The following papers by Enabnit and Urbano, Pauker, and Ervin were presented at the 1969 spring meetings of the Digital Equipment Users Society (DECUS), held at Wakefield, Mass. The papers are reproduced here through the cooperation of the Society, A. J. Cossette, Executive Secretary.

A new approach to on-line, run-time program logic and error debugging using hardware implementation

ROBERT S. ENABIT,[1] *RESEARCH DIVISION, THE GOODYEAR TIME & RUBBER COMPANY, Akron, Ohio 44316*

What is believed to be a somewhat different approach to rapid program debugging has been devised in which: (1) execution of the programmer's logical thinking is automatically debugged by the computer at run time and (2) status errors of any type may be fed back into the computer, which subsequently outputs the immediate steps leading to that error. Debugging is carried out by the computer on an instruction-by-instruction basis, on-line, with all or selected interrupts serviced. A hardware-software implementation package for the PDP-8 is described, which could be adapted to other computers as well.

The hardware capabilities of even small computers today are awesome. Yet, no computer is better than the software that operates it, and until the software program is correct and trouble free, there is no capability whatever. In spite of the relative importance of troubleshooting, debugging software at the operating level is still a guessing game carried out by the programmer with himself as opponent, using the computer only as secretary, file clerk, and referee. As a result, the cost in time and patience for run-time debugging frequently exceeds all other commitment. This is particularly true for on-line and basic language programs that are the backbone of small computer operation.

A general complaint is that current procedures are simply breadboard operations that open the program on demand and spew out volumes of information, most of which is not even remotely pertinent. Then, depending upon the skill of the programmer, the demands and output volume are reduced by a process of successive approximation until error and cause are eventually revealed.

Because of the almost mandatory use of computers in modern technology, we have a large and growing amount of programming, even at assembly language level, being carried out by occasional users. For the part-time programmer, superior debugging skills are difficult to maintain, and the resultant cost in time, money, and frustration is enormous.

Faced with this particular situation ourselves, we tried to devise an approach that would bring more of the computer capability to bear on the debug problem. We used the Digital Equipment Corp. PDP-8 because it was representative, available, and is usually programmed for interrupt operation at assembly language level.

THE STATUS QUO

The philosophy of most current run-time debug procedures seems to encompass some or all of the following techniques:

(1) Provide as much information as possible.
(2) Provide maximum communication between program and programmer.
(3) Always debug in the direction of program flow or execution.
(4) Debug piecemeal, section by section.
(5) Overshoot error, move back a safe distance and step forward.
(6) Suspend normal input-output and simulate operation.
(7) Give the programmer all responsibility for obtaining "what," "where," "how," and "why"; the computer, only "when."

As a result of Items 1 through 4, the troubleshooter ends up spending most of his time looking at and verifying correct information and execution. This procedure is terribly inefficient since errors usually represent a minuscule portion of the total instructions.

The problem inherent in Item 5, with execution already in error and current status achieved via an unknown path, is that one really doesn't know the direction or extent of a safe distance. This approach thus becomes purely trial and error.

With respect to Item 6, suspension of input/output is necessary to prevent altering any logical path or conditions during a debug sequence that might compound the error. In the PDP-8 and similar computers, this entails disabling the interrupt, since the IO service cannot differentiate between input to the program and input to the debug system. Abnormal program operation is thus a prerequisite.

Most disturbing of all is the fact that run-time debuggers, unlike those for assembling and compiling, generally ignore the computer potential and let the programmer do most of the work.

A DIFFERENT APPROACH

If we assume that current techniques are unsatisfactory because the accepted philosophy is wrong, the obvious way to improve would be to invert that philosophy. We would then seek to:

(1) Provide as little information as possible.
(2) Minimize communication.
(3) Debug in reverse from an error and against the flow.
(4) Debug only errors, not the whole program.
(5) Stop at the instant of error and step backward.
(6) Operate the program normally, with interrupts and IO service.
(7) Let the computer do the work instead of the programmer.

