NAT'L INST. OF STAND & TECH
A11106 978031

. Department
Commerce

onal Bureau
tandards

Computer Science and Technology

NBS Special Publication 500-70/1

NBS Minimal BASIC Test Programs—Version 2, User's Manual

Volume 1—Documentation

National Bureau of Standards
Library, E-01 Admin. Bldg.

OCT 1 1981

191061

QC
100
.U57

70/1

NATIONAL BUREAU OF STANDARDS

The National Bureau of Standards[1] was established by an act of Congress on 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's technical work is performed by the National Measurement Laboratory, the National Engineering Laboratory, and the Institute for Computer Sciences and Technology.

THE NATIONAL MEASUREMENT LABORATORY provides the national system of physical and chemical and materials measurement; coordinates the system with measurement systems of other nations and furnishes essential services leading to accurate and uniform physical and chemical measurement throughout the Nation's scientific community, industry, and commerce; conducts materials research leading to improved methods of measurement, standards, and data on the properties of materials needed by industry, commerce, educational institutions, and Government; provides advisory and research services to other Government agencies; develops, produces, and distributes Standard Reference Materials; and provides calibration services. The Laboratory consists of the following centers:

Absolute Physical Quantities[2] — Radiation Research — Thermodynamics and Molecular Science — Analytical Chemistry — Materials Science.

THE NATIONAL ENGINEERING LABORATORY provides technology and technical services to the public and private sectors to address national needs and to solve national problems; conducts research in engineering and applied science in support of these efforts; builds and maintains competence in the necessary disciplines required to carry out this research and technical service; develops engineering data and measurement capabilities; provides engineering measurement traceability services; develops test methods and proposes engineering standards and code changes; develops and proposes new engineering practices; and develops and improves mechanisms to transfer results of its research to the ultimate user. The Laboratory consists of the following centers:

Applied Mathematics — Electronics and Electrical Engineering[2] — Mechanical Engineering and Process Technology[2] — Building Technology — Fire Research — Consumer Product Technology — Field Methods.

THE INSTITUTE FOR COMPUTER SCIENCES AND TECHNOLOGY conducts research and provides scientific and technical services to aid Federal agencies in the selection, acquisition, application, and use of computer technology to improve effectiveness and economy in Government operations in accordance with Public Law 89-306 (40 U.S.C. 759), relevant Executive Orders, and other directives; carries out this mission by managing the Federal Information Processing Standards Program, developing Federal ADP standards guidelines, and managing Federal participation in ADP voluntary standardization activities; provides scientific and technological advisory services and assistance to Federal agencies; and provides the technical foundation for computer-related policies of the Federal Government. The Institute consists of the following centers:

Programming Science and Technology — Computer Systems Engineering.

[1]Headquarters and Laboratories at Gaithersburg, MD, unless otherwise noted; mailing address Washington, DC 20234.
[2]Some divisions within the center are located at Boulder, CO 80303.

National Bureau of Standards
Library, E-01 Admin. Bldg.
DEC 4 1980

Computer Science and Technology

NBS Special Publication 500-70/1

NBS Minimal BASIC Test Programs—Version 2, User's Manual

Volume 1—Documentation

John V. Cugini
Joan S. Bowden
Mark W. Skall

Center for Programming Science and Technology
Institute for Computer Sciences and Technology
National Bureau of Standards
Washington, DC 20234

U.S. DEPARTMENT OF COMMERCE
Philip M. Klutznick, Secretary

Luther H. Hodges, Jr., Deputy Secretary

Jordan J. Baruch, Assistant Secretary for Productivity, Technology and Innovation

National Bureau of Standards
Ernest Ambler, Director

Issued November 1980

Reports on Computer Science and Technology

The National Bureau of Standards has a special responsibility within the Federal Government for computer science and technology activities. The programs of the NBS Institute for Computer Sciences and Technology are designed to provide ADP standards, guidelines, and technical advisory services to improve the effectiveness of computer utilization in the Federal sector, and to perform appropriate research and development efforts as foundation for such activities and programs. This publication series will report these NBS efforts to the Federal computer community as well as to interested specialists in the academic and private sectors. Those wishing to receive notices of publications in this series should complete and return the form at the end of this publication.

National Bureau of Standards Special Publication 500-70/1
Nat. Bur. Stand. (U.S.), Spec. Publ. 500-70/1, 79 pages (Nov. 1980)
CODEN: XNBSAV

Library of Congress Catalog Card Number: 80-600163

U.S. GOVERNMENT PRINTING OFFICE
WASHINGTON: 1980

For sale by the Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402
Price $4.00
(Add 25 percent for other than U.S. mailing)

NBS Minimal BASIC Test Programs - Version 2
User's Manual
Volume 1 - Documentation

John V. Cugini
Joan S. Bowden
Mark W. Skall

Abstract: This publication describes the set of programs developed by NBS for the purpose of testing conformance of implementations of the computer language BASIC to the American National Standard for Minimal BASIC, ANSI X3.60-1978. The Department of Commerce has adopted this ANSI standard as Federal Information Processing Standard 68. By submitting the programs to a candidate implementation, the user can test the various features which an implementation must support in order to conform to the standard. While some programs can determine whether or not a given feature is correctly implemented, others produce output which the user must then interpret to some degree. This manual describes how the programs should be used so as to interpret correctly the results of the tests. Such interpretation depends strongly on a solid understanding of the conformance rules laid down in the standard, and there is a brief discussion of these rules and how they relate to the test programs and to the various ways in which the language may be implemented.

Key words: BASIC; language processor testing; Minimal BASIC; programming language standards; software standards; software testing

Acknowledgments: Version 2 owes its existence to the efforts and example of many people. Dr. David Gilsinn and Mr. Charles Sheppard, the authors of version 1*, deserve credit for construction of that first system, of which version 2 is a refinement. In addition, they were generous in their advice on many of the pitfalls to avoid on the second iteration. Mr. Landon Dyer assisted with the testing and document preparation. It is also important to thank the many people who sent in comments and suggestions on Version 1. We hope that all the users of the resulting Version 2 will help us improve it further.

* issued as an NBS Internal Report; no longer available.

Table of Contents

Section Page

Section Page

Section Page

1 HOW TO USE THIS MANUAL

This manual presents background information and operating instructions for the NBS Minimal BASIC test programs. Readers who want a general idea of what the programs are supposed to do and why they are structured as they are should read sections 2 and 3. These sections give a brief explanation of BASIC, how it is standardized, and how the test programs help measure conformance to the standard. Those who wish to know how to interpret the results of program execution should also read section 3 and then section 4 for the general rules of interpretation and section 5 for information peculiar to individual programs and groups of programs within the test system. Section 6 contains tables of summary information about the tests.

Volume 2 of this publication consists of the source listings and sample outputs for all the test programs.

The test system for BASIC should be helpful to anyone with an interest in measuring the conformance of an implementation of BASIC (e.g., a compiler or interpreter) to the Minimal BASIC standard. This would include 1) purchasers who want to be sure they are buying a standard implementation, 2) programmers who must use a given implementation and want to know in which areas it conforms to the standard and which features to avoid or be wary of, and 3) implementors who may wish to use the tests as a development and debugging tool.

Much of this manual is derived from the technical specifications in the American National Standard for Minimal BASIC, ANSI X3.60-1978 [1]. You will need a copy of that standard in order to understand most of the material herein. Copies are available from the American National Standards Institute, 1430 Broadway, New York, NY 10018. This document will frequently cite ANSI X3.60-1978, and references to "the standard" should be taken to mean that ANSI publication.

The measure of success for Version 2 of the Minimal BASIC Test Programs is its usefulness to you. We at NBS would greatly appreciate hearing about your evaluation of the test system. We will respond to requests for clarification concerning the system and its relation to the standard. Also, we will maintain a mailing list of users who request to be notified of changes and major clarifications. Please direct all comments, questions, and suggestions to:

Project Manager
NBS BASIC Test Programs
National Bureau of Standards
Technology Bldg., Room A-265
Washington, DC 20234

2 THE LANGUAGE STANDARD FOR BASIC

2.1 History And Prospects

BASIC is a computer programming language developed in the mid 1960's by Professors John G. Kemeny and Thomas E. Kurtz at Dartmouth College. The primary motivation behind its design was educational (in contrast to the design goals for, e.g. COBOL and FORTRAN) and accordingly the language has always emphasized ease of use and understanding as opposed to simple machine efficiency. In July 1973, NBS published a "Candidate Standard for Fundamental BASIC" [2] by Prof. John A. N. Lee of the University of Massachusetts at Amherst. This work represented the beginning of a serious effort to standardize BASIC. The first meeting of the American National Standards Technical Committee on the Programming Language BASIC, X3J2, convened at CBEMA headquarters in Washington DC, on January 23-24, 1974, with Professor Kurtz as chairman. The committee adopted a program of work which envisioned development of a nucleus language followed by modularized enhancements. The nucleus finally emerged as Minimal BASIC, which was approved as an ANSI standard January 17, 1978. As its name implies, the language defined in the standard is one which any implementation of BASIC should encompass.

Meanwhile, NBS had been developing a set of test programs, the purpose of which was to exercise all the facilities defined in the standard and thereby test conformance of implementations to the standard. This test suite was released as NBSIR 78-1420-1, 2, 3, and 4, NBS Minimal BASIC Test Programs - Version 1 User's Manual in January 1978. NBS distributed this version to more than 60 users, many of whom made suggestions about how the test suite might be improved. NBS has endeavored to incorporate these suggestions and re-design the tests where it seemed useful to do so. The result is the current Version 2 of the test suite. Appendix A contains a summary of the differences between versions 1 and 2.

In order to provide a larger selection of high level programming languages for the Federal government's ADP activities, the Department of Commerce has incorporated the ANSI standard as Federal Information Processing Standard 68. This means, among other things, that implementations of BASIC sold to the Federal government after an 18 month transition period must conform to the technical specifications of the ANSI standard: hence the NBS interest in developing a tool for measuring such conformance.

ANSI X3J2 is currently (April 1980) working on a language standard for a much richer version of BASIC, which will provide such features as real-time process control, graphics, string manipulation, file handling, exception handling, and array manipulation. The current expectation is for ANSI adoption of this standard sometime in 1982. It is probable that such a standard for a full version of BASIC would be adopted as a Federal Information Processing Standard.

2.2 The Minimal BASIC Language

Minimal BASIC is distinguished among standardized computer languages by its simplicity and its suitability for the casual user. It is simple, not only because of its historic purpose as a computer language for the casual user, but also because the ANSI BASIC committee organized its work around the concept of first defining a core or nucleus language which one might reasonably expect any implementation of BASIC to include, to be followed by a standard for enhanced versions of the language. Therefore the tendency was to defer standardization of all the sophisticated features and include only the simple features. In particular, Minimal BASIC has no facilities for file handling, string manipulation, or array manipulation and has only rudimentary control structures. Although BASIC was originally designed for interactive use, the standard does not restrict implementations to that use.

Minimal BASIC provides for only two types of data, numeric (with the properties usually associated with real or floating-point numbers) and string. String data can be read as input, printed as output, and moved and compared internally. The only legal comparisons, however, are equal or not equal; no collating sequence among characters is defined. Numeric data can be manipulated with the usual choice of operations: addition, subtraction, multiplication, division, and involution (sometimes called exponentiation). There is a modest assortment of numeric functions. One- and two-dimensional arrays are allowed, but only for numeric data, not string.

For control, there is a GOTO, an IF which can cause control to jump to any line in the program, a GOSUB and RETURN for internal subroutines, a FOR and NEXT statement to execute loops while incrementing a control-variable, and a STOP statement.

Input and output are accomplished with the INPUT and PRINT statements, both of which are designed for use on an interactive terminal. There is also a feature which has no real equivalent among the popular computer languages: a kind of internal file of data values, numeric and string, which can be assigned to variables with a READ statement (not to be confused with INPUT which handles external data). The internal set of values is created with DATA statements, and may be read repetitively from the beginning of the set by using the RESTORE statement.

The programmer may (but need not) declare the size of arrays with the DIM statement and specify that subscripts begin at 0 or 1 with an OPTION statement. The programmer may also establish numeric user-defined functions with the DEF statement, but only as simple expressions, taking one argument. The RANDOMIZE statement works in conjunction with the RND function. If RND is called without execution of RANDOMIZE, it always returns the same sequence of pseudo-random numbers for each execution of the program. Executing RANDOMIZE causes RND to return an unpredictable set of values each time.

The REM statement allows the programmer to insert comments or remarks throughout the program.

Although the facilities of the language are modest, there is one area in which the standard sets rather stringent requirements, namely, diagnostic messages. The mandate of the standard that implementations exhibit reasonable behavior even when presented with unreasonable programs follows directly from the design goal of solicitude towards the beginning or casual user. Thus, the standard takes care to define what happens if the user commits any of a number of syntactic or semantic blunders. The need to test these diagnostic requirements strongly affected the overall shape of the test system as will become apparent in later sections.

2.3 Conformance To The Standard

There are many reasons for establishing a standard for a programming language: the promotion of well-defined and well-designed languages as a consequence of the standardizing process itself, the ability to create language-based rather than machine-based software tools and techniques, the increase in programmer productivity which an industry-wide standard fosters, and so on. At bottom, however, there is one result essential to the success of a standard: program portability. The same program should not evoke perniciously different behavior in different implementations. Ideally, the same source code and data environment should produce the same output, regardless of the machine environment.

How does conformance to the standard work towards this goal? Essentially, the standard defines the set of syntactically legal programs, assigns a semantic meaning to all of them and then requires that implementations (sometimes called processors; we will use the two terms interchangeably throughout this document) actually produce the assigned meaning when presented with a legal program.

2.3.1 Program Conformance

Program conformance, then, is a matter of syntax. We look at the source code and determine whether or not it obeys all the rules laid down in the standard. These rules are mostly spelled out in the syntax sections of the standard using a variant of Backus-Naur Form (BNF). They are supplemented, however, by certain context-sensitive constraints, which are contained in the semantics sections. Thus the rules form a ceiling for conforming programs. If a program has been constructed according to the rules, it meets the standard, without regard to its semantic meaning. The syntactic rules are fully implementation independent: a standard program must be accepted by all standard

processors. Further, since we can tell if a program is standard by mere inspection, it would be a reasonably easy job to build a recognizer or syntax checker which could always discover whether or not a program is standard. Unfortunately, such a recognizer could not be written in Minimal BASIC itself and this probably explains why no recognizer to check program conformance has gained wide acceptance. At least one such recognizer does exist, however. Called PBASIC [3], it was developed at the University of Kent at Canterbury and is written in PFORT, a portable subset of FORTRAN. PBASIC was used to check the syntax of the Version 2 Minimal BASIC Test Programs.

