CEGMON

a new monitor
for OSI and UK101 BASIC-in-ROM systems

user's guide

Warranty terms

There is no warranty, either expressed or implied, for CEGMON in any version. CEGMON is sold subject to the distributor's standard conditions of sale, copies of which are available on request. Although it has been extensively tested and is believed to be error-free, there is no guarantee that any of the software will operate under conditions not noted in this User Guide.

Despite the above, we would be grateful for any comments regarding CEGMON or on any possible faults which may appear, to help us in developing future versions.

Licensing and copyright terms

CEGMON and its documentation are protected both by copyright and also by our standard licensing agreement for software (copies available on request).

In essence, you have bought a 2716 EPROM and associated documentation, and a licence to use the software contained on or in it for your own purposes on your own computer.

Copying all or any part of the software onto any medium or into any retrieval system and/or using this on a computer not owned by you is expressly excluded from the terms of your licence and would constitute a breach of contract.

Sold under licence and distributed by Mutek, Quarry Hill, Box, Wiltshire, England

Documentation prepared by Wordsmiths, 19a West End, Street, Somerset BA16 0LQ

Introduction

CEGMON has been developed to provide you with the kind of firmware support that you need to get the most out of your computer. When Ohio Scientific wrote their SYNMON monitor, in the none-too-distant days of expensive computer memory, it is clear that cost was rather more important than flexibility or 'user-friendliness'. To save on cost, OSI's SYNMON 2K monitor chip actually contains monitors for several different machines, and only a meagre ¾K is used for each type. CEGMON uses the full 2K of monitor space. The result is, we hope, a monitor PROM that gives your machine more features and flexibility than any computer in the price range, while retaining the greatest practicable compatibility with the existing firmware and software.

Four versions of CEGMON are available: for standard Superboard or C1; enhanced (32×48 display) Superboard or C1; UK101; and C2 or C4. In the following notes the versions are referred to as C1, C1E, 101 and C2 respectively.

The main features of CEGMON are:

- a **screen-editor** for use with BASIC or assembler programs, linked directly to the system's call for keyboard input.
- a **revised keyboard routine**, giving typewriter-like response, true ASCII key-values and direct access to most graphics.
- a completely **new screen-handler**, formatting screen output within user-definable 'windows', with cursor control and direct 'window'- or screen-clear.
- a **full machine-code monitor** for development of machine-code routines, which includes machine-code load and save in auto-start format, input of text or graphics as well as hex instructions, tabular hexadecimal display of memory contents, memory mover and breakpoint handler for debugging.
- **input and output** from BASIC or assembler vectored through RAM, allowing direct linkage to user-defined I/O routines.
- **compatible design**: entry points of all SYNMON main routines and most subroutines have been retained; OSI-compatible floppy-disc bootstrap provided on all versions, to support OSI's 610 or 470 disc-controller boards.

Start-up

On start-up, the usual array of random 'garbage' characters will appear on the screen. On hitting the BREAK key (RESET — 101), the garbage will be replaced by the CEGMON *'D/C/W/M ?'* prompt at the top of a clear screen. As with SYNMON, answer the prompt with D, C, W or M, to access Disc bootstrap, BASIC Cold start, BASIC Warm start or Monitor respectively.

As with SYNMON, the BREAK reset necessarily resets all the input/output vectors, and also resets the current 'window' for the screen handler to the default values, i.e. text display on the full screen area. Accidental BREAK may be a nuisance in some applications, where the vectors or 'window' definition are important; it's probably a good idea, especially on OSI machines rather than UK101, to insert another switch in series with the BREAK key — and well out of the way! — to prevent accidental BREAK.

Keyboard

The OSI polled keyboard is now decoded in true ASCII format, with one exception: the RUBOUT key is decoded as $\$5F$ (95_{10}) to allow it to be used as a 'delete' key with OSI's BASIC or Assembler.

The major change is that keyboard use with SHIFT-LOCK released is rather more predictable. The two shift keys are now decoded identically; carriage-return (CR), line-feed (LF), RUBOUT and (not 101) escape (ESC) are all accessible regardless of the state of SHIFTs or SHIFT-LOCK. Control (CTRL) returns the same value from alphabetic keys regardless of SHIFT or SHIFT-LOCK — i.e. CTRL-A and CTRL-a both return an ASCII value of 01; but the state of SHIFT-LOCK *is* assessed with CTRL for *non*-alphabetic keys, in order to access otherwise inaccessible characters in the range $\$7B$-$7F$ (123-127_{10}). For the same reason, SHIFT and SHIFT-LOCK pressed together with alphabetic characters still produce a range of garbage characters, in order to allow SHIFT-K to -O to access the up-arrow and other characters in the range $\$5C$-$5F$ (91-95_{10}).

The BASIC line-delete, @, is decoded as SHIFT-0 (zero) as before, but *not* as SHIFT-P. SHIFT-O still functions as a character-delete, now working as a true backspace/delete rather than the crude Teletype underscore; but it is simpler to use RUBOUT, which works regardless of the state of SHIFT or SHIFT-LOCK.

The REPEAT key (not 101) is now used as a second control-key, to access the non-ASCII graphics in conjunction with other keys. Note that BASIC's own input routine masks out most of the control characters and all character-values above ASCII $\$7C$ (124_{10}); however, the masking can be bypassed either by a USR call direct to the keyboard or editor routine, or through a machine-code 'stack trick' via the input vector, as shown in the program example on page 15.

Another program example shows a non-halting 'GET' routine, called by a USR call from BASIC.

Screen handler

CEGMON's screen handler is an entirely new design, applying the 'protected area' and 'scrolling window' concepts of editing terminals to the memory-mapped display system of OSI's microcomputers. All output to the screen is via a 'window' whose position, height and width are all user-definable, and can be changed simply and quickly within programs. Four non-destructive cursor-positioning commands are available, as is a 'window'-clear command, and also a separate total screen-clear.

On start-up, the 'window' is largely identical with the existing formats for C1, C1E, C2 and 101. An obvious difference is that printing starts at the *top* of the screen, with a small allowance made for overscan. Scrolling only occurs when the base-line of the 'window' would be exceeded; until this happens, PRINTing on the screen is very much faster than under SYNMON. In all other respects, though, the start-up 'window' performs in much the same manner as under SYNMON, and most existing programs would not notice any difference.

The cursor controls are as follows:

CHR$(10)	cursor down (line-feed)
CHR$(11)	cursor right
CHR$(12)	cursor home (to top left of 'window')
CHR$(13)	cursor left to start of line (carriage-return)

While this is not a complete set, these controls will be enough for almost all requirements. For example, a practical substitute for the PRINT AT command of some extended BASICs — using cursor-home, cursor-down and cursor-right — is shown in the plotting routine in the program examples in page 18.

