# Thoughts on Small Systems and Monitors

# (mostly the SYM-1)

**BY H. T. GORDON**
**College of Natural Resources**
**University of California**
**Berkeley CA 94720**

I define a "small" system as one that is owned and used by only one person. This still allows more than a ten-fold range in cost and complexity, but the common ground is the man/machine interaction that can be (for the human being) a profound educational experience. My first system was a KIM-1, the creation of Don McLaughlin while he was at MOS Technology. This early SBC launched the new 6502 (in the dim and distant past of 1976!) and, together with brilliant descriptive manuals, played a major role in its popularization. It was—belatedly—emulated by the makers of other microprocessors, and of course surpassed by more complex systems of the multi-board type. As a learning tool it still has few (perhaps no) equals. All of my previous communications, in *DDJ* and elsewhere, were worked out on my KIM.

One of its invaluable assets is the complete and quite well-commented listing of the code of its 2K ROM monitor, a simple "operating system", rich in information on how to program. Like other users I soon became aware that many of its routines were callable by new programs, although this had not been clearly foreseen by whoever wrote the monitor programs. By now, this ought to be recognized as one of the major functions of every monitor: to provide a "library" of efficiently-coded routines accessible to newly-created main programs. In a ROM, the need to squeeze many operations into a limited space will often cause brevity of code to be given priority over maximum speed of execution. Also, subprograms will tend to be special-purpose instead of general-purpose. You can't have everything!

However, one service that ought to be included with the documentation of a monitor listing is a list of user-accessible subroutines, so that users need not waste time searching them out. Such a list may be longer than the number of RTS instructions in the monitor, since a subroutine may have several useful entry points (including some undreamed of by the person who wrote it!) The use of more than one exit point, which can sometimes enhance both code efficiency and timing, is often stigmatized as "unstructured." That's a word I have come to dislike. Everything has structure. The distinction that matters is *intelligibility*, and this is not necessarily lost by using two (or even more) exit points. What must be avoided is excess of any kind, including excessive rigidity and complexity.

**The Upgrading/Customization Problem.** Everyone must sooner or later outgrow a system as confining as an SBC, and the question becomes: upgrade to what? More crucial still, for what? Perusing the articles and ads in microcomputer journals is bewildering. So many competing systems, add-on peripherals, interfaces, languages, complex softwares, and above all uses! I long deferred any upgrading decision, in the hope that major innovations would soon appear. My concept of my own needs gradually crystallized: a system that could sense, measure, analyze, and act on the outside in a way similar—but superior-to the way I do and so do my work for me.

While "dedicated" microprocessors are already automating a variety of instruments, that's not quite what I want. There's a colossal wastage built in to the scientific (and doubtless also the business) world, caused by non-standardization, deliberate avoidance of modularity/interchangeability, and planned obsolescence.

Except at the leading edge, where state-of-the-art can only be had by custom design, "packaged" systems dominate scientific laboratories. Each has its built-in electronics, recorders, etc., so tied together that if one element obsolesces, the whole thing must be thrown away, brushed-aluminum chassis and all. These genies simultaneously serve and enslave their users, who become button-pushers with no comprehension of or power to modify the mechanisms.

The concept that has always appealed to me is more that of the "hi-fi" enthusiasts, where a system consists of several interacting but independently-replaceable components, from many competing sources. True competition, however abhorrent to industrial giants, makes possible a maximum of progressive change at the minimum cost to society. This eccentric, unbundled-components point of view enabled me to resist the lure of the increasingly powerful packaged systems that have entered the market in recent years. However flexible, they must be designed for some least-common-denominator purpose and tend to allocate large resources to things like BASIC interpreters (for me a turn-off). They are like low-cost smoothly-paved roads leading to where everyone else wants to go.

My first upgrade was a Synertek SYM-1. It is even more flexible and I/O-oriented than the KIM. Even future-oriented, since major ICs are socketed and one hopes that upgrading from the 6502 to the 6516 will be easy. The SYM is a masterwork, complex and not free from flaws, that encourages and even compels its user to explore unbeaten paths. It works best only for someone willing to make the effort to under-

stand it.