2.3.2 Implementation Conformance

Implementation conformance is derivative of the more primitive concept of program conformance. In contrast to the way in which program conformance is described, processor conformance is specified functionally, not structurally. The essential requirement is that an implementation accept any standard program and produce the behavior specified by the language standard. That is, the implementation must make the proper connection between the syntax of the program and the operation of the computer system. Note that this is a black box description of the implementation. Whether the realization of the language is done with a compiler, an interpreter, firmware, or by being hard-wired, is irrelevant. Only the external behavior is important, not the internal structure - quite the opposite of the way program conformance is determined.

The difference in the way conformance is defined for programs and processors radically affects the test methodology by which we determine whether the standard is met. The relevant point is that there currently is no way to be certain that an implementation does conform to the standard, although we can sometimes be certain that it does not. In short, there is no algorithm, such as the recognizer that exists for programs, by which we can answer definitively the question, "Is this a standard processor?"

Furthermore, the standard acts as a floor for processors, rather than a ceiling. That is, an implementation must accept and process <u>at least</u> all standard programs, but may also implement enhancements to the language and thus accept non-standard programs as well. Another difference between program and processor conformance is that the description of processor conformance allows for some implementation dependence even in the treatment of standard programs. Thus for some standard programs there is no unique semantic meaning, but rather a set of meanings, usually similar, among which implementations can choose.

3 DETERMINING IMPLEMENTATION CONFORMANCE

3.1 Test Programs As Test Data, Not Algorithms

The test programs do not embody some definitive algorithm by which the question of processor conformance can be answered yes or no. There is an important sense in which it is only accidental that they are programs at all; indeed, some of them, syntactically, are not. Rather their primary function is as test data. It is readily apparent, for instance, that the majority of BASIC test programs are algorithmically trivial; some consist only of a series of PRINT statements. Viewed as test data, however, i.e., a series of inputs to a system whose behavior we wish to probe, the underlying motivation for their structure becomes intelligible. Simply put, it is the goal of the tests to exercise at least one representative of every meaningfully distinct type of syntactic structure or semantic behavior provided for in the language standard. This strategy is characteristic of testing in general: all one can do is submit a representative subset of the typically infinite number of possible inputs to the system under investigation (the implementation) and see whether the results are in accord with the specifications for that system (the language standard).

Thus, successful results of the tests are necessary, but not sufficient to show that the specifications are met. A failed test shows that a language implementation is not standard. A passed test shows that it may be. A long series of passed tests which seem to cover all the various aspects of the language gives us a large measure of confidence that the implementation conforms to the standard.

It can scarcely be stressed too strongly that the test programs do not represent some self-sufficient algorithm which will automatically deliver correct results to a passive observer. Rather they are best seen as one component in a larger system comprising not only the programs, but the documentation of the programs, the documentation of the processor under test, and, not least, a reasonably well-informed user who must actively interpret the results of the tests in the context of some broad background knowledge about the programs, the processor, and the language standard. If, for example, a processor rejects a standard program, it certainly fails to conform to the standard; yet this is a type of behavior which can hardly be detected by the program itself: only a human observer who knows that the processor must accept standard programs, and that this program is standard, is capable of the proper judgment that this processor therefore violates the language standard.

3.2 Special Issues Raised By The Standard Requirements

3.2.1 Implementation-defined Features

At several points in the standard, processors are given a choice about how to implement certain features. These subjects of choice are listed in Appendix C of the standard. In order to conform, implementations must be accompanied by documentation describing their treatment of these features (see section 1.4.2(7) of the standard). Many of these choices, especially those concerning numeric precision, string and numeric overflow, and uninitialized variables, can have a marked effect on the result of executing even standard programs. A given program, for instance, might execute without exceptions on one standard implementation, and cause overflow on another, with a notably different numeric result. The programs that test features in these areas call for especially careful interpretation by the user.

Another class of implementation-defined features is that associated with language enhancements. If an implementation executes non-standard programs, it also must document the meaning it assigns to the non-standard constructions within them. For instance, if an implementation allows comparison of strings with a less-than operator, it must document its interpretation of this comparison.

3.2.2 Error And Exception Reporting

The standard for BASIC, in view of its intended user base of beginning and casual programmers, attempts to specify what a conforming processor must do when confronted with non-standard circumstances. There are two ways in which this can happen: 1) a program submitted to the processor might not conform to the standard syntactic rules, or 2) the executing program might attempt some operation for which there is no reasonable semantic interpretation, e.g., division by zero, assignment to a subscripted variable outside of the array. In the BASIC standard, the first case is called an error, and the second an exception, and in order to conform, a processor must take certain actions upon encountering either sort of anomaly.

Given a program with a syntactically non-standard construction the processor must either reject the program with a message to the user noting the reason for rejection, or, if it accepts the program, it must be accompanied by documentation which describes the interpretation of the construction.

If a condition defined as an exception arises in the course of execution, the processor is obliged, first to report the exception, and then to do one of two things, depending on the type of exception: either it must apply a so-called recovery procedure and continue execution, or it must terminate execution.

Note that it is the user, not the program, who must determine whether there has been an adequate error or exception report, or whether appropriate documentation exists. The pseudo-code in Figure 1 describes how conforming implementations must treat errors. It may be thought of as an algorithm which the user (not the programs) must execute in order to interpret correctly the effect of submitting a test program to an implementation.

The procedure for error handling in Figure 1 speaks of a processor accepting or rejecting a program. The glossary (sec. 19) of the standard defines accept as "to acknowledge as being valid". A processor, then, is said to reject a program if it in some way signifies to the user that an invalid construction (and not just an exception) has been found, whenever it encounters the presumably non-standard construction, or if the processor simply fails to execute the program at all. A processor implicitly accepts a program if the processor encounters all constructions within the program with no indication to the user that the program contains constructions ruled out by the standard or the implementation's documentation.

In like manner, we can construct pseudo-code operating instructions to the user, which describe how to determine whether an exception has been handled in conformance with the standard and this is shown also in Figure 1.

As a point of clarification, it should be understood that these categories of error and exception apply to all implementations, both compilers and interpreters, even though they are more easily understood in terms of a compiler, which first does all the syntax checking and then all the execution, than of an interpreter. There is no requirement, for instance, that error reports precede exception reports. It is the content, rather than the timing, of the message that the standard implies. Messages to reject errors should stress the fact of ill-formed source code. Exception reports should note the conditions, such as data values or flow of control, that are abnormal, without implying that the source code per se is invalid.

Error Handling

```
if program is standard
   if program accepted by processor
      if correct results and behavior
         processor PASSES
      else
         processor FAILS (incorrect interpretation)
      endif
   else
      processor FAILS (rejects standard program)
   endif
else (program non-standard)
   if program accepted by processor
      if non-standard feature correctly documented
         processor PASSES
      else
         processor FAILS (incorrect/missing documentation
                          for non-standard feature)*
      endif
   else (non-standard program rejected)
      if appropriate error message
         processor PASSES
      else
         processor FAILS (did not report reason for rejection)
      endif
   endif
endif
```

* note that all implementation-defined features must be documented (See Appendix C in the ANSI Standard) not just non-standard features.

Exception Handling

```
if processor reports exception
   if procedure is specified for exception
         and host system capable of procedure
      if processor follows specified procedure
         processor PASSES
      else
         processor FAILS (recovery procedure not followed)
      endif
   else (no procedure specified or unable to handle)
      if processor terminates program
         processor PASSES
      else
         processor FAILS (non-termination on fatal exception)
      endif
   endif
else
   processor FAILS (fail to report exception)
endif
```

Figure 1

4 STRUCTURE OF THE TEST SYSTEM

The design of the test programs is an attempt to harmonize several disparate goals: 1) exercise all the individual parts of the standard, 2) test combinations of features where it seems likely that the interaction of these features is vulnerable to incorrect implementation, 3) minimize the number of tests, 4) make the tests easy to use and their results easy to interpret, and 5) give the user helpful information about the implementation even, if possible, in the case of failure of a test. The rest of this section describes the strategy we ultimately adopted, and its relationship to conformance and to interpretation by the user of the programs.

4.1 Testing Features Before Using Them

Perhaps the most difficult problem of design is to find some organizing principle which suggests a natural sequence to the programs. In many ways, the most natural and simple approach is simply to test the language features in the order they appear in the standard itself. The major problem with this strategy is that the tests must then use untested features in order to exercise the features of immediate interest. This raises the possibility that the feature ostensibly being tested might wrongly pass the test because of a flaw in the implementation of the feature whose validity is implicitly being assumed. Furthermore, when a test does report a failure, it is not clear whether the true cause of the failure was the feature under test or one of the untested features being used.

These considerations seemed compelling enough that we decided to order the tests according to the principle of testing features before using them. This approach is not without its own problems, however. First and most importantly, it destroys any simple correspondence between the tests and sections of the standard. The testing of a given section may well be scattered throughout the entire test sequence and it is not a trivial task to identify just those tests whose results pertain to the section of interest. To ameliorate this problem, we have been careful to note at the beginning of each test just which sections of the standard it applies to, and have compiled a cross-reference listing (see section 6.3), so that you may quickly find the tests relevant to a particular section. A second problem is that occasionally the programming of a test becomes artificially awkward because the language feature appropriate for a certain task hasn't been tested yet. While the programs generally abide by the test-before-use rule, there are some cases in which the price in programming efficiency and convenience is simply too high and therefore a few of the programs do employ untested features. When this happens, however, the program always generates a message telling you which untested feature it is depending on. Furthermore, we were careful to use the untested

feature in a simple way unlikely to interact with the feature under test so as to mask errors in its own implementation.

4.2 Hierarchical Organization Of The Tests

Within the constraints imposed by the test-before-use rule, we tried to group together functionally related tests. This grouping should also help you interpret the tests better since you can usually concentrate on one part of the standard at a time, even if the parts themselves are not in order. Section 6.1 of this manual contains a summary of the hierarchical group structure. It relates a functional subject to a sequential range of tests and also to the corresponding sections of the standard. We strongly recommend that you read the relevant sections of the standard carefully before running the tests in a particular group. The documentation contained herein explains the rationale for the tests in each group, but it is not a substitute for a detailed understanding of the standard itself.

Many of the individual test programs are themselves further broken down into so-called sections. Thus the overall hierarchical subdivision scheme is given by, from largest to smallest: system, groups, sub-groups, programs, sections. Program sections are further discussed below under: 4.4.3 Documentation.

4.3 Environment Assumptions

The test programs are oriented towards executing in an interactive environment, but generally can be run in batch mode as well. Some of the programs do require input, however, and these present more of a problem, since the input needed often depends on the immediately preceding output of the program. See the sample output in Volume 2 for help in setting up data files if you plan to run all the programs non-interactively. The programs which use the INPUT statement are 73, 81, 84, 107-113, and 203.

We have tried to keep the storage required for execution within reasonable bounds. Array sizes are as small as possible, consistent with adequate testing. No program exceeds 300 lines in length. The programs print many informative messages which may be changed without affecting the outcome of the tests. If your implementation cannot handle a program because of its size, you should set up a temporary copy of the program with the informative messages cut down to a minimum and use that version. Be careful not to omit printing which is a substantive part of the test itself.

The tests assume that the implementation-defined margin for output lines is at least 72 characters long and contains at least 5 print zones. This should not be confused with the length of a line in the source code itself. The standard requires implementations to accept source lines up to 72 characters long. If the margin is smaller than 72, the tests should still run (according to the standard), but the output will be aesthetically less pleasing.

Finally, the standard does not specify how the tests are to be submitted to the processor for execution. Therefore, the machine-readable part of the test system consists only of source code, i.e., there are no system control commands. It is your responsibility to submit the programs to the implementation in a natural way which does not violate the integrity of the tests.

4.4 Operating And Interpreting The Tests

This section will attempt to guide you through the practical aspects of using the test programs as a tool to measure implementation conformance. The more general issues of conformance are covered in section 3, and of course in the standard itself, especially sections 1 and 2 of the ANSI document.

4.4.1 User Checking Vs. Self Checking

All of the test programs require interpretation of their behavior by the user. As mentioned earlier, the user is an active component in the test system; the source code of the test programs is another component, subordinate to the test user. An important goal in the design of the programs was the minimization of the need for sophisticated interpretation of the test results; but minimization is not elimination. In the best case, the program will print out a conspicuous message indicating that the test passed or failed, and you need only interpret this message correctly. In other cases, you have to examine rather carefully the results and behavior of the program, and must apply the rules of the standard yourself. This interpretation is necessary in:

1. Programs which test that PRINTed output is produced in a certain format

2. Programs which test that termination occurs at the correct time (this arises in many of the exception tests)

3. Programs for which conformance depends on the existence of adequate documentation of implementation-defined features (both those defined in Appendix C of the standard and for any of the error tests that are accepted).

The test programs are an only partially automated solution to the problem of determining processor conformance. Naive reliance on the test programs alone can very well lead to an incorrect judgment about whether an implementation meets the standard.

4.4.2 Types Of Tests

There are four types of test programs: 1) standard, 2) exception, 3) error, and 4) informative. Within each of the functional groups (described above, section 4.2) the tests occur in that order, although not all groups have all four types. The rules that pertain to each type follow. It is quite important that you be aware of which type of test you are running and use the rules which apply to that type.

4.4.2.1 Standard Tests

These tests are the ones whose title does not begin with "EXCEPTION" or "ERROR" and which generate a message about passing or failing at the end of each section. The paragraph below on documentation describes the concept of sections of a test. Since these programs are syntactically standard and raise no exception conditions, they must be accepted and executed to completion by the implementation. If the implementation fails to do this, it has failed the test. For example, if the implementation fails to recognize the key word OPTION, or if it does not accept one of the numeric constants, then the test has failed. Quite obviously, it is you who must apply this rule, since the program itself won't execute at all.

Assuming that the implementation does process the program, the next question is whether it has done so correctly. The program may be able to determine this itself, or you may have to do some active interpretation of the results. See the section below on documentation for more detail.

4.4.2.2 Exception Tests

These tests have titles that begin with the word "EXCEPTION" and examine the behavior of the implementation when exception conditions occur during execution. Nonetheless, these programs are also standard conforming (i.e., syntactically valid) and thus the implementation must accept and process them.

There are two special considerations. The first is the distinction between so-called fatal and non-fatal exceptions. Some exceptions in the standard specify a recovery procedure which allows continued execution of the program, while others

(the fatal exceptions) do not. If no recovery procedure is specified, the implementation must report the exception and then terminate the program. Programs testing fatal exceptions will print out a message that they are about to attempt the instruction causing the exception. If execution proceeds beyond that point, the test fails and prints a message so stating. With the non-fatal exceptions, the test program attempts to discover whether the recovery procedure has been applied or not and in this instance, the test is much like the standard tests, where the question is whether the implementation has followed the semantic rules correctly. For instance, the semantic meaning of division by zero is to report the exception, supply machine infinity, and continue. The standard, however, allows implementations to terminate execution after even a non-fatal exception "if restrictions imposed by the hardware or operating environment make it impossible to follow the given procedures." Because it would be redundant to keep noting this allowance, the test programs do not print such a message for each non-fatal exception. Therefore, when running a test for a non-fatal exception, note that the implementation may, under the stated circumstances, terminate the program, rather than apply the recovery procedure.