Proceeding accordingly, we developed a new run-time debugging system for the PDP-8, including a programmable hardware facility, which: allows completely normal program operation with the interrupt on; provides only error-oriented debugger input-output; and shows the executed program flow leading to the error instead of what happens as a result.

The elements of this system are: a hardware option with control software, a

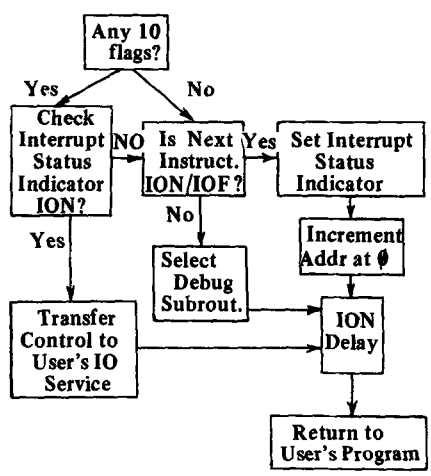

Fig. 1. Debug block diagram.

reverse trace concept, and both logic and instruction error debug routines.

PROGRAMMABLE DEBUG HARDWARE

To determine the instant of error, debugging must be carried out on an instruction-by-instruction basis and the computer should do it automatically. Exit from the user's program following an instruction can be achieved via the interrupt facility by jumping to the debug routine from Address 1. If a continual interrupt is provided, a potential exists for step-by-step program exit and reentry as shown very simply in the following:

Core Address	
Ø	____ (Next address in program)
1	JMP Debug
2Ø Debug,	_____ (Debugging subroutine)
21	_____
22	_____
23	ION
24	JMP I Ø (return to user's addr. held in Ø).

If the interrupt is continuous, "enabling" the interrupt gate (23) initiates a state of interrupt during the debug exit instruction (24), and control again passes to Address 1, thus creating a perpetual loop. It is immediately evident, however, that the difference between endless recycling and successful transfer back to the user's program will always be due to the interceding execution of that single, unvarying, "jump indirect to Ø" instruction (24), equal to a delay of 3.4 microseconds. We, therefore, introduced a corresponding hardware delay between the execution of the "interrupt enable" instruction (23) and the actual "gate enable." This allows program reentry and provides for a continuous automatic debug operation after every program step, with the address of the next instruction always available at Location Ø.

To recognize service needs with this continual state of interrupt, an IO service subroutine is required in the debug routine. This subroutine checks for the presence of IO device flags and, depending on the setting of a program interrupt status indicator, either transfers control to the user's IO service or ignores them. The status indicator is updated when the next program instruction calls for a change in interrupt status, and this program instruction is then skipped.

The basic plan of the debug hardware control routine is shown in Fig. 1. This arrangement also debugs all program IO service, but that feature is optional.

A permanent interrupt delay could, however, lead to execution problems when used without the debug software. ION delay is therefore incorporated in the processor as a separate, independently programmable option. This option also provides the required interrupt and is called, via instruction, 6ØØØ. The basic hardware pattern is shown in Fig. 2.

With an initialization routine that substitutes the "control transfer to debug" at Addresses 1 and 2, the described hardware/software control package provides instruction-by-instruction run-time debugging with all or selected interrupt services maintained. Other than the increased execution time, there are no limitations imposed by the system on either the user's program or the debug routines.

Reverse Trace

Having devised suitable program access to satisfy the new philosophy, the next step was to provide a debug concept that would also fit within this new framework. The principle theme of the inversion is that the debugger must reveal specifically what *has occurred* rather than what possibly *could occur* and do so with utmost austerity.

