoable characters echoed by ring zero, and the characters made available (if ring zero leaves the ring zero echo state while processing them) to ring 4, which would then be woken up.

If the multiplexer delivers up a non-empty shipment of characters without the break_character bit on, and the specific multiplexer has been found to be knowledgeable about multiplexer echo negotiation (i.e., earlier accepted the "start negotiated echo" control order) all characters except the last in the shipment are known to have been echoed by the multiplexer, which has apparently honored the order call which was issued at the time ring zero went into the echoing state. Thus, this only possible if ring zero is in the echoing state. Those characters are counted by ring zero as "having been echoed by ring zero" (as far is ring four is concerned) and are not echoed by ring zero. The last character is checked for stopping ring-zero echo, and if it would not stop ring zero echo, is treated as one of the multiplexer-echoed characters. If it would stop ring zero echo, it is processed as today, indeed takes ring zero out of the echo state, and causes ring 4 to be woken up, as today.

Whenever the multiplexer delivers to the Ring Zero MCS interrupt side a character that takes ring zero out of the echo state, the multiplexer itself will be known to have stopped echoing. Thus, ring zero leaves the echoing state after the multiplexer stops echoing. Note also that ring zero enters the echoing state upon receipt of the echo_negotiate_get_chars call, which is before the multiplexer enters its echoing state (upon processing the receipt of the control order sent at the time ring zero enters the echoing state. Thus, the multiplexer's echoing-state interval is completely and properly contained within that of ring zero.

There is an implicit race condition associated with the start of Multiplexer echoing: when ring zero enters the echoing state and sends the order to the multiplexer to begin echoing characters, ring zero (or ring 4, for that matter) does not know whether or not the multiplexer has processed some input characters, and sent them on their way, which have not yet arrived in ring zero. This is not unlike the case today where characters have been received by the multiplexer and not yet handed to ring zero when ring zero enters the echo state. Today, these characters will eventually be processed (they will be the next characters to be processed) and no harm is done. Yet, when the multiplexer has the ability to echo, the order to start echoing may be invalid when received by the multiplexer if the latter has processed characters not yet seen by ring zero. If these characters are echoable, the ring zero interrupt side would attempt to echo them (which would be in the wrong order with echoing being done by the multiplexer). If they are not echoable, the order for the multiplexer to start echoing is in error, for the unprocessed characters may

produce output (when processed by ring 4) which must precede any echoing done by the multiplexer (e.g., cursor positioning, buffer switching, or any arbitrarily complicated request with arbitrarily complicated effects on the screen.

Thus, the multiplexer must not start echoing (i.e., not honor the order to start echoing) if it has processed (but not had received on the ring zero interrupt side) characters at the time the start_negotiated_echo order is received at the point which processes characters. This determination is made by the "input processor" of the multiplexer based upon a value called the synchronization counter sent with the start_negotiated_echo control order: the value sent by ring zero is the count of all characters received by the ring zero interrupt side since the last character echoed by the multiplexer. The multiplexer input processor will also count characters processed by it since it last echoed a character. Since the start_negotiated_echo control order can only be received when the multiplexer is not in the echoing state, any multiplexer-but-not-ring-zero processed characters could not have been echoed by the multiplexer. Thus, the equality of the multiplexer input processor's count and the count received from ring zero in the start_negotiated_echo control order is necessary and sufficient reason for the multiplexer to enter the echoing state, and begin echoing characters. If the counts are not equal, it must be the case that non-echoed characters are "in transit", and the order must not be honored. Ring zero's shipped count can only be less than the multiplexer's, or equal: it can never be greater.

The above protocol has the effect of multiplexers starting echo only when there are no characters "in transit" between the input processor and the ring zero interrupt side. This would have the effect of each ring 4 echo_negotiate_get_chars being entirely echoed by ring zero or the multiplexer, as there be characters in transit or not at the time of the receipt of the start_negotiated_echo control order by the multiplexer's input processor. We would like the multiplexer to perform as much echoing as possible, even if the start of this echo_negotiate_get_chars lost the race condition. Thus, when ring zero processes (and echoes) an entire delivery of echoable characters from a multiplexer which claims to know how to perform negotiated echo, it will (after having so done) send out another start_negotiated_echo control order, which again, may or may not succeed when it reaches the input processor.

There is a fairly subtle problem here when long communication lines or packet-switched networks present a time delay between the Multics Central System and the echo-negotiating processor that is longer than the inter-character typing time. If the typist continues to type faster than the communication line/system can send characters and receive the control orders, there will always be characters "in the pipe", and not only will all the start_negotiated_echo control orders fail, but more interrupts/communications than today will occur due to the extra control order at every such character! Thus, until the typist scratches his or her head, etc., the system can never synchronize, and will generate extra overhead! With the FNP doing echo-negotation, this should not be a problem, as the round-trip

response time of the network from the FNP through ring zero and out again to the FNP is substantially less than an input character-time. (Note that line speed is completely irrelevant: only the inter-character time of the typist is an issue).