The second special consideration is that in the case of INPUT and numeric and string overflow, the precise conditions for the exception can be implementation-defined. It is possible, therefore, that a standard program, executing on two different standard-conforming processors, using the same data, could cause an exception in one implementation and not in the other. The tests attempt to force the exception to occur, but it could happen, especially in the case of string overflow, that a syntactically standard program cannot force such an exception in a given processor. The documentation accompanying the implementation under test must describe correctly those implementation-defined features upon which the occurrence of exceptions depends. That is, it must be possible to find out from the documentation whether and when overflow and INPUT exceptions will occur in the test programs.

There is a summary of the requirements for exception handling in the form of pseudo-code in section 3.2.2 (Figure 1).

4.4.2.3 Error Tests

These tests have titles that begin with the word "ERROR" and examine how a processor handles a non-standard program. Each of these programs contains a syntactic construction explicitly ruled out by the standard, either in the various syntax sections, or in the semantics sections. Given a program with a syntactically non-standard construction the processor must either reject the program with a message to the user noting the reason for rejection, or, if it accepts the program, it must be accompanied by documentation which describes the interpretation of the

construction. Testing this requirement involves the submission of deliberately illegal programs to the processor to see if it will produce an appropriate message, or if it contains an enhancement of the language such as to assign a semantic meaning to the error. Thus we are faced with an interesting selection problem: out of the infinity of non-standard programs, which are worth submitting to the processor? Three criteria seem reasonable to apply:

1. Test errors which we might expect would be most difficult for a processor to detect, e.g., violations of context-sensitive constraints. These are the ones ruled out by the semantics rather than syntax sections of the standard.

2. Test errors likely to be made by beginners, for example use of a two character array name.

3. Test errors for which there may very well exist a language enhancement, e.g., comparing strings with "<" and ">".

Based on these criteria, the test system contains programs for the errors in the two lists which follow. The first list is for constructions ruled out by the semantics sections alone (these usually are instances of context-sensitive syntax constraints) and the second for plausible syntax errors ruled out by the BNF productions.

Context-sensitive errors:

1. line number out of strictly ascending order

2. line number of zero

3. line-length > 72 characters

4. use of an undefined user function

5. use of a function before its definition

6. recursive function definition

7. duplicate function definition

8. number of arguments in function invocation <> number of parameters in function definition

9. reference to numeric-supplied-function with incorrect number of arguments

10. no spaces around keywords

11. spaces within keywords and other elements or before line number

12. non-existent line number for GOTO, GOSUB, IF...THEN, ON...GOTO
13. mismatch of control variables in FOR-blocks (e.g., interleaving)
14. nested FOR-blocks with same variable
15. jump into FOR-block
16. conflict on number of dimensions among references: A, A(1), A(1,1)
17. conflict on number of dimensions between DIM and reference, e.g., DIM A(20) and either A or A(2,2)
18. reference to subscripted variable followed by DIMensioning thereof
19. multiple OPTION statements
20. OPTION follows reference to subscripted variable
21. OPTION follows DIM
22. OPTION BASE 1 followed by DIM A(0)
23. DIM of same variable twice

Context-free errors:

1. use of long name for array, e.g A1(1)
2. assignment of string to number and number to string
3. assignment without the keyword LET
4. comparison of two strings for < or >
5. comparison of a string with a number
6. unmatched parenthesis in expression
7. FOR without matching NEXT and vice-versa
8. multiple parameters in parameter list
9. line without line-number
10. line number longer than four digits
11. quoted strings containing the quote character or lowercase letters

12. unquoted strings containing quoted-string-characters

13. type mismatch on function reference (using string as an argument)

14. DEF with string variable for parameter

15. DEF with multiple parameters

16. misplaced or missing END-statement

17. null entries in various lists (INPUT, DATA, READ, e.g.)

18. use of "**" as involution operator

19. adjacent operators, such as 2 ^ -4

When developing programs to test for possible enhancements, we also tried to assist the user in confirming what the actual processor behavior is, so that it may be checked against the documentation. For example, the program that tests whether the implementation accepts "<" and ">" for comparison of strings also displays the implicit character collating sequence if the comparisons are accepted. When the implementation accepts an error program be sure to check that the documentation does in fact describe the actual interpretation of the error as exhibited by the test program. If the error program is rejected, the processor's error message should be a reasonably accurate description of the erroneous construction.

There is a summary of the requirements for error handling in the form of pseudo-code in section 3.2.2 (Figure 1).

4.4.2.4 Informative Tests

Informative tests are very much like standard tests. The implementation must accept and process them, since they are syntactically standard. The difference is that the standard only recommends, rather than requires, certain aspects of their behavior. The pass/fail message (described below) and other program output indicates when a test is informative and not mandatory. All the informative tests have to do with the quality (as opposed to the existence) of various mathematical facilities. Specifically, the accuracy of the numeric operations and approximated functions and the randomness of the RND function are the subjects of informative tests. Some of the standard tests also have individual sections which are informative, and again the pass/fail message is the key to which sections are informative and which mandatory. If numeric accuracy is important for your purposes, either as an implementor or a user, you should analyze closely the results of the informative tests.

4.4.3 Documentation

There are three kinds of documentation in the test system, serving three complementary purposes:

1. The user's manual (this document). The purpose of this manual is to provide a global description of the test system and how it relates to the standard and to conformance. At a more detailed level, there is also a description of each functional group of programs and the particular things you should watch for when running that group.

2. Program output. As far as possible, the programs attempt to explain themselves and how they must be interpreted to determine conformance. Nonetheless, they make sense only in the context of some background knowledge of the BASIC standard and conformance (more detail below on output format).

3. Remarks in the source code. Using the REM statement, the programs attempt to clarify their own internal logic, should you care to examine it. Many of the programs are algorithmically trivial enough that remarks are superfluous, but otherwise remarks are there to guide your understanding of how the programs are intended to work.

There is a format for program output consistent throughout the test sequence. The program first prints its identifying sequence number and title. The next line lists the sections of the ANSI standard to which this test applies. After this program header, there is general information, if any, pertaining to the whole program. Following all this program-level output there is a series of one or more sections, numbered sequentially within the program number. Each section tests one aspect of the general feature being exercised by the program. Every section header displays the section number and title and any information pertinent to that section. Then the message, "BEGIN TEST." appears, after which the program attempts execution of the feature under test. At this point, the test may print information to help the user understand how execution is proceeding.

Then comes the important part: a message, surrounded by asterisks, announcing "*** TEST PASSED ***" or "*** TEST FAILED ***". If the test cannot diagnose its own behavior, it will print a conditional pass/fail message, prefacing the standard message with a description of what must or must not have happened for the test to pass. Be careful to understand and apply these conditions correctly. It is a good idea to read the ANSI standard with special attention in conjunction with this sort of test, so that you can better understand the point of the particular section.

There is no pass/fail message for the error tests, since there is, of course, no standard semantics prescribed for a non-standard construction. As mentioned above, error programs usually generate messages to help you diagnose the behavior of the processor when it does accept such a program.

After the pass/fail message will come a line containing "END TEST." which signals that the section is finished. If there is another section, the section header will appear next. If not, there will be a message announcing the end of the program. Note that each section passes or fails independently; all sections, not just the last, must print "*** TEST PASSED ***" for the program as a whole to pass. Figure 2 contains a schematic outline of standard program output.

Format of Test Program Output

```
PROGRAM FILE nn: descriptive program title.
    ANSI STANDARD xx.x, yy.y ...

message if a feature is used before being tested, cf. section 4.1
and general remarks about the purpose of the program

SECTION nn.1: descriptive section title.

interpretive message for error or exception tests
and general remarks about the purpose of this section.

               BEGIN TEST.

function-specific messages and test results

***  TEST PASSED (or FAILED)  ***
          or
***  INFORMATIVE TEST PASSED (or FAILED)  ***
          or
conditional pass/fail message when
it cannot be determined internally.
          or
message to assist analysis of processor
behavior for error program

               END TEST.

SECTION nn.2: descriptive section title.
   .
   .
   .
SECTION nn.m: descriptive section title.
   .
   .
   .
END PROGRAM nn
```

Figure 2

5 FUNCTIONAL GROUPS OF TEST PROGRAMS

This section contains information specific to each of the groups and sub-groups of programs within the test sequence. Groups are arranged hierarchically, as reflected in the numbering system. The sub-section numbers within this section correspond to the group numbering in the table of section 6.1, e.g., section 5.12.1.2 of the manual describes functional group 12.1.2.

It is the purpose of this section to help you understand the overall objectives and context of the tests by providing information supplementary to that already in the tests. This section will generally not simply repeat information contained in the tests themselves, except for emphasis. Where the tests require considerable user interpretation, this documentation will give you the needed background information. Where the tests are self-checking, this documentation will be correspondingly brief. We suggest that you first read the comments in this section to get the general idea of what the tests are trying to do, read the relevant sections of the ANSI standard to learn the precise rules, and finally run the programs themselves, comparing their output to the sample output in Volume 2. The messages written by the test programs are intended to tell you in detail just what behavior is necessary to pass, but these messages are not the vehicle for explaining how that criterion is derived from the standard. Program output should be reasonably intelligible by itself, but it is better understood in the broader context of the standard and its conformance rules.

5.1 Simple PRINTing Of String Constants

This group consists of one program which tests that the implementation is capable of the most primitive type of PRINTing, that of string constants and also the null PRINT. Note that it is entirely up to you to determine whether the test passes or fails by assuring that the program output is consistent with the expected output. The program's own messages describe what is expected. You may also refer to the sample output in Volume 2 to see what the output should look like.

5.2 END And STOP

This group tests the means of bringing BASIC programs to normal termination. These capabilities are tested early, since all the programs use them. Both END and STOP cause execution to stop when encountered, but STOP may appear anywhere in the program any number of times. There must be exactly one END statement in a program, and it must be the last line in the source code. Thus, END serves both as a syntactic marker for the end of the program, and is also executable.

Since the program can't know when it has ended (although it can know when it hasn't), you must assure that the programs terminate at the right time.

5.3 PRINTing And Simple Assignment (LET)

This group of programs examines the ability of the implementation to print strings and numbers correctly. Both constants and variables are tested as print-items. The variables, of course, have to be given a value before they are printed, and this is done with the LET statement.

PRINT is among the most semantically complex statements in BASIC. Furthermore, the PRINT statement is the outstanding case of a feature whose operation cannot be checked internally. The consequence is that this group calls for the most sophisticated user interpretation of any in the test sequence. Please read carefully the specifications in the programs, section 12 of the ANSI standard, and this documentation; the interpretation of test results should then be reasonably clear.

The emphasis in this group is on the correct representation of numeric and string values. There is some testing that TAB, comma, and semi-colon perform their functions, but a challenging exercise of these features is deferred until group 14.6 because of the other features needed to test them.

5.3.1 String Variables And TAB

The PRINTing of strings is fairly straightforward and should be relatively easy to check, since there are no implementation-defined features which affect the printing. The only possible problem is the margin width. The program assumes a margin of at least 60 characters with at least 4 print zones. If your implementation supports only a shorter margin, you must make due allowance for it. The standard does not prescribe a minimum margin.

The string overflow test requires careful interpretation. Your implementation must have a defined maximum string length, and the fatal exception should occur on the assignment corresponding to that documented length. If the implementation supports at least 58 characters in the string, overflow should not occur. Be sure, if there is no overflow exception report, that the processor has indeed not lost data. Do this by checking that the output has not been truncated. A processor that loses string data without reporting overflow definitely fails.

Checking for a TAB exception is simple enough; just follow the conditional pass/fail messages closely. Note that one section of the test should not generate an exception since, even

though the argument itself is less than one, its value becomes one after rounding.

5.3.2 Numeric Constants And Variables

In the following discussion, the terms "significand", "exrad", "explicit point", "implicit point", and "scaled" are used in accordance with the meaning ascribed them in the ANSI standard.

The rules for printing numeric values are fairly elaborate, and, moreover, are heavily implementation-dependent; accordingly conscientious scrutiny is in order. There are two rules to keep in mind. First, the expected output format depends on the value of the print-item, not its source format. In particular, integer values should print as integers as long as the significand-width can accommodate them, fractional values should print in explicit point unscaled format where no loss of accuracy results, and the rest should print in explicit point scaled format. For example "PRINT 2.1E2" should produce "210" because the item has an integer value, even though it is written in source code in explicit point scaled format. Second, leading zeros in the exrad and trailing zeros in the significand may be omitted. Thus, for an implementation with a significand-width of 8 and an exrad-width of 3, the value 1,230,000,000 could print as "1.2300000E+009" at one extreme or "1.23E+9" at the other. The tests generally display the expected output in the latter form, but it should be understood that extra zeros can be tacked on to the actual output, up to the widths specified for the implementation.

The tests in general are oriented toward the minimum requirements of six decimal digits of accuracy, a significand length of six and an exrad-width of two. You must apply the standard requirements in terms of your own implementation's widths, however.

5.4 Control Statements And REM

This group checks that the simple control structures all work when used in a simple way. Some of the same facilities are checked more rigorously in later groups. As with PRINT, END and STOP, these features must come early in the test sequence, since a BASIC program cannot do much of consequence without them. If any of these tests fail, the validity of much of the rest of the test sequence is doubtful, since following tests rely heavily on GOTO, GOSUB, and IF. Note especially that trailing blanks should be significant in comparing strings, e.g. "ABC" <> "ABC ". Subsequent tests which rely on this property of IF will give false results if the implementation doesn't process the comparison properly.

The tests for GOTO and GOSUB exercise a variety of transfers to make sure the processor handles control correctly. If everything works, you should get intelligible, self-consistent output. If the output looks scrambled, the test has failed. There are no helpful diagnostics for failures since it is impossible to anticipate exactly how a processor might misinterpret transfers of control. Look carefully at the sample output for the GOTO and GOSUB programs in Volume 2, to know what to expect.

The IF...THEN tests use a somewhat complex algorithm, so pay attention to the REM statements if you are trying to understand the logic. On the other hand, these tests are easy to use because they are completely self-checking. You need only look for the pass/fail messages to see if they worked. It is worth noting that the IF...THEN test for numeric values depends on the validity of the IF...THEN test for strings, which comes just before.

The error tests are understandable in light of the general rules for interpretation of error programs given earlier.