Obviously, if we are going to provide as little information about the program as possible, the debugger should not output correct information. If we assume, temporarily, that the debugging execution will cease at the instant of error, it follows that a small, continuously updated table of executed instructions would probably contain those crucial operations prior to and responsible for the error, which are the only ones of interest. The previously proposed debug control, by virtue of the interrupt, has access to the address of each new instruction via Address Ø. Transferring these addresses to the table and updating makes available a record of the executed program flow leading directly to the error. The corresponding instructions are available from the program listing so that from an output of this debug table we can backtrack rapidly to the error source. The accumulator content prior to the execution of each instruction is also maintained in the table to reveal why and how errors involving variables originate.

Using this arrangement, communication with the debugger is restricted to a brief initial input and the error table output. Between the two, all communication is dedicated to the user's program during program execution.

LOGIC DEBUGGER

With program access and debugging concept resolved, we need test procedures to determine if an error does, in fact, exist

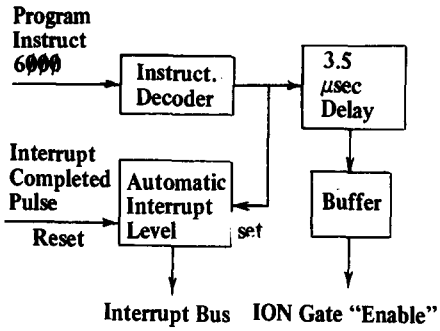

Fig. 2. Hardware block diagram.

and the instant of occurrence, but the computer must do the lion's share of the work. Of course, in the case of output errors, the occurrence is self-evident. This will be dealt with later.

In a program designed to follow a logical pattern to achieve a specific result, there are three possible types of program error: (1) correct user's logic, but wrong instructions; (2) correct instructions to carry out logic, but wrong logic; (3) both.

The end result, however, is the same—the execution eventually ends up in either a loop or in the wrong area at the right time. Loops are almost self-debugging with this system. All we need do is output the table, see what has happened, and backtrack if necessary. (Backtracking is putting the earliest output address or associated information back into the debugger as an error and rerunning.) In all other cases, the initial error for a reverse trace may be obtained by searching for and outputting the table only when the program executed incorrectly and enters a designated "wrong area."

Right and wrong logic is determined by a logic debug routine that compares executed program flow with a logic table read in by the user and obtained from his flow diagram and program listing. This table consists of instruction addresses contained within logical program segments. Figure 3 is a simple flow chart for a program designed to execute along the

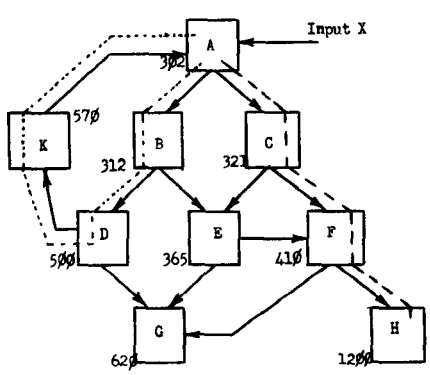

Fig. 3. Flow-chart example.

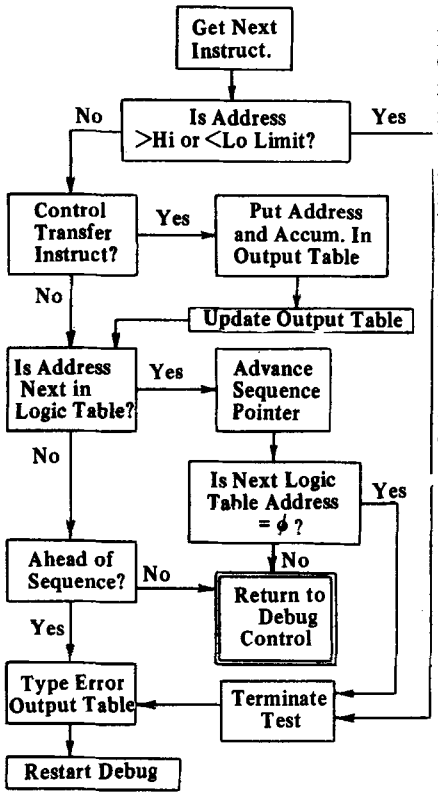

Fig. 4. Logic debug test-block diagram.

dotted path, A, B, D, K, three cycles and the dashed path, A, C, F, H, the fourth, using Input X. The addresses alongside the blocks represent any instructions within those logical segments.

