TECH LIBRARY KAFB, NM
0060538

NASA CONTRACTOR REPORT

NASA CR-13

NASA CR-1397

LOAN COPY: RETURN TO
AFWL (WLIL-2)
KIRTLAND AFB, N MEX

A SURVEY OF CURRENT DEBUGGING CONCEPTS

by Wallace Kocher

Prepared by
WOLF RESEARCH AND DEVELOPMENT CORPORATION
Bladensburg, Md.
for Goddard Space Flight Center

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION • WASHINGTON, D. C. • AUGUST 1969

TECH LIBRARY KAFB, NM

0060538

NASA CR-1397

A SURVEY OF CURRENT DEBUGGING CONCEPTS

By Wallace Kocher

Distribution of this report is provided in the interest of information exchange. Responsibility for the contents resides in the author or organization that prepared it.

Prepared under Contract No. NAS 5-9756-99 by
WOLF RESEARCH AND DEVELOPMENT CORPORATION
Bladensburg, Md.

for Goddard Space Flight Center

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION

For sale by the Clearinghouse for Federal Scientific and Technical Information
Springfield, Virginia 22151 – CFSTI price $3.00

TABLE OF CONTENTS

Section		Page
I	INTRODUCTION	1-1
1-1	Need for Debugging Development	1-1
1-2	Scope of This Report	1-1
1-3	Debugging Aspects, Definitions	1-2
1-4	Categories of Programming Errors	1-3
1-5	Phases of Debugging	1-4
1-6	Error Diagnostic Procedures	1-4
II	IMP-F PROGRAM DEBUGGING	2-1
2-1	Description of the Programs	2-1
2-2	Programmer Comments	2-2
2-3	Interview Guidelines	2-3
III	CURRENT DEBUGGING CONCEPTS	3-1
3-1	Current Concepts	3-1
3-2	Preliminary Testing	3-1
3-3	Checking the Flow Chart	3-1
3-4	Linking the Flow Chart to the Coding	3-3
3-5	Concluding Steps of Preliminary Testing	3-3
3-6	Assembling and Compiling	3-4
3-7	Assembler Aids	3-4
3-8	Compiler Aids	3-5

TABLE OF CONTENTS (Continued)

Section			Page
3-9	Macro-Instructions		3-7
	3-10	Debugging Macros	3-7
	3-11	Advantages of Debugging Macros	3-8
	3-12	The SDC-SHARE Debugging Package	3-9
3-13	Inserting Diagnostic Instructions		3-11
3-14	Dumps		3-12
	3-15	Postmortem Dumps	3-12
	3-16	Snapshot Dumps	3-13
	3-17	Inconveniences of Present Techniques	3-13
	3-18	Efforts to Improve Dumping Techniques	3-14
	3-19	The ALCOR Illinois 7090/7094 Post Mortem Dump	3-15
	3-20	The UNIVAC 1108 Executive Systems	3-18
		3-21 The Snapshot Dump	3-19
		3-22 The Postmortem Dump	3-20
3-23	Traces		3-20
	3-24	Possibilities for Limiting the Trace	3-21
	3-25	Trapping	3-22
	3-26	Static Trace	3-22
	3-27	Improved Approach to Tracing	3-23
	3-28	The SEFLO Tracing Technique	3-25
3-29	Dynamic Debugging		3-25
	3-30	FORTRAN Debugging Languages	3-27

TABLE OF CONTENTS (Continued)

Section			Page
		3-31 One FORTRAN Debugging Language	3-27
		3-32 DIAGNOSE, Another FORTRAN Debugging Language	3-28
		3-33 BUGTRAN, Another FORTRAN Debugging Language	3-30
	3-34	TESTRAN, for IBM System/360	3-32
		3-35 Introduction to TESTRAN	3-32
		3-36 Procedure	3-33
	3-37	The "Rip Stopper" Method	3-35
	3-38	The WATCHR III System	3-36
3-39		TIDY and EDIT for FORTRAN	3-40
		3-40 The TIDY program	3-40
		3-41 The EDIT Program	3-42
3-42		Automatic Flowcharting	3-43
		3-43 The AUTOFLOW System	3-43
3-44		Testing	3-44
		3-45 Purposes	3-44
		3-46 Problems	3-45
		3-47 Logical Tree	3-45
		3-48 Maintaining Test Data	3-45
		3-49 Satellite Test Data	3-46
		3-50 Data Simulation Computer Program	3-47
3-51		Program Correction	3-48
3-52		Restarting	3-49

TABLE OF CONTENTS (Continued)

Section			Page
IV	ON-LINE DEBUGGING		4-1
4-1	On-Line Debugging Techniques		4-1
	4-2	Criteria for an On-Line Debugging System	4-2
	4-3	The Programmer-Controlled Breakpoint	4-2
	4-4	Assembler-Language Program Debugging: The DDT System	4-3
	4-5	Higher-Level Language Debugging	4-7
		4-6 The LISP System	4-7
		4-7 The QUIKTRAN System	4-9
		4-8 The Incremental Compiler	4-9
REFERENCES			R-1
BIBLIOGRAPHY			B-1
APPENDIX - SELECTED DEBUGGING REFERENCES			A-1

SECTION I
INTRODUCTION

1-1. NEED FOR DEBUGGING DEVELOPMENT

Debugging is a general term that refers to the location and correction of errors that occur in computer programs. The debugging activity may consume considerable portions of the total time required for the complete program development. However, the importance of this area of program development has often been minimized. The purpose of this report is to place some emphasis on this important area and to present some of the current debugging concepts which may be considered for implementation.

Computers are continually becoming faster and more complex, and applications becoming more extensive, more intricate, and more time-critical. Thus, the need for a greater effort in the area of debugging is obvious. Two of the major factors which make program errors both harder to find and harder to tolerate are time-constrained programming and closed-loop applications. These are particularly applicable to many of the space-related programs.

1-2. SCOPE OF THIS REPORT

The project that has furnished impetus to this report is the IMP-F (Interplanetary Monitoring Platform). Because the IMP-F computer program complex will form the basis for future IMP efforts, it was necessary to document the present version as fully as possible in order to reduce the lead time and cost of developing the follow-on IMP

complexes. One important area (and the subject of this paper) is debugging. This report includes, in addition to the documentation of the IMP-F debugging effort, a survey of current debugging concepts which may be considered for implementation in future programs.

1-3. DEBUGGING ASPECTS, DEFINITIONS

For the purposes of this report, debugging will be considered to include logical and clerical errors but not machine malfunctions. To be more specific, debugging includes prevention, detection, diagnosis, recovery, and remedy of program errors. It is an on-going process that lasts for the entire lifetime of the program. The following definitions clarify the above mentioned aspects:[1]

a. Prevention - protective action taken against the occurrence of errors or omissions.

b. Detection - determination of the occurrence of an error or of an omission at the earliest point possible so as to limit the consequences.

c. Diagnosis - analysis of the error that was detected.

d. Recovery - provision for a means of bypassing an error condition, especially one of an intermittent nature, with a minimum loss of time and effort.

e. Remedy - rapid remedial action employed to prevent further occurrences of the same or similar errors and omissions.

1-4. CATEGORIES OF PROGRAMMING ERRORS

Programming errors fall into two basic categories - logical and clerical.

Logical errors are those which involve problem analysis. Although they occur less frequently than clerical errors they are usually more serious and more difficult to detect. Since they often occur early in the program writing process, they have a tendency to be overlooked. Frequently they are a result of a misunderstanding of the problem and requirements. Often, in very complex programs, errors result from failure to cover all possible alternatives that may arise, from improper identification of storage areas, etc.

Clerical errors are those errors which involve the input deck or other input media; these errors are sometimes referred to as physical errors, and errors made in writing symbolic instructions. Numbers may be transposed, cards may be omitted or mispunched, a symbolic name may be spelled more than one way, to mention a few types of these errors. Frequently these errors are the result of carelessness. One common source of such errors may be avoided by keeping program listings and decks up to date; old versions should be destroyed or at least marked.

1-5. PHASES OF DEBUGGING

The debugging process is generally thought of as consisting of several phases:[2]

a. Desk checking.

b. Assembled or compiled program checking.

c. Program-testing using test data.

d. Error diagnostic procedures.

e. Running of program using actual data.

1-6. ERROR DIAGNOSTIC PROCEDURES

In this report, emphasis is placed on the phase dealing with error diagnostic procedures. A number of programs and programming techniques are available for diagnostic purposes. Their underlying principle is to provide information about a particular portion or portions of the program as control passes through the specified instructions. The amount and location of the information will vary depending on the nature of the error. Proposals have been made for developing software that will allow debugging to be conducted using the same language as the source program; however, few of these have been implemented. Few if any computer systems or major languages today provide comprehensive symbolic debugging software packages.

SECTION II
IMP-F PROGRAM DEBUGGING

2-1. DESCRIPTION OF THE PROGRAMS

Information concerning the debugging of time-correction and decommutation programs for the IMP-F spacecraft was collected through a series of interviews with programmers and project managers associated with the project.

The IMP-F time-correction and decommutation programs were written in FORTRAN and the UNIVAC 1108 assembler language, SLEUTH II.

a. The time-correction programs perform two basic operations:

   1. Correct range time data by removing discontinuities and filling in missing time data.

   2. Establish fairly precise correlation of spacecraft events with range (earth-based) time signals by correcting for propagation delay.

b. Decommutation operations are concerned with stripping out, formatting, and providing pertinent data for each experimenter.

2-2. PROGRAMMER COMMENTS

In general, the programmers associated with the project felt the diagnostic system that was available within the 1108 Executive system was adequate for their debugging needs. The 1108 diagnostic system provided for snapshot and postmortem dumps. However, some comments and suggestions were made that may provide some insight into debugging procedures that may be considered for implementation in future IMP programs. Among these suggestions were the following:

a. Debugging aids may be too expensive and involve excessive overhead. They may require excessive core storage and increase running time beyond practical limits.

b. Care should be taken when implementing debugging aids not to duplicate aids that are already provided for by the manufacturer's system.

c. During the writing of the IMP-F programs, a change in personnel occurred. The programmers that took over the project felt a trace routine would have been desirable, but that the implementation of a built-in trace routine should have been accomplished during the design phase, since a great deal of work is entailed when such a routine is implemented during later phases.

d. The WOLF programmers assigned to the IMP-F project were able to use snapshot dumps for their tracing needs. However, a problem

associated with the postmortem dump was that whenever the system was lost through a drum failure, illegal instruction, system failure, etc., then the dump was also lost.

e. Imbedded comments were not only useful for debugging purposes but were valuable in estimating running time.

f. Two specific suggestions were made concerning future IMP programs:

   1. The first was to develop routines that would check experimenter's tape for known characteristics such as label, format, size, and special characters, and especially data.

   2. The second was to develop a better method of obtaining test data.

2-3. INTERVIEW GUIDELINES

The following is a copy of the checklist of subjects in conducting the interviews with the IMP-F programmers and project managers.

CHECK LIST

1. Debugging techniques employed.

2. Preliminary phases.

a. Flow chart and coding

b. Clerical errors

c. Logical errors

3. Assembling and compiling phases, suggestions for improvements.

4. Phases beyond assembling.

a. 1108 package adequacy

b. Other concepts

c. Suggestions

5. Testing.

a. Tests employed

b. Generating test data

6. Editing or cleaning-up programs.

7. Management practices.

a. Coordination, etc.

b. Suggestions

8. On-line debugging.

9. State-of-art.

   a. Techniques

   b. Concepts

10. Greatest problem in debugging IMP-F, special or unusual problems.

11. Knowledge of debugging literature.

SECTION III
CURRENT DEBUGGING CONCEPTS

3-1. CURRENT CONCEPTS

This section discusses debugging concepts surveyed for future implementation in programs such as IMP-F. While the concepts described here may not be entirely new or unknown, and while the list may not be complete, nevertheless these concepts represent a typical range of current practices or developments, and as such may suggest known approaches to the subject of debugging.