5.5 Variables

The first of these programs simply checks that the set of valid names is as guaranteed by the standard. In particular, A, A0, and A$ are all distinct. There are no diagnostics for failure, since we expect failures to be rare and it is simple enough to isolate the misinterpretation by modifying the program, if that proves necessary. A later test in group 8.1 tests that the implementation fulfills the requirements for array names.

Default initialization of variables is one of the most important aspects of semantics left to implementation definition. Implementations may treat this however they want to, but it must be documented, and you should check that the documentation agrees with the behavior of the program. Thus this is <u>not</u> merely an informative test; the processor must have correct documentation for its behavior in order to conform.

5.6 Numeric Constants, Variables, And Operations

5.6.1 Standard Capabilities

This group of programs introduces the use of numeric expressions, specifically those formed with the arithmetic operations (+, -, *, /, ^) provided in BASIC. The most troublesome aspect of these tests is the explicit disavowal in the standard of any criterion of accuracy for the result of the operations. Thus it becomes somewhat difficult to say at what point a processor fails to implement a given operation. We

finally decided to require exact results only for integer arithmetic, and, in the case of non-integral operands, to apply an extremely loose criterion of accuracy such that if an implementation failed to meet it, one could reasonably conclude either that the precedence rules had been violated or that the operation had not been implemented at all.

Although the standard does not mandate accuracy for expressions, it does require that individual numbers be accurate to at least six significant decimal digits. This requirement is tested by assuring that values which differ by 1 in the 6th digit actually compare in the proper order, using the IF statement. The rationale for the accuracy test is best explained with an example: suppose we write the constant "333.333" somewhere in the program. For six digits of accuracy to be maintained, it must evaluate internally to some value between 333.3325 and 333.3335, since six digits of accuracy implies an error less than 5 in the 7th place. By the same reasoning, "333.334" must evaluate between 333.3335 and 333.3345. Since the allowable ranges do not overlap, the standard requires that 333.333 compare as strictly less than 333.334. Of course this same reasoning would apply to any two numbers which differed by 1 in the sixth digit.

The accuracy test not only assures that these minimal requirements are met, but also attempts to measure how much accuracy the implementation actually provides. It does this both by comparing some numbers in the manner described above for 7, 8, and 9 decimal digits, and also by using an algorithm to compute any reasonable internal accuracy. Since such an algorithm is highly sensitive to the peculiarities of the system's implementation of arithmetic, this last test is informative only.

5.6.2 Exceptions

The standard specifies a variety of exceptions for numeric expressions. All the mandatory non-fatal exceptions occur when machine infinity is exceeded and they all call for the implementation to supply machine infinity as the result and continue execution. The tests ensure that machine infinity is at least as great as the guaranteed minimum of 1E38, but since machine infinity is implementation-defined, you must assure that the value actually supplied is accurately documented.

It is worth repeating here the general guidance that the timing of exception reports is not specified by the standard. The wording is intentionally imprecise to allow implementations to anticipate exceptions, if they desire. Such anticipation may well occur for overflow and underflow of numeric constants; that is, an implementation may issue the exception report before execution of the program begins. Note that the recovery procedure, substitution of machine infinity for overflow, remains in effect.

Underflow, whether for expressions or constants, is only recommended as an exception, but, in any case, zero must be supplied when the magnitude of the result is below the minimum representable by the implementation. Note that this is required in the semantics sections (7.4 and 5.4) of the standard, not the exception sections (7.5 and 5.5).

5.6.3 Errors

These programs try out the effect of various constructions which represent either common programming errors (missing parentheses) or common enhancements (** as the involution operator) or a blend of the two (adjacent operators). No special interpretation rules apply to these tests beyond those normally associated with error programs.

5.6.4 Accuracy Tests - Informative

Although the standard mandates no particular accuracy for expression evaluation, such accuracy is nonetheless an important measure of the quality of language implementation, and is of interest to a large proportion of language users. Accordingly, these tests apply a criterion of accuracy for the arithmetic operations which is suggested by the standard's requirement that individual numeric values be represented accurate to six significant decimal digits. Note, however, that these tests are informative, not only because there is no strict accuracy requirement, but also because there is no generally valid way for a computer to measure precisely the accuracy of its own operations. Such a measurement involves calculations which must use the very facilities being measured.

The criterion for passing or failing is based on the concept that an implementation should be at least as accurate as a reasonable hypothetical implementation which uses the least accurate numeric representation allowed by the standard. It is best explained by first considering accuracy for functions of a single variable, and then generalizing to operations, which may be thought of as functions of two variables. Given an internal precision of at least d decimal digits, we simply require that the computed value for f(x) (hereinafter denoted by "cf(x)") be some value actually taken on by the function within the domain [x-e,x+e], where

```
e = 10 ^ (int (log10 ( abs (x))) + 1 - d)
```

For example, suppose we want to test the value returned by sin(29.1234) and we specify that d=6. Then:

```
e = 10 ^ (int (log10 (29.1234)) + 1 - 6)
  = 10 ^ (int (1.464) - 5)
  = 1E-4
```

and so we require that csin(x) equal some value taken on by sin(x) in the interval [29.1233, 29.1235]. This then reduces to the test that -.7507297957 <= csin(29.1234) <= -.7505976588.

The motivation for the formula for e is as follows. According to the rule for accuracy of numbers, the internal representation of the argument must lie within [x - e/2, x + e/2]. Now suppose that the internal representation is near an endpoint of the legal interval, and that the granularity of the machine (i.e., the difference between adjacent internal numeric representations) in that region of the real number line is near e (which would be the coarsest allowed, given accuracy of d digits). Given this worst case, we would still want a value returned for which the actual argument was closer to that internal representation than to immediately adjacent representations. This means that we allow for a variation of e/2 when the argument is converted from source to internal form, and another variation of e/2 around the internal representation itself. The maximum allowable variation along the x-axis is then simply the sum of the worst-case variations: e/2 + e/2 = e. This is reasonable if we think of a given internal form as representing not only a point on the real number line, but the set of points for which there is no closer internal form. Then, all we know is that the source argument is somewhere within that set and all we require is that the computed value of the function be true for some (probably different) argument within the set. For accuracy d, the maximum width of the set is of course e.

It should be noted that the first allowed variation of e/2 is inherent in the process of decimal (source) to, e.g., binary (internal) conversion. The case for allowing a variation of e/2 around the internal representation itself is somewhat weaker. If one insists on exact results within the internal numerical manipulation, then the function would be allowed to vary only within the domain [x - e/2, x + e/2], but we did not require this in the tests.

Note that the above scheme not only allows for the discrete nature of the machine, but also for numeric instability in the function itself. Mathematically, if the value of an argument is known to six places, it does not follow that the value of the function is known to six places; the error may be considerably more or less. For example, a function is often very stable near where its graph crosses the y-axis, but not the x-axis (e.g. COS (1E-22)) and very unstable where it crosses the x-axis but not the y-axis (e.g. SIN (21.99)). By allowing the cf(x) to take on any value in the specified domain, we impose strict accuracy where it can be achieved, and permit low accuracy where appropriate. Thus, the pass/fail criterion is independent of both the argument and function; it reflects only how well the implementation computed, relative to a worst-case six-digit machine.

Finally, we must recognize that even if the value of a function is computable to high accuracy (as with COS (1E-22)), the graininess of the machine will again limit how accurately the result itself can be represented. For this reason, there is an additional allowance of e/2 around the result. This implies that even if the result is computable to, say, 20 digits, we never require more than 6 digits of accuracy.

Now all the preceding comments generalize quite naturally to functions of many variables. We can then be guided in our treatment of the arithmetic operations by the above remarks on functions, if we recall that the operations may be thought of as functions of two variables, namely their operands. If we think of, say, subtraction as such a function (i.e. subtract (x,y) = x-y), then the same considerations of argument accuracy and mathematical stability pertain. Thus, we allow both operands to vary within their intervals, and simply require the result of the operation to be within the extreme values so generated. Note that such a technique would be necessary for any of the usual functions which take two variables, such as some versions of arctan.

It should be stressed that the resulting accuracy tests represent only a very minimal requirement. The design goal was to permit even the grainiest machine allowed by the standard to pass the tests; all conforming implementations, then, are inherently capable of passing. Many users will wish to impose more stringent criteria. For example, those interested in high accuracy, or implementors whose machines carry more than six digits, should examine closely the computed value and true value to see if the accuracy is what they expect.

5.7 FOR-NEXT

The ANSI standard provides a loop capability, along with an associated control-variable, through the use of the FOR statement. The semantic requirements for this construction are particularly well-defined. Specifically, the effect of the FOR is described in terms of more primitive language features (IF, GOTO, LET, and REM), which are themselves not very vulnerable to misinterpretation. The tests accordingly are quite specific and extensive in the behavior they require. The standard tests are completely self-checking, since conformance depends only on the value of the control-variable and number of times through the loop. The general design plan was not only to determine passing or failing, but also to display information allowing the user to examine the progress of execution. This should help you diagnose any problems. Note especially the requirement that the control variable, upon exit from the loop, should have the first unused, not the last used, value.

The FOR statement has no associated exceptions, but it does have a rich variety of errors, many of them context sensitive, and therefore somewhat harder for an implementation to detect. As always, if any error programs are accepted, the documentation must specify what meaning the implementation assigns to them.

5.8 Arrays

5.8.1 Standard Capabilities

The standard provides for storing numeric values in one- or two-dimensional arrays. The tests for standard capabilities are all self-checking and quite straightforward in exercising some feature defined in the standard. Note the requirement that subscript values be rounded to integers; the program testing this must not cause an exception or the processor fails.

5.8.2 Exceptions

The exception tests ensure that the subscript out of range condition is handled properly. Note that it is here that the real semantic meaning of OPTION and DIM are exercised; they have little effect other than to cause or prevent the subscript exception for certain subscript values. Since this is a fatal exception, you must check (since the program cannot) that the programs terminate at the right time, as indicated in their messages.

5.8.3 Errors

As with the FOR statement, there are a considerable number of syntactic restrictions. The thrust of these restrictions is to assure that OPTION precedes DIM, that DIM precedes references to the arrays that it governs, and that declared subscript bounds are compatible.

Three of the error programs call for INPUT from the user. This is to help you diagnose the actual behavior of the implementation if it accepts the programs. The first of these, #73, lets you try to reference an array with a subscript of 0 or 1 when OPTION BASE 1 and DIM A(0) have been specified, to see when an exception occurs.

The second, #81, allows you to try a subscript of 0 or 1 for an array whose DIM statement precedes the OPTION statement.

The third program using INPUT, #84, is a bit more complex and has to do with double dimensioning. If there are two DIM statements for the same array, the implementation has a choice of

several plausible interpretations. We have noted five such possibilities and have attempted to distinguish which, if any, seems to apply. Since the only semantic effect of DIM is to cause or prevent an exception for a given array reference, however, it is necessary to run the program three times to see when exceptions occur and when they don't, assuming the processor hasn't simply rejected the program outright. Your input-reply simply tells the program which of the three executions it is currently performing. For each execution, you must note whether an exception occurred or not and then match the results against the table in the program. Suppose, for instance, that you get an exception the first time but not the second or third. That would be incompatible with all five interpretations except number 4, which is that the first DIM statement executed sets the size of the array and it is never changed thereafter. As usual, check the documentation to make sure it correctly describes what happens.

5.9 Control Statements

This group fully exploits the properties of some of the control facilities which were tested in a simpler way in group 4. As before, there seemed no good way to provide diagnostics for failure of standard tests, since the behavior of a failing processor is impossible to predict. Passing implementations will cause the "*** TEST PASSED ***" message to appear, but certain kinds of failures might cause the programs to abort, without producing a failure message. Check Volume 2 for an example of correct output.

5.9.1 GOSUB And RETURN

Most of the tests in this group are self-explanatory, but the one checking address stacking deserves some comment. The standard describes the effect of issuing GOSUBs and RETURNs in terms of a stack of return addresses, for which the GOSUB adds a new address to the top, and the RETURN uses the most recently added address. Thus, we get a kind of primitive recursion in the control structure (although without any stacking of data). Note that this description allows complete freedom in the placement of GOSUBs and RETURNs in the source code. There is no static association of any RETURN with any GOSUB. The test which verifies this specification computes binomial coefficients, using the usual recursive formula. The logic of the program is a bit convoluted, but intentionally so, in order to exercise the stacking mechanism vigorously.

5.9.2 ON-GOTO

The ON-GOTO tests are all readily understandable. The one thing you might want to watch for is that the processor rounds the expression controlling the ON-GOTO to the nearest integer, as specified in the standard. Thus, "ON .6 GOTO", "ON 1 GOTO", and "ON 1.4 GOTO" should all have the same effect; there should be no out of range exception for values between .5 and 1.

5.10 READ, DATA, And RESTORE

This group tests the facilities for establishing a stream of data in the program and accessing it sequentially. This feature has some subtle requirements, and it would be wise to read the standard especially carefully so that you understand the purpose of the tests.

5.10.1 Standard Capabilities

All but the last of these tests are reasonably simple. The last test dealing with the general properties of READ and DATA, although self-checking, has somewhat complex internal logic. It assures that the range of operands of READ and DATA can overlap freely and that a given datum can be read as numeric at one time and as a string at a later time. If you need to examine the internal logic closely, be sure to use the REM statements at the beginning which break down the structure of the READ and DATA lists for you.

5.10.2 Exceptions

The exceptions can be understood directly from the programs. Note that string overflow may or may not occur, depending on the implementation-defined maximum string length. If overflow (loss of data) does occur, the processor must report an exception and execution must terminate. If there is no exception report, look carefully at the output to assure that no loss of data has occurred.

5.10.3 Errors

All of the error tests display results if the implementation accepts them, allowing you to check that the documentation matches the actual behavior of the processor. Some of the illegal constructs are likely candidates for enhancements and thus the diagnostic feature is important here.

5.11 INPUT

This group, like that for PRINT, calls for a good deal of user participation. This participation takes the form, not only of interpreting program output but also of supplying appropriate INPUT replies. The validity of this group depends strongly on the entry of correct replies.

5.11.1 Standard Capabilities

The first program assures that the processor can accept as input any syntactically valid number. It is absolutely essential, then, that you reply to each message with precisely the same set of characters that it asks for. If it tells you to enter "1.E22" you must not reply with, e.g. "1E22". This would defeat one of the purposes of that reply, which is to see whether the processor correctly handles a decimal point immediately before the exponent. Once you have correctly entered the reply, one of several things can happen. If the processor in some way rejects the reply, for instance by producing a message that it is not a valid number, then the processor has failed the test since all the replies are in fact valid according to the standard. To get by this problem, simply enter any number not numerically equal to the one originally requested. This will let you get on to the other items, and will signal a failure to the processor as described below.

