Techniques for program error diagnosis on EDSAC 2

By D. W. Barron and D. F. Hartley

The paper describes the techniques and facilities for program error diagnosis provided in the EDSAC 2 machine-code and Autocode programming systems. The Autocode diagnosis system can normally be used without any knowledge of machine code since some automatic interpretation of diagnostic information is carried out, and all communication with the programmer is in source language. The practical usefulness of the various techniques described is indicated.

Introduction

The detection and diagnosis of program errors is an aspect of programming which has received less attention than it deserves. The checking-out of a program accounts for a not insignificant proportion of the total time taken preparing a problem for the computer, yet many present-day programming systems appear to neglect this fact and offer little assistance to the programmer. Sometimes, when a program goes wrong, the only help given to the programmer is a "core-dump"—a printout of the memory contents in a uniform style—from which he must find the trouble as best he can. Clearly, he requires more than this; if effort can be devoted to writing programs to read other programs into the machine there should be some attempt to help in getting them to work. It is desirable for the programmer to have selective information, both while his program is running and after it has come to an untimely halt. The slight increase in machine time which this involves is balanced by a reduction in overall checkout time.

Difficulties arise when automatic-programming languages are used, since the programmer is essentially one step removed from the operation of the machine, and hence from the effect of his errors. To present diagnostic information in basic machine language means that the programmer must be familiar with machine language, and must have the compiled object program available. We believe that the real purpose of automatic programming is to avoid the use (and consequent teaching) of machine code, and that it is unreasonable to expect knowledge of the latter before program errors can be found.

At Cambridge, interest in techniques for error diagnosis dates back to the early days of EDSAC 1 (see for example Gill, 1951). The purpose of the present paper is to describe the facilities provided to the machine-code programmer of EDSAC 2, and to show how similar features have been provided with the EDSAC 2 Autocode.

It is not proposed to give a full description of EDSAC 2, which can be found elsewhere (Barron and Swinnerton-Dyer, 1960), or of the EDSAC 2 Autocode, which is an algebraic formula language, similar in many respects to Mercury Autocode (Brooker *et al.*, 1961). For a full description of the Autocode, see Hartley (1962). For our present purpose it is sufficient to know that EDSAC 2 is a parallel, binary machine, with core store, and built-in floating-point arithmetic. The core store comprises 17,408 words* plus a *reserved store* which contains 768 words of permanently wired-in programs and 64 words of working space for these programs.

Error stops

It is desirable that any error in a program detected by the hardware of the computer should provide some kind of external indication. This can be a stop, a transfer to a specified part of the user's program (a trap), or, in a fast machine running under control of an internal supervisory or executive system, suspension of the program after some monitor printing. In all cases there is some information which is of immediate interest to the programmer who requires, above all, to know the nature of the fault which caused the machine to stop, and the position reached in the execution of his program. If this information can be presented at the error stop in a form corresponding to the language in which the source program was written, a substantial saving may be made in both machine and programming time.

One of the wired-in programs of EDSAC 2 is the *Report routine*. This prints the contents of the accumulator, the order currently being obeyed in the object program, the position of that order in the store, and the contents of the two modifier (index) registers; it then stops the machine, lighting a special lamp on the control panel. This stop is irrevocable; the computer can only be restarted by reading in another program. The Report routine is entered under various error conditions arising in the execution of an instruction, for example if the function of the current instruction is unassigned, or if an arithmetic overflow is detected. Alternatively it can be entered by transfer of control from another reserved-store routine, for example if the argument presented to the square root routine is negative. If a report occurs during execution of a reserved-store routine the Report routine prints out the last order obeyed in the user's program, rather than the order which actually caused the report.

One of the most common of beginners' mistakes is to refer to an unused store register, either in an arithmetic instruction or a control transfer. Both situations are detected by hardware as a consequence of the fact that

* 16,384 words have been added to the core store since the publication of the paper by Barron and Swinnerton-Dyer.

the store is cleared to binary ones at the start of each program. An attempt to use the contents of an empty register as an instruction produces an unassigned function, with a consequent report, whilst the Arithmetic Unit recognizes a number consisting entirely of binary ones as not representing a valid floating-point number, and therefore reports.