'Window'-clear is called by CTRL-SHIFT-N (CTRL-'up-arrow'), CHR$(30). This clears the current window and homes the cursor location, but does not print the cursor. By changing 'windows', it is very easy to blank areas of the screen selectively without recourse to the 'POKEs into the screen within FOR:NEXT loops' programming demanded by SYNMON. The screen can be cleared completely by CTRL-Z, CHR$(26); this also homes the cursor to the top left of the current window.

Programming with multiple 'windows' is simple in principle, but you will probably need a working knowledge of the screen memory map and a little practice in order to use it to its fullest extent. The definition of the current 'window' is held in five consecutive memory locations, from *$0222-0226* ($546$-$550_{10}$); their values are copied into various other locations during printing, scrolling, 'window'-clear and cursor-home operations. The values at start-up are as follows:

Store name and contents	**location**	**C1**	**C1E**	**C2**	**101**
SWIDTH column width (-1)	$*0222* 546_{10}	$*17* 23_{10}	$*2F* 47_{10}	$*3F* 63_{10}	$*2F* 47_{10}
SLTOP low byte of TOP	$*0223* 547_{10}	$*85* 133_{10}	$*80* 128_{10}	$*80* 128_{10}	$*0C* 12_{10}
SHTOP high byte of TOP	$*0224* 548_{10}	$*D0* 208_{10}	$*D0* 208_{10}	$*D0* 208_{10}	$*D0* 208_{10}
SLBASE low byte of BASE	$*0225* 549_{10}	$*85* 133_{10}	$*40* 64_{10}	$*40* 64_{10}	$CC 204_{10}
SHBASE high byte of BASE	$*0226* 550_{10}	$*D3* 211_{10}	$*D7* 215_{10}	$*D7* 215_{10}	$*D3* 211_{10}

These values, and thus the definition of the current 'window', may be changed at any time within a program.

The width, as defined by SWIDTH, may be changed on its own; but the others should be changed as a set, to avoid the risk of printing at random into system memory. Within BASIC, the simplest way of doing this is to define a two-dimensional BASIC array, such as WI(W,X), where W is the 'window' number, and X points to each of the store locations. The 'window' is then set by a GOSUB call, depending on the current value of W:

```
FOR X=0 TO 4
POKE 546+X, WI(W,X)
NEXT
PRINT CHR$(12);     REM - homes the cursor after setting 'window'
RETURN
```

Even on a 1MHz running speed, this will change the 'window' in around 15 milliseconds — less time than a single raster scan on the monitor. A fuller example of the routine's use is given in the program examples on page 18. It's generally best to home the cursor after resetting the 'window', else at least the first character, and probably more, will be printed 'trailing' on the previously-defined text line. Cursor-home (CHR$(12)) resets the text-line pointer to the top of the new 'window'; and screen-clear (CHR$(26)) and 'window'-clear (CHR$(30)) both home the cursor on completion, but do not print it. If you want text to start further down, you can either print a string of line-feeds (CHR$(10)), or reset the text-line pointer separately. Its two locations are:

LTEXT	*$022B*	555_{10}	low byte of text-line start
HTEXT	*$022C*	556_{10}	high byte of text-line start

The width stored in SWIDTH is *one less* than the number of characters printed per line. The value *must* be less than the nominal width of the screen memory (32_{10} on C1, 64_{10} on all others), or else the scroll will not perform correctly (try it!). Above a value of 127_{10} in SWIDTH, the screen handler will not function at all — if you want to experiment, be careful! SWIDTH also determines the scanning width of the edit cursor — see the section on *Editing* later.

In setting the TOP and BASE pairs, check first that BASE *is* below TOP on the screen (i.e. has a *higher* memory address), and that they are in line with each other on the memory map. If BASE is higher than TOP on the screen, the result will be a two-line scrolling window starting at TOP. If TOP and BASE are out of line with each other, the column width will start as defined by TOP, and all will appear to function normally, except that scroll and 'window'-clear will finish one line higher or lower than BASE.

Note that scrolling and clearing operate on TOP and BASE *inclusive* — TOP thru BASE. A clear, for example, will run from TOP to BASE+SWIDTH *inclusive*, in a column SWIDTH+1 characters wide.

No check is made to ensure that TOP and BASE are within the screen memory — they can print to anywhere within memory. While this does mean that their values must be set with care — as with any POKE instruction — it also means that any memory-mapped device can be accessed by the screen handler, and can be PRINTed to by BASIC, by defining a 'window' in the same way. Examples of this are in PRINTing colour values direct to the C4's colour memory, or PRINTing to an additional display or high-resolution graphics board.

The print speed before scrolling occurs is extremely fast — around 250 cycles per character, giving a screenful from BASIC in well under a second even on a 1MHz C2. The scroll speed for a full screen width is almost identical to SYNMON's, but proportionately faster as the width of the column to be printed is reduced. Printing speed can be slowed as before by POKEing 518_{10} ($0206) with a delay value from 1 to 255.

Machine-code monitor

CEGMON's machine-code monitor has been designed specifically to simplify the development of short machine-code routines, especially those intended to link to BASIC. While it is not as comprehensive as OSI's Extended Monitor (ExMon), it can be co-resident with both BASIC and Assembler, and is immediately available without loading tape — a great advantage for educational users. We expect it to be most useful for developing short routines of up to thirty or so lines, and for testing and debugging larger routines being developed with the Assembler.

This section does assume a basic knowledge of the principles of machine-code programming for the 6502 (the monitor will provide the practice!). Three useful reference books are:

Programming the 6502, Rodnay Zaks (Sybex) — good and mostly complete introduction to the 6502, but earlier editions had a few important inaccuracies.

6502 Assembly Language Programming, Lance A. Leventhal (Osborne/McGraw-Hill) — solid, full of examples of programming for internal routines and I/O through

PIAs, VIAs, ACIAs etc., in assembly mnemonics. Beware the unusual page-numbering!

6502 Software Design, Leo J. Scanlon (Blacksburg) — many practical examples, mostly based around the AIM65 system, but still useful.

On start-up via 'M', the monitor's prompt — a '>' — appears after the screen is cleared. At this point the monitor is in its address/command mode, and normally expecting input from the keyboard. The commands available are:

/ jump to data mode, leaving current address unchanged.
. 'do nothing' — loop back to get address.
L sets load flag — calls for input from the BASIC load vector at *$FFEB*.
S save machine code.
M do memory block move.
T do tabular dump/display of memory contents.
Z set a breakpoint.
R restart from a breakpoint.
U jump to user routine.