3-2. PRELIMINARY TESTING

Preliminary testing, manual checking or desk checking consists of a detailed review of the program by the programmer. This phase of testing may be viewed as having two parts, the first of which is a review of the general logic and degree of completeness. The second part is a manual dry run using sample data during which the programmer keeps track of register contents, modified instructions, switches, etc., while following the flow of the program. Special emphasis may be placed on unlikely situations.

3-3. Checking the Flow Chart

During this phase of testing, the flow chart can and should be utilized both for detecting errors and as a logical tool for debugging procedures. The desired level of detail for such flow charts would show every

decision or branch and every subroutine execution. Seven basic checks should be performed on the flow chart.[3]

a. The beginning and end should be clearly marked and present.

b. Each decision symbol should have at least two alternatives, and each possible result of the test should be represented by some flow line. Such flow lines should be traced so as to insure that the correct path is taken for the specified condition. Each condition should be provided for.

c. Operation symbols should have no more than one exit and entry line.

d. Flow lines must be connected at both ends, each line must originate at some symbol and terminate at a symbol or merge point.

e. On flow charts of that level of detail which includes partial coding, no variable name may appear on the right-hand side of a formula, in a decision symbol, or as output if it has not been defined somewhere in the program in an input list or on the left-hand side of a formula (any statement calling for the assignment of values and quantities). Each variable name which appears at least once in a program in an input list (including the outputs of subroutines) or on the left-hand side of a formula is unused if it does not appear on the right-hand side of a formula, in an output list (including the quantities supplied to subroutines by the main program), or in a conditional test. It is, therefore, unneccessary.

f. Switches must be set or defined at some point previous to usage in the flow chart, and each possible position of the switch must be set at least once somewhere on the chart.

g. Each loop must have the following features:

1. Initialization.

2. A process consisting of one or more statements.

3. One or more tests to terminate loop.

4. A method for reiterating the loop.

3-4. Linking the Flow Chart to Coding

After the programmer is satisfied with the flow chart, the coding is performed. During the coding process, care should be taken to link the flow chart with the coding. If changes in the flow chart are required, the above checks should be considered to ensure that new errors have not been propagated.

3-5. Concluding Steps of Preliminary Testing

The preliminary testing phase progresses from a rough analysis through more detailed analysis to a detailed check of the coding. If time permits, recopying of the diagram may be desirable because close reexamination may disclose errors previously hidden by sloppy drawing or erasing. If time and personnel permit, it is extremely

helpful if a second person can assist in the preliminary checking procedures. A listing of the cards should be made and checked against the coding forms, looking for keypunch errors and sequence.

3-6. ASSEMBLING AND COMPILING

Computer systems supply a certain amount of diagnostic control in their assembly or compilation programs. However, this type of diagnostic aid is in most cases designed to assist the programmer only up to that time at which the computer is able to interpret and execute the instruction sequences. The diagnostic notations produced usually deal with clerical type errors and seldom with logical errors. Some systems produce diagnostics that indicate the severity of the error, i.e., absolute, probable, etc. An error-free compilation or assembly does not indicate that the program is debugged, only that certain types of coding errors have been eliminated. A weakness of both compiler and assembler debugging aids lies with the fact that the statements are examined singly. The processor is for the most part unable to detect logical errors in transfer of control.

3-7. Assembler Aids

Most assemblers produce a program listing with diagnostic notations adjacent to the instruction containing certain errors. Errors of this nature include the following:[4]

a. Undefined or multi-defined symbols.

b. Invalid operation codes.

c. Invalid operands in a symbolic expression.

d. Invalid or omitted address.

e. Improper use of certain pseudo-operations.

f. Invalid names in name field, such as names containing unauthorized special characters, etc.

The diagnostic notations are generally singular. Some systems produce two notations in the event that two detectable errors occur in a single statement.

A list of symbol references (the part of a computer listing sometimes called an external symbol dictionary, or a concordance, or a cross-reference table) may be extremely useful in revising as well as debugging a program. In the event that a change is made that affects the definition of a symbol, it is important that all references to that symbol be checked. Special care is required when a coding sequence is deleted, since it is likely that one or more symbols may be affected.

3-8. Compiler Aids

The diagnostic notations produced by compiler systems are similar to those produced by assemblers. They are generally printed with the listing and where possible make reference to particular statements. The following summary shows typical diagnostics produced by one FORTRAN compiler:[5]

a. Individual statements:

1. Invalid statement - reserved word, excessive length, etc.

2. Mixed arithmetic expressions - containing real and integer constants and variables.

3. Misuse of parentheses - an unequal number of left and right parentheses.

4. Illegal expression - two operators may have been used in succession, etc.

5. Subscripting a fixed-point variable.

b. Interaction of two or more statements:

1. Referenced statement is not present or is unnumbered.

2. Omission of a DIMENSION statement when subscript variables are used.

3. Undefined variables - a variable that appears in an arithmetic expression (on the right side of an arithmetic statement) that is not defined or assigned a value earlier in the program.

c. Capacity of words or of memory:

1. Table limits have been exceeded - during compilation FORTRAN uses a number of tables; if the programmer uses an excessive number of statements or other features a diagnostic notation results.

2. Memory capacity of machine is exceeded.

3. Number exceeding precision of computer - number too large or too small, requiring an excessive number of decimal places ($10^{\pm 38}$) is a typical limit of a 36-bit binary word computer.

d. Logic of program:

1. Flow is unable to reach at least one part of the program because of branches; program flow does not include one or more executable statements.

2. Illegal entry to a DO loop - entering a DO loop at any statement in the range of the loop other than the "DO" statement itself.

3. Excessive levels of nested loops - this limit is determined by the individual computer.

A number of these diagnostic notations would be applicable to any algebraic-language program (a.3, a.4, b.1, b.3, c.1, c.2, and d.1). Others only apply to FORTRAN and certain other compilers (a.1, a.2, a.5, b.2, c.3, d.2, and d.3).

3-9. MACRO-INSTRUCTIONS

3-10. Debugging Macros

Certain special macro-instructions may be used during the running of an object program to provide debugging information that is not normally supplied. Instead of using interruptions or trap features, the macro calls are inserted in

program. These macro-instructions expand into coding that calls for output routines that are under control of the operating system. Loops, when required, are also provided for by these macro-instructions. The information produced may appear in the same form as does a dump or trace.

Existing machine instructions may be modified by macro-definitions to supply diagnostic information. An example of this technique is to modify a storage instruction so as to write out the data being stored. However, a more common technique is the insertion of macro-instructions in the program for the specific purpose of providing diagnostic information.

3-11. Advantages of Debugging Macros

Particular use is made of the writing or printing instructions. Certain condition stipulations may be employed as well as format specifications (similar to those in a dump) such as hexadecimal, decimal, octal, or BCD. Reassembly of the program is required in order to insert the additional coding of the macro-instructions, a process that is not necessary with the use of dumps or traces. There are, however, several advantages to this technique:[6]

a. Symbolic references may be made to locations or blocks of locations that are to be dumped; in the event that coding changes are made, there is generally no need to change the macro calls, which is often the case with control cards when using snapshot dumps since their addresses are absolute; however, many systems provide for symbolic addressing for their dumping routines.

b. Flexibility is provided the programmer; he may specify any sort of dump he wishes. They may resemble traces or dumps or any combination of the two. The macro-instructions used for dumping are generally quite simple in structure, primarily involving coding for a loop, for an output calling sequence, and for storage of certain registers. Because information may be displayed in various formats, the dump may include symbolic information which facilitates the identification of words and registers.

c. Macro-instructions are relatively easy to write. The advantage of writing macro-instructions for dumping purposes over corresponding coding for a monitor system or processor is that instructions must be general in nature and each piece of information must be stored in memory tables or remain in instructions to be interpreted. The coding employed with the use of macro-instructions is specific and is tailored to the user's needs; also, the instructions remain in executable form within the macro-instruction expansion.

3-12. The SDC-SHARE Debugging Package

The debugging package which is included in the SDC-SHARE[7] operating system uses a set of macro-instructions that will produce selective dumps in the source language at any point during the execution of the object program. The execution of these macro-instructions does not affect the operation of the object program. The debugging macro-instructions are defined within the compiler and will be inserted into locations specified by the object code during

compilation. The debugging macro-instructions are classified according to three categories: information macros, conditional macros, and modal macros.

a. Information Macros. This type of macro-instruction is responsible for the actual dumping of information. This information is written on a tape during the execution of the object program after which it is translated and written on an output tape. The format for the output may be included in the information macro, otherwise it will be described in a dictionary by successive location symbols. The dictionary is developed during compilation, each entry containing a location symbol and a corresponding format code to identify the contents of the described cell in memory. Through the use of this dictionary, symbolic output may be produced. The format codes control the printout of all information from the specified location to the location of the next symbol encountered.

b. Conditional Macros. This type of macro-instruction is used to control the execution of information macros. These macro-instructions are placed immediately before the information macros they control; their control is terminated when the next nondebugging macro-instruction is encountered. Through the use of these conditional macros, a programmer may specify conditions which must be satisfied before the information macro can be performed.

c. Modal Macros. This type of macro-instruction permits the program to specify the mode to be used in interpreting subsequent information, or

to set or reset certain parameters used by the debugging system. Besides defining the mode of operation for either conditional or information macros, they also provide internal controls for the debugging supervisor and some control of output format. Modal macros generally remain in effect until countermanded by another modal macro or another macro-instruction is encountered that sets all modal macros to normal conditions.

3-13. INSERTING DIAGNOSTIC INSTRUCTIONS

A relatively simple but effective aid in the debugging process consists of inserting throughout the program temporary diagnostic instructions which will print out messages that display the results of the program at various stages. Generally such insertions are placed at the conclusion of program segments. This technique will provide information as to the flow of the program and it may also be used to indicate program status (the existence of an error condition). In order to ascertain the status of the program, various tests are employed. Through the use of a test deck (a set of test data for which the results at various stages are known), the area in which the program departs from the correct procedure can be detected.

The messages that are printed out may be used to show the overflow of storage blocks or a diverging series of values. Externally controlled status reports containing such information as item counts, important results, calling-sequence parameters, accession assumptions, pointers, and summary totals may be produced. One may devise tests for all parameters and provide meaningful messages. In the event that it is not possible to correct the error, it may be desirable to allow the system to approximate the interim results and

continue, especially in the early stages of the system's development. When the preceding condition occurs, it must be clearly noted. Options should be available to list selected subsets of inputs and outputs from peripheral devices to each routine in the system.

A method of cross-control checking may be provided by producing a printout of program assertions at the beginning of each program segment and of program expectations at the end of each program segment.[8]

The inserted instructions and corresponding messages should be coded in some consistent, recognizable way which will permit their mechanical removal once the program is finalized.

3-14. DUMPS

Dump is a term that is generally applied to a listing of the contents of a storage device; it may include all or part of the internal storage, as well as registers. Dumps may be classified into several categories: snapshot dumps (dynamic) or postmortem dumps (static).

3-15. Postmortem Dumps

The postmortem dump is performed at the termination of the object program. This termination may result from some form of error or by intentional instruction. At times it may be advisable to list the entire memory except possibly the monitor system, since the effects of errors are unpredictable and often wide-spread. In early testing this method of dumping may be the only way to ensure that some unsuspected influencing factor was not overlooked.

3-16. Snapshot Dumps

The snapshot dump is a selected or dynamic listing of registers and specified memory locations printed out upon the execution of strategically located dynamic-dump instructions (breakpoints or checkpoints). The programmer may insert these dump instructions in the program before it is run the first time; or he may insert or change them before a later run, after errors have appeared. This form of dump is useful when the programmer has little or no idea as to the location of the program error. If the program has previously been segmented, it is relatively easy to insert dynamic dump instructions so as to isolate at least the earliest error to some particular segment. Then as the general location of the error is identified, the snapshots (dynamic dumps) are inserted more frequently within smaller areas of the segments. Each segment may be checked individually. The snapshot dump is an extremely useful debugging aid for the programmer (who should know machine language).