One thing the SYM can do is interface with a TTY or CRT terminal, via RS-232 at baud rates from 110 to 4800 (well below the potential of a 6502 CPU). It is obvious that a CRT—not a TTY—is the primary upgrading element of an expanded system. The Commodore PET built one in, while the Apple II, TRS-80, etc., built in only an ASCII keyboard and allow (really require) interfacing to a TV monitor. These systems tend to have less than the 24 x 80 character display of stand-alone CRT terminals, but to some extent compensate this by providing graphics, and sometimes color. Many of their limitations will soon be transcended (probably first by the Apple III). Conventional CRT terminals will soon face superior price/performance competition, and their too-high price is already falling. I therefore decided to buy one, the Televideo 920-B, that seemed a fair value at its current near-$800 mail-order price. It lacks the graphics capability of the 920-C, but I don't mind that since my mind is a symbol-oriented algebraic, not a geometric one. Of course, so-called "printer graphics" of the TTY type are possible. The 920-B is an "intelligent" terminal controlled by an Intel 8035 micro-processor, with a 32K ROM and sockets for 8 high-speed 2114 static RAMs (allowing storage of an "alternate" 24 x 80 character page). Competitive pressure has now made all intelligent terminals so good that I have no idea whether the 920-B is one of the "best," but it is very satisfactory. I like its eleven "function" keys, that send a 3-character message to the computer: 01-XX-OD, where XX ranges from 40 to 4A with the F keys alone, and from 60 to 6A if the shift key is also depressed, for a total of 22 different commands. The user decides how the computer shall interpret them. It also has its own RS-232 interface to a printer, so that the user can print exactly what he sees on the screen.

When I connected the 920-B to the SYM-1, the log-in command (S, J, 1, CR) from the SYM's onboard keyboard did not work. A study of the monitor listing led me to modify the vector in the monitor RAM at $A622 from its set value of $A7 to $71. This worked, proving that having a listing—and being able to read it—is a good thing, and also that the redundancy of *dual* control has merit.

The slowness of 4800-baud serial transmission (that seems to fill the screen at about 80 characters/second) is a shade annoying to a lover of speed. The 920-B can work at 9600 baud (very good for serial), but the SYM designers were not thinking in those terms. Although I've known the theory of communication from books, this first real-life encounter with it has been educational. One sympathizes with the problems of the pioneers trying to drive messages through long transmission lines—losses, noise, errors and the like—and sees the why of parity and checksums. The SYM monitor recognizes the low probability of transmission errors by just ignoring parity. The 920-B doesn't (unless the user so desires) since it might anytime get hooked to a telephone line! It could easily connect via a modem to a timeshared system such as MicroNet.

**Comments on the SYM monitor program.** Like many other KIM-trained machine-code programmers, I am keenly aware of the beauty of both code-efficiency and time-efficiency. Also of higher virtues of clarity, simplicity, modularity, and generality. Hardly ever are all of these wholly compatible.

The writers of the SYM monitor were required to pack as much power and versatility as possible into a 4K-byte space (in a socketed 2332 ROM). The coding gives priority to byte-saving devices, sacrificing timing. They had to complete the task quickly, at the cost of incomplete testing. Version 1.0 had an error in its KIM-format audio-cassette program, corrected in a new version 1.1 ROM (that nonetheless had the log-in-to-CRT flaw mentioned above). Its high-speed audio-cassette interface worked (in my hands) only with Microsette or Data Sound data tape.

The SYM monitor has a command language, similar in kind to that of KIM but far more complex, and designed to be easily user-expansible. I explored the possibility of re-interpreting the keyboard-command keys of the KIM in my EDITHA program (*DDJ* #25; although there are only 4 keys, anything one may desire can be done by implementing key *sequences*. However, the SYM expects to interact with an ASCII keyboard, so that a very large number of single-key commands could be implemented. Those that are recognized by the monitor in its command-entry mode are: B, C, D, E, F, G, L1, L2, LP, M, R, S1, S2, SD, SP, V, and W. Six of the 2-character commands (all but SD) deal with loading and storing of programs with audio cassette or paper tape. The others are useful in program-writing, viewing, running, and debugging (too esoteric to go into detail). A command is acknowledged by displaying it and printing a space. The monitor then shifts to a hex-digit entry mode, that can accept (and print) up to 3 sets of 4 hex digits, delimited by a comma (or hyphen), to serve as numerical parameters for the command. Pressing the CR (carriage return) key then causes execution if the command is valid, i.e., interpretable by the monitor. Only the M command is defined for all 4 parameter possibilities (0, 1, 2, or 3). If an undefined number of parameters has been keyed-in, control shifts to an "unrecognized syntax" vector. An unrecognized command character shifts control to a vector for that.

The computer elite (and some not-so-elite) will naturally view a mere command language as an antiquated anachronism, when so many wondrous high-level languages exist! I do some reading in the HLL domain but see nothing irresistably great there. A language can imprison the mind. True, you can encode complex logic in symbols that may seem simple, and shift the arduous task of programming and antibugging machine code to an abler mind than your own. But errors are instructive, tutelage can dim vision, and few gains are not balanced by (often unseen) losses.