The logical information read into the debug table could be

3Ø2/312/5ØØ/57Ø/5ØØ/5ØØ/321/41Ø/
12ØØ/Ø/5ØØ/365/62Ø

Program execution limits are also defined during read-in to allow protection of specified areas. The debug test is carried out according to the scheme indicated in Fig. 4.

Program addresses matching those in the logic table must appear in true table sequence. Addresses already verified are ignored. The test program is terminated by zero, and following addresses stipulate additional errors. In the example given, the complete loop is verified, then its repetition verified twice more using Address 5ØØ. Further looping is in error, as are the other excluded branches. Output on error consists of those control transfer addresses executed immediately in advance of the error, the next program address, and the last verified address in the logic table.

As previously stated, the program executes the logical flow, encounters a forbidden path, or ends up in a minor loop. If the logical execution is correct but the programmer's logic is not, any program output will be in error and we can immediately use the instruction debug test next to be described. A loop is debugged by calling for output to reveal its origin and subsequently using this address with instruction debug to obtain the cause. When the origin of a logical flow error is not apparent from the control transfer addresses outputted, the flow may be backtracked or the logic table refined for the immediate error area. However, since some form of the error is continually available after the first pass, we could turn at any time to the more versatile and efficient instruction debug test.

The principal intent of logic debug is to provide an error for use with instruction debug when none is readily apparent. Additionally, it can be used to determine whether the program or the programmer's logic is at fault. This routine locates flow errors by checking executing addresses against input-defined program flow, and outputs program control addresses leading to sequence deviation or specified wrong areas.

INSTRUCTION DEBUG

The logic debug subroutine attempts to give the computer more responsibility in finding "what" and "where" in addition to "when" by using the inverse debug feature. Instruction debug is designed to do the same with "how" and "why" for the error.

This subroutine performs any one of the following selected test options prior to the execution of every encountered instruction at run time. (1) Looks for a specified address in the program control register. (Note that, with every instruction interrupted, the content of Address Ø and that of the program control register are always identical when control is transferred to debug.) (2) Performs a specified test on the content of any address in memory. (3) Tests the accumulator content each time a specified address appears in the program control register.

The input to instruction debug can thus be any address, variable, or accumulator operation known or suspected of being in error. The debugger will run the program and output addresses of the executed steps leading to that error. The elementary logic of instruction debug is shown in Fig. 5.

As in logic debug, output is restricted to a series of control addresses and the accumulator content prior to error. Addresses put in the output table may be those of every sequential instruction or only control transfers, the choice being directed by a switch register setting. A typical procedure in using this subroutine would be to output only control instructions to determine the earliest effect

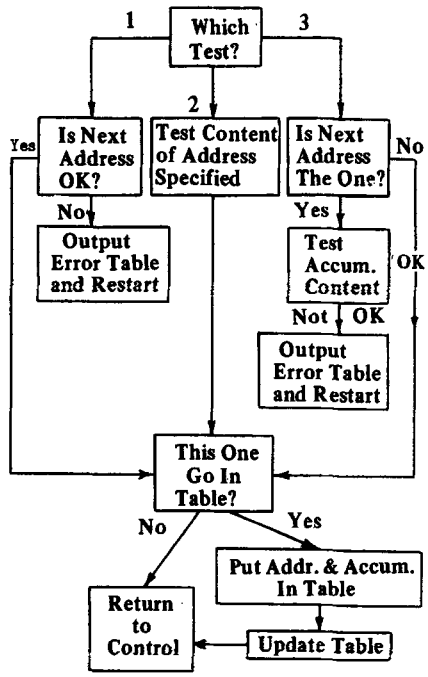

Fig. 5. Instruction debug test-block diagram.

of an error, then using the associated variable or accumulator as error input, to follow with another execution that outputs sequential addresses.