3-17. Inconveniences of Present Techniques

The dumping techniques still widely used today, even in some of the most modern and advanced computers, are claimed to be obsolete, and produce very inconvenient listings.[9] Criticisms of the dumping techniques include the following:

1. If the program has run for some time, earlier parts which may be meaningful are lost.

2. Great amounts of useless information are sometimes generated.

3. The listings are difficult to analyze. One must perform a great deal of decoding; know the main characteristics of a particular machine, its internal language, the operating system, and the object (machine) codes corresponding to instructions written in the source language.

3-18. Efforts to Improve Dumping Techniques

Various attempts have been made to refine and improve the dumping techniques. These are, in essence, attempts to overcome the previously listed criticisms.

a. The use of symbolic address designations for the purpose of dumping is provided for by certain processor systems. The advantages of this approach are evident - consider the advantages of symbolic coding over machine language coding. In order to facilitate such a technique a symbol table associated with the program being debugged must be accessible. The dumping routine must convert the symbolic addresses on cards to absolute addresses.

b. The operating system must assume temporary control from the object program during the time that the dumped information is being written out. A common way of accomplishing this is through the use of a trap. The trap is a special form of a conditional breakpoint which is activated by the computer itself or by conditions imposed by the operating system, or, by a combination of these. The trap causes control to pass to a specified memory location within the operating system. A method of

returning control to the object program must also be provided for. When the operating system encounters debugging control instructions, it must provide means by which the trap will occur at the specified locations. A record must be made of the location in the object program from which control was transferred to the operating system.

c. The efforts made on improving dumping techniques have been directed toward the pinpointing of dump locations, increasing the selectivity of the storage locations to be dumped, and editing of the printed output.

3-19. The ALCOR Illinois 7090/7094 Post Mortem Dump

The ALCOR Illinois 7090/7094 Post Mortem Dump (PM-Dump) provides an example of improved techniques in the area of postmortem dumping.

a. The objectives of this effort include the following:[10]

1. If the program runs successfully, the execution time must not be increased.

2. Specialized knowledge in excess of what was needed to write the program should not be required for understanding the data provided by PM-Dump.

3. Events occurring before the failure that have a possible influence on its cause should be recorded in PM-Dump.

4. All information that is not pertinent to the determination of the cause of the failure should be avoided.

5. Information pertaining to specific machine characteristics should be held to a minimum.

b. The PM-Dump is not a single program but consists of several programs, each of which generally pertains to one of the various stages of program processing (translation, loading, execution).

1. During the translation phase the compiler generates dump information which relates the object code generated by the compiler to the original source-language program. This dump information provides the basis for failure analysis. Since the information which relates the source program to the object code is available only at compilation time, the compiler must save this information. The modification of an existing compiler may be a difficult and tedious task. The dump information is moved to secondary storage until later when it is used at dump time. A loading map is useful but it is not necessary. Such a map would be helpful in locating the point of failure and in determining how to access the proper dump information.

2. At the beginning of program execution, a short routine indicates to the operating system where control should be transferred in the event of an unsuccessful termination. This is the only routine that slows program

execution if a dump is requested. The routine consists of about 100 machine instructions which set switches in the operating system or initialize a few error routines.

3. At the time of the failure, the operating system transfers control to a connecting routine which saves the relevant contents of memory (the program and working storage areas) for the main dumping routine. This routine then calls on the main dumping routine and passes to it such information as the type of failure and the contents of the instruction counter. Because of the relatively long length (about 3600 machine instructions) of the main dumping program, it is desirable to load it into main memory only after a failure occurs. This is accomplished through the use of the connecting routine.

4. The main dumping routine coordinates and analyzes the information made available to it by the previous routines, and presents the programmer an analysis of the state of the program and of its history at the time of failure. The object code at execution time is not affected by the dumping routine except in the case of one call. During compilation all preparations are made for the dumping process and the analysis is performed only after the failure has occurred. The dumping routine provides the programmer with the following information:

(a) Lists local information such as names and values of variables and object codes which concern the failure point in the program.

(b) The name of the subprogram or procedure is listed if the failure point was not in the main program.

(c) A listing of names and present values of variables is produced beginning with the block in which the failure point occurs and proceeding to surrounding blocks until all blocks in a given procedure have been processed. Unless the block terminates in the main programs, the point from which it was called is determined and is considered a new failure point.

3-20. The UNIVAC 1108 Executive Systems

The UNIVAC 1108 executive systems provide for both postmortem and snapshot dumps.[11] Snapshot dumps may be initiated when a source language element is processed to relocatable element form or when relocatable elements are combined by the collector into an absolute program. Postmortem dumps are initiated by a control statement. Dumped information is written in a diagnostic file, and is read back for editing after the program being tested has terminated. Library subroutines write out dumps, save and restore all

of the program's environment. The snapshot facility may be employed by any processor.

3-21. The Snapshot Dump

The selective nature of the snapshot dump is achieved through the use of seventeen procedures. These are divided into three categories - conditional, dump, specification.

a. The conditional procedures may precede or be interspersed among a list of dump procedures. The purpose of the conditional procedures is to determine when or if dump procedures are to be activated.

b. The dump procedure generates a calling sequence which writes out the information comprising the desired dump. If no conditional procedures are used, the dump procedures will always produce output. These instructions enable the programmer to select the locations and amount of information to be dumped.

c. The specification procedures provide a buffer area and format specifications. A number of standard formats are provided by the system and these would suffice in most cases; however, for unusual situations, one may define special formats. An area of core is defined into which information from tapes or drums is read. The programmer is also able to save dumps up to a certain point in execution and then delete them at his discretion. The overall control of calls to debugging procedures is maintained by two instructions which

activate or deactivate references made to the debugging procedures.

3-22. The Postmortem Dump

Each diagnostic routine that is part of the program is processed serially. A common exit in diagnostic library routine will check to see if the next instruction is a call to the diagnostic system. This technique ensures that a series of calls will not be interrupted by the activities of another subprogram.

The postmortem dump executive control statement is used to dump current core memory following the execution of a task. These dumps may consist of overlay segments, elements, or specified parts of elements. Several options are available to the programmer for selecting output format and defining core areas to be dumped.

3-23. TRACES

A trace is an interpretive diagnostic technique which provides an analysis of each executed instruction resulting in a listing of such information as the contents of words and registers as they are modified and the order in which the instructions are performed. This technique does not require recompilation and is particularly useful for the programmer who is having difficulty following the logic flow since the trace provides a means by which one may follow the course of a program as it is executed.

However, a detailed trace producing a line of information for each instruction greatly slows down the execution of the program and generates enormous and excessive listings. This approach, particularly diastrous on large systems, is generally discouraged and reserved for use as a last resort.

Another problem associated with the tracing technique is the lack of flexibility of many systems in that the programmer cannot provide his own subroutines for extended usage, therefore, the system software must provide the capability for selective or conditional tracing techniques. Because of the increasing sophistication of both computer hardware and software, as exemplified by address indexing, indirect addressing, automatic allocation of relocatable subroutines, numerous resident monitor and library routines, paging, chaining, and both manual and automatic overlay facilities, there are increasing difficulties in following the flow of a program, so that it is often desirable for the system software to have an efficient trace routine available.

3-24. Possibilities for Limiting the Trace

Possibilities for limiting the trace include the following:

a. Tracing only a single type of instruction such as storage, loading, arithmetic, branching, index, shifting, skipping, or input/output instruction.

b. Tracing a combination of these specified instructions.

c. Tracing only address modification.

d. Limiting the number of tracings performed on a single loop.

e. Tracing in only selected parts of the program while other parts run at normal speed.

f. Tracing only a single register or combination of registers.

3-25. Trapping

As in the dumping technique, the operating system must assume temporary control from the object program during that time the diagnostic information is being written out. This is generally accomplished through the use of hardware interrupts or interpretive software routines (traps).

The IBM 7090 provides a special transfer trapping mode which enables the object program to run at normal speed and to slow down only when a transfer is executed and diagnostic information is written out. While running at normal speed the flow of control passes sequentially from one instruction to the next, except in the case of a skip after which it passes to the next instruction. This technique is particularly useful when one is only trying to establish the structure of the program.

3-26. Static Trace

A static trace technique was developed by the Mitre Corporation for use on the IBM 7090.[12] It is referred to as the Masked Search Program. Through the use of specification cards, the programmer specifies a mask and a block of storage to be searched by the program. The Masked Search Program then lists the locations that are in accordance with the specification cards. An address mask may be used to

identify all locations that are related to a particular entry point in a subroutine; after obtaining a list of the locations having this entry point as their address, a listing of the storage area being searched would normally tell where in the search area the subroutine is being entered. Through the use of an op-code mask the programmer can locate all instructions which change memory. One may also determine how certain storage locations are used.

3-27. Improved Approach to Tracing

An approach increasing the speed and efficiency of trace techniques is presented in a paper by Eugene I. Grunby.[13] This approach is basically an attempt to allow the programmer to exercise his skill and ingenuity in developing the trace routine, and to provide flexibility in the selection of subroutines thereby eliminating unwanted processing.

The method of allowing for this flexibility in selecting subroutines is based on the use of a single trace routine which makes available the same basic information to each of a number of subroutines. The basic information provided by the trace routine generally includes the origin and destination addresses that represent a change in sequential addressing. This approach enables the subprograms to be independent. A set of subprograms each of which serves a specific function may be developed. Each of these subprograms may call another, thus a composite of functions can be selected to meet task requirements. The programmer is able to select standard subprograms in any combination or he may construct his own.

Mr. Grunby's approach also includes a method of eliminating excessive and useless printout. The first step is to eliminate all but the most essential information, the

addresses or origin and destination involved in a branch instruction. A second step involves the use of a cyclic table. Trace information is stored in a core table and is used for a selective listing at the end of the job, but because of the limitations of memory available, the trace information is stored on a cyclic basis, overlaying from the top to bottom. Options should be provided to allow the programmer to select the length of the table and make the most efficient use of this limited storage area. Thirdly, use of format options or other options which maximize output on the printed line and that are generally supplied by most systems should be employed when the table is being listed.

Among the most significant applications for the tracing technique just described (as implemented on the UNIVAC 1107) are the following:[14]

a. Dumping of preselected memory locations at the time of each traced instruction.

b. Continuous testing for branches to invalid destinations, and initiate an error routine when the event occurs.

c. Continuous testing to determine if and when an instruction or I/O operation is exceeding storage limitations and which one is at fault, and initiation of an error routine when this event occurs.

d. Computation of the distribution of operation codes in executing program; it may at the programmer's request include library and resident monitor routines.

e. Continuous detection and recording of points where characteristic overflow, characteristic underflow, divide overflow, occur.

f. Combinations of the preceding applications.

3-28. The SEFLO Tracing Technique

SEFLO, SEquence-FLOw,[15] a tracing technique, is described in a paper by Abrom Hisler. This technique may be used either for FORTRAN V or SLEUTH II on the UNIVAC 1108. SEFLO enables the programmer to follow changes in contents of selected registers and storage locations. The package consists of two subroutines and the sequence cards which call for the contents of trace whenever requested during running of the object program.

3-29. DYNAMIC DEBUGGING

Tracing and dumping each has its advantages and limitations. Any concept that employs the advantages of each in a single technique can be referred to by a number of titles, but for the purpose of this discussion shall be referred to as dynamic debugging. This concept implies that the process takes place as the program is in progress and that a certain degree of analysis is performed, producing diagnostic information in addition to a listing such as that produced by traces and dumps.[16] The basic concept of dynamic debugging is simply to provide the programmer with a snapshot-type dump each time that predefined criteria are met.

