A Set of Debugging and Monitoring Facilities to Improve the Diagnostic Capabilities of a Compiler
The National Bureau of Standards1 was established by an act of Congress March 3, 1901. The Bureau's overall goal is to strengthen and advance the Nation's science and technology and facilitate their effective application for public benefit. To this end, the Bureau conducts research and provides: (1) a basis for the Nation's physical measurement system, (2) scientific and technological services for industry and government, (3) a technical basis for equity in trade, and (4) technical services to promote public safety. The Bureau consists of the Institute for Basic Standards, the Institute for Materials Research, the Institute for Applied Technology, the Center for Computer Sciences and Technology, and the Office for Information Programs.

THE INSTITUTE FOR BASIC STANDARDS provides the central basis within the United States of a complete and consistent system of physical measurement; coordinates that system with measurement systems of other nations; and furnishes essential services leading to accurate and uniform physical measurements throughout the Nation's scientific community, industry, and commerce. The Institute consists of a Center for Radiation Research, an Office of Measurement Services and the following divisions:

- Applied Mathematics — Electricity — Mechanics — Heat — Optical Physics —

THE INSTITUTE FOR MATERIALS RESEARCH conducts materials research leading to improved methods of measurement, standards, and data on the properties of well-characterized materials needed by industry, commerce, educational institutions, and Government; provides advisory and research services to other Government agencies; and develops, produces, and distributes standard reference materials. The Institute consists of the Office of Standard Reference Materials and the following divisions:


THE INSTITUTE FOR APPLIED TECHNOLOGY provides technical services to promote the use of available technology and to facilitate technological innovation in industry and Government; cooperates with public and private organizations leading to the development of technological standards (including mandatory safety standards), codes and methods of test; and provides technical advice and services to Government agencies upon request. The Institute also monitors NBS engineering standards activities and provides liaison between NBS and national and international engineering standards bodies. The Institute consists of a Center for Building Technology and the following divisions and offices:

- Engineering and Product Standards—Weights and Measures—Invention and Innovation—Product Evaluation Technology—Electronic Technology—Techni-
- cal Analysis—Measurement Engineering—Building Standards and Code Serv-
- ices 4—Housing Technology 4—Federal Building Technology 4—Structures, Ma-

THE INSTITUTE FOR COMPUTER SCIENCES AND TECHNOLOGY conducts research and provides technical services designed to aid Government agencies in improving cost effectiveness in the conduct of their programs through the selection, acquisition, and effective utilization of automatic data processing equipment; and serves as the principal focus within the executive branch for the development of Federal standards for automatic data processing equipment, techniques, and computer languages. The Center consists of the following offices and divisions:

- Information Processing Standards—Computer Information—Computer Services
- Systems Development—Information Processing Technology.

THE OFFICE FOR INFORMATION PROGRAMS promotes optimum dissemination and accessibility of scientific information generated within NBS and other agencies of the Federal Government; promotes the development of the National Standard Reference Data System and a system of information analysis centers dealing with the broader aspects of the National Measurement System; provides appropriate services to ensure that the NBS staff has optimum accessibility to the scientific information of the world, and directs the public information activities of the Bureau. The Office consists of the following organizational units:


---

1 Headquarters and Laboratories at Gaithersburg, Maryland, unless otherwise noted; mailing address Washington, D.C. 20234.
2 Part of the Center for Radiation Research.
3 Located at Boulder, Colorado 80302.
4 Part of the Center for Building Technology.
A Set of Debugging and Monitoring Facilities to Improve the Diagnostic Capabilities of a Compiler

Elizabeth Fong

Systems Development Division
Institute for Computer Sciences and Technology
National Bureau of Standards
Washington, D.C. 20234

NBS Technical Notes are designed to supplement the Bureau's regular publications program. They provide a means for making available scientific data that are of transient for limited interest. Technical Notes may be listed or referred to in the open literature.