If the processor accepts the reply, the program then tests that six digits of accuracy have been preserved. If so, you will get a message that the test is OK, and may go on to the next reply. If not, you will get a message indicating that the correct value was not received, and the program will ask if you want to retry that item. If you simply mistyped the original input-reply, you should enter the code for a retry. If your original reply was correct, but the processor misinterpreted the numeric value, there is no point to retrying; just go ahead to the next item. The program will count up all the failures and report the total at the end of the program.

The next program, for array input, assures that you can enter numbers into an array, and that assignments are done left to right, so that a statement such as "INPUT I, A(I)" allows you to control which element of the array gets the value. Also, it is here (and only here) that the standard's requirement for checking the input-reply before assignment is tested. Your first reply to this section of the test must cause an exception, and you must be allowed to re-enter the entire reply, otherwise the test fails. The rest of the program is self-checking.

The program for string input comes next and, as with the numeric input program two considerations are paramount: 1) you should enter your replies exactly as indicated in the message and 2) all input replies are syntactically valid and therefore if the

implementation rejects any of them, it fails the test. A potentially troublesome aspect of this program is that the prompting message cannot always look exactly like your reply. In particular, your replies will sometimes include blanks and quotes. It is impossible to PRINT the quote character in Minimal BASIC, so the number-sign (#) is used instead. For ease of counting characters, an equals (=) is used in the message to represent blanks. Therefore, when you see the number-sign, type the quote and when you see the equals, type the blank. If you forget, the item will fail and you will have a chance to retry, so make sure that a reported failure really is a processor failure and not just your own mistyping before bypassing the retry. As with the numeric input, if the processor rejects one of the replies, simply enter any reply whose evaluation is not equal to that of the prompting message to bypass the item and force a failure. The second section of the string input program does not use the substitute characters in the message; rather you always type exactly what you see in the message surrounded by quotes.

The program for mixed input follows the conventions individually established by the numeric and string input programs. Its purpose is simply to assure that the implementation can handle both string and numeric data in the same reply.

5.11.2 Exceptions

Unlike the other groups, where each exception type is tested with its own program, all the mandatory exceptions for INPUT are gathered into one routine. There are two reasons for this: first, there are so many variations worth trying that a separate program for each would be impractical, and second, the recovery procedures are the same for all input exception types. It is, then, both economical and convenient to group together all the various possibilities into one program. Underflow on INPUT is an optional exception and has a different recovery procedure, governed by the semantics for numeric constants rather than INPUT. It, therefore, is tested in its own separate program.

The conformance requirements for input exceptions are perhaps the most complex of any in the standard. It is worthwhile to review these requirements in some detail, and then relate them to the test. The standard says that "unquoted-strings that are numeric-constants must be supplied as input for numeric-variables, and either quoted-strings or unquoted-strings must be supplied as input for string-variables." Since the syntactic entities mentioned are well-defined in the standard, this specification seems clear enough. Recall, however, that processors can, in general, enlarge the class of syntactic objects which they accept. In particular, a processor may have an enhanced definition of quoted-string, unquoted-string, numeric-constant, or, more generally,

input-reply, and therefore accept a reply not strictly allowed by the standard, just as standard implementations may accept, and render meaningful, non-standard programs. The result is that the conditions for an input exception may depend on implementation-defined features, and thus a given input-reply may cause an exception for one processor and yet not another. Note that the same situation prevails for overflow - the exception depends on the implementation-defined maximum string length and machine infinity. Thus, "LET A = 1E37 * 100" may cause overflow on one standard processor, but not another.

When running the program then, a given input-reply need not generate an exception if there is a documented enhancement which describes its interpretation. Of course, such an enhancement must not change the meaning of any input-reply which is syntactically standard. Note that, of the replies called for in the program, some are syntactically standard and some are not; they should, however, all cause exceptions on a truly minimal BASIC processor, i.e. one with no syntactic enhancements, with machine infinity = 1E38 and with maximum string length of 18.

Another problem is that, for some replies, it is not clear which exception type applies. If, for instance, you respond to "INPUT A,B,C" with: "2,,3", it may be taken as a wrong type, since a numeric-constant was not supplied for B, or as insufficient data, since only two, not three, were supplied. In such a case, as with all exception reports, it is sufficient if the report is a reasonably accurate description of what went wrong, regardless of precisely how the report corresponds to the types defined in the standard.

As with all non-fatal exceptions, it is permitted for an implementation to treat a given INPUT exception as fatal, if the hardware or operating environment makes the recovery procedure impossible to follow. The program is set up with table-driven logic, so that each exception is triggered by a set of values in a given DATA statement. If you need to separate out some of the cases because they cause the program to terminate, simply delete the DATA statements for those cases. REM statements in the program describe the format of the data.

After that lengthy preliminary discussion, we are now ready to consider how to operate and interpret the test. The program will ask you for a reply, and also show you the INPUT statement to which it is directed, to help you understand why the exception should occur. Enter the exception-provoking reply, exactly as requested by the message. If all goes well, the implementation will give you an exception report and allow you to re-supply the entire input-reply. On this second try, simply enter all zeros, exactly as many as needed by the complete original INPUT statement, to bypass that case - this will signal the program that that case has passed, and you will then receive the next message.

Now, let us look at what might go wrong. If the implementation simply accepts the initial input-reply, the program will display the resulting values assigned to the variables and signal a possible failure. If the documentation for the processor describes an enhancement which agrees with the actual result, then that case passes; otherwise it is a failure.

Suppose the implementation reports an exception, but does not allow you to re-supply the entire input-reply. At that point, just do whatever the processor requires to bypass that case. You should supply non-zero input to signal the program that the case in question has failed.

When the program detects an apparent failure (non-zeros in the variables) it allows you to retry the whole case. As before, if you mistyped you should reply that you wish to retry; if the processor simply mishandled the exception, reject the retry and move on to the next case.

Figure 3 outlines the user's operating and interpretation procedure for the INPUT exception test.

5.11.3 Errors

There is only one error program and it tests the effect of a null entry in the input-list. The usual rules for error tests apply (see section 4.4.2.3).

Instructions for the INPUT exceptions test

```
Inspect message from program
Supply exact copy of message as input-reply
If processor reports exception
then
   if processor allows you to re-supply entire reply
   then
      enter all zeros (exactly enough to satisfy original
            INPUT request)
      if processor responds that test passed
      then
         test passed
      else (no pass message after entering zeros)
         zeros not assigned to variables
         test failed (recovery procedure not followed)
      endif
   else (not allowed to re-supply entire reply)
      supply any non-zero reply to bypass this case
      test failed (recovery procedure not followed)
   endif
else (no exception report)
   if documentation for processor correctly describes syntactic
      enhancement to accept the reply
   then
      test passed
   else (no exception and incorrect/missing documentation)
      test failed
   endif
endif
```

Figure 3

5.12 Implementation-supplied Functions

All conforming implementations must make available to the programmer the set of functions defined in section 8 of the ANSI standard. The purpose of this group is to assure that these functions have actually been implemented and also to measure at least roughly the quality of implementation.

5.12.1 Precise Functions: ABS,INT,SGN

These three functions are distinguished among the eleven supplied functions in that any reasonable implementation should return a precise value for them. Therefore they can be tested in a more stringent manner than the other eight which are inherently approximate (i.e. a discrete machine cannot possibly supply an exact answer for most arguments).

The structure of the tests is simple: the function under test is invoked with a variety of argument values and the returned value is compared to the correct result. If all results are equal, the test passes, otherwise it fails. The values are displayed for your inspection and the tests are self-checking. The test for the INT function has a second section which does an informative test on the values returned for large arguments requiring more than six digits of accuracy.

5.12.2 Approximated Functions: SQR,ATN,COS,EXP,LOG,SIN,TAN

These functions do not typically return rational values for rational arguments and thus may only be approximated by digital computers. Furthermore, the standard explicitly disavows any criterion of accuracy, making it difficult to say when an implementation has definitely failed a test. Because of these constraints, the non-exception tests in this group are informative only. We can, however, quite easily apply the ideas developed earlier in section 5.6.4. As explained there, we can devise an accuracy criterion for the implementation of a function, based on a hypothetical six decimal digit machine. If a function returns a value less accurate even than that of which this worst-case machine is capable, the informative test fails.

To repeat the earlier guidance for the numeric operations: this approach imposes only a very minimal requirement. You may well want to set a stricter standard for the implementation under test. For this reason, the programs in this group also compute and report an error measure, which gives an estimate of the degree of accuracy achieved, again relative to a six-digit machine. The error measure thus goes beyond a simple pass/fail report and quantifies how well or poorly the function value was computed. Of course, the error measure itself is subject to inaccuracy in its own internal computation, and no one

measurement should be taken as precisely correct. Nonetheless, when the error measures of all the cases are considered in the aggregate, it should give a good overall picture of the quality of function evaluation. Since it is based on the same allowed interval for values as the pass/fail criterion, it too measures the quality of function evaluation independent of the function and argument under test. It does depend on the internal accuracy with which the implementation can represent numeric quantities: the greater the accuracy, the smaller the error measure should become. As a rough guide, the error measures should all be < 10^(6-d), where d is the number of significant decimal digits supported by the implementation (this is determined in the standard tests for numeric operations, group 6.1). For instance, an eight decimal digit processor should have all error measures < .01.

Another point to be stressed: even though the results of these tests are informative, the tests themselves are syntactically standard, and thus must be accepted and processed by the implementation. If, for instance, the processor does not recognize the ATN function and rejects the program, it definitely fails to conform to the standard. This is in contrast to the case of a processor which accepts the program, but returns somewhat inaccurate values. The latter processor is arguably standard-conforming, even if of low quality.

This group also contains exception tests for those conditions so specified in the ANSI standard. Most of these can be understood in light of the general guidance given for exceptions. The program for overflow of the TAN function deserves some comment. Since it is questionable whether overflow can be forced simply by encoding pi/2 as a numeric constant for the source code argument, the program attempts to generate the exception by a convergence algorithm. It may be, however, that no argument exists which will cause overflow, so you must verify merely that if overflow occurs, then it is reported as an exception. For instance, if several of the function calls return machine infinity, it is clear that overflow has occurred and if there were no exception report in such a case, the test fails. Also, as a measure of quality, the returned values with a given sign should increase in magnitude until overflow occurs, i.e. all the positive values should form an ascending sequence, and the negative values a descending sequence.

5.12.3 RND And RANDOMIZE

Unlike the other functions, there is no single correct value to be returned by any individual reference to RND, but only the properties of an aggregation of returned values are specified. The standard says that these values are "uniformly distributed in the range 0 <= RND < 1". Also, section 17 specifies that in the absence of the RANDOMIZE statement, RND will generate the same pseudorandom sequence for each execution of a program;

conversely, each execution of RANDOMIZE "generates a new unpredictable starting point" for the sequence produced by RND. The RND tests follow closely the strategy put forth in chapter 3.3.1 of Knuth's The Art of Computer Programming [4], which explains fully the rationale for the programs in this group.

5.12.3.1 Standard Capabilities

The first two programs test that the same sequence or a novel sequence appear as appropriate, depending on whether RANDOMIZE has executed. Note that you must execute both of these programs three times apiece, since the RND sequence is initialized by the implementation only when execution begins. The next three programs all test properties of the sequence which follow directly from the specification that it is uniformly distributed in the range 0 <= RND < 1. If the results make it quite improbable that the distribution is uniform, or if any value returned is outside the legal range, then the test fails. Of course, any implementation could pass simply by adjusting the RND algorithm or starting point until a passing sequence is generated. In order to measure the quality of implementation, you can run the programs with a RANDOMIZE statement in the beginning and then observe how often the test passes or fails. Note that, if you use RANDOMIZE, these programs should fail a certain proportion of the time since they are probabilistic tests.

5.12.3.2 Informative Tests

There are several desirable properties of a sequence of pseudorandom numbers which are not strictly implied by uniform distribution. If, for instance, the numbers in the sequence alternated between being <= .5 and > .5, they might still be uniform, but would be non-random in an important way. These tests attempt to measure how well the implementation has approached the ideal of a perfectly random sequence by looking for patterns indicative of nonrandomness in the sequence actually produced. Like the tests for standard capabilities, these programs are probabilistic and any one of them may fail without necessarily implying that the RND sequence is not random. If a high quality RND function is important for your purposes, we suggest you run each of these programs several times with the RANDOMIZE statement. If a given test seems to fail far more often than likely, it may well indicate a weakness in the RND algorithm.

5.12.4 Errors

The tests in this group all use an argument-list which is incorrect in some way, either for the particular function, or because of the general rules of syntax. As always, if the processor does accept any of them, the documentation must be consistent with the actual results. Note that the ANSI standard contains a misprint, indicating that the TAN function takes no arguments. The tests are written to treat TAN as a function of a single variable.

5.13 User-defined Functions

The standard provides a facility so that programmers can define functions of a single variable in the form of a numeric expression. This group of tests exercises both the invoking mechanism (function references) and the defining mechanism (DEF statement).

5.13.1 Standard Capabilities

These programs test a variety of properties guaranteed by the standard: the DEF statement must allow any numeric expression as the function definition; the parameter, if any, must not be confused with a global variable of the same name; global variables, other than one with the same name as the parameter, are available to the function definition; a DEF statement in the path of execution has no effect; invocation of a function as such never changes the value of any variable; the set of valid names for user-defined functions is "FN" followed by any alphabetic character. The tests are self-checking. As with the numeric operations, a very loose criterion of accuracy is used to check the implementation. Its purpose is not to check accuracy as such, but only to assure that the semantic behavior accords with the standard.

5.13.2 Errors

Many of these tests are similar to the error tests for implementation-supplied functions, in that they try out various malformed argument lists. There are also some tests involving the DEF statement, in particular for the requirements that a program contain exactly one DEF statement for each user function referred to in the program and that the definition precede any references.

5.14 Numeric Expressions

Numeric expressions have a somewhat special place in the Minimal BASIC standard. They are the most complex entity, syntactically, for two reasons. First, the expression itself may be built up in a variety of ways. Numeric constants, variables, and function references are combined using any of five operations. The function references themselves may be to user-defined expressions. And of course expressions can be nested, either implicitly, or explicitly with parentheses. Second, not only do the expressions have a complex internal syntax, but also they may appear in a number of quite different contexts. Not just the LET statement, but also the IF, PRINT, ON...GOTO, and FOR statements, can contain expressions. Also they may be used as array subscripts or as arguments in a function reference. Note that when they are used in the ON...GOTO, as subscripts, or as arguments to TAB, expressions must be rounded to the nearest integer.

The overall strategy of the test system is first to assure that the elements of numeric expressions are handled correctly, then to try out increasingly complex expressions in the comparatively simple context of the LET statement, and finally to verify that these complex expressions work properly in the other contexts mentioned. Preceding groups have already accomplished the first task of checking out individual expression elements, such as constants, variables (both simple and array), and function references. This group completes the latter two steps.