The main restrictions placed on dynamic debugging are the resultant size of available memory and the ingenuity required of the programmer writing the analysis program. The following is an illustrative list of the diagnostic information this technique may provide.[17]

a. Contents of the location counter, showing the current location of the program.

b. Contents of all other control and arithmetic registers along with the status of such things as overflow indicators.

c. Contents of certain memory locations or blocks of locations. The option to specify mode and format may be desirable; the locations and formats are specified in the debugging program and formats are likely to change from one point in the program to another.

d. A record of the occurrences of a certain condition, as tallied in a counter. Through the use of such a counter, parameters may be specified for various tests, i.e., listing will be produced a certain number of times or not produced until after the condition has occurred a specified number of times.

e. A record (desirable under certain circumstances) of a specified number of the last executed branches.

3-30. FORTRAN Debugging Languages

Since FORTRAN was used on the IMP-F project and is particularly applicable to this type of programming, it is well to consider more specifically those debugging aids associated with FORTRAN. Since the compiler naturally checks for simple clerical errors, and since the FORTRAN language is essentially closed with respect to vocabulary, syntax, and structure, the compiler is able to check the basic tests outlined in discussions of preliminary testing.

3-31. One FORTRAN Debugging Language

Since earlier FORTRAN compilers did not include an independent debugging facility, this often forced programmers to rely on the machine language for diagnostic purposes. A debugging language in a FORTRAN format is now available.[18] This language was designed so that reference may be made to specific statements within the body of the FORTRAN program without becoming an actual part of that program. Many of the statements in the FORTRAN language itself such as GO TO, CALL, RETURN, and STOP are included in this system. Much use is made of subroutines. Conditional situations are particularly applicable to the use of logical IF statements. The DUMP statement provides a convenient method for printing out information when there are no particular format requirements. The DUMP statement, much like the FORTRAN READ and WRITE statements, specifies a list of information to be listed; these specifications may be within the statement or in another statement referenced by the DUMP statement. Through the use of a Hollerith field or quotation marks one may provide notes or messages with the information listing.

The LIST statement enables one to reference the same information by more than one DUMP statement without repeating a list with each DUMP statement. When necessary one may dump the entire program with the statement LIST PROGRAM. One may cause dumps at certain intervals, thus limiting the amount of output through the use of an ON statement.

A single card must proceed each debugging request or group of debugging requests. This card may set maximum limits on the number of requests honored, regardless of type and the maximum number of lines that may be printed, all other diagnostic information being lost. This control card almost forces the programmer to set explicit limits on his debugging request. However, the limit control features may be left off of the control cards; thus, no upper bounds are set.

Since this FORTRAN debugging system allows the execution of a program composed of many parts and having been compiled separately, it is necessary to specify not only which statements are to be affected, but in which subprogram (function or subroutine) those statements are to be found. Each subprogram is referenced by its name. A code is used to indicate that there are no more debugging requests.

3-32. DIAGNOSE, Another FORTRAN Debugging Language

DIAGNOSE[19] is a debugging program that is used to detect the following three common errors which occur during the execution of FORTRAN-63 programs.

a. Erroneous subscripts.

b. Undefined variables.

c. Erroneous DO-loop parameters.

When one of the above conditions occurs during the execution of a program, operation is halted and an error message written on a standard outpu_ unit and an error routine is activated for dumping purposes. The error message is composed of a statement number, variable name, and type of error.

The input when operating with DIAGNOSE consists of source decks, binary decks, and data. DIAGNOSE produces a new souce deck by inserting calls to certain library subroutine in the original source deck. The logical flows and the program results are in no way affected up to the point at which the error occurs.

DIAGNOSE makes two passes through the source deck, as follows:

a. The first pass produces four lists and outputs part of the original program along with other information and a scratch tape. The four lists are:

   1. Arrays and dimensions.

   2. Statement numbers.

   3. Terminal statement numbers and DO loops.

   4. Statement numbers of replacement and CALL statements.

b. The second pass performs analysis on DO statements, replacement statements, IF statements, and CALL statements. During the second pass adjustments are made on statement numbers to insure that the flow of the program remains unchanged. For each

of the above statements, DIAGNOSE may insert CALL statements to two or more routines that do the following:

1. Identifies statement currently being checked.

2. Checks subscripts.

3. Checks the value that has been assigned to a particular variable.

4. Checks variable parameters of a DO loop.

DIAGNOSE produces an intermediate program that will require approximately twenty percent more memory and roughly doubles the running time of the original program. DIAGNOSE requires a compilable program (one that is free of syntax errors).

3-33. BUGTRAN, Another FORTRAN Debugging Language

BUGTRAN[20] is another debugging system that applies to FORTRAN. The checkpoint method requires nearly exact intermediate results which are frequently unavailable; and the trace routine, as it is commonly used, is cumbersome and expensive. Thus, BUGTRAN was developed to help overcome these shortcomings.

a. The BUGTRAN system includes a variable trace, flow trace, trace of program entries and exits, snapshot dumps, conditional termination of the program, and an option to print the comments of the source program. The option to apply BUGTRAN to only specified parts of the program is provided for. The system does not place any requirements

on the compiler, such as generating symbol tables, or affecting the operation of the program. A mechanical means of removing the debugging from the program is also provided for. The following is a summary of BUGTRAN features:

1. Check Variables - a list of variables is specified in a BUGTRAN control statement. If one of these variables appears on the left-hand side of an arithmetic substitution statement or as the index of a DO statement, it will generate a call to the BUGTRAN output routine.

2. Check Flow - all GO TO, ASSIGN and IF statements generate calls to the BUGTRAN output routine.

3. Dump - a statement number along with the variables to be dumped is specified in a BUGTRAN control card. The dump is executed immediately before the execution of the specified statement. This is accomplished through a generated call to the output routine.

4. Check Entries - a call is generated to the output routine each time that a SUBROUTINE, FUNCTION, END or RETURN statement is encountered.

5. Print Comments - all comments generate a call to the BUGTRAN output routine, causing the comment to appear in the output.

6. Terminate - a statement number and a conditional IF clause are specified on a BUGTRAN control card. The condition is tested immediately before the execution of the specified statement, and if the condition is satisfied the program is terminated.

b. A syntax table approach is used to interpret the FORTRAN and BUGTRAN statements. A syntax table is first used to interpret the BUGTRAN control cards. Another syntax table is then generated based on the type modifications specified by the control card to process the FORTRAN statements. Only those FORTRAN statements which are specified are provided for in the syntax table. The major advantages of the syntax table are the following:

1. It is possible to use the same scanner to recognize the various types of statements by merely changing the syntax tables.

2. Changes in the source language require modification of the syntax tables only.

3-34. TESTRAN, IBM System/360

3-35. Introduction to TESTRAN

TESTRAN[21], or test translator, is a facility offered by the IBM System/360 Operating System. TESTRAN aids in detecting faulty logic by providing printed information concerning the actual operation of the program. This facility describes the changing contents of storage areas, registers, and control blocks, and shows the manner in which control flows from one group of instructions to another.

Requests for TESTRAN services are coded in a TESTRAN source module. Each of these statements is a coded TESTRAN macro-instruction, which is replaced with a series of constants by the assembler. In effect, these constants are a control statement that directs the TESTRAN interpreter to perform a specific operation. The decision to perform the next sequential macro-instruction or a logical branch to another macro-instruction is determined by the operation being performed.

3-36. Procedure

The structure of a TESTRAN statement resembles that of a basic assembler language statement. Every statement includes an operation code and one or more operands. A symbolic name may precede the operation code and a comment may follow the operands. The operation code and first operand combine to define the type of operation to be performed and are used as generic names for statements.

a. The general functions of TESTRAN statements are the following:

   1. Recording functions which provide dumps and traces of the problem program.

   2. Linkage functions which control linkage to the TESTRAN interpreter.

   3. Decision-making functions which provide condition testing and condition branching.

   4. Branching functions which provide unconditional branching and subroutine capabilities.

5. Assignment functions which control values of variables in the problem program and of special variables used in decision making.

b. During execution, TESTRAN is able to test for predefined error conditions and take corrective action when necessary. Usually, after corrective action is first taken, the final results of the program are still erroneous, but continued processing would permit the possibility of finding the additional errors.

c. The TESTRAN statements are combined with the program to be tested by either of the following methods:

1. The TESTRAN and the problem program source modules are assembled together and result in a single module.

2. The TESTRAN and the problem program source modules are assembled separately, resulting in separate object modules; these are then processed by the linkage editor to form a single load module.

d. The single load module is then loaded and executed. Requests for testing services are interpreted by the TESTRAN interpreter which is a part of the control program that receives control during program interruptions. Test information along with control information copied from the unloaded form of the load module is placed in a TESTRAN data set by the TESTRAN interpreter.

e. The TESTRAN editor prints the test information in the form of dumps and traces. Like the assembler and the linkage editor, the TESTRAN editor is a processing program that is executed as a job step. It puts test information into a meaningful symbolic format by using control information copied from the load module. This control information includes symbol tables and a control dictionary for each object module that is part of the load module. Both the control dictionary, which is produced as a standard feature of the assembler, and the symbol table, which is an optional feature of the assembler, are placed in the load module as an optional feature of the linkage editor.

3-37. The "Rip Stopper" Method

A major part of the conventional debugging effort rests in going back from that point at which the error manifests itself to its point of origin. Frequently, the faulty program destroys the information that would permit tracking activity. A concept that would permit the identification of an error condition in time to preserve the necessary diagnostic information is presented in a paper by Mark Halpern.[22]

The "rip stopper" concept, presented by Mr. Halpern, requires a processor that is able to operate in two modes (debugging and production). During operation in the debugging mode, the processor would require the following programmer-supplied information:

a. Minimum and maximum limits, and where appropriate, increment size for each numerical variable in the program.

b. Limits within which transfers are legitimate.

While operating in the debugging mode, all computation and transfer of control would be executed interpretively. If any limits are violated, action specified by the programmer is initiated.

Provision should be made for the option to generate, upon demand, coding that is interpretively executed, like that produced by the debugging mode, with the purpose of computing the execution time of the compiled instructions rather than testing their validity.

The print-out of diagnostic information should be structured, interpreted, captioned, and ordered so as to follow the steps taken in normal debugging procedures. Tables should be in tabular form, character strings in linear form, numerics in the appropriate external format and base mode, and labels and comments should be used freely. Parts of a single logical item that are scattered throughout the memory must be collected and presented in a unified form.

3-38. The WATCHR III System

An excellent example of a comprehensive debugging package is the WATCHR III[23] system that was developed at New York University for the CDC 6600 computer. It has the ability to provide traces of selected instructions or of all instructions, of changes affecting selected variables, of flow of the object program, of changes affecting selected registers, of selected loads, and of selected subroutine calls. Traps may be provided on selected addresses, selected locations stored into, selected addresses loaded from, and selected operation codes. Provision is made for dumping part of core at any time and anywhere within

the field length, and all of the user's registers at any time. The dump may be in octal, integer, floating point, or alphanumeric; excessive duplication is suppressed. Error checking facilities are provided for out-of-bounds jumps, memory references, arithmetic errors, etc., indefinite and infinite results, invalid stores and loads, infinite loops, incorrect values for selected variables. When an error condition is encountered, a trace of the preceding 600 instructions is automatically given. The WATCHR III system makes provision for the user to examine core or registers during the run, and for recovery if fatal errors occur.

The system will operate on any central processor binary code, simulates the action of the CPU, and collects information as directed by switch settings. At any time switches may be set or unset, selections may be made. Also at any time the object program may be placed under or removed from under WATCHR's control.

The following is a summary of WATCHR III features and options:[24]

a. Traces:

1. Instruction trace - instructions may be selected by means of their operation codes for tracing; any operation code or combination of operation codes may be selected for tracing.

2. Register trace - registers in any combination or order may be selected for tracing. Thereafter, whenever a selected register acquires a new numerical value, both the old and new values are printed out.