U.S. DEPARTMENT OF COMMERCE, Frederick B. Dent, Secretary
NATIONAL BUREAU OF STANDARDS, Richard W. Roberts, Director
Issued March 1973
## CONTENTS

1. Introduction .................................................. 1

2. Scope and Sources ............................................. 2

3. Descriptions of Debugging and Monitoring Facilities ........ 3
   3.1 Compile Time Checks ....................................... 3
       3.1.1 Syntax Checking ........................................ 3
       3.1.2 Static Concordance of Variables and Labels .......... 4
       3.1.3 Language Flagging Features ............................ 4
       3.1.4 Logical Segmentation of Programs .................... 5
       3.1.5 Static Control Structure Concordance ............... 5
   3.2 Link/Load Time Checks ..................................... 5
       3.2.1 Formal and Actual Argument Checks .................. 6
       3.2.2 Static Subroutine Structure Analysis ............... 6
   3.3 Execution Time Checks ..................................... 6
       3.3.1 Dynamic Trace of Subroutine Calls ................. 7
       3.3.2 Backward Trace of Subroutine Calls Upon Error Termination ........................................ 7
       3.3.3 Variable Trace .......................................... 7
       3.3.4 Snapshot ................................................. 7
       3.3.5 Flow Trace ................................................ 7
       3.3.6 Array Bounds Checking ................................. 7
       3.3.7 Effective Address Check ............................... 8
       3.3.8 Value of the Control Variable Upon Exit ........... 8
       3.3.9 GOTO Checks ............................................. 8
       3.3.10 Truncation Error Warning ............................. 8

4. Conclusions ................................................... 9

5. References ................................................... 20
A SET OF DEBUGGING AND MONITORING FACILITIES TO IMPROVE THE DIAGNOSTIC CAPABILITIES OF A COMPILER

Elizabeth Fong

Increasing concern with the quality of computer software today makes it important to evaluate critically the debugging facilities available in high-level languages. This paper presents a collection of program debugging and monitoring facilities to improve the diagnostic capabilities of a compiler. A distinction is made between debugging and monitoring facilities performed at compile time, at link/load time and at execution time. These facilities are described in terms of this breakdown with a conscious attempt to move the detection of errors from execution time to compile or link/load time, and to collect information when the information is available during the compilation process.

Key words: Compiler; debugging; error diagnostic; high-level programming languages; monitoring; procedural-oriented languages.

1. INTRODUCTION

Software production requires large amounts of both human and machine resources. The recent trend has been towards a more systematic development of software and in particular towards producing "quality" software that is effective and maintainable. During the program development stage, most of the programmer's effort is spent in debugging. Program debugging is still a very uncertain process. Professionals do not agree on how to get the bugs out of a program systematically, or how to construct the most reliable program. Tools, both automated and manual, exist but surprisingly, much of the work in this field has been described only in unpublished reports or passed on through oral tradition. Interest in debugging techniques has been aroused recently, and a collection of papers devoted to the topics of debugging and maintenance of large software systems can be found in Reference [1].

The programming activity is rooted in the traditional environment, that is, writing the program in a high-level programming language, such as FORTRAN, ALGOL, COBOL, or PL/1 with the usual compile-link/load-go situation. To aid the programmers in the stage of program development, it is advantageous to build debugging aids and monitoring features into a high-level programming language system.

This paper presents a collection of program debugging and monitoring facilities to improve the diagnostic capabilities of a compiler. A distinction is made between debugging and monitoring facilities performed

1Figures in brackets indicate the literature reference at the end of this paper.
at compile time, at link/load time and at execution time. These facilities are described in terms of this breakdown with a conscious attempt:

- to move, as much as possible, the detection of errors from execution time to compile or link/load time in order to avoid overhead in an executing program, and
- to collect information when the information is available during the compilation process. At the user's option, this information could be output at a later time.

2. SCOPE AND SOURCES

Many programmers think of debugging aids as "dumps" (the display of the contents of storage cells) and "traces" (the display of the control flow during execution). However, these basic tools have become considerably more sophisticated over the years. In particular, very powerful debugging facilities have been built for the on-line interactive environment [2,3] giving the user a high-level language with which to control the setting of breakpoints and to allow interrogation of individual storage cells.

