PROBE!

PROBE is a disassembler program specially written for the Sharp PC 1500/Tandy PC2 for the purpose of exploring the computer's operating system. The program reads machine code routines and automatically translates them into assembly language. The package can be operated in two modes: in one a sequential list of assembly language instructions is printed, in the second the program automatically makes subroutine jumps when it encounters them, to produce a flowsheet of assembly language matching the order in which the subroutines are read by the operating system in practice.

Background

For the computer to understand and react to the commands that its user expects it to obey to requires an operating system, in the form of a ROM, that runs to some 16000 individually numbered addresses. Each address contains a number that is interpretable by the CPU (central processing unit) as part of a machine code instruction. The organization of the operating system is exceedingly complex. Its design involved creation of commonly used machine code subroutines, most of which are accessed by 'vector jump' instructions for speed, and an enormous set of other subroutines, each related to specific computer functions such as mathematical operations, reading keyboard input, displaying output, etc. etc.

For people interested in seeing how the manufacturer has achieved particular functions in machine code, it is necessary to peek each address in the appropriate subroutine (found either by trial and error or with help from the Sharp or Tandy Technical Manuals) and back interpret the machine code into comprehensible assembly language instructions, again by reference to the Technical Manuals. In practice, this process is tedious and highly vulnerable to human errors. It is also complicated by the fact that simple PEEKs into the ROM give decimal numbers that have to be converted to hexadecimal (hex) code to match the data in the Technical Manuals. Preliminary explorations of the operating system soon reveal that the whole process is compounded by the way subroutines are piled within subroutines within subroutines and so on. In theory it should be possible to make a listing of the assembly code, say, for a CLOAD operation. In practice it is almost impossible to do by hand.

PROBE was developed to overcome these difficulties. It is useful to have clear assembly language sequences (with address numbers and data in hex code) that simply follow operating system addresses in ascending numerical order. It is also useful to be able to follow the sequences of conditional and unconditional subroutine jumps, instead of simple ascending numerical order of commands. The two modes of PROBE make these desirable goals a reality. The program contains a short machine code sequence to retrieve data from memory addresses; everything else is written in Basic.

Hardware necessary for PROBE

Operation of PROBE requires a PC 1500 or PC2 computer plus printer, and a memory extension of 8k or more. The package has been designed to give a clear printed output on the pocket computer printer, but the program can be readily run to give output on an alternative printer via the serial interface. Only a single pen colour is used. Access to the computer Technical Manual is an essential adjunct for intelligent operation of the package.

Operation of the PROBE package

Load the program from cassette with the command CLOAD "PROBE". The program is a long one and takes about 11 minutes to load. The program is initiated by the command RUN <ENTER>.

The computer first requires a few inputs to determine its operating requirements. The first prompt is for a choice of MODE 1 OR 2?. An input of 1 or 2 is all that is necessary: anything else will be ignored. In Mode 1 the output will be a simple sequential listing of assembly language from a selected start address. In Mode 2 the program recognizes unconditional jump instructions (JMP, SJP, VEJ, VMJ) and jumps to the address specified, returning to the previous address when it meets the RTN instruction (except after a JMP instruction, of course!). PROBE allows for 15 levels of sub-routine jumps within subroutine jumps, which should be more than adequate for operating system code.

If you have selected Mode 2 operation, you will be asked to choose Conditional jumps (Y/N)?. If you input Y <ENTER> the program will make conditional vector subroutine jumps (VCS, VCR, VHS, VHR, VZS, VZR and VVS) in the same way as it makes unconditional jumps. Any other input means the conditional vector jumps will be listed but ignored.

The next prompt is for ME0 or ME1 (0/1)? The memory addresses in the pocket computers are stored in two banks, ME0 and ME1, so you must first specify which memory bank is to be searched. The bulk of the operating system and user memory is within ME0. Inputs other than 0 or 1 will be ignored. Please note that in the unmodified computer as manufactured, even with all accessories attached, the ME1 bank appears to be used only in very small blocks that are concerned with input/output constants. There are no machine code routines as such within ME1, so the provision to scan it with PROBE is really only an academic nicety.

The next prompt is for PU SET (0) or RESET (1)?. This prompt relates only to probes in the CE-158 interface ROM area (&8000 to &9999 in ME0). For probes not involving this area simply key <ENTER>. If you wish to probe the CE-158 ROM you need to specify which subbank of memory is involved - PU set or reset - consult page 95 in the Sharp Technical Manual for details.