3. Program trace - the logical flow or sequence of instructions is made available through the use of the MAP option. Pairs of addresses will be printed out periodically. The occurrence of a branch is indicated with starting of a new pair of signals.

4. Memory reference (load) trace - analogous to the trace of changed locations except that it provides for a long comment giving the function served by the variable being loaded. The purpose of this feature is to enable the programmer to see how the program under observation actually operates.

5. Subroutine call trace - analogous to the register trace. Return jumps are traced and space is provided for comments denoting the purpose of each subroutine.

b. Traps:

The trap is used to initiate the trap routine. A trap routine is any instruction or set of instructions the programmer may wish to execute if trap conditions are satisfied. Traps always occur before execution of the instruction in question. Any number of traps may be set and they will be able to operate simultaneously.

1. Trap on operation code - the programmer may assign the starting address of a trap routine to any operation code or combination of operation codes.

2. Trap on address - when program control arrives at a specified address a trap routine is initiated.

3. Trap on storage into a specified location - analogous to a trap on address except that the addresses selected are locations into which stores may be made or are expected to be made.

4. Trap on load from a specified location - analogous to a trap on storage into a specified location except that the address chosen is to be trapped on when its contents are loaded into a register.

c. Reports:

1. Snapshot dump - a parameter list is used to dump current contents of selected locations.

2. Interpreted dump - this dump may be called on as often as desired and may be in octal, integer, floating point, BCD, or alphanumeric. Provision is also made for giving an alphanumeric label to each dump.

3. Reports - these are given automatically when a fatal error is encountered. Reports consist of the following:

   a. Up to 50 most recent branch instructions.

   b. Up to 50 most recent return jumps.

c. Up to 50 most recent stores.

d. Up to 600 most recent executed instructions.

A program running under the control of WATCHR III will take from 40 to 300 times longer. WATCHR III requires 16K of control memory. The number of test runs required in the preparation of a program should be about 5 times less with the judicious use of the package. With the employment of WATCHR III the overall usage of computer time should be less than or equal to the total computer time used without WATCHR III. The total program preparation time should be shortened by a factor of 5 to 10.

3-39. TIDY AND EDIT FOR FORTRAN

Two programs have been developed for the purpose of editing FORTRAN programs. The employment of such programs would greatly facilitate future changes in the program and the debugging of such changes. The correction of errors that appear after the program has been in operation for some time would be greatly simplified. The following is a summary of these two systems.

3-40. The TIDY Program

TIDY[25] processes the old FORTRAN program routine-by-routine, and prepares and punches a new version of the program. The new program has the following characteristics.

a. Statement numbers are left-justified and in ascending consecutive order.

b. Only those statements which are referenced by other statements are assigned statement numbers.

c. Statement number references are updated to conform to the new statement number assignments.

d. FORMAT statement are collected from within the body of each routine and placed at the end.

e. The only FORMAT and CONTINUE statements retained are those that are referenced within the routine.

f. Spaces are deleted or inserted as necessary to ensure uniformity and improve readability.

g. Comment cards are inspected for comments starting in the statement number field; if found, they are right-shifted so as to start in column seven.

h. Consecutive blank comment cards in excess of two are deleted.

i. All statements in each new routine are labeled in card columns 73 through 79 with a unique letter-number combination. The alphabetic character(s) indicate routine, while the number indicates the position of the statement in the routine.

j. Column 80 of each END statement is punched with a " - " sign (11 punch). This will permit automatic page ejection when listing TIDY-processed routines on machines that have the "x-skip" feature.

k. Certain FORTRAN II statements are rewritten as FORTRAN IV statements.

A number of options are available which enable one to modify some of the above characteristics. In addition to the editing function, TIDY offers a limited set of diagnostics. Errors and trouble areas such as missing or duplicate statement numbers, incorrect parenthesis counts, illegal DO-loop indexing, illegal statements, and inaccessible parts of the program are noted. Pseudo-statement numbers are generated when references are found to nonexisting statement numbers to enable the programmer to make corrections with a minimum of repunching. One option provides a list of corresponding old and new statement numbers.

3-41. The EDIT Program

EDIT[26] is very much like TIDY. Characteristics a, d, f, g, and j of TIDY are also performed by EDIT. The major emphasis of EDIT deals with the renaming of variables. The variable names used in writing FORTRAN programs often make the flow of the program hard to follow. Natural conditions which are largely responsible for the above condition include the following:[27]

a. Programmers may use variable names that are convenient to work with but are devoid of meaning.

b. Awkward expedient variable names often result from hasty changes made during the debugging phase.

c. Poor selection of variable names was made initially, but later, after the program is almost completed, the programmer arrives at a greatly improved symbolism that he would prefer to use.

It is often desirable to change the names of variables but this would entail an amount of work that is in excess of the value gained. However, EDIT provides a simple solution as detailed in source document.

3-42. AUTOMATIC FLOWCHARTING

Recently, various proprietary systems have been put on the market that aid in debugging the original source program and provide documentation, by generating from the source program a flowchart and diagnostics that give a picture of the program logic in its earliest stages, even before assembly if desired, thus facilitating debugging.

3-43. The AUTOFLOW System

One of these flowcharting systems is called AUTOFLOW[28], marketed by Applied Data Research, Inc. AUTOFLOW can generate flowcharts directly from source programs written in assembler language, FORTRAN, or COBOL. The input to the computer is the AUTOFLOW program and the user's source deck (or tape); the resulting computer printout is a flowchart using standard symbols (decision, processing, connectors, terminals, etc.), plus a listing of the cross-reference table, and a listing of diagnostics (if any). The program comments become a significant part of the flowchart.

This type of flowchart can make instantly visible to the programmer any errors in missing destinations in branching instructions, undefined external references, errors in address arithmetic, etc.

After the initial flowchart has assisted in correcting the coding, then the programmer has the option of improving the clarity of the flowchart for final documentation by repunching and rerunning the deck, adding special chart-oriented codes to the assembler language, FORTRAN, or COBOL cards. Through the use of these codes, the level of the flowchart detail may be made more meaningful.

3-44. TESTING

3-45. Purposes

Program testing has two basic purposes.[29]

a. To insure that the program has been coded correctly and that the coding matches the logical design.

b. To insure that the logical design matches the basic requirements of the task as it is defined in the job specification.

As each segment of the program is developed, rigorous test data should be prepared and that segment should be tested. After testing each segment separately, the entire sytem is ready for testing.

3-46. Problems

The fact that the program is operative and runs to a satisfactory completion does not insure that all of the exceptional conditions, and their permutations and combinations, have been tested. The consideration of exceptional and unusual conditions frequently accounts for a large percentage of the program's instructions. Unless one is careful to provide for these exception conditions in his test data, it is quite possible to reach the end of the program while checking out only a small proportion of the program. Because of the many combinations and permutations of conditions inherent in a given program, it is practically impossible to test all conditions which may actually occur. Thus it is not uncommon for a program that has been operating successfully for some time to fail one day due to an unusual set of conditions.

3-47. Logical Tree

The use of a logical tree can be a valuable aid in selecting the various combinations of input data for testing purposes. This technique also aids in locating program errors. The systematic formulation of test data will eliminate duplication of test situations and will significantly expand the number of different test conditions. Through the systematic selection of all realistic combinations of input data, the programmer may test all segments of a given program.

3-48. Maintaining Test Data

It may be desirable to maintain a test sequence that may be run periodically. One cannot assume that a feature which worked on one version will work on another. It may

also be desirable to provide a set of test data that will generate most system error diagnostics.[31]

3-49. Satellite Test Data

Data that are used in testing programs that are to process satellite telemetry data are derived from the following sources.[32]

a. Data generated manually by the programmer.

b. Data acquired from the satellite during ground testing.

c. Actual data tapes from previous satellites which have the same basic telemetry format.

The first two sources have a major limitation in that the test data produced do not reflect adequately the actual perturbations present in telemetry data. The third source partially overcomes the above limitations but it too is of limited utility. Often none of the data available from previous satellites have a suitable format or have been obtained via the same set of data links as the given satellite. Also information concerning the specific type and location of noise perturbations in a given data tape is not available to the programmer. Thus the program may not detect a particular perturbation in the data and the programmer will be unaware of this deficiency.

3-50. Data Simulation Computer Program

Because of these shortcomings and the necessity of high reliability, another source of data has been developed - Data Simulation Computer Program. The benefits derived from this program include the following:

a. Test data will be provided that realistically take into account the various specified noise conditions.

b. An increased degree of reliability other than achieved through the use of other sources.

c. Time and effort spent in generating test data is reduced.

d. Reduction in clerical and keypunch errors.

e. Reduce time of testing cycle.

f. Useful criteria for accepting contractor-produced programs.

The Data Simulation Program[33] was written for the UNIVAC 1107 computer. A high degree of flexibility was incorporated to provide for differences in telemetry systems and formats, varying degrees and types of noise conditions, variations in experiment-data characteristics, differences in formats, etc. The user provides a deck of punched cards containing all definitions and parameters for the simulation along with user subroutines for special computations. The primary output is a digital tape which contains the simulated

test data. A secondary output is a listing of input parameters, selected data channel and record printouts, errors inserted during simulation, and summary statistics for each simulated file.

3-51. PROGRAM CORRECTION

Basically there are two approaches to program correction - re-compilation and load-time patching. When a large program and a small error are involved, re-compilation is inefficient and is of greater expense than the correction warrants. Load-time patching does not provide a listing thus the documentation does not match the card deck. For the programmer who knows only the higher-level language there is no choice, he must re-compile. The option is available to the programmer who has the program listing and knows the assembly language.

A third possibility for making program corrections exists.[34] This approach employs a miniature assembly program, packaged as a closed subroutine, which is loaded with the object program that is to be corrected. The routine is activated by the programmer who specifies the input and output buffers and key words which indicate patching locations in the input stream. The key-word signals that a patch location follows; the input stream is diverted to a work area, where the symbolic language patch is assembled. When the key word for the end of the patch is encountered, control will be transferred to the patch or back to the object program as directed.

Several advantages are incurred through the use of this subroutine technique. It is faster than either of the other two techniques and it provides documentation. Another

advantage over the two conventional methods is that it is not confined to unconditional load-time changes. It may be called upon to modify the object program at any time during the run, and these changes may be made contingent on any condition that may be tested by the programmer. Several versions of the same procedure may be tested during one run, this is made possible by having the constant availability of an assembler. The reduced turn-around time can be an important consideration.

3-52. RESTARTING

An area closely related to testing is that of restarting. Often much time and expense may be saved by allowing for the restarting of a program at various points. By employing this technique one does not need to start at the beginning of the program each time an error condition occurs. The technique of intermittent restarting may be implemented by incorporating a sequence of checks, perhaps at the end of each segment. If any of these checks reveals an error at that point, operations should cease and the program restarted at the last point that was known to be correct after corrections are implemented. In the event that a program is to be restarted at an intermediate point, due to an error check stop, operator mishap or machine malfunction, it is necessary to save information about the status of the program at that point. Such information would include the contents of all memory storage areas, both internal and external. One approach to saving this information is to dump memory at every check point, always saving the last dump. Upon restarting, this information would be relocated.[35]

SECTION IV
ON-LINE-DEBUGGING

4-1. ON-LINE DEBUGGING TECHNIQUES

Although the main emphasis of this paper is intended to be on conventional or batch debugging, some mention of on-line debugging is in order. The current consensus among computer professionals is that on-line applications represent the wave of the future.[36] On-line computer activity probably represents about one percent of the total present computer activity; however, in five years it may represent fifty percent and in ten years it may represent nearly all computer activity.[37]

On-line debugging is conducted by a programmer who is in direct communication with the computer. The typewriter or teletype is most commonly used. The programmer makes changes, tests, then again makes changes in a continuous process with quick response from the computer until satisfactory results are achieved. On small computers or in the early days of computing when the computer was completely dedicated to the programmer and his program, the on-line ("hands-on") mode of debugging was a standard practice; but now it is not used as much, where programs are run "remote" on large computers. However, the arrival of large-scale time-sharing systems has made this mode of operation feasible on large computers.