The universities in teaching programming to students, have found that they need an extensive set of debugging facilities. "Forgiving" compilers have emerged. These include not only error detection but error correction features as well. Examples include Cornell Computing Language (CORC) [4], Waterloo FORTRAN (WATFOR and WATFIV) [5], and Purdue Fast FORTRAN Translator (PUFFT) [6].

Program monitors are another programming aid. These can be viewed as a logical extension of the debugging aids described above. A program monitor is a program built to measure the behavior of another program. In general, the program monitors that are available commercially operate in two phases: data gathering and data presentation.

The idea of combining debugging and monitoring facilities as an augmentation to high-level languages has been attempted in the EXDAMS Project [7]. Both static and dynamic displays may be exhibited at the request of the programmer.

Dijkstra's notion of structured programming [8], and Wirth's concept of program development as the process of successive refinement [9] are attempts to move programming from a private art to a new level of precision. The essential ingredient in achieving a correct final program quickly seems to lie in the area called "good" programming practices. Examples of "good" programming practice include the exhibition of structured readable source statement as output, planting special checks and messages for the use of undeclared variables, and elimination of wild GOTO transfers, etc. Some of these ingredients could be implemented on a compiler.
3. DESCRIPTIONS OF DEBUGGING AND MONITORING FACILITIES

In a compiler environment, the program text is coded, keypunched and fed into a compiler which produces object code. The object code and a system library are link-edited and loaded by a link/loader which produces absolute code ready to be executed. (see Figure 1). Three distinct times can be identified at which debugging and monitoring can occur: compile time, link/load time, and execution. Each individual type of check is described in terms of this breakdown.

Figure 1

Debugging and monitoring features described below are categorized according to these stages. Some features may occur either at compile time or link-load time depending on the implementation. This three-part breakdown hypothesizes that the earlier an error is caught, the easier it is to deal with. Therefore, any tool or technique that moves detection of certain errors from execution time to compile time is valuable. This breakdown does not apply to the interpretive environment.

3.1 Compile Time Checks

The types of checks performed by the compiler are usually referred to as static checks because the problems they address are invariant with execution time and independent of the values of the input data. The following are descriptions of some program debugging and monitoring facilities that could be performed at compile time.

3.1.1 Syntax Checking. One of the first tasks done by a compiler is to scan the source program statements. Every high-level language has its own established set of syntax rules. Error situations occur when the compiler cannot translate a source statement, or the process of syntax-directed parsing is blocked.
Examples of syntax errors include misspelling of reserve words, punctuation errors such as unmatched parentheses, missing operators, illegal sequences of operators in expressions, or missing statement labels. In a class of university-developed compilers (e.g., CORC, PUFFT, WATFOR) with "forgiving" features built in, extensive syntax checking is done. These compilers also do some error-correction during source language scanning. For example, CORC attempts to define various categories of misspelling and always tries to repair the error and continue to compile executable code. All compilers have syntax checking facilities, however the extent of checking varies.

3.1.2 Static Concordance of Variables and Labels. At compile time, all the variables and labels are entered into a symbol table or an intermediate file. This information could be made available during compile time to produce one of the most beneficial aids to debugging. This facility is sometimes called a cross reference listing. The usual format consists of the symbolic name followed by the concordance of the line numbers where the variable is being defined and the line numbers where this variable is being referenced. Some compilers also print out data type and relocation information. An example appeared in Figure 2. In this case, the statement label concordance appeared as a separate listing. The advantages of such a facility would be to detect a variable or a label being defined but never referenced or vice versa. For a block-oriented language such as ALGOL 60 or PL/1, the variables and labels may be declared as local within a block. Exhibiting a static concordance of variables and labels plus their scope of definition would aid debugging. Most of the compilers provide this facility. There are also programming support packages available which can generate such a concordance.