The Trace

EDSAC 2 incorporates a built-in *Trace* facility which enables a record to be kept of the jump orders obeyed in a program, this being achieved by a combination of hardware and software, using a program-interrupt technique. The Trace is normally inactive, and is brought into action by operating a key on the control panel. In the Trace mode, the object program is interrupted after each jump order, and a routine in the reserved store is entered which adds details of the jump to a list, also held in the reserved store, and then resumes the object program. In the case of conditional jumps the interruption takes place only if the condition is satisfied so that the jump actually occurs. The Trace is by-passed during execution of subroutines in the reserved store, so that only jumps in the user's program are recorded. Note particularly that this is completely different from the system, often described as a trace, in which the object program is executed interpretively, with printing of information after some or all of the orders. This is prohibitively slow, and undiscriminating in its output, whereas the EDSAC 2 system produces a highly compressed list, and only slows the program down by about a factor of two.

The Trace record in the reserved store consists of a series of items, each of which gives the source and destination of a jump, a count of the number of times this jump has been obeyed in sequence, and a count of the number of times that this jump and the previous item in the record have been obeyed in sequence. Thus a single repeated jump produces one item in the list, and a cycle of two repeated jumps takes up two items in the list. At any time the 41 most recent entries are retained in the list. As it stands this record is not in a convenient form for the programmer, and service routines are provided to edit the list and to print it in a more suitable form. There is one routine which gives details of the last two jumps obeyed; this is useful if the program has jumped to an unexpected part of the store. The most used routine, however, is one which analyses the Trace record and prints it in the form of a series of jumps and cycles. Briefly, the routine operates as follows. Single and double cycles have already been detected and contracted by the built-in Trace program. The service routine continues this process by detecting every sequence of items (or *cycle*) which occurs in the Trace record more than once. The aim is to contract the record so as to reduce the amount of printing, and to do this in such a way that the cycles produced resemble the natural sub-units or flow diagram of the program. The effect of the service routine is to remove the details of each cycle to a separate list and to replace each occurrence of the cycle in the main record by an appropriate reference. In this process the cycles are taken in their order of frequency so that inner cycles are detected and removed before outer cycles, thereby obtaining a maximum contraction. The service routine prints the main record as a series of items each of which is either a jump or a reference to a cycle. Each jump is printed as the source address, the order in that location, the destination address, and the order in that location. The main record is followed by a list of the items comprising each cycle, some of which may, of course, be references to other (i.e. inner) cycles.

The Report and Trace facilities were devised by Dr. D. J. Wheeler. No description of the EDSAC 2 implementation has previously been published, but the facilities were proposed in Wilkes, Wheeler and Gill (1957).

Post Mortems

When a program has stopped or been stopped, the contents of selected parts of the store can be printed by the Post Mortem service routine. A wide range of output formats is allowed: a register can be interpreted as an order pair, fixed-point integer, fixed-point fraction, or floating-point number, in which case it can be printed in floating-decimal form or rounded off to the nearest integer. (The internal representation is floating-binary.) For non-numerical work binary, base-4, and octal formats are available. The contents of different parts of the store can be printed in different formats; the required regions of store are specified, together with the format, on an input paper tape which is read by the service routine. The addresses which specify the regions of store may be absolute addresses, or they may be symbolic addresses previously defined by the program. A further service routine prints a list of all the symbolic addresses used, together with the corresponding absolute addresses.

Dynamic checking

Post Mortem routines are not a great help if a program has gone seriously astray, since relevant information may have been destroyed. In this situation as in others (notably when a program appears to run correctly but gives the wrong answers) it is a great help to have information printed during the operation of the program so as to obtain a dynamic picture of what is going on. There are two ways of achieving this in machine-language programs. The assembly routine allows sections of a program to be declared optional; such sections are ignored unless a certain key on the control panel is depressed during assembly. Thus the programmer can include additional printing instructions which can be obeyed during checkout, but omitted when the program is known (or at least believed) to work. This technique requires anticipation on the part of the programmer; a more powerful facility is afforded by the

Check-Point routine. This routine enables a programmer to establish check-points (or break-points) in a program, and to specify for each check-point the information to be printed, and the occasions on which printing is to occur.