Much of the work done in the area of on-line debugging has been described only in unpublished reports or passed on orally. However, an excellent paper by Thomas G. Evans and D. Lucille Darley presents a survey of on-line debugging techniques.[38] Included in their paper are some possible future developments as well as a list of references.

4-2. CRITERIA FOR AN ON-LINE DEBUGGING SYSTEM

The following principles may be considered as good criteria for an on-line debugging system.[39]

a. Flexible control over the program must be provided to the programmer. The programmer must be able to specify this control in terms of natural units, small and large, of the language in question and be able to carry this control down to the finest level of detail.

b. Using the notation of the language of the program the programmer must be able to examine and "incrementally" modify both data and program at any time.

c. The debugging control language should be designed so that a minimum of typing is required and information provided the programmer is compatible, concise, and aids rapid comprehension.

d. Provision should be made for the automatic updating of the user's symbolic file to reflect the in-core representation of the program.

4-3. THE PROGRAMMER-CONTROLLED BREAKPOINT

Perhaps the central notion of on-line debugging is the programmer-controlled breakpoint.[40] This concept allows the programmer to specify, generally in symbolic forms, a point or points in the program at which he wishes to interrupt the flow and return to the debugging routine. Upon entering the debugging routine, the state of the active

registers is stored in order to permit subsequent continuation from the breakpoint. The programmer examines his program at this breakpoint and may make any desired changes before continuing.

4-4. ASSEMBLER-LANGUAGE PROGRAM DEBUGGING: THE DDT SYSTEM

When considering on-line debugging facilities a separation may be made between those that deal with the assembler language and those that deal with higher level languages.

DDT[41] (Digital Debugging Tape) is one of the better known on-line debugging programs that deals with an assembler-language program. One of the most important characteristics of this program is the care devoted to the design of the typing conventions. Single-letter commands and a structure in which frequently desired states can be easily reached from the present state minimize typing and aid rapid interaction. Convenient ways of typing contents of a given register in various formats such as symbolic, decimal, or octal are also provided. A number of extensions have been made on the DDT system. The following is a summary of some of these extensions.[42]

a. Because DDT can accept instructions in symbolic assembler-language form, it already nearly serves as an "on-line assembler" capable of processing the on-line writing and testing of small programs. In conventional DDT, the introduction, in a line of coded input, of a symbol not previously defined by the programmer results in an error. Edwards and Minsky have added to the conventional DDT an "undefined symbol"

feature. This feature results in a special symbol table entry. These entries are linked together and when the symbol is ultimately defined by the programmer its previous occurrences are filled in appropriately.

b. DDT provides an unlimited freedom to patch a program. This is accomplished by inserting the desired coding in some available space, then planting a transfer to this insertion whenever desired. Sometimes extensively patched programs result. The process of editing or cleaning up such a program is long and susceptible to errors. At least two attempts have been made to facilitate this editing effort.

1. One version of such an editing facility was developed by Deutsch and Lamson.[43] In response to a request by the programmer to insert a specified piece of symbolic coding, the debugging program performs the following:

   (a) Edits the changes into the symbolic program stored on the drum.

   (b) Assembles the addition into a "patch area" of core and automatically links the resulting code to the main program by copying instructions and inserting transfers.

Thus, the patched binary program in core is equivalent to the edited symbolic version stored on the drum. At the termination of the debugging session the updated symbolic program is stored among the programmer's files.

2. Another effort to solve the same problem was developed for the M-460 computer.[44] As in the previous approach, the on-line programmer presents insertions, deletions, or a mixture of both, written in a symbolic assembler language, to the debugging program. The debugging program performs the following:

   (a) Symbolic changes are stored along with the original symbolic program. At the end of the debugging session, both are edited and the programmer is provided with an updated symbolic file.

   (b) Instead of the patch being made to correspond with the programmer's changes, the part of the program affected by the change is relocated appropriately in core. This relocation process is possible only because the relocation information resulting from the assembly of the program is collected into a list structure which is used by the debugging program whenever a change is called for and it is then updated accordingly. The list

structure is also used by the relocation loader. The symbol table passed by the assembler to the debugging program must also be updated each time.

3. Although this approach may be time-consuming, it has several advantages over the "automatic patching" approach:

   (a) The patched binary and the edited symbolic program may behave differently in situations where the location of words in core relative to each other is important.

   (b) Core may be left in a rather confusing state which may require relatively frequently reassembly for readability.

c. In addition to the flexibility in the placement and moving of breakpoints provided for in DDT, a facility which permits the programmer to make the breakpoints conditional has been added to some systems. Tests are supplied on-line by the programmer and are executed when the breakpoint condition is reached to determine if control is to be turned over to the programmer. Several systems have implemented the use of "canned tests." Older versions and a few of the newer versions of DDT provide an option whereby the programmer can specify a certain number of times a point must occur before the break can be executed.

d. The provision for some form of instruction-by-instruction tracing may be desirable. Generally such a feature has been omitted in favor of breakpoints. Tracing facilities may share much of the breakpoint machinery. The programmer should be able to specify a location in his program and ask either for control or for a printing or both whenever a specified point is reached and associated conditions are satisfied.[45]

e. Another desirable but not widely found feature of current on-line debugging systems is extensibility. Extensibility provides the capability in terms of available primitives.[46]

f. An option often found in the DDT system allows the programmer to conduct a search between specified limits in core for all words matching a given word in the bits specified by a given mask.[47]

4-5. HIGHER-LEVEL LANGUAGE DEBUGGING

4-6. The LISP System

The concepts or techniques of on-line assembler language debugging are the same as those employed in the on-line debugging systems of higher-level languages, the systems for higher-level languages are generally better developed and more widely used.

One well-known on-line debugging system developed for higher-level languages is the LISP[48] system. The following is a summary of some of the features of various LISP[49] systems.

a. Extensive tracing facilities were originally made available to the on-line programmer. This feature was later extended and made conditional in both the MAC and M-460 LISP systems.

b. An editing program, not a conventional text editor but a program permitting the user to modify the list in which LISP functions are stored for interpretation, is provided for on some LISP systems. The editing feature has greatly facilitated the ease in making program changes.

c. Conditional breakpoints which may be inserted at any point in a LISP function definition were provided for in some LISP systems. Conditional breakpoints and tracking make it possible to use the full capacity of the LISP language for on-line composition of the conditions. This permits relatively simple coding of an elaborate logical condition for which the counterpart in assembly language might be quite complex. Greater selectivity may be achieved by suppressing irrelevant tracking and breakpoints through the "canning" of a few special predicates used in writing conditions. It is possible to find an error condition at a breakpoint while running a test case, call the editor to make a correction, run the program on a simpler test case to verify

the correctness of the change, then continue with the execution of the original test case from the breakpoint.

d. The MAC and M-460 LISP systems contain both an interpreter for LISP functions stored as list structures and a compiler for LISP functions in symbolic code. These interpreted and compiled functions may be intermixed. The interpreter makes implementation of debugging facilities relatively simple. The insertion of breakpoints at arbitrary locations is easily implemented by modifying the list structure corresponding to the program.

4-7. The QUIKTRAN System

QUIKTRAN[50] is a debugging system based on the interpretation of FORTRAN statements. Modifications of the FORTRAN program are made freely by inserting and deleting statements. The capability of examining and modifying variables is provided. A number of modes of tracking are available. Extensive run-time diagnostics are made possible by the interpretive mode. The system employs a form of non-conditional breakpoint capability. The non-conditional breakpoint means that a statement can be inserted at any point in the program which, when reached, has the effect of transferring control to the programmer.

4-8. The Incremental Compiler

The concept of an "incremental compiler" is presented by K. Lock at the California Institute of Technology.[51]

The basic idea is to compile each program statement separately and place the resulting code, together with a copy of the symbolic form of the statement, certain pointers, and other information, depending on the type of statement, in a contiguous block of core. These blocks are linked together in lists. With the implementation of this concept only those portions of a program to be changed need to be recompiled. Insertions and deletions at the statement level proceed with modifications of the list. The breakpoint capability occurs at statement level since control is returned to the monitor between each statement.

REFERENCES

1. Computer Program Documentation and Debugging (Washington: CEIR, Inc., 1968) p. 5.

2. Gordan Davis, An Introduction to Electronic Computers (New York: McGraw-Hill Book Co., 1965) p. 199.

3. Herbert D. Leeds and Gerald M. Weinberg, Computer Programming Fundamentals, 2nd ed. (New York: McGraw-Hill Book Co., 1966) pp. 362, 363.

4. CEIR, Inc., op. cit., p. 45.

5. Philip M. Sherman, Programming and Coding Digital Computers, (New York: John Wiley and Sons, Inc., 1963) pp. 404, 405.

6. Ibid., pp. 413, 414.

7. J. R. Dingeldine and D. E. Bear, Reference Manual for the SDC-SHARE Operating System Volume 9 - Debugging (Santa Monica: System Development Corporation, 1964) pp. 3, 4, 12, 19.

8. CEIR, Inc., op. cit., p. 41.

9. Rudolf Bayer, et al, The ALCOR Illinois 7090/7094 Post Mortem Dump (Illinois: Boeing Scientific Research Laboratories, 1967), p. 1.

10. Ibid., pp. 2, 3.

11. UNIVAC 1108 Executive Programmer's Reference (Sperry Rand Corporation, 1966), Section 17, pp. 1-15.

12. G. S. Stoller, Masked Search Program, (Bedford: Mitre Corporation, 1965), p. 2.

13. Eugene I. Grunby, An Improved Approach to Trace Routines, (Greenbelt: Goddard Space Flight Center, 1965), pp. 1-4.

14. Ibid., pp. 3, 4.

15. A. Hisler, "SEFLO-Sequence Flow, a Program Debugging Tool," (Greenbelt: Goddard Space Flight Center, 1968), pp. 1, 2.

16. Daniel D. McCracken, Harold Weiss and Lee Tsia-Hwa, Programming Business Computers, (New York: John Wiley and Sons, Inc., 1959), p. 236.

17. Ibid., p. 237.

18. Leeds, op. cit., p. 387.

19. J. A. Thompson, Diagnose, A Routine to Debug FORTRAN Programs, (Oak Ridge: Union Carbide Corporation, 1965).

20. Earl H. Ferguson and Elizabeth Berner, "Debugging Systems of Source Language Level," Communications of ACM 6, 8 (August 1963), pp. 430-434.

21. IBM System/360 Operating System TESTRAN (IBM Corporation 1967), pp. 8-13.

22. Mark Halpern, "Computer Programming: The Debugging Epoch Opens," Computers and Automation 14, 11 (November 1965), pp. 28, 29.

23. E. Draughon, WATCHR III, A Program Analyzing and Debugging System for the CDC 6600 User's Manual (New York: New York University, 1966).

24. Ibid.

25. Harry M. Murphy, Jr., TIDY, A Computer Code for Renumbering and Editing FORTRAN Source Programs, (Albuquerque: Air Force Weapons Laboratory, 1966).

26. Forest McMains, EDIT, A FORTRAN Program for Renaming Variables in a Source Program , (Dover, New Jersey, Picatinny Arsenal, 1967).

27. Ibid.

28. AUTOFLOW Computer Documentation System (Applied Data Research, Inc., 1967).

29. Dick H. Brandon and Frederick Kirch, "Standards for Computer Programming," Computers and Automation (May 1964), p. 22.

30. Joan C. Miller and Clifford J. Maloney, "Systematic Mistake Analysis of Digital Computer Programs," Communications of the ACM 6, 2 (February 1963), pp. 58-62.

31. CEIR, Inc., op. cit., p. 42.

32. Bernard G. Narrow and Richard C. Lee, A Generalized Satellite Telemetry Data Simulation Program, (Greenbelt: Goddard Space Flight Center, 1966), p. 2.