5.14.1 Standard Capabilities In Context Of LET-statement

This test tries out various lengthy expressions, using the full generality allowed by the standard, and assigns the resulting value to a variable. As usual, if this value is even approximately correct, the test passes, since we are interested in semantics rather than accuracy. The program displays the correct value and actual computed value. This test also verifies that subscript expressions evaluate to the nearest integer.

5.14.2 Expressions In Other Contexts: PRINT, IF, ON-GOTO, FOR

Please note that the PRINT test, like other PRINT tests, is inherently incapable of checking itself, and therefore you must inspect and interpret the results. The PRINT program first tests the use of expressions as print-items. Check that the actual and correct values are reasonably close. The second section of the program tests that the TAB call is handled correctly. Simply verify that the characters appear in the appropriate columns.

The second program is self-checking and tests IF, ON-GOTO and FOR, one in each section. As with other tests of control statements, the diagnostics are rather sparse for failures. Check Volume 2 for an example of correct output.

5.14.3 Exceptions In Subscripts And Arguments

The exceptions specified in section 7 and 8 apply to numeric expressions in whatever context they occur. These tests simply assure that the correct values are supplied, e.g., machine infinity for overflow, zero for underflow, and that the execution continues normally as if that value had been put in that context as, say, a numeric constant. Sometimes this action will produce normal results and sometimes will trigger another exception, e.g., machine infinity supplied as a subscript. Simply verify that the exception reports are produced as specified in the individual tests.

5.14.4 Exceptions In Other Contexts: PRINT, IF, ON-GOTO, FOR

As in the immediately preceding section, these tests make sure that the recovery procedures have the natural effect given the context in which they occur. As usual for exception tests, it is up to you to verify that reasonable exception reports appear. The PRINT tests also require user interpretation to some degree.

5.15 Miscellaneous Checks

This group consists mostly of error tests in which the error is tied not to some specific functional area but rather to the general format rules for BASIC programs. If you are not already thoroughly familiar with the general criteria for error tests, it would be wise to review them (sections 3.2.2 and 4.4.2.3 of this document) before going through this group. A few tests require special comment and this is supplied below in the appropriate subsection.

5.15.1 Missing Keyword

Many implementations of BASIC allow programs to omit the keyword LET in assignment statements. This program checks that possibility and reports the resulting behavior if accepted.

5.15.2 Spaces

Sections 3 and 4 of the ANSI standard specify several context sensitive rules for the occurrence of spaces in a BASIC program. The standard test assures that wherever one space may occur, several spaces may occur with no effect, except within a quoted- or unquoted-string. There are certain places where spaces either must, or may, or may not appear, and the error programs test how the implementation treats various violations of the rules.

5.15.3 Quotes

These programs test the effect of using either a single or double quote in a quoted string. Some processors may interpret the double quote as a single occurrence of the quote character within the string. The programs test the effect of aberrant quotes in the context of the PRINT and the LET statements.

5.15.4 Line Numbers

The first of these programs is a standard, not an error, test. It verifies that leading zeros in line numbers have no effect. The other programs all deal with some violation of the syntax rules for line numbers. When submitting these programs to your implementation, you should not explicitly call for any sorting or renumbering of lines. If the implementation sorts the lines by default, even when the program is submitted to it in the simplest way, the documentation must make this clear. Such sorting merely constitutes a particular type of syntactic enhancement, i.e., to treat a program with lines out of order as if they were in order. Similarly, an implementation may discard duplicate lines, or append line numbers to the beginning of lines missing them, as long as these actions occur without special user intervention and are documented. Of course, processors may also reject such programs, with an error message to the user.

5.15.5 Line Longer Than 72 Characters

This program tests the implementation's reaction to a line whose length is greater than the standard limit of 72. Many implementations accept longer lines; if so the documentation must specify the limit.

5.15.6 Margin Overflow For Output Line

This is not an error test, but a standard one. Further, it involves PRINT capabilities and therefore calls for careful user interpretation. Its purpose is to assure correct handling of the margin and print zones, relative to the implementation-defined length for each of those two entities. After you have entered the appropriate values, the program will generate pairs of output, with either one or two printed lines for each member of the pair. The first member is produced using primitive capabilities of PRINT and is intended to show what the output should look like. The second member of the pair is produced using the facilities under test and shows what the output actually looks like. If the two members differ at all, the test fails. It could happen, however, that the first member of the pair does not produce the correct output either. You should, therefore, closely examine the sample output for this test in Volume 2 to understand what the expected output is. Of course the sample is exactly correct only for implementations with the same margin and zone width, but allowing for the possibly different widths of your processor, the sample should give you the idea of what your processor must do.

5.15.7 Lowercase Characters

These two tests tell you whether your processor can handle lowercase characters in the program, and, if so, whether they are converted to uppercase or left as lowercase.

5.15.8 Ordering Strings

This program tests whether your implementation accepts comparison operators other than the standard = or <> for strings. If the processor does accept them, the program assumes that the interpretation is the intuitively appealing one and prints informative output concerning the implicit character collating sequence and also some comparison results for multi-character strings.

5.15.9 Mismatch Of Types In Assignment

These programs check whether the processor accepts assignment of a string to a numeric variable and vice-versa, and if so what the resulting value of the receiving variable is. As usual, make sure your documentation covers these cases if the implementation accepts these programs.

6 TABLES OF SUMMARY INFORMATION ABOUT THE TEST PROGRAMS

This section contains three tables which should help you find your way around the programs and the ANSI standard. The first table presents the functional grouping of the tests and shows which programs are in each group and the sections of the ANSI standard whose specifications are being tested thereby. The second table lists all the programs individually by number and title, and also the particular sections and subsections of the standard to which they apply. The third table lists the sections and subsections of the standard in order, followed by a list of program numbers for those sections. This third table is especially important if you want to test the implementation of only certain parts of the standard. Be aware, however, that since the sections of the standard are not tested in order, the tests for a given section may rely on the implementation of later sections in the standard which have been tested earlier in the test sequence.

6.1 Group Structure Of The Minimal BASIC Test Programs

Group	Program Number	ANSI Section
1 Simple PRINTing of string constants	1	(3,5,12)
2 END and STOP	2-5	(4,10)
2.1 END	2-4	(4)
2.2 STOP	5	(10)
3 PRINTing and simple assignment (LET)	6-14	(5,6,9,12)
3.1 string variables and TAB	6-8	(6,9,12)
3.2 numeric constants and variables	9-14	(5,6,9,12)
4 Control Statements and REM	15-21	(10,18)
4.1 REM and GOTO	15-16	(10,18)
4.2 GOSUB and RETURN	17	(10)
4.3 IF-THEN	18-21	(10)
5 Variables	22-23	(6)
6 Numeric Constants, Variables, and Operations	24-43	(5,6,7)
6.1 Standard Capabilities	24-27	(5,6,7)
6.2 Exceptions	28-35	(5,7)
6.3 Errors	36-38	(7)
6.4 Accuracy tests - Informative	39-43	(7)
7 FOR-NEXT	44-55	(10,11)
7.1 Standard Capabilities	44-49	(10,11)
7.2 Errors	50-55	(11)
8 Arrays	56-84	(6,7,9,15)
8.1 Standard Capabilities	56-62	(6,7,9,15)
8.2 Exceptions	63-72	(6,15)
8.3 Errors	73-84	(6,15)

Group Structure of the Minimal BASIC Test Programs (cont.)

Group	Program Number	ANSI Section
9 Control Statements	85-91	(10)
9.1 GOSUB and RETURN	85-87	(10)
9.2 ON-GOTO	88-91	(10)
10 READ, DATA, and RESTORE	92-106	(3,5,14)
10.1 Standard Capabilities	92-95	(3,5,14)
10.2 Exceptions	96-101	(14)
10.3 Errors	102-106	(3,14)
11 INPUT	107-113	(3,5,13)
11.1 Standard Capabilities	107-110	(3,5,13)
11.2 Exceptions	111-112	(3,5,13)
11.3 Errors	113	(13)
12 Implementation-supplied Functions	114-150	(7,8,17)
12.1 Precise functions: ABS, INT, SGN	114-116	(8)
12.2 Approximated functions: SQR, ATN, COS, EXP, LOG, SIN, TAN	117-129	(7,8)
12.3 RND and RANDOMIZE	130-142	(8,17)
12.3.1 Standard Capabilities	130-134	(8,17)
12.3.2 Informative tests	135-142	(8)
12.4 Errors	143-150	(7,8)
13 User-defined Functions	151-163	(7,16)
13.1 Standard Capabilities	151-152	(7,16)
13.2 Errors	153-163	(7,16)

Group Structure of the Minimal BASIC Test Programs (cont.)

Group		Program Number	ANSI Section
14	Numeric Expressions	164-184	(6,7,8,10, 11,12,16)
14.1	Standard Capabilities in context of LET-statement	164	(6,7,8,16)
14.2	Expressions in other contexts: PRINT, IF, ON-GOTO, FOR	165-166	(7,10,11,12)
14.3	Exceptions in subscripts and arguments	167-171	(6,7,8,16)
14.4	Exceptions in other contexts: PRINT, IF, ON-GOTO, FOR	172-184	(7,10,11,12)
15	Miscellaneous Checks	185-208	(3,4,9,10,12)
15.1	Missing keyword	185	(9)
15.2	Spaces	186-191	(3,4)
15.3	Quotes	192-195	(3,9,12)
15.4	Line numbers	196-201	(4)
15.5	Line longer than 72 characters	202	(4)
15.6	Effect of zones and margin on PRINT	203	(12)
15.7	lowercase characters	204-205	(3,9,12)
15.8	Ordering relations between strings	206	(3,10)
15.9	Mismatch of types in assignment	207-208	(9)

6.2 Test Program Sequence

PROGRAM NUMBER 1
NULL PRINT AND PRINTING QUOTED STRINGS.
REFS: 3.2 3.4 5.2 5.4 12.2 12.4

PROGRAM NUMBER 2
THE END-STATEMENT.
REFS: 4.2 4.4

PROGRAM NUMBER 3
ERROR - MISPLACED END-STATEMENT.
REFS: 4.2 4.4

PROGRAM NUMBER 4
ERROR - MISSING END-STATEMENT.
REFS: 4.2 4.4

PROGRAM NUMBER 5
THE STOP-STATEMENT.
REFS: 10.2 10.4

PROGRAM NUMBER 6
PRINT-SEPARATORS, TABS, AND STRING VARIABLES.
REFS: 6.2 6.4 9.2 9.4 12.2 12.4

PROGRAM NUMBER 7
EXCEPTION - STRING OVERFLOW USING THE LET-STATEMENT.
REFS: 9.5 12.4

PROGRAM NUMBER 8
EXCEPTION - TAB ARGUMENT LESS THAN ONE.
REFS: 12.5

PROGRAM NUMBER 9
PRINTING NR1 AND NR2 NUMERIC CONSTANTS.
REFS: 5.2 5.4 12.4

PROGRAM NUMBER 10
PRINTING NR3 NUMERIC CONSTANTS.
REFS: 5.2 5.4 12.4

PROGRAM NUMBER 11
PRINTING NUMERIC VARIABLES ASSIGNED NR1 AND NR2 CONSTANTS.
REFS: 5.2 5.4 6.2 6.4 9.2 9.4 12.4

PROGRAM NUMBER 12
PRINTING NUMERIC VARIABLES ASSIGNED NR3 CONSTANTS.
REFS: 5.2 5.4 6.2 6.4 9.2 9.4 12.4

PROGRAM NUMBER 13
FORMAT AND ROUNDING OF PRINTED NUMERIC CONSTANTS.
REFS: 12.4 5.2 5.4

PROGRAM NUMBER 14
PRINTING AND ASSIGNING NUMERIC VALUES NEAR TO THE MAXIMUM AND MINIMUM MAGNITUDE.
REFS: 5.4 9.4 12.4

PROGRAM NUMBER 15
THE REM AND GOTO STATEMENTS.
REFS: 18.2 18.4 10.2 10.4

PROGRAM NUMBER 16
ERROR - TRANSFER TO A NON-EXISTING LINE NUMBER USING THE GOTO-STATEMENT.
REFS: 10.4

PROGRAM NUMBER 17
ELEMENTARY USE OF GOSUB AND RETURN.
REFS: 10.2 10.4

PROGRAM NUMBER 18
THE IF-THEN STATEMENT WITH STRING OPERANDS.
REFS: 10.2 10.4

PROGRAM NUMBER 19
THE IF-THEN STATEMENT WITH NUMERIC OPERANDS
REFS: 10.2 10.4

PROGRAM NUMBER 20
ERROR - IF-THEN STATEMENT WITH A STRING AND NUMERIC OPERAND.
REFS: 10.2

PROGRAM NUMBER 21
ERROR - TRANSFER TO NON-EXISTING LINE NUMBER USING THE IF-THEN-STATEMENT.
REFS: 10.4

PROGRAM NUMBER 22
NUMERIC AND STRING VARIABLE NAMES WITH THE SAME INITIAL LETTER.
REFS: 6.2 6.4

PROGRAM NUMBER 23
INITIALIZATION OF STRING AND NUMERIC VARIABLES.
REFS: 6.6

PROGRAM NUMBER 24
PLUS AND MINUS
REFS: 7.2 7.4

PROGRAM NUMBER 25
MULTIPLY, DIVIDE, AND INVOLUTE
REFS: 7.2 7.4

PROGRAM NUMBER 26
PRECEDENCE RULES FOR NUMERIC EXPRESSIONS.
REFS: 7.2 7.4

```
PROGRAM NUMBER 27
   ACCURACY OF CONSTANTS AND VARIABLES.
   REFS:   5.2  5.4  6.2  6.4 10.4

PROGRAM NUMBER 28
   EXCEPTION - DIVISION BY ZERO.
   REFS:   7.5

PROGRAM NUMBER 29
   EXCEPTION - OVERFLOW OF NUMERIC EXPRESSIONS.
   REFS:   7.5

PROGRAM NUMBER 30
   EXCEPTION - OVERFLOW OF NUMERIC CONSTANTS.
   REFS:   5.4  5.5

PROGRAM NUMBER 31
   EXCEPTION - ZERO RAISED TO A NEGATIVE POWER.
   REFS:   7.5

PROGRAM NUMBER 32
   EXCEPTION - NEGATIVE QUANTITY RAISED TO A NON-INTEGRAL POWER.
   REFS:   7.5

PROGRAM NUMBER 33
   EXCEPTION - UNDERFLOW OF NUMERIC EXPRESSIONS.
   REFS:   7.4

PROGRAM NUMBER 34
   EXCEPTION - UNDERFLOW OF NUMERIC CONSTANTS.
   REFS:   5.4  5.6

PROGRAM NUMBER 35
   EXCEPTION - OVERFLOW AND UNDERFLOW WITHIN SUB-EXPRESSIONS
   REFS:   7.4  7.5

PROGRAM NUMBER 36
   ERROR - UNMATCHED PARENTHESES IN NUMERIC EXPRESSION.
   REFS:   7.2

PROGRAM NUMBER 37
   ERROR - USE OF '**' AS OPERATOR.
   REFS:   7.2

PROGRAM NUMBER 38
   ERROR - USE OF ADJACENT OPERATORS.
   REFS:   7.2

PROGRAM NUMBER 39
   ACCURACY OF ADDITION
   REFS:   7.2  7.4  7.6

PROGRAM NUMBER 40
   ACCURACY OF SUBTRACTION
   REFS:   7.2  7.4  7.6
```