The check-point routine works in two phases. The first phase follows assembly, and is essentially an extension of the assembly process. During this phase a data tape is read specifying the positions of the check-points, and the action to be taken at each one. A list is placed in the store giving the details of each check-point, and jump orders (blocking orders) are placed at appropriate positions in the object program, the orders which were previously in these positions being preserved in the check-point list. At the end of phase one, the object program is started. Phase two occurs every time control reaches one of the check-points in the object program, when control is transferred by the blocking order to the check-point routine. First the contents of the arithmetic registers, the modifier registers, and the page-layout counters are preserved. Then the list for the check-point is consulted, and any printing there called for takes place, usually on an output channel other than that being used by the object program. Finally the arithmetic registers, modifier registers and page-layout counters are restored, the original output channel is reselected, the order which was originally at the check-point is obeyed, and then control is transferred back to the object program. A simplified version of this routine allows a sequence of characters to be printed every time control reaches a check-point.

Clerical and syntactic errors

We have so far been dealing with the errors which show up during the operation of a program. There is another class of errors which are much easier to deal with; these are the clerical errors where, by mispunching or misunderstanding, a meaningless instruction appears on the tape, or a symbolic address is left undefined. The assembly routine carries out a fairly thorough check of the program as it is being read in. If an error is found an indication is printed, the rest of the item (instruction or number) is ignored, and assembly is resumed. However, if there have been any such "input reports" the program is not actually obeyed: at the end of the program tape the machine stops with the "Report" lamp on the control panel illuminated. Thus one scan produces a listing of most of the errors of this nature. (If there is more than one error in a single item, only the first is listed.) For each error there is printed an identifying number (which can be checked against a standard list to find the cause of the report—for example, "two minus signs in number") and an indication of the location of the offending item on the tape. This is the address of the register in which the item would have been stored; additionally, the machine halts every time an error is found so that the operator may mark the input tape.

Small corrections can often be incorporated in programs by using the second input channel. An instruction is terminated by a carriage return, but for convenience of printing programs this can be followed by a line feed or line feeds which are ignored by the assembly routine. The tape code is so arranged that the line feed can be converted, by overpunching, into a symbol which will select the second input channel for subsequent program input. At the end of the correction another tape character switches back to the first input channel, so that the machine resumes reading of the main program tape. Larger corrections are usually made by copying the tape and inserting new material from a keyboard. This is done using reproducers, designed at the Mathematical Laboratory, which operate at a speed of 25 characters per second. The duplicated tapes are checked on a comparator, also designed at the Laboratory, which works at a speed of about 400 characters per second.

Discussion

In the preceding sections the facilities available to the machine-code programmer have been described. These are part of a programming system that is built round an elaborate symbolic assembly routine, and represent those facilities which experience has shown to be most used by programmers. (EDSAC 2 is run on an "open-shop" principle.) Some facilities have been tried, and discarded because they proved to be little used; outstanding among these is the Comparison Post Mortem. This facility was part of the assembly process, and checked the program being read from tape against the program already in the store, printing details of any discrepancies. This was useful whilst the machine was output-limited, but the installation of a 300 characters per second tape punch made it reasonable to post-mortem larger areas of store, and the Comparison Post Mortem fell into disuse, and was omitted from the second version of the assembly program.

In general it is true to say that the facilities most popular with programmers are those which are easy to use. Thus the check-point routines are disliked because the data tape specifying the check-points has to be prepared in an inconvenient form. The one exception to this rule that we have encountered is the facility for interpolating corrections on the second input channel which, though very simple, is seldom used.

The Autocode system

The diagnostic features of the EDSAC 2 Autocode were designed to provide similar facilities to those available for the machine-code programmer, with two overriding requirements: that they be simple to use, and that they require no knowledge of the machine-code language and the internal structure of the computer. The following sections describe how the machine-code facilities were adapted to meet these needs, but first we consider some of the general principles involved in this task.