**A Preliminary Monitor-Enhancement Program.** The "unrecognized command" mode is free-wheeling and congenial for machine-linguists. In its command-entry state, the SYM monitor is sent any ASCII character it will not recognize. It echoes this, and a space, then awaits 1, 2, or 3 four-digit hex parameters (more could be had, but I prefer to limit complexity). I reserve the last one keyed in for the address of the program to be run. Therefore it is akin to the monitor's own G (for GO) command, except that the user can provide parameters for the program to use. The unrecognized-command vector is reset to the address of a simple 23-byte interpreter (INPRET, cf. listings) that copies the second keyed-in parameter to zero-page to serve as an indirect address for program instructions, then jumps-indirect to the last meter to shift control to the program, whose address is the *real* command. The program in the listings is moderately complex but not optimized (as much of my earlier work was). It is an exercise in interaction between the SYM-1 and the 920-B, a sketch of something useful that may become even more so, perhaps even worth putting into an E-PROM. I was driven to write it because the SYM monitor expects to work with "dumb" peripherals (or even with only its own keyboard and hex display, in a quite ingenious way). There is no provision for teamwork with an intelligent terminal, and its TTY-style programs for displaying, modifying, and moving memory are restrictive.

My new programs implement only some of the possible intercommunication between the SYM and the 920-B. While this specific system may now be a unique one, it may arouse some interest in the teamwork concept, that is feasible for any SBC working with any intelligent terminal. The main program, DISPAG (at $0C00 in the listings), is implemented by the command sequence: T XYY, ZZZZ, C00 CR (where CR is the carriage-

return key). Any unrecognized character other than T works as well, since it will shift control to the vector preset in $A66D-E, that in the example is the address of INPRET ($0D00). ZZZZ is the starting address of the block of memory to be displayed, e.g., C00 will display the program itself (note that leading zeroes need not be keyed in.) YY is the hex number of *lines* to be displayed, that must range from 1 to $18. Any other value will yield an error message and return to monitor control. X controls the display format, and need not be keyed in if the "unispace" format is desired; any hex digit from 1 to F will command the "lister" format. This operation is controlled by subroutined LNFORM, that could be modified to allow other formats.

The display will therefore be from 1 to 24 lines of bytes (as hex-digit pairs) of memory. The first four (packed) hex digits are the address of the first byte in the line, separated by one space from the byte sequence . In the unispace format, each byte is followed by one space and each line has exactly 16 bytes. In the lister format, every opcode is preceded by an extra space, so that the *instruction* sequence is more legible. This is useful only for programs, and is meaningless for data tables. To avoid splitting one instruction between two successive lines, the lister format allows up to 18 bytes on one line. It uses my old BYTCNT program, all-too-familiar to *DDJ* readers! Parenthetically, I note that one more 6502 byte-count program, not very efficient, has been published (BYTE, May 1980, p. 190).

When all the lines are displayed, control shifts to the NEXCHR program. This frees the 920-B keyboard for any of its own programmed operations. To return to the SYM monitor, the CR key must be pressed twice in succession. In this mode the display can be edited in any way desired. Wholly new progams can be created on the CRT screen. NEXCHR merely echoes everything the 920-B sends to it, a mode that persists unless a double-CR or a RUBOUT signal is received. This last is a bit dangerous, since it shifts control to a progam sequence (starting at LNREAD) that can wreck the SYM memory if used carelessly. Its purpose is to transfer a sequence of hex-bytes from the CRT screen (really the 920-B memory) to a specified region of the SYM memory. It uses most of the code in the listings, and was the trickiest to write.

For the 920-B to SYM transfer to work correctly, the revised (or brand-new) program on the screen must obey the following restrictions:

1. Each line must start with 4 characters, followed by a space. These will be read but not used in any way. The reason for this is to make it unnecessary to alter the address information that DISPAG has written, when an existing program is being revised (or just moved to another memory location.) It is an inconvenience when a new program is being constructed, unless the user wishes to keep track of addresses, but at this early stage I prefer not to augment the program complexity.
2. The lowest program line must be followed by a completely blank line (unless it is at the very bottom of the screen display, in which case a blank line will be automatically created by the read operation). This is needed to terminate the transfer.
3. The line just above the first program line must also start with 4 characters and a space (again read but not used), and then must have exactly 4 hex characters that specify the SYM RAM address to which the program is to be moved. E.g., 0200 will cause transfer to start at that location in the SYM RAM. The cursor must be to the *right* of that address on the screen when the RUBOUT key is pressed. Since the top line provides the only address information, this must be correct!

It would be possible—perhaps desirable—to include various anti-goof guards (such as a requirement that the RUBOUT key be pressed twice in succession to initiate the transfer) but only use-experience can truly dictate how a program of this type ought to be refined. A few guards do exist. Either a non-hex character or an incomplete hex byte (only one digit) will cause return to monitor control.

The program lines need not be limited to 16 bytes, or be written in any particular format, though the lister format is the most legible. When the transfer has been completed, control shifts to the VERIF program, that will use the DISPAG logic to display the revised or new program that is now in the SYM memory, and again shift to the NEXCHR mode. This allows further revision, and another transfer. This can be kept up until the user is satisfied that the program is ready to be tested. In my experience, it has proved desirable to test tricky logic in the form of subroutines, even when it will ultimately be coded in-line to optimize time-and code- efficiency. Debugging a complex in-line program can be difficult! There are times when I wonder whether really adequate testing is even possible. And is a poorly-tested program really debugged?