If none of these are given as a reply, the system expects *four* hex digits to make up an address (see *Error-handling* later). On completion of the address (the 'current address'), the system prints a '/', then the contents of that address as a hex pair, and a space. The system is then in the data mode loop, and the following commands are available:

. return to address mode.
/ re-open current address, to correct a mis-type.
G start execution at the current address.
' enter text entry loop.
, increment current address.
LF (line feed) — increment current address, do CR/LF, display new current address and contents on next line.
CR (carriage return) — as for LF, but do CR only; display by overwriting on same line.
^ (up-arrow, SHIFT-N) — as for LF, but *decrement* current address.

Otherwise the system expects both digits of a hex pair, stores the complete byte at the current address, and loops back to the start of the data mode for a new command or value.

Command/address mode:

/ *jump to data mode*
On start-up the current address is set to *$0000*; thereafter it is not changed on a restart, such as on an error recovery.

L *load*
The machine-code load flag at *$FB* is set; the system then restarts at the beginning of the data mode loop. It then expects input from the ACIA — either from tape or from an RS-232 serial interface — in the old SYNMON format of '.' to define address, '/' to define data, data transmitted in the form of hex pairs separated by a CR, and concluded by '.', an address and 'G' for an auto-start. The only difference from SYNMON's load is that the former contents of the current address are displayed on the current line as well as the address and its new contents. The load loop can run at up to 4800 baud on a 1MHz machine.

Note that normal error-checking is disabled during load — see *Error-handling* below — and that the system will not accept input from the keyboard until the load flag is cleared, either in your program, or by a re-entry into the monitor's command-mode. As with loading tapes for BASIC or Assembler, loading can be halted by hitting the SPACE bar; the monitor then restarts in its command mode, with normal error-checking resumed.

S *save*

Syntax: *.Saaaa,bbbb>cccc* where *aaaa* is the start address of the code to be saved, *bbbb* is the last address *inclusive* of the code, and *cccc* is the restart address — either to the beginning of the routine for autostart, or to the monitor for further work (see *List of locations, routines and subroutines* later). Code is saved from *aaaa* to *bbbb* inclusive — *aaaa* thru *bbbb*; the routine automatically provides the ',' and '>' prompts. It then waits until the RETURN key is pressed on the keyboard — to give you time to start your recorder — and then prints out the code in the SYNMON load format. (It will start after any key is pressed; but RETURN is advised, since it will also output ten nulls to the tape before the actual 'save' starts). The 'start' and 'go' addresses and hex codes are displayed on the screen; the CR which separates each data byte, however, is output direct to the ACIA. As a result, *this routine is not fully vectored for user-defined output* — it can only be used through the ACIA, to cassette port or RS-232 interface.

On completion of the save, the BASIC save flag is cleared, and the system restarts in the command mode.

M *memory block move*

Syntax: *.Maaaa,bbbb>cccc* where *aaaa* is the start of the code to be moved, *bbbb* is the end inclusive — *aaaa* thru *bbbb* — and *cccc* is the new start location. The routine does not erase the code at the previous locations, though it may over-write it if the new locations overlap the old. *If the new start is between the old start and end addresses, it will over-write the remaining code before it has been copied* — if you need to do this kind of move, copy to a 'safe' area first, and then copy back to the new area.

T *tabular display*

Syntax: *.Taaaa,bbbb* where *aaaa* is the start address of the code to be displayed, and *bbbb* is the last address *inclusive* — *aaaa* thru *bbbb*. The ',' prompt is supplied by the routine. The contents of the memory are displayed as a table of eight-byte blocks (sixteen-byte blocks — C2), each block preceded by the address of the first byte of the block (on C1s, the address is printed between each line of the table). If you want to display more than a screenful, it's advisable to slow the print speed down by placing a delay value in *$0206* before calling the *T* routine; to send the display out to a printer, set the OUTVEC 'save' flag at *$0205* to *$FF* before you start.

On completion, the system restarts in the command/address mode, displaying the '>' prompt.

Z *zero — set a breakpoint*

Syntax: *.Zaaaa* where *aaaa* is the address at which the breakpoint is to be inserted. *Z* sets up at *$01C0* (OSI's IRQ/BRK address) the pointer to CEGMON's breakpoint handler; saves the current contents of the chosen address in BRKVAL; and replaces it with a BRK opcode (*$00*). Note that a breakpoint cannot be set at any ROM address! — see *Using breakpoints* later.

The routine then exits back to the command mode, displaying the '>' prompt.

R *restart from breakpoint*
Collects its start address and the contents of the registers, processor status and stack pointer from the break-table, and restarts the program at that address by executing an RTI instruction — see *Using breakpoints* later. Inappropriate use of *R* will usually cause a system hang-up or crash — it should *only* be used to restart from a break-point.

U *jump to user routine*
Causes the system to 'jump-indirect' to a routine whose start address is held in *$0233-34* — the low byte of the address in *$0233*, the high byte in *$0234*. Useful for calls to regularly-used locations like the Assembler restart, or where the 'current address' should be left unchanged.

Data mode:

. *exit to command/address mode*
When in the data mode, '.' *must* be typed before calling for any of the command mode's commands or for a new address. If it is forgotten, the command mode's command letters will be treated as errors; while the intended new 'address' will be treated as two hex pairs, the second overwriting the first at the unchanged current address!

/ *re-open current address*
Leave current address unchanged, to place a new value at the current address — used if the value just typed was incorrect.

G *'go'*
Sets all registers to *$00*, and starts execution at the current address. Usually used with the syntax *.aaaaG* — but make sure that the '.' command precedes the *aaaa* address!

' *start text mode*
The text mode expects ASCII text rather than hex digits. Control characters such as the 'window'-clear and cursor controls, and also graphics characters, can also be typed direct into memory. Where the text is to be printed to screen later via the BASIC output vector (OUTVEC), errors may be corrected by RUBOUT, but both it and the character it 'deletes' will be stored in memory; otherwise no editing is possible without exiting back to the data mode. Each new character is stored directly and the current address is incremented.

A second ' ' will exit back to the data mode on the same line; a ',' will be printed after it, for clarification, but the current address will not be further incremented. The text-entry mode can also be exited by typing a CR (carriage return); this returns to the data mode, but printing on the *next* line, displaying the updated current address and its contents.

, *increment current address*
Used to space succeeding entries into memory. If more than one ',' is typed, the current address will be incremented accordingly, and the contents of the 'skipped' addresses will be left unchanged.

LF *line-feed - increment current address, display on next line*
This is the same as on OSI's ExMon — the current address is incremented, a CR/LF (carriage-return/line-feed) is issued, and the new current address and its contents are displayed on the next line.