In order to avoid the necessity of using machine code it is essential that all communication between programmer and diagnostic routines should be in terms of the source language. This requires a certain amount of reverse translation, which has hitherto been left for the programmer to do, using an object program printout together with a listing of storage allocation. To effect reverse translation automatically, diagnostic routines must have access to a store allocation map, which is a natural by-product of translation and assembly processes, and since diagnostic communication is normally required during a post-mortem phase, this map can be conveniently retained in any form of auxiliary storage that is available. Thus, by keeping certain information linking source programs with translated object programs it is possible to provide diagnostic routines comparable to the facilities available for machine-code programming. Moreover, just as the machine-code programmer can interpret the meaning of basic diagnostic information, so also to a certain extent can a diagnostic routine provide more information when dealing with object programs produced by translators according to well-defined rules. For these purposes it is essential that compiled programs should be altered only at source-language level so that diagnostic routines can produce concise yet reliable information.

Fundamental to the working of all diagnostic routines provided with the EDSAC 2 Autocode is the *label list* produced by the translator. At run time, this is held in backing store together with a copy of the object program, and gives the machine addresses corresponding to source program labels, cycle instructions and library subroutines, and the base addresses of subscripted variables. A name list is not required, since the Autocode works with a fixed set of names for variables.

Autocode Reports and Rescue

Syntactic errors in an Autocode program are detected during the input phase of the translation process. Each error produces an Autocode Input Report, which consists of an identifying number for the fault, the nearest label, and the position of the offending item relative to that label. The remainder of the item is ignored and reading proceeds, but the machine stops at the end of the program tape: in this way most of the syntactic errors are detected in one pass.

However, if a report occurs during the running of an Autocode object program, the information printed is meaningless to the programmer as it stands, and must be interpreted in the context of the source program. In many automatic programming systems a listing is provided of compiled object program and storage allocation, to permit the interpretation of this kind of diagnostic information at machine-language level. In the EDSAC 2 system this procedure has been mechanized, and is effected by a service routine called *Autocode Rescue*. This routine is kept on paper tape, and is read into the computer by the operator whenever a report occurs in an Autocode object program. It is fortunate that the Report routine preserves the address of the order causing the report. Thus, the Rescue routine uses this address together with the label list associated with the object program to determine the approximate position of the report in the source program and, further, if this is in a library subroutine, it is usually possible to locate the position of the call to this subroutine from the link in a similar manner. Both the function and the address of the order causing the report give further information: for example, if the function is to store the content of the accumulator, then an overflow must have occurred, and the address gives the variable to which a value was being assigned at the time. Certain error conditions in library subroutines are dealt with by transferring control to an order with a characteristic unassigned function. When the consequent report is analysed by the Rescue routine, the characteristic function and its relative position within the subroutine indicate the cause of the report.

The following are typical examples of outputs from this service routine:

```
UNSET NUMBER IN X—AFTER LABEL 6
OVERFLOW IN LIBRARY 1—AFTER REPEAT 2
PROGRAM OVERWRITTEN AT Z(385)—AFTER
    LABEL 3
```

The first of these indicates that an instruction after label 6 (but before the next label) has used the variable X without a value having previously been assigned to it. The second indicates that an overflow occurred inside library subroutine number 1 called after the second cycle instruction. In the third example subscripted variable Z has overrun its allocated store, thereby overwriting the program with a number, and the machine has reported on attempting to obey this part of the program.

Although the information provided by Autocode Rescue is not as precise as a machine-code report, it is usually sufficient to enable a large proportion of errors to be found easily and quickly. The routine is capable of distinguishing between more than 50 types of error stop, ranging from accumulator overflow to suspected machine failure.