3.1.3 Language Flagging Features. Due to historical development, most programming languages (e.g., FORTRAN) have undergone modifications and upgrading (e.g., FORTRAN II, Basic FORTRAN, FORTRAN IV, and extended FORTRAN). Moreover, each implementation of the language may contain differences. Flagging can be done during the source language scanning phase to detect language syntactic features which deviate from some "standard" definition such as ASA FORTRAN [10], or additional features from the previous versions. A message could be printed out by the compiler as a warning to the user that the particular feature is unique to this version of the compiler. Very few compilers have this flagging facility. An idea closely related to this can be found in a COBOL compiler which prints the message: "REDEF Name feature not the same as previously defined version". (See Figure 3). It is possible to determine the language feature differences by compiling the previously compiled program with the new compiler, however, the
point here is to detect deviations specifically against a "standard" language definition, or to be able to identify a subset of the language that existed earlier. To make a full semantic check of a program is not possible.

3.1.4 Logical Segmentation of Programs. A large program may be considered as consisting of small logical segments. This permits a modular organization. Modularity aids debugging because it breaks up the program into manageable sized pieces. In ALGOL, there are BEGIN-END statements which create a logical segment. In COBOL, a paragraph or a section also is considered to be a logical segment. A formatting facility could be built into the compiler which would cause more readable source program statements to be printed out. Examples are: alignment of BEGINs with corresponding ENDS with nested inner blocks properly indented; hierarchical data declaration in COBOL or PL/1 are properly indented to reflect structures even when the input is not punched in the proper column; or reordering of source statements in order to group declaration statements, executable statements, and format statements. Figure 4 shows an example of an ALGOL program printout with block number and level number generated by the compiler to reflect logical segments. This facility, although not a debugging feature in the sense that it helps to uncover errors, could greatly add to the readability of a program. Producing readable documentation also helps another person in maintaining the program.

3.1.5 Static Control Structure Concordance. The normal flow of control in a program is sequential. However, control may be transferred elsewhere in the program by a transfer of control command, e.g., GOTO, RETURN. A topological structure diagram showing every transfer of control at the request of the user would aid debugging. Figure 5 contains such a static control structure printout, showing that a transfer of control is occurring at line number n to line number m. Some compilers do not provide a total topological control structure trace but merely print out messages such as "Control can never reach the next statement" or "There is no path to this statement" (See Figure 6 and 7).

3.2 Link/Load Time Checks

In most batch-oriented installations, compilation produces a binary program file, usually stored on direct access storage. Subprograms and the main program are compiled separately and bound together by the link/ loader which associates external references and adjusts addresses. If an external reference is to a system library routine, it link/loads this routine. The information available at this point enables the system to perform the following type of debugging or monitoring checks:
3.2.1 Formal and Actual Argument Checks. A subroutine or function declaration consists of the name of the subroutine or function followed by a list of actual parameters.

For a programming language such as ASA Standard FORTRAN, the definition of the language specifies that the actual arguments must agree in order, number and type with the corresponding formal arguments. A check on all three items would insure that the program is adhering to a "standard". In order to perform checks at link/load time, it is necessary to carry more information than the usual external symbol definition, for example, the list of formal argument names and their data types, the corresponding actual argument names and their data types. The advantages of performing the checks at link/load time rather than execution time is that the check would be done only once, rather than each time the subprogram is executed. The tests that could be performed include the following:

a) If the subroutine or function does contain a non-standard return, test that the return label is indeed a label.

b) In the case of FORTRAN, if one of the arguments of the subroutine or function is a subroutine name or a function name, the compiler could check to determine if the name is being declared as an external procedure.

c) Some implementations of FORTRAN compilers allow the number of formal arguments to differ from the number of actual arguments. If the standard language definition, e.g., ASA FORTRAN standard, calls for a match in the number, the check might insure that the program is adhering to the appropriate standard.

3.2.2 Static Subroutine Structure Analysis. The static information about subroutine structure consists of caller and callee relationship derived from the program text. In most languages, this caller-callee relationship is invariant at execution time. This information is available at link/load time and usually contained in the external symbol definition table. The link/loader could optionally produce a concordance of the source language subprogram caller/callee names. An example of the "CALL" concordance (Figure 8) and "CALLED-BY" concordance (Figure 9) was developed at NBS on the UNIVAC 1108 computer.