33. Ibid., pp. 1-4.

34. Halpern, op. cit., p. 30.

35. Sherman, op. cit., p. 407.

36. CEIR, Inc., op. cit., p. 48.

37. Ibid.

38. Thomas, G. Evans and Lucille D. Darley, On-Line Debugging Techniques: A Survey (Bedford: Air Force Cambridge Research Laboratories, 1966).

39. Ibid., p. 48.

40. Ibid., p. 39.

41. Ibid., p. 39.

42. Ibid., p. 39.

43. L. P. Deutsch and B. W. Lamson, "DDT Time Sharing Debugging System Reference Manual," (California: University of California, 1965).

44. Evans, op. cit.

45. Ibid.

46. Ibid.

47. Ibid.

48. Daniel G. Bobrow, et al, The BBN LISP System, (Bedford: United States Air Force, 1967).

49. Evans, op. cit.

50. T. M. Dunn and J. H. Morrissey, "Remote Computing - An Experimental System," Proceedings SJCC, 1964.

51. K. Lock, "Structuring Programs for Multi-Program Time-Sharing On-Line Applications," Proceedings of FJCC, 1965.

BIBLIOGRAPHY

Books

Davis, Gordon B., An Introduction to Electronic Computers, New York: McGraw-Hill Book Company, 1965.

Leeds, Herbert D., and Weinberg, Gerald M., Computer Programming Fundamentals, 2nd Ed. New York: McGraw-Hill Book Company, 1966.

McCracken, Daniel D., Weiss, Harold, Tsia-Hwa, Lee, Programming Business Computers, New York: John Wiley and Sons, Inc., 1959.

Sherman, Phillip M., Programming and Coding Digital Computers, New York: John Wiley and Sons, Inc., 1963.

Manuals

Applied Data Research, Inc., AUTOFLOW Computer Documentation System, October 1967.

Deutsch, P. L. and Lamson, B. W., DDT Time Sharing Debugging System Reference Manual, Document #30.40.10 (Rev.), University of California, May 1965.

Dingeldine, J. R., and Bear, D. E., Reference Manual for the SDC-SHARE Operating System Volume 9 - Debugging System. Report by System Development Corporation, Santa Monica, California, December 1964.

Draughon, E., WATCHR III, A Program Analyzing and Debugging System for the DCD 6600 User's Manual. Report by AEC Computing and Applied Mathematics Center Courant Institute of Mathematical Sciences, New York University, July 1966.

IBM Corporation. IBM System/360 Operating System TESTRAN. Systems Reference Library. Form C28-6648-0 File Number S360-37. February 1967.

Sperry Rand Corporation. UNIVAC 1108 Executive Programmer's Reference Manual, 1966.

Reports

Bayer, Rudolf, et al. The ALCOR Illinois 7090/7094 Post Mortem Dump (AD 660014). Information Sciences Report No. 3, Information Sciences Laboratory, Boeing Scientific Research Laboratories, August 1967.

Bobrow, Daniel G., et al., The BBN LISP System. Report prepared for United States Air Force, Bedford, Massachusetts. July 1967.

Dunn, T. M., and Morrissey, J. H., "Remote Computing - An Experimental System," Proceedings SJCC, 1964.

Evans, Thomas G., and Darley, D., Debug, an Extension to Current On-Line Debugging Techniques, Report by Air Force Cambridge Research Labs., Bedford, Mass., November 1964.

Evans, Thomas G., and Darley, D. Lucille, On-Line Debugging Techniques: A Survey. (AD650016). Report by Air Force Cambridge Research Laboratories, Office of Aerospace Research, L. G. Hanscom Field, Bedford, Massachusetts, January 1967.

Grunby, Eugene I. An Improved Approach to Trace Routines, (N65-29802, X-545-65-15) Report by Goddard Space Flight Center, Greenbelt, Matyland. February 1965.

McMains, Forest. EDIT, A FORTRAN Program for Renaming Variables in a Source Program. A report by Data Processing Systems Office, Picatinny Arsenal, Dover, New Jersey, July 1967.

Murphy, Harry, M., Jr. TIDY, A Computer Code for Renumbering and Editing FORTRAN Source Programs. A report by Air Force Weapons Laboratory, Kirtland Air Force Base, New Mexico. October 1966.

Narrow, Bernard G. and Lee, Richard C., A Generalized Satellite Telemetry Data Simulation Program. Report by Goddard Space Flight Center, Greenbelt, Maryland. November 1966.

Stoller, G. S., Masked Search Program (AD 610062) A Technical Documentary Report No. ESD-TDR-64-629) Prepared by Mitre Corporation, Bedford, Massachusetts. January 1965.

Thompson, J. A., DIAGNOSE, A Routine to Debug FORTRAN Programs (N66-23235). Report for Oak Ridge National Laboratory operated by Union Carbide Corporation for the U.S. Atomic Energy Commission, June 1965.

Articles and Periodicals

Brandon, Dick H. and Kirch, Frederick "Standards for Computer Programming" Computers and Automation, May 1964, pp. 22-28, 43.

Ferguson, H. Earl, and Berner, Elizabeth, "Debugging Systems at a Source Language Level," Communications of ACM 6, 8 (August 1963), 43-434.

Halpern, Mark, "Computer Programming: The Debugging Epoch Opens," Computers and Automation 14, 11 (November 1965), 28-31.

Lock, K., "Structuring Programs for Multi-Program Time-Sharing On-Line Applications," Proceedings of FJCC, 1965.

Miller, Joan C., and Maloney, Clifford J., "Systematic Mistake Analysis of Digital Computer Programs," Communications of the ACM, 6, 2 (February 1968), 58-62.

Unpublished Material

CEIR, Inc., Institute for Advanced Technology, "Computer Program Documentation and Debugging," Documentation and Debugging Seminar, Washington, 1968.

Hisler, Abrom. "SEFLO-SEQUENCE FLOW, A Computer Program Debugging Tool," Preliminary report by Goddard Space Flight Center, Greenbelt, Maryland, January 1968.

APPENDIX

SELECTED DEBUGGING REFERENCES

The following is a selected list of sources that deal in some way with debugging. Abstracts of many appear in the Technical Abstract Bulletin Index published by the Defense Documentation Center (DDC), the Research and Development Reports abstract journal published by the U. S. Department of Commerce, the Scientific and Technical Aerospace Reports (STAR) index published by the National Aeronautics and Space Administration, and the Computing Reviews published by the Association for Computing Machinery. Key words and phrases have been noted for some of the sources. "AD" numbers refer to DDC documents and "S" numbers refer to STAR documents.

APPENDIX

Books

Davis, Gordon B., An Introduction to Electronic Computers, New York, McGraw-Hill Book Company, 1965.

Leeds, Herbert D., and Weinberg, Gerald M., Computer Programming Fundamentals, 2nd Ed., New York, McGraw-Hill Book Company, 1966.

Pages 358-394

Program testing, preliminary testing, assembly output, test cases, philosophy of segmentation, aids to testing, SHARE (DB) IBM 7090, dumping, conditional debugging macros, tracing, program testing, program testing in FORTRAN, FORTRAN debugging system

Library of Congress 65-21588

McCracken, Daniel D., Weiss, Harold, Tsia-Hwa, Lee, Programming Business Computers, New York, John Wiley and Sons, Inc. 1959.

Randell, B., and Russell, L. J., ALGOL 60 Implementation, A.P.I.C. Studies in Data Processing, No. 5, Academic Press, 1964, xir +418.

Compiler, with error-checking and debugging facilities

Source: ACM Computing Review

Scott, Theordore G., Computer Programming Techniques, Garden City, New York, Doubleday and Company, Inc., 1964.

Sherman, Philip M., Programming and Coding Digital Computers, New York, John Wiley and Sons, Inc., 1966.

Stark, Peter A., Digital Computer Programming, New York, Macmillan Company, 1967.

Program debugging

Manuals

Appel, Klaus, Reference Manual for Easy, An Automatic Programming System for the ALWAC Computer, Report by Uppsala University, Sweden, October 1963.

ALWAC debugging systems

Deutsch, P. L. and Lamson, B. W., DDT Time Sharing Debugging System Reference Manual, Document #30.40.10 (Rev.), University of California, May 1965.

Dingeldine, J. R., Reference Manual for the SDC - SHARE Operating System, Volume 9, Debugging System, Report by System Development Corporation, Santa Monica, California, December 1964.

SHARE, debugging, SOS macros

AD-456058

Draughon, E., WATCHR III - A Program Analyzing and Debugging System for the CDC 6600 - User's Manual, Report by Courant Institute of Mathematical Sciences, New York University, New York, July 1966.

CDC 6600, debugging system

N68-81549

Griffith, E. L., SCF Computer Program Systems Manual Trace Program, Report by System Development Corporation, Santa Monica, California, November 1963.

Trace, debugging tool

AD-428179

IBM. IBM 7090/7094 IBSYS Operating System Version 13 IBJOB Processor Debugging Package. Systems Reference Library, (January 1965).

File No. 7090-27

Form C28-63930

IBM. IBM System/360 Operating System Programmer's Guide to Debugging. File No. S360-20, Form C28-6670-0. 1967.

IBM Corporation. IBM System/360 Operating System TESTRAN. Systems Reference Library. Form C28-6648-0 File Number S360-37. February 1967.

Sperry Rand Corporation. UNIVAC 1108 Executive Programmer's Reference Manual, 1966.

Reports

Ammerman, Anne B., Diesen, Larry R., and Thombs, Herman W., Displaytron - A Graphical Display Oriented Conversational FORTRAN Facility for and IBM 360/40 Computer, Report by Naval Weapons Laboratory, Dahlgren, Virginia, July 1967.

Displaytron, on-line, FORTRAN IV, 360/40

AD-656583

Armour Research Foundation of Illinois Institute of Technology, Advanced Studies of Computer Programming, Report prepared for U. S. Army Signal Research and Development Agency, January 1961.

MOBIDIC program debugging systems

AD-251951

Arnold, L. J., 160-A Utility Program Descriptions Milestone 11 160-A Core Dump onto Printer, Report by System Development Corporation, Santa Monica, California, December 1963.

Core dump

AD-427900

Barbier, John, and Morrissey, Computer Compiler Organization Studies, Report by Morrissey (John) Associates, Inc., New York, May 1966.

N67-40182

AD-658196

Bayer, Rudolf, et al., The ALCOR Illinois 7090/7094 Post Mortem Dump (AD 660014). Information Sciences Report No. 3, Information Sciences Laboratory, Boeing Scientific Research Laboratories, August 1967.

Bell Aerosystems Company, Aero-Space Environment Simulation System (ASESS), Volume III: Program Modification and Implementation on the Digital Computer, Report prepared for A.F.S.S., Electronic Systems Division, Buffalo, New York, November 1964.

FORTRAN IV, snapshot, program checkout

N66-20024

AD-610715

Best, G. C., United Kingdom Atomic Energy Authority, Harwell, England; Electronics and Applied Physics Division, AUTOTOGGLE - An Aid to PDP8 Program Debugging, July 1966.

Debugging program for PDP8 computer

N66-39821

Source: NASA/STAR

Bobrow, D. G., Darley, D. L., Deutsch, L. P., Murphy, D. L., and Teitelman, W., The BBN 940 LISP System Interim Scientific Report, Report by Bolt, Beranek, and Newman, Inc., Cambridge, Massachusetts, July 1967.

BBN 940 LISP System, tracking, conditional breakpoints, debugging

AD-656771

N67-36746

Brown, W. S., "An Operating Environment for Dynamic-Recursive Computer Programming Systems," of VIII(6), June 1965, pp. 371-77.

Clark, George A., Jr.,"Technical Problems of Simulation Development," Report by Defense Supply Agency, Alexandria, Virginia, 1967.

AD-813899

Dunn, T. M., and Morrissey, J. H., "Remote Computing - An Experimental System," Proceedings SJCC, 1964.