Another thorny point of the present scheme is the initialization of the synchronization counters: If they count characters processed since the last multiplexer-echoed character, what value are they to have if the multiplexer has never echoed any characters? It will never be able to start echoing, since successful inception of echoing depends upon the values of the synchronization counters. This is solved by another new control order, "init_echo_negotiation", which must be supported by all multiplexers supporting echo negotiation. Ring Zero MCS will maintain, for each channel which has ever dealt with echo negotiation, a "synchronized" flag, initially off. As long as this flag is off, received characters are not counted against the synchronization counter. When a call to echo_negotiate_get_chars is made upon ring zero, and there are no undelivered (to ring 4) characters in ring zero, the "synchronized" flag will be inspected. If on, action will proceed as above. If off, new control order will be issued to the multiplexer. If not honored, ring zero enters the echoing state as usual, knowing that the multiplexer will not ever echo. If honored, the echoing state is not entered. Characters will arrive, non-echoed, from the multiplexer, and cause ring 4 wakeups. At the time the multiplexer's input processor (which maintains the multiplexer's synchronization counter) receives the init_echo_negotiation control order, it zeroes its synchronization counter, begins counting characters (it must be in the non-echoing state) thereafter, and sends a new type of interrupt to Ring Zero MCS, ACK_ECHNEGO_START. When ring zero receives this interrupt, it zeroes its synchronization counter, turns on the "synchronized" bit, and begins counting received non-echoed characters. Characters are currently not being echoed: they are being delivered to ring 4 unechoed as they arrive. The first character received in ring 4 will cause ring 4 to promptly make another echo_negotiate_get_chars call (as ring 4 is still in its echoing state), and all proceeds as above, when synchronized.

One more service needed of MCS echo negotiation is the ability to stop MCS echoing on demand of ring 4, when the latter receives some non-keyboard-initiated event (e.g., a console message) which will affect the display. When such an event occurs, ring 4 will call into ring zero to stop echoing, and make calls to determine precisely what has been echoed, i.e., what is on the screen, with the certain knowledge that MCS will echo no more until instructed to do so. Today, this is done (idea due to Spencer Love) by an echo_negotiate_get_chars call with zero "screen length left". Ring zero, when so called, will return what it has received, indicate how much it has echoed, and stop echoing in a unitary operation. The zero screen length left will prevent it from re-entering the echoing state (or staying in it), which it would otherwise do. The complex scheme described in r0-inter.text (BSGROI >) was never implemented, and was discarded in light of this superior idea.

When multiplexer echoing is involved, not only ring zero, but the multiplexer, must leave the echoing state when such a call is made. Although ring zero could leave the echoing state instantly when such a call is received, ring 4 cannot proceed knowing that echoing has stopped until there is positive verification that the multiplexer has stopped echoing. Thus, the following protocol has been devised: At the time of an asynchronous ring 4 event (Emacs interrupt), Ring 4 will be prepared to try and retry the call to call hcs_$echo_negotiate_get_chars with a screen-length-left in a loop, until the entry indicatest that echoing has indeed stopped. The entry will return a zero code and actually return characters if echoing has stopped, or return error_table_$line_status_pending if it has not, in which case ring 4 is expected to block. In this way will ring 4 wait for the multiplexer to acknowledge stopping echoing. Upon receipt of such a call, ring zero will, if in the echoing state, and has not already done so, send out a new control order to the multiplexer, "stop_negotiated_echo", and set a bit saying that it has done so. If called in this way when in the echoing state, and the order call is supported, ring zero will return error_table_$line_status_pending. This code will be returned for each such call until the echoing state is exited as is to be described. When the multiplexer input processor (which does the actual echoing) receives this control order, it will stop echoing (leave its echoing state), having echoed and forwarded all it is going to echo) if in its echoing state, and send another new type of interrupt, ACK_ECHNEGO_STOP, to the MCS interrupt side. The MCS interrupt side will receive this after it has processed all characters processed by the multiplexer input processor before the latter received the control order, and thus, ring zero knows at this time exactly how much was echoed and that no more will be echoed. At this time, the MCS interrupt side takes ring zero out of the echoing state, turns of the bit saying that the control order was issued, and wakes up ring 4. Ring 4 wakes up, retries the call, receives a zero error code, and real characters (including a count of how many were echoed), and echoing does not restart, because of the zero screen-left passed in as input on the last (indeed each) of these calls.

One further point needs be mentioned. A Multiplexer may find that it cannot honor a start_negotiated_echo control order because it has output which it has already accepted from Ring Zero MCS queued internally. fnp_multiplexer, for instance, cannot send the control order to the FNP until the internally queued output is taken by the FNP, lest echoed output, echoed by the FNP, appear in the wrong sequence with respect to Multics Central System output. The multiplexer will, in this case, return the code error_table_$invalid_write to Ring Zero MCS, which will set a special flag bit indicating that when the multiplexer asks for more output (via the SEND_OUTPUT interrupt), another attempt is to be made to send the control order to the multiplexer before any more output is given to it. Such a "failure" is not considered to be a case of the multiplexer rejecting (or not honoring) the control order. This scenario will almost certainly occur in the case mentioned above where a second attempt at synchronization for a given echo negotiation causes a start_negotatiated_echo control order to be sent after ring zero has echoed characters following a muliplexer synchronization failure.

(END)

(END)