PROGRAM NUMBER 41
ACCURACY OF MULTIPLICATION
REFS: 7.2 7.4 7.6

PROGRAM NUMBER 42
ACCURACY OF DIVISION
REFS: 7.2 7.4 7.6

PROGRAM NUMBER 43
ACCURACY OF INVOLUTION
REFS: 7.2 7.4 7.6

PROGRAM NUMBER 44
ELEMENTARY USE OF THE FOR-STATEMENT.
REFS: 11.2 11.4

PROGRAM NUMBER 45
ALTERING THE CONTROL-VARIABLE WITHIN A FOR-BLOCK.
REFS: 11.2 11.4

PROGRAM NUMBER 46
INTERACTION OF CONTROL STATEMENTS WITH THE FOR-STATEMENT.
REFS: 11.2 11.4 10.2 10.4

PROGRAM NUMBER 47
INCREMENT IN THE STEP CLAUSE OF THE FOR-STATEMENT DEFAULTS TO A VALUE OF ONE.
REFS: 11.2 11.4

PROGRAM NUMBER 48
LIMIT AND INCREMENT IN THE FOR-STATEMENT ARE EVALUATED ONCE UPON ENTERING THE LOOP.
REFS: 11.2 11.4

PROGRAM NUMBER 49
NESTED FOR-BLOCKS.
REFS: 11.2 11.4

PROGRAM NUMBER 50
ERROR - FOR-STATEMENT WITHOUT A MATCHING NEXT-STATEMENT.
REFS: 11.2 11.4

PROGRAM NUMBER 51
ERROR - NEXT-STATEMENT WITHOUT A MATCHING FOR-STATEMENT.
REFS: 11.2 11.4

PROGRAM NUMBER 52
ERROR - MISMATCHED CONTROL-VARIABLES ON FOR-STATEMENT AND NEXT-STATEMENT.
REFS: 11.4

PROGRAM NUMBER 53
ERROR - INTERLEAVED FOR-BLOCKS.
REFS: 11.4

PROGRAM NUMBER 54
ERROR - NESTED FOR-BLOCKS WITH THE SAME CONTROL VARIABLE.
REFS: 11.4

PROGRAM NUMBER 55
ERROR - JUMP INTO FOR-BLOCK.
REFS: 11.4

PROGRAM NUMBER 56
ARRAY ASSIGNMENT WITHOUT THE OPTION-STATEMENT.
REFS: 6.2 6.4 9.2 9.4 15.2 15.4

PROGRAM NUMBER 57
ARRAY ASSIGNMENT WITH OPTION BASE 0.
REFS: 6.2 6.4 9.2 9.4 15.2 15.4

PROGRAM NUMBER 58
ARRAY ASSIGNMENT WITH OPTION BASE 1.
REFS: 6.2 6.4 9.2 9.4 15.2 15.4

PROGRAM NUMBER 59
ARRAY NAMED 'A' IS DISTINCT FROM 'A$'.
REFS: 6.2 6.4

PROGRAM NUMBER 60
NUMERIC CONSTANTS USED AS SUBSCRIPTS ARE ROUNDED TO NEAREST INTEGER.
REFS: 6.4 5.4

PROGRAM NUMBER 61
NUMERIC EXPRESSIONS CONTAINING SUBSCRIPTED VARIABLES.
REFS: 6.2 6.4 7.2 7.4

PROGRAM NUMBER 62
GENERAL SYNTACTIC AND SEMANTIC PROPERTIES OF ARRAY CONTROL STATEMENTS: OPTION AND DIM.
REFS: 15.2 15.4

PROGRAM NUMBER 63
EXCEPTION - SUBSCRIPT TOO LARGE FOR ONE-DIMENSIONAL ARRAY.
REFS: 6.5

PROGRAM NUMBER 64
EXCEPTION - SUBSCRIPT TOO SMALL FOR TWO-DIMENSIONAL ARRAY.
REFS: 6.5

PROGRAM NUMBER 65
EXCEPTION - SUBSCRIPT TOO SMALL FOR ONE-DIMENSIONAL ARRAY, WITH DIM.
REFS: 6.5 15.2 15.4

PROGRAM NUMBER 66
EXCEPTION - SUBSCRIPT TOO LARGE FOR TWO-DIMENSIONAL ARRAY, WITH DIM.
REFS: 6.5 15.2 15.4

PROGRAM NUMBER 67
EXCEPTION - SUBSCRIPT TOO SMALL FOR ONE-DIMENSIONAL ARRAY, WITH OPTION BASE 1.
REFS: 6.5 15.2 15.4

PROGRAM NUMBER 68
EXCEPTION - SUBSCRIPT TOO LARGE FOR ONE-DIMENSIONAL ARRAY, WITH DIM AND OPTION BASE 1.
REFS: 6.5 15.2 15.4

PROGRAM NUMBER 69
EXCEPTION - SUBSCRIPT TOO LARGE FOR TWO-DIMENSIONAL ARRAY, WITH DIM AND OPTION BASE 0.
REFS: 6.5 15.2 15.4

PROGRAM NUMBER 70
EXCEPTION - SUBSCRIPT TOO SMALL FOR ONE-DIMENSIONAL ARRAY, WITH OPTION BASE 0.
REFS: 6.5 15.2 15.4

PROGRAM NUMBER 71
EXCEPTION - SUBSCRIPT TOO SMALL FOR TWO-DIMENSIONAL ARRAY, WITH DIM AND OPTION BASE 0.
REFS: 6.5 15.2 15.4

PROGRAM NUMBER 72
EXCEPTION - SUBSCRIPT TOO SMALL FOR TWO-DIMENSIONAL ARRAY, WITH DIM AND OPTION BASE 1.
REFS: 6.5 15.2 15.4

PROGRAM NUMBER 73
ERROR - DIM SETS UPPER BOUND OF ZERO WITH OPTION BASE 1.
REFS: 15.4

PROGRAM NUMBER 74
ERROR - DIM SETS ARRAY TO ONE DIMENSION AND REFERENCE IS MADE TO TWO-DIMENSIONAL VARIABLE OF SAME NAME.
REFS: 15.4 6.4

PROGRAM NUMBER 75
ERROR - DIM SETS ARRAY TO ONE DIMENSION AND REFERENCE IS MADE TO SIMPLE VARIABLE OF SAME NAME.
REFS: 15.4 6.4

PROGRAM NUMBER 76
ERROR - DIM SETS ARRAY TO TWO DIMENSIONS AND REFERENCE IS MADE TO ONE-DIMENSIONAL VARIABLE OF SAME NAME.
REFS: 15.4 6.4

PROGRAM NUMBER 77
ERROR - REFERENCE TO ARRAY AND SIMPLE VARIABLE OF SAME NAME.
REFS: 6.4

PROGRAM NUMBER 78
ERROR - REFERENCE TO ONE-DIMENSIONAL AND TWO-DIMENSIONAL VARIABLE OF SAME NAME.
REFS: 6.4

PROGRAM NUMBER 79
ERROR - REFERENCE TO ARRAY WITH LETTER-DIGIT NAME.
REFS: 6.2

PROGRAM NUMBER 80
ERROR - MULTIPLE OPTION STATEMENTS.
REFS: 15.4

PROGRAM NUMBER 81
ERROR - DIM-STATEMENT PRECEDES OPTION-STATEMENT.
REFS: 15.4

PROGRAM NUMBER 82
ERROR - ARRAY-REFERENCE PRECEDES OPTION-STATEMENT.
REFS: 15.4

PROGRAM NUMBER 83
ERROR - ARRAY-REFERENCE PRECEDES DIM-STATEMENT.
REFS: 15.4

PROGRAM NUMBER 84
ERROR - DIMENSIONING THE SAME ARRAY MORE THAN ONCE.
REFS: 15.4

PROGRAM NUMBER 85
GENERAL CAPABILITIES OF GOSUB/RETURN.
REFS: 10.4

PROGRAM NUMBER 86
EXCEPTION - RETURN WITHOUT GOSUB.
REFS: 10.5

PROGRAM NUMBER 87
ERROR - TRANSFER TO NON-EXISTING LINE NUMBER USING THE GOSUB-STATEMENT.
REFS: 10.4

PROGRAM NUMBER 88
THE ON-GOTO-STATEMENT.
REFS: 10.2 10.4

PROGRAM NUMBER 89
EXCEPTION - ON-GOTO CONTROL EXPRESSION LESS THAN 1.
REFS: 10.5

PROGRAM NUMBER 90
EXCEPTION - ON-GOTO CONTROL EXPRESSION GREATER THAN NUMBER OF LINE-NUMBERS IN LIST.
REFS: 10.5

PROGRAM NUMBER 91
ERROR - TRANSFER TO NON-EXISTING LINE NUMBER USING THE ON-GOTO-STATEMENT.
REFS: 10.4

PROGRAM NUMBER 92
READ AND DATA STATEMENTS FOR NUMERIC DATA.
REFS: 5.2 14.2 14.4

PROGRAM NUMBER 93
READ AND DATA STATEMENTS FOR STRING DATA.
REFS: 3.2 5.2 14.2 14.4

PROGRAM NUMBER 94
READING DATA INTO SUBSCRIPTED VARIABLES.
REFS: 14.2 14.4

PROGRAM NUMBER 95
GENERAL USE OF THE READ, DATA, AND RESTORE STATEMENTS.
REFS: 14.2 14.4

PROGRAM NUMBER 96
EXCEPTION - NUMERIC UNDERFLOW WHEN READING DATA CAUSES REPLACEMENT BY ZERO.
REFS: 5.5 14.4

PROGRAM NUMBER 97
EXCEPTION - INSUFFICIENT DATA FOR READ.
REFS: 14.5

PROGRAM NUMBER 98
EXCEPTION - READING UNQUOTED STRING DATA INTO A NUMERIC VARIABLE.
REFS: 14.5

PROGRAM NUMBER 99
EXCEPTION - READING QUOTED STRING DATA INTO A NUMERIC VARIABLE.
REFS: 14.5

PROGRAM NUMBER 100
EXCEPTION - STRING OVERFLOW ON READ.
REFS: 14.5

PROGRAM NUMBER 101
EXCEPTION - NUMERIC OVERFLOW ON READ.
REFS: 14.5

PROGRAM NUMBER 102
ERROR - ILLEGAL CHARACTER IN UNQUOTED STRING IN DATA STATEMENT.
REFS: 3.2 14.2

```
PROGRAM NUMBER 103
   ERROR - READING QUOTED STRINGS CONTAINING SINGLE QUOTE.
   REFS:   3.2 14.2

PROGRAM NUMBER 104
   ERROR - READING QUOTED STRINGS CONTAINING DOUBLE QUOTE.
   REFS:   3.2 14.2

PROGRAM NUMBER 105
   ERROR - NULL DATUM IN DATA-LIST.
   REFS:  14.2

PROGRAM NUMBER 106
   ERROR - NULL ENTRY IN READ'S VARIABLE-LIST.
   REFS:  14.2

PROGRAM NUMBER 107
   INPUT OF NUMERIC CONSTANTS.
   REFS:   5.2 13.2 13.4

PROGRAM NUMBER 108
   INPUT TO SUBSCRIPTED VARIABLES.
   REFS:  13.2 13.4

PROGRAM NUMBER 109
   STRING INPUT.
   REFS:   3.2 13.2 13.4

PROGRAM NUMBER 110
   MIXED INPUT OF STRINGS AND NUMBERS.
   REFS:  13.2 13.4

PROGRAM NUMBER 111
   EXCEPTION - NUMERIC UNDERFLOW ON INPUT CAUSES REPLACEMENT BY
   ZERO.
   REFS:   5.6 13.4

PROGRAM NUMBER 112
   EXCEPTION - INPUT-REPLY INCONSISTENT WITH INPUT VARIABLE-LIST.
   REFS:  13.4 13.5  3.2  5.2

PROGRAM NUMBER 113
   ERROR - NULL ENTRY IN INPUT-LIST.
   REFS:  13.2

PROGRAM NUMBER 114
   EVALUATION OF ABS FUNCTION.
   REFS:   8.4

PROGRAM NUMBER 115
   EVALUATION OF INT FUNCTION.
   REFS:   8.4
```