Next you will be asked for a Start address?. Here you should key in the number of the first address of your chosen subroutine. If you have chosen Mode 2 operation the routine will now commence operation, stopping automatically when it meets the final RTN instruction in the sequence where it started.

In Mode 1 operation you will be asked to input an End address?. This allows you to choose your own finishing point, by entering an address number. If you enter 0 or &FFFF in response to this prompt, the program will stop operation automatically when it first encounters a RTN instruction.

Detailed points about PROBE

All numbers output during the operation of PROBE are in hex code, not decimal. The ampersand (&) is omitted in the interests of brevity and clarity. But do remember that if you <u>enter</u> start and end addresses in hex code, you <u>must</u> include the ampersand. Thus &BCD4 will be accepted as an input, BCD4 will cause an ERROR 1 message.

The start address chosen must be the start of a proper machine code subroutine. The Technical Manuals indicate the start addresses of many subroutines: others can be found by peeking VEJ addresses (see Technical Manual for details), and in practice many new start addresses will be turned up by jump instructions that appear within a chosen subroutine. If an improper start address is chosen, the machine code will be read with a frame shift error. It is, perhaps remarkably, possible that an entire but spurious routine can be output in PROBE by frameshift reading of the addresses: usually the program will crash with an error message after a few apparently normal commands. Because some ROM addresses appear to contain no data or reference numbers only the machine code reading frame varies in different areas of the operating system. It is therefore not possible to use PROBE to obtain a continuous readout of the entire operating system.

Unconditional and conditional branch instructions are listed but not reacted to in Mode 2 operation. This is because most branch instructions are associated with program loops, and the output would get stuck in a loop of endless repetitions if it attempted to follow branches. Some subroutine jumps are also involved in loop sequences, so that Mode 2 operation may still sometimes become caught in a loop. Judicious use of Mode 1 and Mode 2 operation, with the optional use of conditional vector jumps, is usually adequate for rapid resolution of the design of most operating system subroutines. It is advisable to keep an eye on the output, in Mode 2 operation, so the program can be interrupted with the <BREAK> key if it seems to have encountered a loop.

Address numbers, set to the left of the assembly language output, are provided intermittently for orientation in both operating modes. In Mode 2, address numbers are also provided at the start of each subroutine jump or return.

A few informal points

Calculations with hexadecimal numbers, for example to determine the point to which a branch is made within a routine, can be awkward. From the keyboard of a computer loaded with PROBE it is possible to convert decimal numbers to hex by the following operations: CLEAR, M=[decimal number], GOSUB 6100, GOSUB 6210, PRINT A$.

Although PROBE was designed to interpret machine code routines in the operating system, it can, of course, be used to interpret ones own routines, providing they are contained in spaces not used by PROBE itself. The most obvious space is the RESERVE MEMORY area. The value of PROBE in this situation is that it will provide a neatly printed assembly language version of a user's machine code routines for future reference. Just for fun, if PROBE is run with &7750 as start address, it will output its own short machine code routine for peeking memory addresses - a rare example of computer self-reference!

Caution

While every care has been taken to ensure the accuracy and reliability of PROBE, no liability can be assumed for any loss or damage arising from its use.

APPENDIX

Explanations and glossary for terms used in these instructions

Computers do everything by numbers. Imagine 8 matchboxes, each individually numbered so that its contents can be examined. In boxes 1, 4, 6 and 8 there is a bead; the remaining boxes are empty. Suppose now that you handle the boxes according to a small, unambiguous set of instructions - here are some examples.

I: look at the contents of box 5.
II: if there is a bead, do nothing, if not, place one in box 7.
III: look at the contents of box 7.
IV: if there is a bead, do nothing, if not, place one in box 5.

Already we have a small but useful instruction set. If the instructions are acted on in the sequence I, II, III, IV, we end up with a bead in boxes 1, 4, 6, 7 and 8. However, if we acted on them in the sequence III, IV, I, II, we end up with beads in boxes 1, 4, 5, 6 and 8. This may be of no consequence to the beads or the matchboxes, but if the number of boxes were increased to several thousand, and the equivalent of a bead or no bead is translated to lighting up or not lighting up a particular dot on a computer display, or storing a particular number in an electronic memory, then we have created a system with some power.