**The Game of ROM-Creation.** Even the minority of programmers who have read the ROMs created by someone else do not fully appreciate the problem until they toy with the idea of creating their own. It's the chasm between critic an artist, or rather between a builder of sand-castles and a sculptor in marble.

My coding would surely seem very sand-castly to the writer(s) of the SYM monitor, who make heavy use of the stack and hardly touch zero-page—while I haven't even one overt stack-save, and transfer information freely via the on-chip registers, and use zero-page with wild abandon. The last is possible because the monitor graciously cedes zero-page to users, using instead its own half-page of RAM staring at $A600. The only zero-page locations it uses are the eight

of RAM starting at $A600. The only zero-page locations it uses are the eight from F8-FF. Its stack usage is compelled by a seemingly iron rule that registers must be saved and restored by all subroutines, except when information is to be returned in the accumulator, the status register, or both. The 6502 instruction set allows a 6-byte code sequence for stack-saving (16 cycles) and for restoring (20 cycles) its four major registers PAXY. However, the SYM monitor frequently saves bytes by using a JSR SAVER (3 bytes but 103 cycles!) as the first instruction of a subroutine. SAVER is an intricate 48-byte program that rearranges the stack so that, when it has returned, PAXY is left on the stack. The subroutine that has used SAVER exits by a JMP to one of three entry-points of a program that will restore all registers (29 cycles), only AXY (41 cycles), or only XY (55 cycles) before its RTS termination. This allows three options with only six more bytes added to the subroutine, but the timing penalty is high (I think it's why serial transmission can't exceed 4800 baud.) The philosophic contrast shows in my 62 bytes of code from $0D2F in the listings, that contain 13 different entry points to a single RTS (and 2 others, ESCOUT and TWOCHR, that are potential entry points), averaging less than 5 code bytes per subroutine. This economy is possible only because nothing is saved. Panegyrists of discipline might also frown on the anarchy of my subroutine LNFORM, that uses the Y index for "parameter-passing" and PAX in a self-centered way. Its calls to the monitor subroutines OUTBYT and SPACE, on the other hand, involve no fewer than 28 stack save/restore instructions, plus the musical-chairs stack manipulation of the monitor's SAVER subroutine, for a total of 39 stack operations! Since LNFORM only needs to protect the X and Y registers from alteration by OUTBYT, OUTBYT does a lot of work to svae LNFORM the 8 bytes it would need to do it itself. The *entire* KIM-1 monitor has only *ten* stack pull/push opcodes, but does more saving in zero-page—simpler but with the possibility of conflict, that is of course a powerful argument for using the stack.

One thing is for sure. ROMs tend to use a lot of subroutines. The KIM monitor has 115 JSRs, and it's the dominant opcode. The SYM monitor has 269 JSRs (in twice the ROM space). Knowers of the 6502 will expect that the LDA will be a close second (as it is in the KIM and probably in the SYM, that does not have a static analysis of opcode usage frequency). That is a powerful argument for speeding up subroutine calls, as the lonely genius who designed the Signetics 2650 years ago foresaw. I have no idea who he was (though I am sure the gender is right), but I hope the story of the 2650 will someday be written, since foresight is to me the most miraculous of all human qualities. I hardly need to expatiate, for readers of *DDJ*, on the virtues of subroutines: easy debugging, easy rewriting, and above all availability to any program.

Re-reading my main program (from $0C00 to 0C73), I see that I created seven subroutines, without quite knowing why. All are used exactly once and could have been coded in-line. Why not? LNFORM CVHEX, and RNOSCP are complex enough so that the need for tailorability was perhaps subconsciously obvious. The others are moronic drudgery, and may just indicate something like battle-fatigue. It's wrong to assume that everything constructed by a frail human being is a gem of purest ray serene! We get tired, we get fed-up, we don't feel like recalculating relative-branch offsets, so we do it the easy way. This kind of feeling must have pervaded the SYM monitor-writer(s) near the end of the task. Michelangelo too, when he put the finishing touches on the Sistine Chapel, at an age not far from my own.

In fact, the subroutine-library concept that I lauded at the start of this communication is elusive. Who knows what will be useful to many other workers? No monitor has a bytecount routine, but we now know it's useful because so many have been created. The SYM provides a block-move routine, that many KIM users had to write for themselves. However, its byte-search program can only find single bytes (my EDITHA program can also find sequences of two or three bytes). What length of byte-string is likely to be needed? ROM-design is bedeviled by dilemmas, and is analogous to the design of the CPU itself, likewise frozen in silicon.