CR *carriage-return - increment current address, display on current line*
This differs from OSI's ExMon, where CR is used to exit to the command mode. Its use here is mainly to allow fast tape load without scrolling; it is identical to LF, except that a carriage-return only (without line-feed) is issued. See also the use of CR in the ''' (text) mode above.

^ *up-arrow (SHIFT-N) - decrement current address, display on next line*
This is the same as in ExMon. The routine is identical to LF, except that the current address is decremented rather than incremented.

Using breakpoints
Breakpoints are a useful part of the debugging toolkit for machine-code work. They force the program execution to halt, rather like the STOP command in BASIC. In CEGMON's case, the halt then presents the system registers for view and alteration as required.

In the 6502 processor, the breakpoint is forced when the processor executes a BRK instruction, opcode $00; so breakpoints are set by overwriting an existing instruction with a BRK opcode. CEGMON does this with the *Z* command: a $00 is stored at the chosen address, and the current contents are saved, to be restored by the breakpoint handler when the breakpoint is hit. There are two restrictions on setting breakpoints: first, that the BRK instruction must replace an *instruction* byte, since if it is placed in the data or address bytes of an instruction, it will simply be treated as part of that data or address; and second, that since the breakpoint is set by replacing the existing instruction byte with BRK, breakpoints cannot be set at ROM addresses. Only one breakpoint can be set at a time, and is automatically cleared by the breakpoint handler after hitting the breakpoint.

To test a program by using breakpoints, set a breakpoint at a likely location, and start the program running with a *.aaaaG* or *U* command. If nothing different happens, or if the program hangs up, either the breakpoint was never reached, or was set incorrectly and interpreted as data or address. When the program hits the breakpoint, a check is made that this is a BRK and not an IRQ interrupt (which is ignored); the registers and corrected program counter (see Zaks, p.111, 235; Leventhal, 14-2, 3) are saved in a table; the previous opcode is restored at the breakpoint, replacing the BRK opcode; and the routine then jumps to the monitor data-mode loop, pointing to the beginning of the break-table. What you will see on the screen is a CR/LF done, followed by

00E0/aa

where aa is the contents of the A register at the time the breakpoint was reached.

$00E0 is the beginning of the break-table — the same as ExMon's. You can then use the data mode loop to examine and/or modify the registers and program counter. They are stored as follows:

00E0 A register — accumulator
E1 X register
E2 Y register
E3 P register — processor status flags, in hexadecimal form
E4 K register — stack pointer
E5 PCL — low byte of program counter
E6 PCH — high byte of program counter

The address shown by *E5* and *E6* should be the same as the breakpoint address that you set; if not, you have a loose BRK in your program somewhere!

While in the data mode, you can change these values; you can also exit as usual to the command mode to set another breakpoint — or reset this one — with the *Z* command. When you've finished, and want to restart, type .R (don't forget the '.'!). This collects the registers' values and the program counter from the break table, and restarts execution. If you've only looked at the break table, without changing any values, the program will simply carry on where it left off, as if nothing had happened, and will do so until it finds another breakpoint or reaches its own conclusion.

One minor problem does occur when testing programs with the Assembler still in memory, such as after an A3 assembly. The Assembler uses BRK as 'return to command-mode' statement; setting a breakpoint via CEGMON's *Z* command will over-write the Assembler's own jump and cause it to 'return' to CEGMON when you restart it after testing your routine. If you are working with the Assembler, note down the contents of *$01C0-01C2 before setting any breakpoint with Z*, and restore them before restarting the Assembler.

Error handling

CEGMON's error handling in the machine-code monitor is similar to that in OSI's ExMon. In the command mode, only the *first* letter after the '>' prompt or a '.' command may be a command letter; thereafter, only hex digits are allowed until a complete four-digit hex address is built up. The same applies within the commands themselves — only complete four-digit addresses are allowed, as the syntax given for each command shows. (Note that the system supplies any ',' or '>' prompts). Within the data mode, excluding its text input mode, the same applies: the *first* character in each time round the loop after a previous instruction or value may be an instruction, such as ',' or LF; thereafter, the system expects hex digits to make up hex pairs. A '.' or '/' may be typed at any time, to exit back to the command or data modes (but note that this will leave a part-complete address or data-byte only partly rotated into place, and almost certainly incorrect). *All other characters are invalid.* Within the text entry mode, *no* characters are invalid; the only restricted characters are ''' and CR, which exit back to the main data mode loop.

During text entry from the keyboard, invalid characters will be printed, but immediately followed by an '?' and CR/LF, and the restart '>' prompt. You are then back in the command mode. The current address, however, is unchanged, as can be seen if you then re-enter the data mode by typing a '/'. Control characters like CTRL-Z — to clear the screen — are recognised, but are decoded as errors on completion; CTRL-Z clears the screen, but a '?' and the '>' prompt are then printed.

During a tape load, this *error checking is disabled.* The inevitable 'glitch' characters that precede the start of each record on cassette would halt the load before anything had been loaded if this was not done. This does leave the load open to errors. However, if a digit is invalid it is simply ignored; the following CR bumps up the current-address counter as normal, and only the contents of that address are affected. If a CR is lost, the addresses will be out of step with the data. Normal error-checking is resumed only when the load-flag at *$FB* is cleared. As mentioned earlier, this is done automatically by the system at the entry to the command mode, and also if the SPACE bar is hit during load; it is also cleared by a BREAK reset. If your program auto-starts without entering the monitor's command mode, you will need to clear the flag by storing a null (*$00*) in it.

On short programs a load can be checked simply with the *T* tabular display; on

larger programs a checksum loader should probably be bootstrapped in, as on OSI's ExMon and Assembler — although we have found OSI's own checksum loader to be less reliable than a straight load! The digit-by-digit load format is surprisingly reliable with a reasonable tape and tape recorder: when testing CEGMON we used it to save the Assembler and the CEGMON source code — more than 20K in all — and re-loaded it at 4800 baud into a C2, with no detectable errors.

Editing

Two types of editing are available within CEGMON: limited in-line editing, using RUBOUT; and screen editing, using a second cursor. The backspace or RUBOUT routine is linked mainly to CEGMON's screen-handler; the screen editor replaces the call to the keyboard routine in the BASIC input vector's routine, and the keyboard is called through it.

The screen-editor is available for any program which calls for input via the input vector at *$FFEB* — which includes not just BASIC but OSI's Assembler and ExMon as well. However, neither form of editing is available in CEGMON's machine-code monitor: the keyboard routine is called direct, in order to maintain compatibility with SYNMON and to avoid internal incompatibility problems; while RUBOUT is treated simply as another invalid character in both command and data modes. In the data mode, you can re-open the current location with '/'; otherwise, if you see a mistake, it's simplest either to restart the command or data by typing '.' or '/', or else to crash it deliberately by hitting the SPACE bar, thus restarting in the command mode. Editing isn't really practical or necessary in the monitor; and even the longest command is only just over a dozen keystrokes, after all!