Autocode Trace

As with the Report, the record produced by the EDSAC 2 Trace facility, described previously, is of little interest as it stands if the source program was written in Autocode. There are two reasons for this. First, the items are printed as jumps from one absolute store address to another, so that even if the compiled object program and label list were available, the necessary cross-referencing would be most tedious. Secondly, some of the items in the Trace record may be irrelevant to the Autocode program, since the Trace will record jumps within system and library subroutines as well as jumps in the object program proper. Thus a separate service routine is provided; this is basically similar to the machine-code routine, but overcomes the difficulties described above. From the nature of object programs compiled by the Autocode translator, it is possible to determine which items of the Trace record represent control transfers corresponding to source-program instructions, i.e. jumps, cycles and entries to system and library subroutines. All other items are removed from the Trace record, which is then analysed into cycles by the same method as used in the machine-code service routine. Before printing the record, the label list is used to translate from absolute machine addresses to Autocode names. A typical output from this routine might be as follows:

ACTUAL PRINTING		NOTES
	→ 10	jump to label 10
	→ 3 >	jump to label 3 on a "greater than" condition
	INTEGER	use of the system function INTEGER
6	CYCLE 1	six times round cycle 1 (details below)
	RETURN	return from subroutine
CYCLE 1 =		
	→ 6	jump to label 6
	PRINT	use of the PRINT routine
10	REPEAT 3	ten times round the third cycle instruction

Post Mortems and Optional Printing

An Autocode Post Mortem is provided to enable the values of working variables to be obtained after a program has been run. The programmer specifies on an input tape the Autocode names of the variables required; these names are printed with the values of the variables for easy reference. Another service routine produces a listing of the storage allocation, the absolute addresses corresponding to labels, and the compiled object program. However, this listing is deliberately arranged in such a way that it cannot be read into the machine as an object program unless it is completely repunched by hand on a keyboard perforator. Attempts to modify compiled programs at machine-code level are discouraged. This practice is directly contrary to our philosophy of providing a self-sufficient automatic programming system, and in addition it injects a degree of uncertainty into the form of compiled programs which seriously limits the extent to which automatic diagnostic routines can be used. Fortunately translation is sufficiently rapid (of order five seconds for an average program) to make modification of compiled programs unnecessary. For fully developed programs there is a separate service routine which punches on paper tape a compiled program in base-32 form: this can be re-input at high speed (1,000 characters per second), but again does not permit editing or correction. It is not considered necessary, or desirable, to produce an object program listing as an automatic by-product of translation. The listing produced by the service routine is intended as a diagnostic feature to be used when all else fails: experience has shown that it is rarely used—the most frequent use was in checking the translator!

There is a facility by which sections of program may be made optional. This is virtually identical with that provided for machine-code programmers: groups of instructions are either included in the program or ignored, according to the position of a control panel key during input. The facilities of a check-point routine are provided by the system of *optional printing*, employed by the programmer as follows. Any calculation instruction may be followed by the symbols *n, where n is an integer from 0 to 5 inclusive. (A calculation instruction is equivalent to an ALGOL assignment statement.) Normally these symbols are ignored by the translator, but if the key numbered n on the control panel is depressed during input of the program, certain additional orders are included in the compiled object program following the calculation instruction. While the same key is depressed at run time, these orders cause the printing of *n, followed by the number evaluated by the calculation instruction; if the key is raised the orders have no effect. This printing usually takes place on an output channel other than that being used for the main output of the program. Thus the programmer can specify optional printing at selected points in his program, allocating numbers to give different modes or different degrees of complexity of dynamic printing as convenient. The options required for a particular run must be indicated by key settings during input of the program, and printing of any or all of these options can be obtained by manipulating the keys whilst the program is running. This facility is easy to use, and is very popular with Autocode programmers.

Conclusions

The including of optional printing in a program is a powerful technique for error detection, but it presupposes anticipation by the programmer, and experience suggests that other facilities are helpful, especially to the novice and non-professional programmer. Probably the most important facility after optional printing is adequate monitoring of error stops. Above all, however, the diagnostic system must be flexible, and must provide *selective* printing.

When a computer is run under control of an operating system there is a great temptation to restrict the diagnostic facilities in order to keep jobs flowing through the system. We believe this to be a mistake: the system should include elaborate trapping, monitoring and checking procedures, and the programming language must allow the user to specify the diagnostic information he requires. Checking-out a program involves some interaction between programmer and machine, and one should be prepared to sacrifice some efficiency in the processing of jobs in order to make this interaction more effective. Efficiency is not measured entirely by the proportion of time the computer is doing "useful" work: the overall time between presentation of a problem and production of correct results is another measure, and one can gain here at the expense of absolute machine efficiency.