One essential of ROM-design is mastery of the CPU instruction set. The SYM monitor shows great skill in complex assembler logic, and is usually very well coded, but there are spots where recoding would save both bytes and time. That kind of thing is rare in the work of Jim Butterfield, who "thinks in 6502", so I assume that the SYM writer(s) learned on a different kind of machine. One gets "conditioned" to a set one uses all the time—much like one of the human languages—so there'll be touches of awkardness if one shifts to a less familiar set (or language), recognizable by a "native speaker." There's also the "personal style" factor, but some styles command admiration even though they differ from one's own.

A ROM also has an "architecture," as a CPU does, and this may be true of any program (even my rough sketch in the listings). The SYM ROM is basically a command-language interpreter. Having read the low-level "instruction," it first branches to one of the four "legal" parameter-blocks (0, 1, 2, or 3 parameters). Each of these checks the "legal" command-characters and, if one is found, shifts control to the proper execution coding. If not, control shifts to the "unrecognized-command" vector, that defaults to an error-message (very simple and clever). This is something more than a set of isolated routines filed so they can be retrieved by keying the right code, since the ensemble serves a common purpose. Perhaps the only component of my new program that merits addition to a monitor is the old BYTCNT routine; all else is system-dependent.

I am reminded of a brief—but highly didactic—exposition of the principles of system design (chapters 1 and 3 in the book *Systems Concepts*, edited by R. F. Miles, Wiley-Interscience, 1973). A "system" is defined as a large, complex, man-made set of concepts and/or elements used to satisfy some human need. All components are integrated to yield a set of optimum outputs from a set of given inputs (some of them random, so that the exact performance at any instant is unpredictable). It works semi-automatically: machines always perform some of the functions and human beings always perform other functions. The design goal is to maximize "effectiveness" for a given cost—that implies trade-offs. I now shift from paraphrase to a key quotation: "Because it is generally impossible to find a single number which realistically represents the effectiveness of a complex system, there is a good deal of subjectiveness, as represented by judgment, as well as objectiveness, as represented by analysis, in systems engineering." The system is an entity such that "... optimization of each subsystem independently will not lead in general to a system optimum, and ... improvement of a particular subsystem actually may worsen the overall system." Food for thought there! Less arguable is the concept that "no system can be all things to all people, all of the time" so that "the fundamental mission of a system should not be jeopardized, nor its fundamental objectives significantly compromised, in order to accommodate events of extremely low probability" since that would make the system "too complex to be workable." This means that someone must guess right the probability of future events! Wow! In the saying *per aspera ad astra*, only the *aspera* are certainties.