As mentioned earlier, RUBOUT is actually decoded as *$5F* (ASCII underscore) rather than *$7F*, the true ASCII DELETE. On pressing RUBOUT, the character immediately to the left of the cursor is deleted, and the cursor backspaces into its space; if the beginning of the line is reached, the cursor will skip to the end of the previous line, as defined by SWIDTH. In principle, it will continue doing this as long as RUBOUT is pressed, until it reaches the beginning of the top line of the current 'window', where it will stop. In practice, when linked to BASIC, it will backspace up to the beginning of the current program line, or to the '?' of an INPUT request from BASIC.

At the same time, it will normally delete the last character typed in BASIC or the Assembler. But there are some difficulties, particularly in BASIC, because OSI's BASIC-in-ROM was originally intended for use with Teletypes, not video systems — hence the careful storing of the character count and terminal width in 14_{10} and 15_{10} respectively. These two locations, and the 'hangover' from Teletype days, cause two problems with RUBOUT in BASIC.

The first comes if you overshoot the buffer by typing more than 72_{10} characters in one go — you then see a 'half-battleship' on OSI systems. This is CTRL-G, the ASCII BELL code. On Teletypes, this is a non-printing character — it rings the bell. But on video systems — all those involved here — it *is* printed, but *not* stored in the buffer, since there is no room for it. RUBOUT will then delete the last valid character from the buffer — the last before any 'half-battleships' — but will delete the last 'half-battleship' rather than the relevant character from the screen: the backspace will be out of step with the true delete. Getting round this problem completely would cost about as much code as the entire editor, and would make editing incompatible with

anything other than BASIC — and we prefer to keep CEGMON as flexible as possible. If you get caught out in this way, by over-running the buffer, use RUBOUT with care; then LIST the relevant line, and edit it as required with the screen editor.

The other problem is more a matter of habit, particularly for owners of Superboards or UK101s with overscan problems on their display. If you've been in the habit of limiting the display width by limiting the terminal-width value (i.e. POKEing 15 with, say, 22 rather than the default value of 72) — DON'T! Limit the display width by POKEing the same value into SWIDTH — i.e. POKE 546, 22. The reason is the same as with the BELL code: if you limit the terminal width, the backspace will get out of step with the true delete, because while the characters displayed are linked to terminal-width, the backspace is necessarily linked to SWIDTH. As long as the terminal-width value in 15 (*$0F*) is *greater* than SWIDTH, no problem will arise. 'Terminal width' should still be used to define the line-length of output to a printer or terminal.

The screen editor is directly linked to all the usual calls for keyboard input. The editor itself is turned on or off by typing CTRL-E; when turned off, the keyboard returns a character exactly as under SYNMON, as though the editor did not exist. Whenever the editor is turned on from the keyboard, the edit cursor — a white square — appears at the beginning of the current line. The editor can also be turned on or off within programs by POKEing the edit flag at 516_{10}, *$0204* — POKE 516, 255 turns the editor on, POKE 516, 0 turns it off. You can also set the start-position of the edit cursor within programs by POKEing values into its store locations: CURSLO, at 561_{10} (*$0231*), holds the low byte of its address, and CURSHI at 562_{10} (*$0232*) holds the high byte. The edit cursor can be moved around on the screen with the following keys:

CTRL-A	(CTRL-a)	left
CTRL-S	(CTRL-s)	up
CTRL-D	(CTRL-d)	right
CTRL-F	(CTRL-f)	down

While editing, the values of these are not sent out to the calling routine (e.g. to BASIC) at all — they are solely used to move the edit cursor. The cursor 'wraps round' if it is moved off the screen vertically, and is not limited to the height of the current window; its movement horizontally, however, is limited to the width of the current 'window' as defined by SWIDTH, because of yet more difficulties caused by BASIC's terminal-width values. For this reason, you'll probably find it best to set SWIDTH to the full visible screen width when editing.

As you move the edit cursor around, the character beneath it at each moment is stored, and replaced when the cursor moves away. If you then press the ESC key (or CTRL-Q on UK101), the character 'beneath' the edit cursor is copied to the main cursor, and sent out to BASIC, the Assembler or whatever; both cursors then move one place to the right. Entire lines — up to the limit of the BASIC or Assembler input buffers — can thus be copied by ESC from anywhere on the screen.

Since any key other than the editor's control-keys can be typed as usual, being printed by the main cursor, a wide variety of editing is possible. A mistyped line can be copied up to the mistake; the mistake corrected by typing; the old mistake skipped by the edit cursor; and the remainder of the line copied again. A mistakenly deleted line can simply be copied back into the program as long as the line is still on the screen. Lines can be renumbered or re-ordered simply by re-typing the line number and copying the rest (though the old line will still be there until it is

deleted). A program that has gone 'out of memory' can be copied line by line, converted to multiple statements wherever possible, and skipping every unnecessary space. And by changing the 'window'-definitions — as is shown in the program examples — a group of lines can be LISTed, protected in a non-scrolling area, and copied as required into a new program. Program conversions between different machines become simple, too, for only the machine-specific parts — usually screen and keyboard values and a few POKE locations — need be changed in each line.

Input/output

Except in the machine-code monitor, all input and output in CEGMON is through BASIC's vectors at *$FFEB-FFF9*. Since these all call their routines via JMP-indirect calls through further vectors stored in a table in page 2, from *$0218-0221*, any special input or output routines for printers or special programming purposes can be linked directly to BASIC or elsewhere by changing the vectors in the table in page 2. The program examples show two routines — a program to skip BASIC's masking of control and graphic characters on input; and a TRACE routine called by BASIC, via its CTRL-C check vector, between the execution of each BASIC statement.

In the machine-code monitor, neither load, save nor keyboard input are vectored, simply to maintain compatibility with OSI's original system. All output to the screen, however, does go through the BASIC output vector, and can be sent to a printer or whatever simply being setting the BASIC 'save' flag at *$0205* to *01*.