Erickson, W. J., Pilot Study of Interactive Versus Non-interactive Debugging, Report by System Development Corporation, Santa Monica, California, December 1966.

Determining economics of different types of computer systems

Evans, Thomas G., and Darley, D., Debug, an Extension to Current On-Line Debugging Techniques, Report by Air Force Cambridge Research Laboratories, Bedford, Massachusetts, November 1964.

Debug computer program, UNIVAC M-460

AD-168825

Evans, Thomas G., and Darley, D. Lucille, On-Line Debugging Techniques: A Survey, Report by Air Force Research Laboratories, L. G. Hanscom Field, Massachusetts, 1966.

Gildea, Robert A. J., Evaluation of ADAM: An Advanced Data Management System, Report by Mitre Corporation, Bedford, Massachusetts, August 1967.

Debugging facilities, suggestions

N68-12069

AD-661273

Grant, E. E., An Empirical Comparison of On-Line and Off-Line Debugging, A Report by System Development Corporation, Santa Monica, California, May 1966.

On-line and off-line debugging

AD-633907

Griffith, E. L., Utility Program Descriptions Milestone 11 Memory Dump Routine, Report by System Development Corporation, Santa Monica, California, May 1964.

Dump

AD-446860

Grunby, Eugene I., An Improved Approach to Trace Routines, Report for National Aeronautics and Space Administration, Goddard Space Flight Center, Greenbelt, Maryland, February 1965.

Trace routines, proposes solutions to excess of printed diagnostic material, considerably extended execution time, also provisions for programmers to provide own coded subroutines, UNIVAC 1107

N65-29802

Source: NASA/STAR

Hisler, Abrom, Propellant Utilization Time Trace (PUTT) A Documentation Case Study, Report by Telemetry Computation Branch, Goddard Space Flight Center, Greenbelt, Maryland, September 1967.

Debugging techniques used on PUTT

I-560-67-399

IBM Corporation, Computer Programming Techniques for Intelligence Analyst Application, Report No. 1, Report by Thomas J. Watson Research Center, Yorktown Heights, New York for RADC Griffiss AFB, New York, August 1964.

Automated debugging techniques

AD-605267

N64-29932

Informatics, Inc., Display Oriented Computer Usage System, Interim Technical Report October 64 - April 66, Bethesda, Maryland, June 1966.

FORTRAN, on-line, DOCUS

AD-487385

Jacoby, K., and Layton, H., "Automation of Program Debugging," Preprints of papers presented at the 16th National Meeting of the ACM, Los Angeles, September 5-8, 1961; ACM, New York.

Source: ACM Computing Review

Kramfus, I. R., and Yakushin, Debugging Program for Training Computers.

N67-27843

Lampson, Butler W., "Interactive Machine Language Programming," Proc. AFIPS Fall Joint Computer Conference, Part 1, pp. 473-481.

SDS 930, macro-assembler and debugging facility

Source: ACM Computing Review

Magnuson, Robert A., Extended Use of Macro Assemblers, Report by Research Analysis Corporation, McLean, Virginia, July 1965.

Debugging, techniques

AD-470105

McMains, Forrest, Edit, A FORTRAN Program for Renaming Variables in a Source Program, Technical Report by Picatinny Arsenal, Dover, New Jersey, July 1967.

Edit, FORTRAN IV

AD 659-340

Mitre Corporation, First Congress on the Information System Sciences, Session 12, Programming Information Processing Automata, Bedford, Massachusetts, October 1963.

Debugging

AD-422475

Murphy, Harry M., TIDY, A Computer Code for Renumbering and Editing FORTRAN Source Programs, Report by AFSC, Air Force Weapons Laboratory, Kirtland Air Force Base, New Mexico, June 1966.

TIDY, FORTRAN, editing, renumbering

N67-20677

AD-642099

Narrow, Bernard G., and Lee, Richard C., A Generalized Satellite Telemetry Data Simulation Program, Report by NASA Goddard Space Flight Center, Greenbelt, Maryland, November 1966.

Data simulation program which generates test tapes used for debugging and testing

Paul, Manfred, and Wiehle, Hans Ruediger, Bayer, Rudolf, Gries, David, The ALCOR Illinois 7090/94 Post Mortem Dump, Report by Boeing Scientific Research Laboratories, Seattle, Washington,

Post mortem dump technique

7090 Algol-60

Stoller, G. S., Masked Search Program, Report by Mitre Corporation, Bedford, Massachusetts; January 1965.

Masked search program, static trace, debugging or modifying 7090 program

N65-35592

AD-610062

Sutherland, W. R., On-Line Graphical Specification of Computer Procedures, Report by Massachusetts Institute of Technology, Lexington, Massachusetts, May 1966.

On-Line, Debugging

AD-639734

Sweeney, M. J., 160-A Diagnostic Program (SFCHEX) Milestone 5, Report by System Development Corporation, Santa Monica, California, July 1965.

CD6 160A, dump, selective dump, diagnostic program

AD-469872

Thompson, J. A., DIAGNOSE, A Routine to Debug FORTRAN Programs, Report for Oakridge National Laboratory, Tennessee, June 1965.

DIAGNOSE, CDC 1604, debugging, erroneous subscripts and DO-loop parameters, use of variables that have no values assigned to them

N66-23235

Source: NASA/STAR

Yourdon, Edward, "A Debugging Environment for Real-Time Systems," Real-Time Systems Design. Information and Systems Institute, Cambridge, Massachusetts, 1967, pp. 137-151.

Dynamic Debugging, Real-Time

Source ACM Computing Reviews

Articles and Periodicals

Apple, C. T., "The Program Monitor - a Device for Program Performance Measurement," Proc. ACM 205h National Conference, pp. 66-75.

IBM, evaluating software performance

Source: ACM Computing Review

Barron, D. W., and Hartley, D. F., "Techniques for Program Error Diagnosis on EDSAC 2," Computer Journal, 6, 1 (April 1963), 44-49.

Post-mortem, trace, irrevocable stops on mistake conditions

Source: ACM Computing Review

Boehm, E. M., and Steel, T. B., Jr., "The SHARE 709 System: Machine Implementation of Symbolic Programming," Journal of ACM, 6, 2 (April 1959), pp. 134-140.

Brandon, Dick H. and Kirch, Frederick "Standards for Computer Programming," Computers and Automation, May 1964, pp. 22-28, 43.

Chapin, Ned, "Logical Design to Improve Software Debugging - a Proposal," Computer Automation, 15, 2 (February 1966), pp. 22-24.

Hardware features to assist programmer, console trace, small memory snapshot, elaboration of run and dump

Source: ACM Computing Review

Constantine, Larry L., "Design and Reduction of Bugs," Concepts in Program Design, pp. 115-126.

Rules of program design to help reduce bugs

Source: ACM Computing Review

Evans, Thomas G., and Darley, D. Lucille, "Debug - an Extension of current on-line Debugging Techniques, Communications of ACM, 8, 5 (May 1965), pp. 321-326.

Debug, checkout of assembly language programs, UNIVAC M-460, RAP, TIC, STUD prelocating assembler, typewriter program for inspection of memory and control of program execution, text-editing

Source: ACM Computing Review

Ferguson, H. Earl, and Berner, Elizabeth, "Debugging Systems at a Source Language Level," Communications of ACM, 6, 8 (August 1963), pp. 430-434.

FORTRAN preprocessor called BUGTRAN traces, dumps, output format which includes symbolic variable names and statement numbers

Source: ACM Computing Review

Greenwald, I. D., and Kane, Maureen, "The SHARE 709 System: Programming and Modification," Journal of ACM, 6, 2 (April 1959), pp. 218-133.

Halpern, Mark, "Computer Programming: the Debugging Epoch Opens," Computer Automation, 14, 11 (November 1965), pp. 28-31.

References, debugging, arresting, identifying, correcting, examples

Source: ACM Computing Review

King, Claude F., "Computer Programming for Control of Space Vehicles," Peaceful Uses of Automation in Outer Space, pp. 409-414.

Source: ACM Computing Review

Lietzke, Majorie P., "A Method of Syntax - Checking ALGOL 60," Communications of ACM, 7, 8, (August 1964), 475, 478.

SHARE ALGOL 60 TRANSLATOR, Syntax checker used as diagnostic, compiler, recursive subroutines, error recovery, error correction

Source: ACM Computing Review

Lunelli, M., and Macchi, V., "Technichee tempi per la messa a punto dei programmi (techniques and times for program debugging)," Atti del convegno sui linguaggi simbolici di programmazione, AICA - January 1962, 90-95, (Italian).

Debugging, time requirements

Source: ACM Computing Review

McCarthy, J., Boilen, S., Fredkin, E., and Lickliaer, J.C.R., "A Time-Sharing Debugging System for a Small Computer," Proc. AFIPS 1963 Spring Joint Computer Conference, Detroit, Michigan, May 1963.

Typewriter-console command, time-shared, TX-0 computer at MIT, interrupt and examine any location in memory

Source: ACM Computing Review

Miller, Joan C., and Maloney, Clifford J., "Systematic Mistake Analysis of Digital Computer Programs," Communications of the ACM, 6, 2 (February 1963), 58-62.

Perlis, A.J., "Construction of Programming Systems Using Remote Editing Facilities," Proc. of IFIP Congress 65, Vol. 1, 229.

Description of Debugging aids, experimental programming languages

Source: ACM Computing Review

Schwartz, Jules I., Coffman, Edward G., and Weissman, Clark, "Potentials of a Large-Scale Time-Sharing System," Proceedings of 2nd Congress Info. Sys. Sci., 15-32.

Time-sharing, large-scale

Source: ACM Computing Review

Senko, M.E., "A Control System for Logical Block Diagnosis with Data Loading," Communications of the ACM, 3, 4 (1960), 236-240.

Smith, Lyle B., "A Comparison of Batch Processing and Instant Turnaround," Communications of ACM, 10, 8 (August 1967), 495-500.

Instant turnaround, Syntax correction, provides a more nearly correct program when logic debugging begins

Source: ACM Computing Review

ver Steeg, R. L., "TALK - a High-Level Source Language Debugging Technique with Real-Time Data Extraction," Communications of ACM, 7, 7 (July 1964), 418-419.

Technique for extracting the values of specified data during the execution of a compiled program, extracted data is processed by a separate editing program producing an annotated listing

Source: ACM Computing Review

Weinberg, G. M., "An Experiment in Automatic Verification of Programs," Communications of ACM, 6, 10 (October 1963), 610-613.

Common mistakes in writing and keypunching, FORTRAN symbolic debugging system, multiple access computer system

Source: ACM Computing Review

Wengert, R. E., "A Simple Automatic Derivative Evaluation Program," Communications of ACM, 7, 8 (August 1964), 463-464.

Debugging tool for programs which contain derivatives

Source: ACM Computing Review
Related to paper by R. D. Wilkens

Wilkens, R. D., "Investigation of a New Analytical Method for Numerical Derivative Evaluation," Communications of ACM, 7, 8 (August 1964), 465-471.

Numerical derivative evaluation uses as a debugging too, detecting and pinpointing overflow

Source: ACM Computing Review
Related to paper by Wengert, R. E.

Wilkerson, M., "The JOVIAC Checker, an Automatic Checkout System for Higher Level Language Programs," Proc. Western Joint Computer Conference, Los Angeles, California, (May 9-11, 1961), 397-404.

Recording test results

Zimmerman, Luther L., "On-Line Program Debugging - A Graphic Approach," Computers and Automation, 16, 11 (November 1967), 30-34.

GBUG an on-line assembly language debugging program, graphic terminals

Source: ACM Computing Review

Unpublished Material

C-E-I-R Inc., Institute for Advanced Technology, "Computer Program Documentation and Debugging," Documentation and Debugging Seminar, Washington, 1968.

Hisler, Abrom, "SEFLO - Sequence Flow A Computer Program Debugging Tool," Goddard Space Flight Center, Greenbelt, Maryland, January 1968.