3.3 Execution Time Checks

Dynamic information about the actual running of the program is obtainable at this time. To embed tests, the compiler usually inserts
code at appropriate points to be executed in conjunction with the worker program. Every test introduced will decrease the running efficiency of the worker program and it is necessary to exercise care when deciding what to measure and where to embed tests. Attention also has to be given to where the tests are to be inserted to assure uniform checks.

3.3.1 Dynamic Trace of Subroutine Calls. Actual subroutine paths could be traced at execution time. This information not only aids in debugging but is very useful for program activity analysis. LEAP (Lambda Efficiency Analysis Program) [11], a software monitor, contains a special section to perform subroutine structure analysis. A tree-like diagram of the paths through the hierarchy of subprogram calls from the main program allows the user to identify the most significant call chains. Figure 10 shows a portion of a dynamic subroutine trace developed at NBS for the UNIVAC 1108. It consists of ten subroutines with its call chain properly reflected by the indented printout.

3.3.2 Backward Trace of Subroutine Calls Upon Error Termination. Upon error termination, an identification of where the error occurred plus a backward trace of the subroutine calls aids debugging. Figure 11 shows an example of backward trace of subroutine calls when the program terminated with an error. This example is generated on the UNIVAC 1108 FORTRAN compiler. This compiler uses an extra word in the calling sequence to store the "walk-back" or the backward reference to permit this kind of backward trace.

3.3.3 Variable Trace. This is a dynamic display of the specified variable and its content at each instant in time. The display usually occurs as an instruction by instruction accounting of information or at every instance of a value change. It is not only useful for debugging, but also useful in spotting the value changes of certain variables for program analysis.

3.3.4 Snapshot. This is similar to the variable trace except that the variables and their values are recorded periodically on entering or exiting certain regions of the program.

3.3.5 Flow Trace. This is the dynamic display of every branch point of a running program. The trace records the decision points and exhibits the branches taken.

3.3.6 Array Bounds Checking. This checking is sometimes built into a computer as a precaution against altering of values incorrectly. The value of the subscript is tested to determine if it is within the specified dimension of the array element, and if it is also an integer constant. The check is useful in monitoring array elements and spotting the activity level of various parts of the table.
3.3.7 Effective Address Check. This is a feature provided for memory protection. Bounds registers are set to certain permitted address ranges and every effective address value is checked against these bound registers. In some cases, it is implemented as a hardware feature to avoid system overhead. This feature is especially important in a multiprogramming environment.

3.3.8 Value of the Control Variable Upon Exit. In an iteration loop such as FORTRAN-DO or ALGOL-FOR, the value of the control variable upon exit is undefined if the exit is due to exhaustion of the loop, otherwise it is the same as it was immediately preceding the execution of the exit condition. The undefined situation is handled by implementors in different ways. Some compilers even try to "guess" what the user intended. The trouble starts when programmers make use of these undefined situations on a particular implementation of the compiler, and later discover that the same program does not work correctly on a different implementation. Debugging this type of error is very difficult because it involves an understanding of the semantics of different compiler implementation of undefined situations. One of the debugging facilities for this particular undefined situation could be to report all later use of the control variable or intentionally set the value of the control variable to "undefined" (minus zero or some such number) when the exit of the loop is due to exhaustion of the loop.

3.3.9 GOTO Checks. Recently, there has been considerable interest in eliminating the GOTO statement [12, 13]. Dijkstra claims the use of GOTO statements is undesirable, and avoiding it would increase the readability and proveability of the program. When GOTO cannot be avoided, and when label variables are allowed e.g. FL/1, the following kinds of checks could be performed by the compiler:

(a) Flag error if the transfer is made to itself or to a non-executable statement.

(b) Flag error if the transfer label value is negative or undefined.

(c) Flag error if the transfer label value is outside of the user's assigned program space.

(d) Flag warning if the transfer label goes within an iteration loop.

3.3.10 Truncation Error Warning. On an arithmetic or MOVE-data operation, some bits may be dropped due to computer word-length. Overflow to the left or to the right of the computer word is called truncation error. If such overflow is sus-
4. CONCLUSIONS

Increasing concern with the quality of computer software today makes it important to evaluate critically the debugging facilities available in high-level languages. The debugging and monitoring aids described above are particularly useful during early implementation and initial system integration stages. They could be automated by embedding these checks at appropriate points in the compilation-execution process when all the needed information is available.