The input and output vectors are as follows:

Page $FF vector		**through**	**normally points to**		**locations/contents in decimal**	
INVEC	*$FFEB*	*$0218-19*	INPUT	*$FB46*	536 - 70	537 - 251
OUTVEC	*$FFEE*	*$021A-1B*	OUTPUT	*$FF9B*	538 - 155	539 - 255
CCVEC	*$FFF1*	*$021C-1D*	CTRLC	*$FB94*	540 - 148	541 - 251
LDVEC	*$FFF4*	*$021E-1F*	SETLOD	*$FE70*	542 - 112	543 - 254
SVVEC	*$FFF7*	*$0220-21*	SETSAV	*$FE7B*	544 - 123	545 - 254

Compatibility and conflicts

OSI's system software, as represented by its BASIC, Assembler, ExMon and SYNMON, was clearly not designed as a system at all. The Assembler and BASIC cannot be co-resident, and ExMon's disassembler crashes BASIC by overwriting the tail-end of BASIC's all-important memory scan subroutine (from *$00BC* to *$00D3*). BASIC's keyboard buffer starts at *$0013*, the Assembler's at *$0080*. And so on. In designing CEGMON, one of our main concerns was to build a monitor that could co-exist with all of these conflicting requirements and still contain the kind of features we wanted.

Apart from the difficulties over BASIC's terminal width and character-counter, the other major problem is with the use of zero-page stores. Apart from SYNMON's five locations at the top end — *$FB-$FF* — only one pair is reasonably 'safe', and even that is used as a temporary store by the Assembler. The *$E4-E5* pair is thus used in

four ways by CEGMON: as a temporary pointer for the edit cursor, but only during each call to the keyboard; as a temporary store for the 'go' address of the machine-code save, but only during the actual save; as the store for the 'new start-of-block' address during a memory block move; and as part of the break-table, for the stack-pointer and PCL contents. This is another reason why the keyboard is called direct by CEGMON's machine-code monitor. The only time when the conflict may be important is when you are both testing and saving a routine with the Assembler still in memory, such as after an A3 assembly to memory. The *$F9-FA* pair is also used by CEGMON as a temporary store for addresses during screen-clear, tabular display, save and block move.

The other important point is that CEGMON uses some of the old 'free RAM' starting at *$0222*: up to *$022E* for the screen handler's store-locations and subroutines, and to *$0232* for the editor's stores. The monitor *U* command jump vectors through the *$0233-0234* pair, so the 'free' RAM under CEGMON starts at *$0235*, 565_{10}. Programs written to start at *$0222* can still be run on CEGMON, though only with the editor disabled and the old screen-handler called instead of the new.The edit flag at *$0204*, 516_{10} *must* contain 0, and CTRL-E must not be typed at any time, or the edit-cursor will be written at random into memory, probably causing a program crash. The old screen handler may be called by changing the output vector: POKE 538, 149 enables output through the old screen handler at *$BF2D* on OSI machines, while POKE 538, 155 enables output through the new routine. (When returning to the new handler, it's a good idea to home the cursor with CHR$(12) or one of the screen-clear calls). Note that a BREAK reset will not only reset the vector to point to the new screen handler, but also over-write the RAM area up to *$0234* — so you may need to disable BREAK as well. No problem will arise with either the Assembler or ExMon, since the cassette versions of both of these start higher up in memory.

Under the OS-65D disc operating system, the entire RAM from *$0200* is used; 65D has its own I/O routines, and ignores those normally used by BASIC-in-ROM. CEGMON contains a bootstrap to boot up 65D, but from then on 65D is self-contained, and CEGMON's special features like the editor and screen handler are ignored. Patches for 65D, to enable it to use CEGMON's editor and screen-handler, will be made available shortly.

Locations, routines and subroutines

The following is a list of various useful points within CEGMON. As can be seen, the locations of the SYNMON equivalents have in general been retained; but note that in most cases the way in which they work will be somewhat different — output to display in the machine code monitor goes via the screen-handler rather than direct to screen memory, for example. One other important point is that, within the machine-code monitor, the Y register is always reset to zero, and most of its routines assume this to be the case — beware of this if you use these routines in your own programs.

Locations

For break-table locations — *00E0-E6* — and their functions, see p.8.

BTABK	*00E4*	part of break-table, but also used as a pair to store 'go' address in during
BTABCL	*00E5*	save; new block start address in move; and edit cursor location during each call to keyboard.

BRKVAL	*00E7*	store for opcode moved by *Z* when setting a breakpoint.
LOTO	*00F9*	store for 'to' addresses in save, move and tabular display — see NOTEND.
HITO	*00FA*	
STORE	*00FC*	store for current data during data mode and most other routines.
LOFROM	*00FE*	store 'current address' for most routines — the 'from' address in save, move
HIFROM	*00FF*	and tabular display.
DOBRK	*01C0*	location for IRQ/BRK jump; set to 'JMP $FA4F' by *Z*.
CURDIS	*0200*	cursor displacement on current line.
OLDCHR	*0201*	stores current character during SCREEN; exits containing char 'beneath' the cursor.
NEWCHR	*0202*	park for new char for SCREEN.
LDFLAG	*0203*	BASIC load flag: *00* - no load; *FF* - load from ACIA.
EDFLAG	*0204*	EDITOR flag: *00* - disable edit cursor; *FF* enable edit cursor.
SVFLAG	*0205*	BASIC save flag: *00* - skip save; *01* - enable save to ACIA.
SDELAY	*0206*	print-delay value for SCREEN; delay is delay-value times approx. 400 machine-cycles (i.e. times 400 micro-seconds at 1MHz).
COUNTR	*0214*	auto-repeat counter for GETKEY.
SCRTCH	*0215*	returns from GETKEY with final ASCII value of key.
LSTCHR	*0216*	pre-shift value of last key left here by GETKEY to test auto-repeat.
CCFLAG	*0212*	BASIC CTRL-C flag: *00* - enables CTRL-C break; *01* disables CTRL-C break.
DISP	*022F*	edit-cursor displacement from start of editor's current line.
CURCHR	*0230*	store for char 'beneath' edit cursor.
CURSLO	*0231*	contain start of edit cursor's current line on screen.
CURSHI	*0232*	
USERLO	*0233*	contain location of start of user routine called by machine-code monitor's
USERHI	*0234*	*U* command.

Main entry points

RESET	*FF00*	start of BREAK/RESET routine.
NEWMON	*FE00*	'reset' entry to m/c monitor — reset stack, vectors/pointers, clear decimal mode. Recommended re-entry point for auto-load m/c tapes.
MENTRY	*FE0C*	non-reset entry to m/c monitor — clear screen, zero 'current address'.
MSTART	*F97E*	entry to command/address mode.
DATALN	*FA2E*	entry to data-mode loop — prints 'current address' and its contents.
SAVEMC	*FA7E*	start of m/c save routine.
DISK	*FC00*	entry to disc bootstrap (at *F700* on C2).