**A Brief Afterthought on FORTH.** To me, FORTH has been the most tantalizing of the existing HLLs. Perhaps the word is infuriating, since the FORTH enthsiasts—like the Rosicrucians or the initiates to the ancient Eleusynian mystery—won't tell you what it is. In comparison, my own much more miniscule programs come with a surfeit of explanatory comment (tinseled with allusions). When I glanced at a listing of FORTH, its most striking quality was the virtual absence of comment. Adam Osborne recently observed (*InfoWorld* 2(8):7, 1980) that the success of an HLL depends less on its intrinsic merit thn on hhow hard it's pushed. Whatever the demerits of BASIC—and they are legion—being unexplained is not one of them. Dozens of books expound it in great detail, and some are brilliant. Where is the book that describes how FORTH works, from the ground up, in a painstakingly detailed, translucent an vivid way?

```
Main Program, 3 sequences: DISPAG, NEXCHR, VERIF.  Normal entry
     from Monitor via INFRET (cf. text for details.)

0C00  20 4D 63  DISPAG  JSR CRLF   (carriage return + line feed)
   3  A6 01             LDX ZBAS+1 (base Page Hi in X reg.)
   5  A5 00             LDA ZBAS   (  "  Lo in Acc.)
   7  20 F4 82          JSR OUTXAH (print HiLo base address)
   A  20 42 83          JSR SPACE  (print one space)
   D  A0 00             LDY #0     (zero Y index for this line)
   F  20 90 0C          JSR LNFORM (print one line of bytes)
  12  98                 TYA        (# of bytes printed, Y to Acc.)
  13  18                 CLC        (clear carry for add)
  14  65 00              ADC ZBAS   (add to base address Lo)
  16  85 00              STA ZBAS   (store new Lo value)
  18  90 02              BCC NOPINC (carry clear, no Hi change)
  1A  E6 01              INC ZBAS+1 (increment Hi address)
```

```
  1C  CE 4E A6  NOPINC DEC P1L    (decrement line-count)
  1F  DØ DF            BNE DISPAG (if not Ø, do next line)

ØC21  2Ø 58 8A  NEXCHR JSR INTCHR (get char. from term. kybd.)
   4  C9 ØD            CMP #$ØD   (a CR?)
   6  DØ Ø7            BNE RUBTES (if not, go test a RUBOUT)
   8  2Ø 58 8A         JSR INTCHR (get next character)
   B  C9 ØD            CMP #$ØD   (a second CR?)
   D  FØ 44            BEQ EXIT   (if yes, back to monitor)
   F  C9 7F    RUBTES  CMP #$7F   (a RUBOUT? be careful!)
  31  DØ EE            BNE NEXCHR (no, go get another char.)

 (memory move from terminal memory to SYM memory now begins)

ØC33  2Ø 73 ØD  LNREAD JSR RNOSCP (move one line as ASCII chars.)
   6  BØ 3B            BCS EXIT   (error, not an even #)
   8  3Ø 2E            BMI VERIF  (line all spaces, verify move)
   A  2Ø B3 ØC         JSR CVHEX  (convert ASCII to hex bytes)
   D  BØ 34            BCS EXIT   (error, non-hex conversion)
   F  A6 Ø7            LDX ZBAS+7 (load # of chars. moved)
  41  EØ Ø5            CPX #5     (more than four?)
   3  BØ Ø5            BCS MVRAM   (if yes, move bytes to RAM)
   5  2Ø A4 ØD         JSR STORAD (if four, use as base address)
   8  FØ E9            BEQ LNREAD (and read next line)

ØC4A  A4 Ø6      MVRAM LDY ZBAS+6 (load # of bytes to be moved)
   C  88        WYDEC DEY         (decrement #)
   D  B5 Ø7            LDA ZBAS+7,X (pick up top byte in buffer)
   F  91 ØØ            STA (ZBAS),Y (store in RAM address)
  51  CA                DEX          (decrement X index twice
   2  CA                DEX           for next lower hex byte)
   3  DØ F7            BNE WYDEC   (if not Ø, do next byte)

 (now reset base address for next line-read and total byte count)

ØC55  A2 FC                 LDX #$FC  (set X index for 4 increments)
   7  A5 Ø6       ADDOP    LDA ZBAS+6 (load # of bytes moved)
   9  18                   CLC        (clear carry for add)
   A  75 Ø4                ADC ZBAS+4,X (add to base address Lo)
 ØC5C  95 Ø4               STA ZBAS+4,X (store new addr. Lo)
   E  9Ø Ø2                BCC HISAME  (if carry clear, same Hi)
  6Ø  F6 Ø5                INC ZBAS+5,X  (increment addr. Hi)
   2  E8           HISAME  INX           (up two for byte count)
   3  E8                   INX
   4  DØ F1                BNE ADDOP  (if not Ø, add to count)
   6  FØ CB                BEQ LNREAD (if Ø, go read next line)

 (the verify sequence is reached when LNREAD read all spaces)

ØC68  2Ø DØ ØC  VERIF JSR SETAD  (reset base addr.)
   B  2Ø DD ØC         JSR LNCNT  (set # lines to display)
   E  2Ø AC ØD         JSR ZERCNT (zero byte count locations)
  71  FØ 8D            BEQ DISPAG (really a forced branch)