The list of features is not geared to any particular high-level programming language; however, some of the features described are applicable only to particular language constructs. Techniques of implementation and the question of how to invoke and suppress these debugging and monitoring facilities have not been addressed here.

Commercially available compilers usually provide some debugging facilities; however, certain trade-off decisions are made which usually sacrifice the extent of providing debugging aids in favor of efficiency of the compiler. Such a list might prove to be useful as an evaluation criteria in determining the capabilities of a compiler.

These debugging and monitoring facilities described above could be automated to aid programmers during program development stage. Certain bugs are very much problem and situation dependent. Such bugs are difficult even to anticipate. For instance, it would be in general impossible for the compiler to isolate a sequence of code and determine that it is a non-terminating loop. There also exists a whole class of "Timing Bugs" which, when certain combination of situation occurs, the program behaves abnormally.

The problems in the area of debugging and monitoring are many: The question of how much to output, what should be outputted, how to recover from error conditions, what should be the default situation, etc. All of these questions are still open and the decisions are usually left with the implementor.

There are other automated testing techniques for validating purposes, but they are beyond the scope of this paper. Such techniques include benchmark tests, exhaustive exercising of the program with different input data, and proof of correctness using formal logic, etc. Testing methods for validation purposes are very different from those mentioned here, which are limited to debugging and monitoring aids for program development purposes.

The author would like to thank Mrs. Frances E. Holberton for giving her very valuable suggestions. The author also wishes to acknowledge Dr. Rona Stillman for reading the paper. Dr. Selden Stewart and Mr. Justin Walker provided some of the computer examples.
Figure 3
BEGIN

STRING STACK1001; COMPAKE(211, PROGSU);  
RULE(TITLE(12), 3, CHAIN(211, 2, ARROW(1)), 2, PHONE(194), 1, COMM(10));
PROC(12));
STRING ARROW RECOGNIZE(51;50); TABLE(1111111101);
INTEGER ARRAY VNAME(1:1001);
STRING INC(1:1001);
STRING ARRAY RES(1:1:75);
STRING BACKND(101);
FORMAT FORM2(A, XI, SD01)
LOCAL LABEL DONE;
FORMAT FORM1(A, SD01);
INTEGER GENERATE;

INTEGER PROCEDURE INDEX(ITEM, CI):
STRING CI;
VALUE * ITEM;
BEGIN INTEGER K;
FOR K = (1, 100) DO
IF ITEM(K) EQL C THEN GO TO MATCH;
K = C;
MATCH: INDEX(K);
END INDEX:

BEGIN INTEGER NUNELIST;
STRING LIST;
VALUE LIST;
BEGIN INTEGER 1, 13, MM;

Figure 4
C******************************************
C*****
C***** KFI = 433
C*****
C*****
C*****INTEGER FUNCTION OF LOGICAL ARGUMENT (TEST 3)
7 FUNCTION KFI(AW,B)
8 LOGICAL AW,B
9 IF (AW,B) GO TO 4331
10 4330 IF (, NOT, AW,B) GO TO 4332
11 RETURN
12 4331 KFI = 2
13 GO TO 4330
14 4332 KFI = 0
15 RETURN
16 END

TRANSFERS....
FROM LINE# TO LINE# FROM LINE# TO LINE# FROM LINE# TO LINE# FROM LINE# TO LINE# FROM LINE# TO LINE#
15 RETURN 13 10 11 RETURN 10 14 9 12

Figure 5
<table>
<thead>
<tr>
<th>CARD NO.</th>
<th>SEVERITY</th>
<th>DIAGNOSTIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>630</td>
<td>I</td>
<td>THERE IS NO PATH TO THIS STATEMENT</td>
</tr>
<tr>
<td>1069</td>
<td>FE</td>
<td>UNDEFINED STATEMENT NUMBERS, SEE BELOW</td>
</tr>
</tbody>
</table>

**UNDEFINED LABELS**

221

Figure 7
CALL CONCORDANCE.