We hope to describe in a later paper how the programming system currently being designed at Cambridge for a new computer attempts to meet these requirements, and to describe some new facilities for error diagnosis which are being incorporated.

The techniques described in this paper are the result of a continuous process of development in which many members of the Laboratory staff, past and present, have taken part. We wish to acknowledge these contributions: it is impossible to assign individual credit for all the ideas, but a large share must certainly go to Dr. D. J. Wheeler. One of us (D. F. H.) is indebted to the Department of Scientific and Industrial Research for a Research Studentship.

References

BARRON, D. W., and SWINNERTON-DYER, H. P. F. (1960). "The Solution of Simultaneous Linear Equations using a Magnetic Tape Store," *The Computer Journal*, Vol. 3, p. 32.

BROOKER, R. A., *et al.* (1961). *Mercury Autocode Manual*, Ferranti Ltd., List CS242A, 2nd edition.

GILL, S., (1951). "The diagnosis of mistakes in programmes on the EDSAC," *Proc. Roy. Soc.* A, Vol. 206, p. 538.

HARTLEY, D. F. (1962). *EDSAC 2 Autocode Programming Manual*, University Mathematical Laboratory, Cambridge, 2nd edition.

WILKES, M. V., WHEELER, D. J., and GILL, S. (1957). *The Preparation of Programs for an Electronic Digital Computer*, Addison-Wesley, 2nd edition.

A convention to distinguish letter O from numeral zero

By H. McG. Ross

To avoid confusion when alpha-numeric data and programs are read into computers, a convention is suggested to distinguish between letter O and numeral zero.

For a considerable time difficulties have been experienced in computer work because of confusion between the letter O and numeral zero. With the increasing use of modern symbolic-programming methods, with their greater flexibility and increased opportunity to mix letters and numerals, the position has got worse and has now reached the stage when it is felt that something needs to be done about it.

A variety of suggestions on this point have been made from time to time, including the following.

1. Letter O should be "fatter" than zero.
2. Letter O should be a rectangular shape with rounded corners; this is quite widely used, particularly with ALGOL, but it is not easy to write.
3. There should be a dot in the middle of letter O; however, it is found that one's eye tends to halt at this when reading.
4. Zero may be split at the top, or at top and bottom (when it may be confused with two parentheses), or at the sides. Sometimes this has been used for letter O.
5. Zero should appear as Ø. This is widely used in meteorology and in British Government work, but a string of such zeros (as is common in data-processing work) is definitely unsightly, and there is confusion with phi, which is fairly widely used as a mathematical function or to mean figure-shift.
6. Letter O, or sometimes zero, has been like an inverted Q.

No one of these conventions has become widely accepted, and at the risk of another non-starter the following series of conventions is now put forward.

1. For printed documents (from typewriters, and line-at-a-time printers, etc.), letter O should appear 0 and zero 0.
2. For handwriting, if there is no possibility of confusion, do not bother to introduce any distinction.
3. For handwriting, if confusion might arise, write letter ʘ and zero Ø.

Convention 1 has the advantages that the eye will run on for ordinary reading but the difference may be found on closer scrutiny; it is satisfactory for capital and small letters; it may be satisfactory for future alpha-numeric optical character-recognition systems.

Convention 3 is very easy to write, and the mark in zero hints at the oblique line of scheme 5 above.

A similar problem arises with confusion between capital letter I, lower-case letter ℓ and numeral one. Here the solution is easier, and a corresponding set of conventions could be:

1. For printing, letter I, letter l, and numeral |, 1 or 1.
2. For handwriting, letter I, letter ℓ or *l*, numeral | or 1.

(The only difficulty here is to write the serif of 1 small enough and at a slope, to avoid confusion with 7; some people might like to follow the Continental European practice of a small bar through the seven, 7.)

No difficulty has been found in following these conventions for all the printing machines used with Ferranti data-processing systems, which will in future be equipped as standard in this way.

Acknowledgement is given for contributions made by Messrs. R. W. Bemer, G. E. Felton, and C. R. Merton, and for permission to publish by Ferranti Ltd.