Subroutines

INPUT	*FB46*	BASIC input routine — get a char from keyboard or ACIA.
OUTPUT	*FF9B*	General output routine, to SCREEN and ACIA.
OLDSCR	*FF95*	As for output, but screen-handling done by *$BF2D* rather than SCREEN.
TENULL	*FFAC*	outputs ten nulls to ACIA.
CTRLC	*FB94*	BASIC's CTRL-C check — called between execution of each BASIC statement.
SETLOD	*FE70*	sets BASIC load flag, clears save flag; *decrements* load flag to set it.
SETSAV	*FE7B*	sets BASIC save flag.
TAPIN	*FB57*	collects char from ACIA; exits via EDITOR if SPACE hit.
TAPOUT	*FCB1*	output to tape (*BF15* on C2).
RSACIA	*FCA6*	initialise ACIA (*BF22* on C2).
SCNCLR	*FE59*	clear entire screen; exits with X and Y registers zero.
SCREEN	*F836*	new screen handler.
ENDCHK	*FBCF*	checks if top or base of screen overshot — if Y=0, carry clear if top overshot, if Y=2, carry set if base overshot.
SCOUT	*FF8C*	print char at cursor location.
CURHOM	*FFD1*	resets TEXT line pointer to TOP; do STX *$0200* to reset cursor at TOP.

EDITOR	*FABD*	entry to screen editor — see main text, p.11.
GETKEY	*FD00*	wait till key pressed, return with ASCII value in A register.
KEYWRT	*FCBE*	write-to-keyboard invert for C1 (*F7BE* on C2).
KYREAD	*FCCF*	read-A-from-keyboard invert for C1 (*F7CF* on C2).
KEY2XR	*FCC6*	read-X-from keyboard invert for C1 (*F7C6* on C2).
KDELAY	*FCDF*	approx. 6500 cycle delay; exits with X and Y registers zero (*F7DF* on C2).
DELAY2	*FCE1*	approx. (400 × Y-register) cycles delay (*F7E1* on C2).
TRIQAD	*FFBD*	collect three addresses: first stored in (FE) pair, second in (F9), third in (E4).
TWOQAD	*F9A6*	collect two addresses: first stored in (FE) pair, second in (F9).
GETQDE	*F9B5*	collect address, store in (FE). Note: call GETNEW first!
GETPRC	*F9BE*	collect hex pair for data byte, store in FC. Note: call GETNEW first!
GETNEW	*FE8D*	get new char; print it to display before returning.
GETCHR	*FEE9*	get char from keyboard or ACIA.
MCACIA	*FE80*	get char from ACIA, strip off any top bit before returning.
ASCHEX	*FE93*	strip ASCII digit to hex; set to 80_{16} if not hex.
ROLSTR	*FEDA*	roll new nibble into (FE) if X=2, or into FC if X=0.
ADVTOD	*FEAC*	print address in (FE), space, value in FC to display.
QDDATD	*FEB6*	print address in (FE) to display.
PRDATD	*FEBD*	print data byte in FC to display.
HEXOUT	*FECA*	strip byte in A register to lower nibble; print nibble as ASCII hex to display.
PRBYTE	*FEF0*	print data at 'current address' pointed to by (FE) to display. Assumes Y=0!
CRLF	*FBF5*	print carriage-return/line-feed to display.
SPCOUT	*FBE6*	print ASCII space to display.
BUMP	*FEF9*	increment 'current address' at (FE).
NOTEND	*FBEB*	compare (FE) with (F9); carry clear if (FE) is less.
SWAP	*FDE4*	memory block move. Expects start address in (FE), end address in (F9), new start of block in (E4); assumes Y=0.

Program examples and ideas

Routine to bypass BASIC input masking

As mentioned earlier, BASIC's input routine masks out most of the control and graphic characters. If want to be able to send cursor controls, graphics for display and the like to BASIC — from keyboard or tape — BASIC's masking has to be bypassed. A USR call to the keyboard, as the next example shows, bypasses the input routine entirely; while this example changes the input vector to point to an 'unmasking' routine, allowing input of the 'non-standard' characters from tape.

BASIC calls for input by a subroutine at *$A357*, which itself calls another subroutine at *$A386* — the actual call for input through the input vector. The masking is in the first subroutine, *$A357*. The routine which follows thus 'throws away' these two subroutine levels as soon as it is called, and then does their work of collecting a BASIC line and placing it in the input buffer before returning to BASIC.

It is shown here as if typed in with CEGMON's monitor, starting at *$0240*, each line being followed by its mnemonic and comment.

To use the routine from BASIC, two POKEs are needed. POKE 536, 64: POKE 537, 2 points the input vector to this routine, while POKE 536, 70: POKE 537, 251 returns input back to the normal routine.

```
0240/24 68          ;PLA
0241/24 68          ;PLA - bypass one RTS call
0242/24 68          ;PLA
0243/24 68          ;PLA - bypass second RTS call
0244/24 4C,53,02    ;JMP to start of loop
0247/24 20,E5,A8    ;JSR $A8E5 - for backspace here
024A/24 CA          ;DEX - decrement char-count after backspace
024B/24 10,08       ;BPL +8 - to get new char if count not zero
024D/24 20,E5,A8    ;JSR $A8E5 - for @ line-delete
0250/24 20,6C,A8    ;JSR $A86C - for CR/LF
0253/24 A2,00       ;LDX #$00 - reset char counter
0255/24 20,46,FB    ;JSR $FB46 - get char from keyboard or ACIA
0258/24 C9,01       ;CMP #$01 - to mask off nulls from tape
025A/24 90,F9       ;BCC -7 - get another char if null
025C/24 20,99,A3    ;JSR $A399 - check for ctrl-O
025F/24 C9,0D       ;CMP #$0D - carriage-return?
0261/24 F0,1C       ;BEQ +28 - exit if CR
0263/24 C9,0A       ;CMP #$0A - line-feed?
0265/24 D0,04       ;BNE +4 - skip next check if not LF
0267/24 E0,00       ;CPX #$00 - first char in line?
0269/24 F0,EA       ;BEQ -22 - if yes, ignore; get new char
026B/24 C9,40       ;CMP #$40 - @
026D/24 F0,DE       ;BEQ -34 - do line-delete if @
026F/24 C9,5F       ;CMP #$5F - backspace
0271/24 F0,D4       ;BEQ -44 - if yes, do backspace
0273/24 E0,47       ;CPX #$47 - char count exceeds buffer?
0275/24 B0,0B       ;BCS +11 - skip store-in-buffer if full
0277/24 95,13       ;STA $13,X - else store in BASIC line buffer
0279/24 E8          ;INX - increment char counter
027A/24 20,E5,A8    ;JSR $A8E5 - print the char
027D/24 D0,D6       ;BNE -42 - always jump, to get new char
027F/24 4C,66,A8    ;JMP $A866 - do CR/LF, exit to BASIC
0282/24 A9,07       ;LDA #$07 - the 'bell' char
0284/24 E0,FE       ;CPX #$FE - check char count not too great!
0286/24 F0,C5       ;BEQ -57 - if yes, cancel line
0288/24 D0,EF       ;BNE -17 - else inc. counter, print 'bell'
```

Non-halting 'GET'from the keyboard

CEGMON's keyboard routine will wait until a key is pressed before returning with its value; but there are many cases where this halt is a nuisance — such as in real-time games or in foreground/background work. The BASIC routine below copies the first part of the keyboard routine into page-2 memory, and then modifies it to bypass the halt, delay and auto-repeat loops. When called by a USR(X) statement in BASIC, it returns the ASCII value — or a null if no key was pressed — to BASIC via its INVAR routine at *$AFC1*.