ROUTINE A  CALLED B
            C
            D
            E
            F
            G
            H
            J
            K

ROUTINE B  CALLED C
            D
            E
            F
            G
            H
            J
            K

ROUTINE C  CALLED D
            E
            F
            G
            H
            J
            K

ROUTINE D  CALLED E
            F
            G
            H
            J
            K

ROUTINE E  CALLED F
            G
            H
            J
            K

ROUTINE F  CALLED G
            H
            J
            K

ROUTINE G  CALLED H
            J
            K

ROUTINE H  CALLED J
            K

ROUTINE J  CALLED K

ROUTINE MAIN  CALLED A

Figure 8
CALLED-BY CONCORDANCE:

SUBROUTINE A WAS CALLED BY MAIN

SUBROUTINE B WAS CALLED BY A

SUBROUTINE C WAS CALLED BY A

SUBROUTINE D WAS CALLED BY A

SUBROUTINE E WAS CALLED BY A

SUBROUTINE F WAS CALLED BY A

SUBROUTINE G WAS CALLED BY A

SUBROUTINE H WAS CALLED BY A

SUBROUTINE I WAS CALLED BY A

SUBROUTINE J WAS CALLED BY A

SUBROUTINE K WAS CALLED BY A

Figure 9
SUBR1 - (170) SUBROUTINE SUBPROGRAM
          WITHOUT AN ARGUMENT

ASA REF. - E.4.1

RESULTS

ERROR TERMINATION IN NEXPI$ ROUTINE
NEXPI$ CALLED AT SEQUENCE NUMBER 00120 OF SUBR2
SUBR2 CALLED AT SEQUENCE NUMBER 00102 OF MAIN PROGRAM

Figure 11
5. REFERENCES


### ABSTRACT

Increasing concern with the quality of computer software today makes it important to evaluate critically the debugging facilities available in high-level languages. This paper presents a collection of program debugging and monitoring facilities to improve the diagnostic capabilities of a compiler. A distinction is made between debugging and monitoring facilities performed at compile time, at link/load time and at execution time. These facilities are described in terms of this breakdown with a conscious attempt to move the detection of errors from execution time to compile or link/load time, and to collect information when the information is available during the compilation process.
JOURNAL OF RESEARCH reports National Bureau of Standards research and development in physics, mathematics, and chemistry. Comprehensive scientific papers give complete details of the work, including laboratory data, experimental procedures, and theoretical and mathematical analyses. Illustrated with photographs, drawings, and charts. Includes listings of other NBS papers as issued.

Published in two sections, available separately:

- **Physics and Chemistry (Section A)**
  Papers of interest primarily to scientists working in these fields. This section covers a broad range of physical and chemical research, with major emphasis on standards of physical measurement, fundamental constants, and properties of matter. Issued six times a year. Annual subscription: Domestic, $17.00; Foreign, $21.25.

- **Mathematical Sciences (Section B)**
  Studies and compilations designed mainly for the mathematician and theoretical physicist. Topics in mathematical statistics, theory of experiment design, numerical analysis, theoretical physics and chemistry, logical design and programming of computers and computer systems. Short numerical tables. Issued quarterly. Annual subscription: Domestic, $9.00; Foreign, $11.25.

**TECHNICAL NEWS BULLETIN**

The best single source of information concerning the Bureau's measurement, research, developmental, cooperative, and publication activities, this monthly publication is designed for the industry-oriented individual whose daily work involves intimate contact with science and technology—for engineers, chemists, physicists, research managers, product-development managers, and company executives. Includes listing of all NBS papers as issued. Annual subscription: Domestic, $6.50; Foreign, $8.25.

**BIBLIOGRAPHIC SUBSCRIPTION SERVICES**

The following current-awareness and literature-survey bibliographies are issued periodically by the Bureau:

**Cryogenic Data Center Current Awareness Service** (Publications and Reports of Interest in Cryogenics).

A literature survey issued weekly. Annual subscription: Domestic, $20.00; foreign, $23.00.

**Liquefied Natural Gas.** A literature survey issued quarterly. Annual subscription: $20.00.