In either case, further backtracking, if required, is accomplished by using the most sequentially remote outputted address as a debug search error and restarting.

Compound error, that grim reaper of programming, presents no additional problems since reverse debugging with backtracking reveals branch deviation, the instant of compounding, and the individual sources.

Characteristically, the supplied error input word is subtracted from the current value of the designated test status word. Although the error trigger is initially set to output on zero difference, the option is provided to alter that instruction during test selection if desired. The user thus retains complete control over the test conditions.

The effect of instruction debug approximates putting a program error back into the computer and letting the computer tell the programmer how and why it occurred without simulation or ambiguity. If the error terminates in output, this is indeed a very good simile.

DEBUG PACKAGE

The following hardware and techniques were employed in assembling a complete debug system for the PDP-8. In keeping with the austerity inherent in the new philosophy, the software package is

restricted to three pages in memory (384 words), or only two pages with the omission of logic debug. Included is an initialization routine that modifies Addresses 1 and 2 for transfer of control to debug, prints out the starting address of the error output table (used when forcing output for loops), and provides for user input to the debugger. Input/output communications are brief and in octal notation to correspond with program assembly listings.

Input is requested by abbreviated queries initiated by the debugger. Input, showing the computer type-out underlined, is as follows for the logic debug subroutine.

S 265
(Desired starting address in user's program)
P 1
(Debug program, Ø or 1. Logic = 1)
LIM 177/4ØØØ
(User program limits)
LOG 225/265/646/etc.
(Logical flow)

Similarly, instruction debug is called when the response to P is Ø and will be followed by:

ADD 171
(Address to be tested)
VAL 3556
(Test value)
TST 1
(Test option, Ø or 1. Following the selection with / allows a new test instruction to be typed in)

Test options are 1 and Ø. Option 1, shown, checks the accumulator each time the instruction designated by ADD is executed. If Option Ø, which tests the content of the address ADD, is called, one additional query results:

SET Ø
(Initial setting of ADD)

This brief numerical input is the only user communication with debug. The output, together with the program listing, will provide all other information required.

Again, in the best interest of minimum communication, only six executed addresses, their related accumulator content, and the current program control address are outputted. With such highly pertinent information, this seems adequate in most cases, and the backtracking technique is available for the rest.

An example of a completed debug execution is shown in the following:

```
OUTP  SA75Ø1   (Output table address)

S     264Ø
P     Ø
ADD   171      } Debug Input
VAL   Ø        } Information
TST   Ø/765Ø
SET   Ø

For instructions = * No = /
                                  } Normal
Net reg. = ØØØ1  Space = 473Ø    } Program
IR2 (Ø, ØC5, 5)                   } IO

ØЗØØ/ØØØØ
Ø3Ø5/Ø261   } Debug output showing
1315/ØØØØ   } program control
1322/Ø261   } approaching error and
2124/ØØØØ   } accum. before execution.
2134/ØØØØ
2142        Next address
S           (Auto Restart)
```

This particular run was testing for a nonzero condition at Address 171, which occurred upon execution of the command at 2141 (since 2142 is next). The output table indicated that this resulted from an illogical transfer from Address 3Ø5 to the routine just ahead of Address 1315. A rerun using Address Ø, testing for Address 1315 and outputting sequential addresses rather than control transfers, immediately revealed the cause to be a tape reader error.

The input shown is the maximum ever used for instruction debug and includes a change in the sense of the test specification. Although the software is expandable, the complete debugging system, as now packaged, is composed of: the programmable hardware interrupt and delay which provides program exit every instruction; the hardware control routine which determines IO status, selects test, and provides program reentry; initialization routine; logic debug subroutine to check program flow and provide evidence of error; instruction debug subroutine to locate address or variable in error; and reverse trace and output routines to permit backtracking to the source of error.