In practice, memory addresses in the PC 1500 function as the equivalent of groups of 8 matchboxes, so that each can represent a binary number from 0 to 255 (&00 to &FF in hex code), and sequences of instructions are stored as numbers (machine code) within the numbered addresses as well as the data on which the instructions operate. To peek into a memory therefore doesn't tell you whether the number in the memory is an instruction or a datum. Most of the instructions have been placed in addresses that can be read, but not altered (glued-up matchboxes with windows?!), and they comprise the operating system of the computer. The remainder have been left free for the user to fill with instructions and data. The sole complication is that because memory addresses are so long they need numbers of 4 hex digits, each memory can hold only 2 hex digits, so to <u>store</u> the reference address of a memory, for example &A064, <u>within</u> the memory requires two memory addresses - one for &A0, one for &64.

Thus, in any small sequence of addresses, it is possible to pack a few instructions followed by one or two boxes of data, or unrelated instructions. Any given set of machine code instructions can contain commands to skip forwards or backwards past a certain number of memory boxes (BRANCH), to JUMP to a new memory address, or to jump to a subroutine that begins somewhere else then return to the original place in the sequence (SJP). Sometimes such instructions are made conditional, that is they are carried out only if certain conditions - the setting of various flags for machine code routines - are met. Because a maximum of two hex digits can be stored in a single memory, the branch instructions are limited to ± 255 addresses, whereas the jump instructions specify a full 4-digit new address number (and therefore take up an extra memory themselves).

The PC 1500 operating system uses a lot of clever tricks to pack the maximum power and flexibility into the minimum number of memory addresses. One important trick is to save the number of command codes necessary to initiate jumps to subroutines that are used very commonly. This is known as **VECTOR JUMP**ing. The memory addresses from &FF00 to &FFF6 have been filled, in pairs, with the address numbers of commonly used subroutines. Normally, a command to make the system jump to a subroutine at address, say, &BCDE, would require three memory addresses, containing the codes for <JUMP to subroutine at> <&BC> <&DE>, respectively. But if &BC and &DE have been stored, let us say in addresses &FFC0 and &FFC1, the vector jump command makes it possible to say <JUMP to subroutine indicated by numbers stored in the pair of memories starting at &FF> <&C0> - i.e. it takes two commands. This is a **VMJ** command format, and vector jumps of this type can be conditional or unconditional. An even shorter version of the same idea is embodied in the VEJ command. There are 28 commands, all of which mean VEJ, but the reference number is also embodied in the command. Thus <&C0> is interpreted as <JUMP to subroutine indicated by numbers stored in the pair of memories starting at &FFC0> - the jump can now be made in response to a single command. For all vector jumps it is the built-in assumption that the reference address always begins &FF__ that allows for the shortening.

By now, it should be clear that the operating system memory addresses are filled with different numbers, each of which is part of a machine code command - either the command itself or related address references. If a program like PROBE is to be used to disentangle the command sequence intelligently, it is essential that each routine - related set of machine code commands - should be read properly from the intended start point, that is, in the correct **READING FRAME**. Consider this sequence of machine code instructions: &4A &48 &A5 &A7 &BA &02 &AE &62 &59 &9A.

If we disassemble this sequence from the first code (&4A) we read <LDI XL, 48> <LDA (A78A)> <ADC XL> <STA (6259)> <RTN>. This is a routine that reads a number stored in &A7BA, adds it to &48 and stores the result in address &6259. Suppose, now, we started to interpret the code from the second number (&48). This time we would read <LDI XH A5> <CPA (8A02)> <STA (6259)> <RTN>. The first two instructions are valid, but will not achieve the intended result of the subroutine. Suppose we started to disassemble from the &BA command. We would read <JMP (02AE)> <DEC UL> <ANI 9A>. Once again, each instruction makes sense, but because the reading frame is wrong the resultant routine is nonsense. Such frameshift errors usually come to grief when a code is encountered that cannot be interpreted as a command, but it is often possible to misread a long section of machine code before this happens.

The final point worth elaborating in this appendix concerns the use of electronic flip-flops called PU and PV, which the designers of the PC 1500 have used to help pack quarts of some memory sections - related to the printer and the interface units - into a pint pot of address locations. The CE 150 operating system is read properly only when PV is set: PROBE can handle this automatically. But the interface operating system is read slightly differently when PU is set or reset: the PROBE user therefore has the option to choose the setting of PU.