  73  6Ø          EXIT RTS        (back to monitor)

 (main program ended, now come 4 of its subroutines)

ØC74  A2 Ø1     BYTCNT LDX #1  (cf. DDJ #22 for much comment!)
   6  2C ØF 8Ø         BIT IRQBRK  (an Ø8 BIT-mask in SYM ROM)
   9  DØ Ø8            BNE HALFOP

   B  C9 2Ø            CMP #$2Ø
   D  FØ ØE            BEQ THREE
   F  29 9F            AND #$9F
  81  DØ ØB            BNE TWO
   3  29 15    HALFOP  AND #$15
   5  C9 Ø1            CMP #1
   7  FØ Ø5            BEQ TWO
   9  29 Ø5            AND #5
   B  FØ Ø2            BEQ ONE
   D  E8         THREE INX
   E  E8           TWO INX
   F  6Ø           ONE RTS

ØC9Ø  A2 1Ø     LNFORM LDX #$1Ø  (set X to display 16 bytes)
   2  AD 4F A6 NEXTOP LDA P1H  (is format control parameter)
   5  FØ Ø5            BEQ SAMEOP (if Ø, not lister format)
   7  B1 ØØ            LDA (ZBAS),Y  (load opcode byte)
   9  2Ø 74 ØC         JSR BYTCNT (get its bytecount, in X)
   C  B1 ØØ    SAMEOP  LDA (ZBAS),Y   (load byte)
   E  2Ø FA 82          JSR OUTBYT   (print it)
  A1  CA                DEX          (decrement count)
  A2  DØ Ø3             BNE UNISPC   (if not Ø, only one space)
  A4  2Ø 42 83          JSR SPACE    (print one space)
  A7  2Ø 42 83 UNISPC   JSR SPACE    ( "     "    "  )
  AA  C8                INY          (increment Y to get next byte)
  AB  CØ 1Ø             CPY #$1Ø     (if Y=16+, will set carry)
  AD  8A                TXA          (check if X = Ø)
  AE  DØ EC             BNE SAMEOP   (if not, same instruction)
  BØ  9Ø EØ             BCC NEXTOP   (if Y not yet 16, next one)
  B2  6Ø                RTS          (line display completed)

ØCB3  B5 Ø7      CVHEX LDA ZBAS+7,X  (load topmost ASCII char.)
   5  2Ø 75 82          JSR ASCNIB  (convert to hex, lo nibble)
   8  BØ 15             BCS FINIS   (error, not a hex character!)
   A  95 Ø7             STA ZBAS+7,X  (store conversion)
   C  CA                DEX           (to pick up next char.)
   D  8A                TXA           (move to Acc. to test if odd #)

ØCBE  4A                LSR A  (bit Ø to carry, set if odd)
   F  BØ F2             BCS CVHEX  (if odd, convert hi half)
  C1  B5 Ø8             LDA ZBAS+8,X  (reload hi conversion)
   3  ØA                ASL A   (and do 4 left-shifts to
   4  ØA                ASL A     move into hi nibble)
   5  ØA                ASL A
   6  ØA                ASL A
   7  15 Ø9             ORA ZBAS+9,X  (insert lo nibble)
   9  95 Ø9             STA ZBAS+9,X  (store whole hex byte)
   B  8A                TXA     (test if X now Ø)
   C  DØ E5             BNE CVHEX  (if not, do next byte)
   E  18                CLC   (clear carry, no-error exit)
   F  6Ø                RTS

ØCDØ  A2 FE      SETAD LDX #$FE  (set X for 2 ops)
   2  38                SEC   (set carry for subtract)
   3  B5 Ø2      SUBIT LDA ZBAS+2,X  (load address, LO, HI)
   5  F5 Ø4            SBC ZBAS+4,X (sub count, Lo, Hi)
   7  95 Ø2            STA ZBAS+2,X (store new base address)
   9  E8               INX    (increment X to do HI)
   A  DØ F7            BNE SUBIT  (if not Ø, go do it)
   C  6Ø               RTS
```

```
ØCDD  A5 Ø2      LNCNT LDA ZBAS+2  (load byte count Lo)
   F  A2 Ø4            LDX #4  (set X for 4 rotate ops)
  E1  18               CLC (clear carry to insert a Ø)
   2  66 Ø3      LNCALC ROR ZBAS+3  (rotate count Hi right)
   4  6A               ROR A     (  "      "    Lo right)
   5  CA               DEX   (four ops done?)
   6  DØ FA            BNE LNCALC  (if not Ø, continue)
   8  AA               TAX   (move line count from A into X)
   9  66 Ø3            ROR ZBAS+3  (rotate the carry in)
   B  FØ Ø1            BEQ STOLIN (if Ø, no fractional line)
   D  E8               INX   (if not, add one whole line)
   E  8E 4E A6  STOLIN STX P1H  (store line count for DISPAG)
  F1  6Ø               RTS

 (the next program page starts with the unrecognized-command
  interpreter, INPRET.  Subroutines meant for interaction of
  the SYM and the 920-B follow, not all being used in the
  working program, then two new essential subroutines.)

ØDØØ  AD 4E A6  INPRET LDA P1L  (load line-count parameter)
   3  FØ 11            BEQ EROUT (zero is an error, so exit)
   5  C9 19            CMP #$19  (upper display limit is $18)
   7  BØ ØD            BCS EROUT  (if more, exit)
   9  A2 Ø1            LDX #1  (set for base address move)
   B  3D 4C A6   PTOZP LDA P2L,X  (pick up addr. parameter)
   E  95 ØØ            STA ZBAS,X  (store in zero page)
  1Ø  CA               DEX