Program execution is unavoidably slowed by debugging every instruction. With the separate flag service in the debug control, however, modifications can be made in the debugger to select and modify interrupt handling without disturbing the user's IO service. Even the most peculiar interrupt situations can be easily resolved.

In summary, the PDP-8 debug package appears to have the following worthwhile features: It allows normal operation with interrupts. Input is brief, guided, octal, and versatile. It debugs every instruction but outputs only error-oriented information. It debugs all program errors, including those that are hardware initiated. It backtracks errors from result directly to source. It reduces the number of debug operations required. It employs the computer to a better advantage.

The system has been appropriately designated as a "programmed on-line inverted logic debugger" or, in short, Program OIL.

CONCLUSIONS

Inverting the normal philosophy of current debugging techniques has led to the development of some interesting hardware and software arrangements.

The programmable hardware interrupt and delay, with its associated software control routine, permit program entry and exit via the interrupt facility. The arrangement also permits instruction-by-instruction debugging without interference to normal interrupt operation. This is believed to add new dimensions to debugging capability.

The inverse debugging technique described and implemented provides only that output information directly related to the error and produces an effect somewhat similar to that of running the logic in reverse from the instant of occurrence.

Using these features, we were able to develop a successful PDP-8 run-time debug system that does, in fact, reverse the philosophy inherent in most debug routines and, as initially specified:

(1) Requires minimal input.
(2) Provides brief dedicated output.
(3) Debugs both logic and execution in reverse.
(4) Virtually ignores everything but errors.
(5) Reveals the error source directly from the error.
(6) Permits normal on-line operation with interrupt IO service.
(7) Lets the computer do the debugging.
(8) As an added feature, requires little core space.

The hardware/software package described is specific to basic assembly language and the PDP-8. Since it appears universally applicable, we plan to make the package available to other PDP-8 users. We see no reason, however, why the same concepts and methods would not apply equally to any other computer having interrupt facilities. Application to byte-oriented computers and high-level languages has also been studied and appears feasible.

NOTE

1. I wish to thank B. C. Kent and K. A. Spriggel, of the Goodyear Research laboratory, for their valuable assistance in preparing and testing the hardware and software for the debug system. Appreciation is also expressed to the Goodyear Tire & Rubber Company and the Research Division for their support of this project and permission to publicize the result.

A spectral analysis program for the processing of neuro-electric data

FRANCESCA URBANO, STEPHEN G. PAUKER, *and* FRANK R. ERVIN, *STANLEY COBB LABORATORIES FOR PSYCHIATRIC RESEARCH, MASSACHUSETTS GENERAL HOSPITAL, Boston, Massachusetts 02114*

The Spectral Analysis Program is designed to perform the following kinds of spectral analysis, using the Cooley-Tukey Fast Fourier Transform Algorithm: cross correlation, cross power spectrum, Fourier transform, inverse Fourier transform, and double Fourier transform (the process of transforming two real functions simultaneously). These operations are performed on two blocks of core that can be filled through time-sequential sampling of an external analog signal or from previously sampled data that have been stored on DECtape. Through the use of nine Teletype commands, the user can manipulate input and output and select a particular type of spectral analysis. Output may be displayed on a 30D scope stored on DECtape or turned into hard copy by a Calcomp plotter used for processing neuro-electric data at the Stanley Cobb Laboratories for Psychiatric Research.

Equipped[1] with an 9K PDP-7 computer, two DECtape units, and some standard input-output devices, we set about the task of providing neurophysiologists at the Stanley Cobb Laboratories for Psychiatric Research with a flexible tool for the analysis of human brain-wave activity. The rationale for our approach to this problem, the Spectral Analysis Program that was the solution, and the applications that it made possible are the topics to which this paper is devoted.