Once loaded, the loader can be deleted by NEW; the 'GET' routine is unaffected by a BASIC cold-start, but the two POKEs to set up the USR call (in line 100) would

have to be replaced. Lines 200 to 220 give a very brief example of the routine's syntax and use.

```
 10 FOR X=1 TO 80:M=PEEK(64767+X):POKE (575+X),M:NEXT
 20 FOR X=1 TO 3:READ M:POKE (633+X),M:NEXT
 30 DATA 76,208,253          :REM - JMP $FDD0
 40 FOR X=1 TO 15:READ M:POKE(655+X),M:NEXT
 50 DATA 141,19,2            :REM - STA $0213
 60 DATA 76,110,253          :REM - JMP $FD6E
 70 DATA 32,64,2             :REM - JSR $0240
 80 DATA 168,169,0           :REM - TAY, LDA #$00
 90 DATA 76,193,175          :REM - JMP $AFC1
100 POKE 11,150:POKE 12,2
200 P=USR(X):IF P THEN P$=CHR$(P):GOTO 220
210 PRINT,"Nothing":GOTO 200
220 PRINTP$,,"Something":GOTO 200
```

TRACE for BASIC

In debugging BASIC programs it's useful to know what statements are being executed when, or even executed at all. A 'trace' routine can sometimes help in this. Since BASIC checks between each statement for a CTRL-C break, the following routine uses the CTRL-C vector to print out the current line number before the check is made. When this 'trace' is turned on, BASIC prints out 'IN (line number)' between each statement.

A couple of points should be noted. One is that the routine can only be turned on by a *single* POKE — otherwise BASIC would become 'lost' between the two POKEs — and thus only the high-order part of the vector address is changed. If you move the routine from its location here — so that it can co-reside with the BASIC 'GET' above, for example — it *must* start in memory with its low-order address byte the same as that of the CTRL-C routine in CEGMON, at *$xx94*.

The other point is that the 'trace' will print out *every* line number it sees. In a ten-thousand FOR:NEXT timing loop, it will print that line number ten thousand times — though the line number of the NEXT will only appear when the last loop is done. In these cases, other line numbers will be 'buried' in amongst the mass of the loop number; so turn the 'trace' off with a POKE 541,251 before entering these loops, and turn it on again afterwards. The 'trace' also takes an appreciable time for each line number, and should not be used where timings are critical.

```
10 FOR X=660 TO 669: READ R: POKE X,R: NEXT
20 DATA 169,255     : REM - LDA #$FF
30 DATA 133, 95     : REM - STA $5F - set 'type' flag
40 DATA  32, 83,185: REM - JSR $B953 - print 'IN (line no.)'
50 DATA  76,148,251: REM - JMP $FB94 - do ctrl-C check; exit
60 REM
70 REM - POKE 541,2   turns trace ON
80 REM - POKE 541,251 turns trace OFF
90 REM
100 NEW
```

BASIC 'plot' routine

OSI's BASIC does not have a 'PRINT AT' statement; but CEGMON's cursor-controls make simple X/Y plotting practicable without complex POKE calculations. The following BASIC subroutine is for a C2's 32 × 64 display, assuming 28 printable lines and a centre-zero for the axes. Note the POKE 14, 0 in line 10020: it zeroes the 'current character count', to prevent BASIC from issuing a CR/LF each time the count goes past its 'terminal width' value. The routine presumes that the X and Y values and the plotting character P$ have been defined elsewhere. A faster way of handling the same problem would be to define two strings, one of 28 line-feeds (Y$) and one of 64 cursor-rights (X$), and print the correct number with a LEFT$(Y$,Y) and LEFT$(X$,X) call.

```
10000 PRINT CHR$(12);
10010 FOR AY=0 TO (14-(14-Y));PRINT CHR$(10);:NEXT
10020 POKE 14,0
10030 FOR AX=0 TO (32+X):PRINT CHR$(11);:NEXT
10040 PRINT P$;
10050 RETURN
```

Changing 'windows'

The short demonstration program below should give some ideas on how to change 'windows', to allow several lines to be PRINTed without scrolling the screen. The DATA lines within the program are those for all but standard Superboard/C1 systems; the DATA lines for these are below the main listing.

A simple way of finding the right values to POKE into the 'window'-table is to PEEK the edit-cursor position. Like the screen-handler, the edit-cursor is moved on a line-by-line basis, with a displacement stored elsewhere. PEEK(562) gives the high-order address of the cursor, while PEEK(561) + PEEK(559) gives its true low-order address.

```
10 FOR X=0 TO 2:FOR Y=0 TO 4:READ A:WI(X,Y)=A:NEXT Y,X
20 DATA 25,72,210,200,210
30 DATA 16,75,211,203,211
40 DATA 25,136,209,8,210
50 PRINT CHR$(26);
60 W=0:GOSUB 500
70 PRINT "  ^   ^   ^   ^   ^   ^   ^"
80 PRINT "-9 -6 -3  0 +3 +6 +9";
90 W=1: GOSUB 500
100 PRINT "Damping
110 PRINT "of oscillation";
120 W=2:GOSUB 500
130 AY=10:BY=1:ST=-1:GOSUB 300
140 W=1:GOSUB 500
150 PRINT "Expanding
160 PRINT "oscillation";
170 W=2:GOSUB 500
180 AY=1:BY=10:ST=1:GOSUB 300
190 GOTO 50
```

```
290 REM *** Graph-print subroutine
300 FOR Y=AY TO BY STEP ST
310 FOR X=0 TO 6.2 STEP .2
320 PRINT TAB(11+(Y*SIN(X)))"*"
330 NEXT X,Y
340 RETURN
490 REM *** Window-change subroutine
500 FOR X=0 TO 4:POKE 546+X, WI(W,X):NEXT
510 PRINT CHR$(30);
520 RETURN


20 DATA 25,166,209,228,209
30 DATA 16,72,210,168,210
40 DATA 25,70,209,134,209
```