  11  1Ø F8            BPL PTOZP  (get LO)
  13  6C 4A A6         JMP (P3L)  (jump indirect to program)
  16  6Ø         EROUT RTS     (if error, back to moniter)

ØD17  2Ø 5D ØD  CURSLD JSR CURASK  (asks cursor position)
   A  A2 Ø3     TRICHR LDX #3   (set for 3 characters)
   C  2Ø 58 8A  LODCHR JSR INTCHR  (get a character)
   F  95 Ø3            STA ZBAS+3,X (store in zero-page)
  21  CA               DEX     (decrement for next char.)
   2  DØ F8            BNE LODCHR  (if not Ø, get char.)
   4  6Ø               RTS

ØD25  2Ø 61 ØD  CURSET JSR SETCUR (signal to set cursor)
   8  2Ø 4F ØD         JSR YXSEN  (send coordinates)
   B  6Ø               RTS

 (series of multiple-entry character-send routines)

ØD2F  A9 Ø7       BEEP LDA #7   (terminal beep sound)
  31  DØ 36            BNE ONECHR (forced branch, output)
  33  A9 Ø8     CULEFT LDA #8  (move cursor left)
  35  DØ 32            BNE ONECHR
  37  A9 ØA     CUDOWN LDA #$ØA  (move cursor down)
  39  DØ 2E            BNE ONECHR
  3B  A9 ØB     CURSUP LDA #$ØB  (move cursor up)
  3D  DØ 2A            BNE ONECHR
  3F  A9 ØC     CRIGHT LDA #$ØC  (move cursor right)
  41  DØ 26            BNE ONECHR
   3  A9 ØD     CARETN LDA #$ØD  (carriage return)
   5  DØ 22            BNE ONECHR
   7  A9 1F     NEWLIN LDA #$1F  (= a CR,LF)
   9  DØ 1E            BNE ONECHR
   B  A9 1E       HOME LDA #$1E  (send cursor home)

   D  DØ 1A            BNE ONECHR
   F  A5 Ø4      YXSEN LDA ZBAS+4 (cursor Y in zero-page)
  51  A6 Ø3            LDX ZBAS+3 (cursor X  "     "    )
   3  DØ 1Ø            BNE TWOCHR (send both coordinates)
   5  A2 4B     TOGPAG LDX #$4B  (toggle to 2cd screen page)
   7  DØ ØA            BNE ESCOUT (send ESC, then char.)
   9  A2 36     SENLIN LDX #$36  (send cursor line)
   B  DØ Ø6            BNE ESCOUT
   D  A2 3F     CURASK LDX #$3F (terminal asked cursor position)
   F  DØ Ø2            BNE ESCOUT
  61  A2 3D     SETCUR LDX #$3D (set term. to new cursor status)
   3  A9 1B     ESCOUT LDA #$1B  (load ESC character)
   5  2Ø AØ 8A  TWOCHR JSR TOUT  (send the char. in Acc.)
   8  8A               TXA    (shift char. in X into Acc.)
   9  2Ø AØ 8A  ONECHR JSR TOUT   (send the character)
   C  6Ø               RTS   (sole exit for all routines)

  (now two special subroutines for main program use)

 ØD6D  A2 FF       RDALL LDX #$FF  (set to read whole line)
   F  A9 2Ø            LDA #$2Ø  (including space chars.)
  71  DØ Ø4            BNE SETOP (forced branch)
   3  A2 FB     RNOCSP LDX #$FB  (exclude first 4 chars.)
   5  A9 21            LDA #$21  (  "     space chars.)

ØD77  85 Ø5       SETOP STA ZBAS+5 (store select comparand)
   9  86 Ø7            STX ZBAS+7 (save X index)
   B  2Ø 59 ØD         JSR SENLIN (command send cursor line)
   E  2Ø 43 ØD         JSR CARETN (CR, so echo overwrites)
  81  A6 Ø7            LDS ZBAS+7 (restore X index)

  83  2Ø 58 8A  MORCHR JSR INTCHR (accept a sent character)
   6  C5 Ø5            CMP ZBAS+5 (excluded by comparand?)
   8  9Ø Ø7            BCC TESEND (if yes, go test end-of-line)
   A  E8               INX        (increment X index)
   B  3Ø F6            BMI MORCHR (if negative, get next char.)
   D  95 Ø8            STA ZBAS+8,X (if not, store in z.p. buffer)
   F  1Ø F2            BPL MORCHR (forced branch, continue)
  91  C9 ØD     TESEND CMP #$ØD  (a CR? it's end-of-line signal)
   3  DØ EE            BNE MORCHR (if not, continue)
   5  2Ø 37 ØD         JSR CUDOWN (after the exit CR, cursor down,
   8  2Ø 33 ØD         JSR CULEFT  left, sets to end of next line)
   B  E8               INX      (X+1 is actual # of chars. stored)
   C  86 Ø7            STX ZBAS+7  (save character count)
   E  8A               TXA        (move from X into Acc.)
   F  4A               LSR A  (right-shift, divide by 2, =# of bytes)
  AØ  85 Ø6            STA ZBAS+6  (save count of hex bytes)
  A2  8A               TXA       (tests for char. count of Ø)
  A3  6Ø               RTS

 (note: above termination logic is OK for the RNOCSP entry, but not
  for the RDALL entry that has no exclusion logic)

ØDA4  A5 Ø9       STORAD LDA ZBAS+9 (load base address HI)
   6  85 Ø1            STA ZBAS+1 (store for indirect address)
   8  A5 ØB            LDA ZBAS+11 (load base address LO)
   A  85 ØØ            STA ZBAS   (and store for indirect)
   C  A9 ØØ      ZERCNT LDA #Ø    (to re-zero count locations)
   E  85 Ø3            STA ZBAS+3  (first the Hi)
  BØ  85 Ø2            STA ZBAS+2  (then the Lo)
  B2  6Ø               RTS
```

**End**