BACKGROUND

The traditional measure of brain-wave activity is the electroencephalogram, or EEG, which consists of a time-varying voltage measured between two points on the skull or in the brain (see Fig. 1). There are many factors that influence the waveform of an EEG. Two such factors are: (1) location of the probe on the skull and (2) the physiological condition of the test patient—asleep vs awake, anxious vs calm, tired vs alert. In broad terms, the prime objective of the investigator in studying the EEG is to detect patterns or changes in the waveform that correspond to behavioral state. For example, in the ideal situation, the researcher would like to find that a specific EEG was characteristic of the schizophrenic but did not occur in the nonschizophrenic. He might then be able to diagnose schizophrenia as well as determine how his patient was responding to treatment. The soundness of such an approach is based on the assumption that "brain wave phenomena are intimately involved with the state of the organism and are not just irrelevant noise [Hanley, Walter, Rhodes, & Adey, 1969]."

METHODS OF EEG ANALYSIS

The era preceding large, high-speed digital computers found EEG analysis in a qualitative state. Three basic techniques, all visual in nature, were available to the investigator:

(1) He could look for a dominant frequency in the waveform. Motivated by the impression that the EEG was an inherently oscillatory phenomenon, the investigator found some important clues as to brain function. A high frequency EEG was found to correlate with a hyperalert, anxious state, while a low-frequency wave was indicative of a drowsy, calm state.

(2) The detection of symmetry in EEG recorded from physiologically symmetric points in the brain was another visual technique with limited usefulness. In the normal brain, recording from two identical structures yields two identical waveforms. Asymmetry has led neurophysiologists to suspect malfunction or obstruction in one of the hemispheres. Indeed, brain tumors and blood clots have been found in these cases. But the capacity for refinement of this clinical tool is unlimited. The pressing need is for more sophisticated quantitative analysis techniques.

(3) The visual detection of unique patterns in EEG has produced three major results: (a) The discovery of the 3/sec spike and waveform characteristic of *petit mal* epilepsy. (b) The separation of sleep into five stages, distinguishable both by the pattern of their EEG and other physical signs. (c) The discovery of the alpha wave and blocking phenomena; this pattern is a sinusoidal 8- to 12-cps wave, recorded from the occipital region of the head and present in the awake, relaxed patient. Its most striking feature is its disappearance, or "blocking," upon presentation of a stimulus.

One cannot ignore the importance of the above discoveries. Yet, like the former methods of visual inspection, visual pattern recognition (3) has had limited success as it completely precludes the possibility of other patterns hidden in noise.

The ability of the present-day computer to perform complicated mathematical analysis on large masses of data has revolutionized EEG analysis and given investigators a powerful quantitative tool.

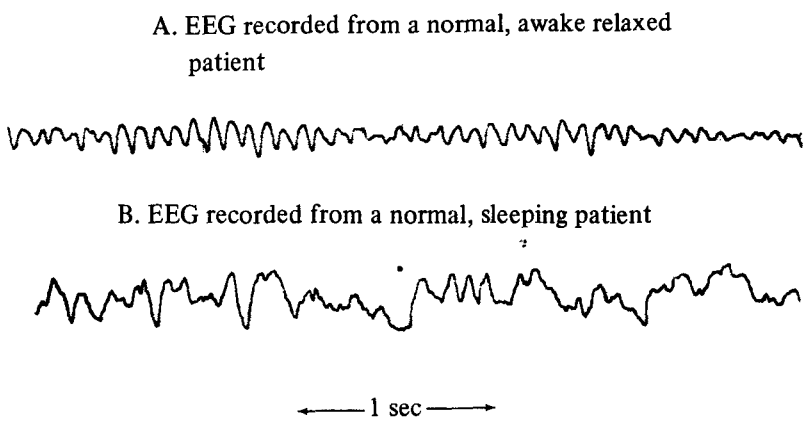

Fig. 1. Some typical EEGs.