PROGRAM NUMBER 116
   EVALUATION OF SGN FUNCTION.
   REFS: 8.4

PROGRAM NUMBER 117
   ACCURACY OF SQR FUNCTION.
   REFS: 7.6 8.4

PROGRAM NUMBER 118
   EXCEPTION - SQR OF NEGATIVE ARGUMENT.
   REFS: 8.5

PROGRAM NUMBER 119
   ACCURACY OF ATN FUNCTION.
   REFS: 7.6 8.4

PROGRAM NUMBER 120
   ACCURACY OF COS FUNCTION.
   REFS: 7.6 8.4

PROGRAM NUMBER 121
   ACCURACY OF EXP FUNCTION.
   REFS: 7.6 8.4

PROGRAM NUMBER 122
   EXCEPTION - OVERFLOW ON VALUE OF EXP FUNCTION.
   REFS: 8.5

PROGRAM NUMBER 123
   EXCEPTION - UNDERFLOW ON VALUE OF EXP FUNCTION.
   REFS: 8.4 8.6

PROGRAM NUMBER 124
   ACCURACY OF LOG FUNCTION.
   REFS: 7.6 8.4

PROGRAM NUMBER 125
   EXCEPTION - LOG OF ZERO ARGUMENT.
   REFS: 8.5

PROGRAM NUMBER 126
   EXCEPTION - LOG OF NEGATIVE ARGUMENT.
   REFS: 8.5

PROGRAM NUMBER 127
   ACCURACY OF SIN FUNCTION.
   REFS: 7.6 8.4

PROGRAM NUMBER 128
   ACCURACY OF TAN FUNCTION.
   REFS: 7.6 8.4

PROGRAM NUMBER 129
   EXCEPTION - OVERFLOW ON VALUE OF TAN FUNCTION.
   REFS: 8.5

PROGRAM NUMBER 130
RND FUNCTION WITHOUT RANDOMIZE STATEMENT.
REFS: 8.2 8.4

PROGRAM NUMBER 131
RND FUNCTION WITH THE RANDOMIZE STATEMENT.
REFS: 8.2 8.4 17.2 17.4

PROGRAM NUMBER 132
AVERAGE OF RANDOM NUMBERS APPROXIMATES 0.5 AND 0 <= RND < 1.
REFS: 8.4

PROGRAM NUMBER 133
CHI-SQUARE UNIFORMITY TEST FOR RND FUNCTION.
REFS: 8.4

PROGRAM NUMBER 134
KOMOLGOROV-SMIRNOV UNIFORMITY TEST FOR RND FUNCTION.
REFS: 8.4

PROGRAM NUMBER 135
SERIAL TEST FOR RANDOMNESS.
REFS: 8.4

PROGRAM NUMBER 136
GAP TEST FOR RND FUNCTION.
REFS: 8.4

PROGRAM NUMBER 137
POKER TEST FOR RND FUNCTION.
REFS: 8.4

PROGRAM NUMBER 138
COUPON COLLECTOR TEST OF RND FUNCTION.
REFS: 8.4

PROGRAM NUMBER 139
PERMUTATION TEST FOR THE RND FUNCTION.
REFS: 8.4

PROGRAM NUMBER 140
RUNS TEST FOR THE RND FUNCTION.
REFS: 8.4

PROGRAM NUMBER 141
MAXIMUM OF GROUP TEST OF RND FUNCTION.
REFS: 8.4

PROGRAM NUMBER 142
SERIAL CORRELATION TEST OF RND FUNCTION.
REFS: 8.4

PROGRAM NUMBER 143
ERROR - TWO ARGUMENTS IN LIST FOR SIN FUNCTION.
REFS: 7.2 7.4 8.2 8.4

```
PROGRAM NUMBER 144
   ERROR - TWO ARGUMENTS IN LIST FOR ATN FUNCTION.
   REFS:   7.2  7.4  8.2  8.4

PROGRAM NUMBER 145
   ERROR - TWO ARGUMENTS IN LIST FOR RND FUNCTION.
   REFS:   7.2  7.4  8.2  8.4

PROGRAM NUMBER 146
   ERROR - ONE ARGUMENT IN LIST FOR RND FUNCTION.
   REFS:   7.2  7.4  8.2  8.4

PROGRAM NUMBER 147
   ERROR - NULL ARGUMENT-LIST FOR INT FUNCTION.
   REFS:   7.2  7.4  8.2  8.4

PROGRAM NUMBER 148
   ERROR - MISSING ARGUMENT LIST FOR TAN FUNCTION.
   REFS:   7.2  7.4  8.2  8.4

PROGRAM NUMBER 149
   ERROR - NULL ARGUMENT-LIST FOR RND FUNCTION.
   REFS:   7.2  7.4  8.2  8.4

PROGRAM NUMBER 150
   ERROR - USING A STRING AS AN ARGUMENT FOR AN
   IMPLEMENTATION-SUPPLIED FUNCTION.
   REFS:   7.2  7.4  8.2  8.4

PROGRAM NUMBER 151
   USER-DEFINED FUNCTIONS.
   REFS:  16.2 16.4  7.2  7.4

PROGRAM NUMBER 152
   VALID NAMES FOR USER-DEFINED FUNCTIONS.
   REFS:  16.2

PROGRAM NUMBER 153
   ERROR - SUPERFLUOUS ARGUMENT-LIST FOR USER-DEFINED FUNCTION.
   REFS:  16.4

PROGRAM NUMBER 154
   ERROR - MISSING ARGUMENT-LIST FOR USER-DEFINED FUNCTION.
   REFS:  16.4

PROGRAM NUMBER 155
   ERROR - NULL ARGUMENT-LIST FOR USER-DEFINED FUNCTION.
   REFS:   7.2  7.4 16.2 16.4

PROGRAM NUMBER 156
   ERROR - EXCESS ARGUMENT IN LIST FOR USER-DEFINED FUNCTION.
   REFS:  16.4
```

PROGRAM NUMBER 157
ERROR - USER-DEFINED FUNCTION WITH TWO PARAMETERS.
REFS: 16.2 16.4 7.2 7.4

PROGRAM NUMBER 158
ERROR - USING A STRING AS AN ARGUMENT FOR A USER-DEFINED FUNCTION.
REFS: 7.2 7.4 16.2 16.4

PROGRAM NUMBER 159
ERROR - USING A STRING AS AN ARGUMENT AND PARAMETER FOR A USER-DEFINED FUNCTION.
REFS: 7.2 7.4 16.2 16.4

PROGRAM NUMBER 160
ERROR - FUNCTION DEFINED MORE THAN ONCE.
REFS: 16.4

PROGRAM NUMBER 161
ERROR - REFERENCING A FUNCTION INSIDE ITS OWN DEFINITION.
REFS: 16.4

PROGRAM NUMBER 162
ERROR - REFERENCE TO FUNCTION PRECEDES ITS DEFINITION.
REFS: 16.4

PROGRAM NUMBER 163
ERROR - REFERENCE TO AN UNDEFINED FUNCTION.
REFS: 16.4

PROGRAM NUMBER 164
GENERAL USE OF NUMERIC EXPRESSIONS IN LET-STATEMENT.
REFS: 6.2 6.4 7.2 7.4 8.2 8.4 16.2 16.4

PROGRAM NUMBER 165
COMPOUND EXPRESSIONS AND PRINT.
REFS: 7.2 7.4 12.2 12.4

PROGRAM NUMBER 166
COMPOUND EXPRESSIONS USED WITH CONTROL STATEMENTS AND FOR-STATEMENTS.
REFS: 7.2 7.4 10.2 10.4 11.2 11.4

PROGRAM NUMBER 167
EXCEPTION - EVALUATION OF NUMERIC EXPRESSIONS ACTING AS FUNCTION ARGUMENTS.
REFS: 7.5 8.4 16.4

PROGRAM NUMBER 168
EXCEPTION - OVERFLOW IN THE SUBSCRIPT OF AN ARRAY.
REFS: 6.4 6.5 7.5

PROGRAM NUMBER 169
EXCEPTION - NUMERIC UNDERFLOW IN THE EVALUATION OF NUMERIC EXPRESSIONS ACTING AS ARGUMENTS AND SUBSCRIPTS.
REFS: 6.4 7.4 7.6 8.4

PROGRAM NUMBER 170
EXCEPTION - NEGATIVE QUANTITY RAISED TO A NON-INTEGRAL POWER IN A SUBSCRIPT.
REFS: 7.5 6.2

PROGRAM NUMBER 171
EXCEPTION - LOG OF A NEGATIVE QUANTITY IN AN ARGUMENT.
REFS: 8.5 16.2

PROGRAM NUMBER 172
EXCEPTION - SQR OF NEGATIVE QUANTITY IN PRINT-ITEM.
REFS: 8.5 12.2

PROGRAM NUMBER 173
EXCEPTION - NEGATIVE QUANTITY RAISED TO A NON-INTEGRAL POWER IN TAB-ITEM.
REFS: 7.5 12.2

PROGRAM NUMBER 174
EXCEPTION - EVALUATION OF NUMERIC EXPRESSIONS IN THE PRINT STATEMENT.
REFS: 7.5 8.5 12.2

PROGRAM NUMBER 175
EXCEPTION - UNDERFLOW IN THE EVALUATION OF NUMERIC EXPRESSIONS IN THE PRINT STATEMENT.
REFS: 7.4 7.6 8.6 12.2

PROGRAM NUMBER 176
EXCEPTION - NEGATIVE QUANTITY RAISED TO A NON-INTEGRAL POWER IN IF-STATEMENT.
REFS: 7.5 10.2

PROGRAM NUMBER 177
EXCEPTION - EVALUATION OF NUMERIC EXPRESSIONS IN THE IF-STATEMENT.
REFS: 7.5 10.2

PROGRAM NUMBER 178
EXCEPTION - UNDERFLOW IN THE EVALUATION OF NUMERIC EXPRESSIONS IN THE IF-STATEMENT.
REFS: 7.4 7.6 10.2

PROGRAM NUMBER 179
EXCEPTION - LOG OF ZERO IN ON-GOTO-STATEMENT.
REFS: 8.5 10.2

PROGRAM NUMBER 180
EXCEPTION - EVALUATION OF NUMERIC EXPRESSIONS IN THE ON-GOTO STATEMENT.
REFS: 7.5 10.2 10.5

PROGRAM NUMBER 181
EXCEPTION - UNDERFLOW IN THE EVALUATION OF THE EXP FUNCTION IN THE ON-GOTO STATEMENT.
REFS: 7.4 8.6 10.2 10.5

PROGRAM NUMBER 182
EXCEPTION - NEGATIVE QUANTITY RAISED TO A NON-INTEGRAL POWER IN FOR-STATEMENT.
REFS: 7.5 11.2

PROGRAM NUMBER 183
EXCEPTION - EVALUATION OF NUMERIC EXPRESSIONS IN THE FOR-STATEMENT.
REFS: 7.5 11.2

PROGRAM NUMBER 184
EXCEPTION - UNDERFLOW IN THE EVALUATION OF NUMERIC EXPRESSIONS IN THE FOR-STATEMENT.
REFS: 7.4 7.6 11.2

PROGRAM NUMBER 185
ERROR - MISSING KEYWORD LET.
REFS: 9.2 9.4

PROGRAM NUMBER 186
EXTRA SPACES HAVE NO EFFECT.
REFS: 3.4

PROGRAM NUMBER 187
ERROR - SPACES AT THE BEGINNING OF A LINE.
REFS: 3.4 4.4

PROGRAM NUMBER 188
ERROR - SPACES WITHIN LINE-NUMBERS.
REFS: 3.4 4.4

PROGRAM NUMBER 189
ERROR - SPACES WITHIN KEYWORDS.
REFS: 3.4

PROGRAM NUMBER 190
ERROR - NO SPACES BEFORE KEYWORDS.
REFS: 3.4

PROGRAM NUMBER 191
ERROR - NO SPACES AFTER KEYWORDS.
REFS: 3.4

PROGRAM NUMBER 192
ERROR - PRINT-ITEM QUOTED STRINGS CONTAINING SINGLE QUOTE.
REFS: 3.2 12.2 12.4

PROGRAM NUMBER 193
ERROR - PRINT-ITEM QUOTED STRINGS CONTAINING DOUBLE QUOTES.
REFS: 3.2 12.2 12.4

PROGRAM NUMBER 194
ERROR - ASSIGNED QUOTED STRINGS CONTAINING SINGLE QUOTE.
REFS: 3.2 9.2

PROGRAM NUMBER 195
ERROR - ASSIGNED QUOTED STRING CONTAINING DOUBLE QUOTES.
REFS: 3.2 9.2

PROGRAM NUMBER 196
LINE-NUMBERS WITH LEADING ZEROS.
REFS: 4.2 4.4

PROGRAM NUMBER 197
ERROR - DUPLICATE LINE-NUMBERS.
REFS: 4.4

PROGRAM NUMBER 198
ERROR - LINES OUT OF ORDER.
REFS: 4.4

PROGRAM NUMBER 199
ERROR - FIVE-DIGIT LINE-NUMBERS.
REFS: 4.2

PROGRAM NUMBER 200
ERROR - LINE-NUMBER ZERO.
REFS: 4.4

PROGRAM NUMBER 201
ERROR - STATEMENTS WITHOUT LINE-NUMBERS.
REFS: 4.2 4.4

PROGRAM NUMBER 202
ERROR - LINES LONGER THAN 72 CHARACTERS.
REFS: 4.4

PROGRAM NUMBER 203
EFFECT OF ZONES AND MARGIN ON PRINT.
REFS: 12.4 12.2

PROGRAM NUMBER 204
ERROR - PRINT-STATEMENTS CONTAINING LOWERCASE CHARACTERS.
REFS: 3.2 3.4 12.2

PROGRAM NUMBER 205
ERROR - ASSIGNED STRING CONTAINING LOWERCASE CHARACTERS.
REFS: 3.2 3.4 9.2

PROGRAM NUMBER 206
   ERROR - ORDERING RELATIONS BETWEEN STRINGS.
   REFS: 3.2 3.4 3.6 10.2

PROGRAM NUMBER 207
   ERROR - ASSIGNMENT OF A STRING TO A NUMERIC VARIABLE.
   REFS: 9.2

PROGRAM NUMBER 208
   ERROR - ASSIGNMENT OF A NUMBER TO A STRING VARIABLE.
   REFS: 9.2

6.3 Cross-reference Between ANSI Standard And Test Programs

Section 3: Characters and Strings

3.2: Syntax
1 93 102 103 104 109 112 192 193 194 195 204 205 206

3.4: Semantics
1 186 187 188 189 190 191 204 205 206

3.6: Remarks
206

Section 4: Programs

4.2: Syntax
2 3 4 196 199 201

4.4: Semantics
2 3 4 187 188 196 197 198 200 201 202

Section 5: Constants

5.2: Syntax
1 9 10 11 12 13 27 92 93 107 112

5.4: Semantics
1 9 10 11 12 13 14 27 30 34 60

5.5: Exceptions
30

5.6: Remarks
34 96 111

Section 6: Variables

6.2: Syntax
6 11 12 22 27 56 57 58 59 61 79 164 170

6.4: Semantics
6 11 12 22 27 56 57 58 59 60 61 74 75 76 77
78 164 168 169

6.5: Exceptions
63 64 65 66 67 68 69 70 71 72 168

6.6: Remarks
23

Cross-reference between ANSI Standard and Test Programs (cont.)

Section 7: Expressions

7.2: Syntax
24 25 26 36 37 38 39 40 41 42 43 61 143 144 145
146 147 148 149 150 151 155 157 158 159 164 165 166

7.4: Semantics
24 25 26 33 35 39 40 41 42 43 61 143 144 145 146
147 148 149 150 151 155 157 158 159 164 165 166 169 175 178
181 184

7.5: Exceptions
28 29 31 32 35 167 168 170 173 174 176 177 180 182 183

7.6: Remarks
39 40 41 42 43 117 119 120 121 124 127 128 169 175 178
184

Section 8: Implementation-Supplied Functions

8.2: Syntax
130 131 143 144 145 146 147 148 149 150 164

8.4: Semantics
114 115 116 117 119 120 121 123 124 127 128 130 131 132 133
134 135 136 137 138 139 140 141 142 143 144 145 146 147 148
149 150 164 167 169

8.5: Exceptions
118 122 125 126 129 171 172 174 179

8.6: Remarks
123 175 181

Section 9: The Let-Statement

9.2: Syntax
6 11 12 56 57 58 185 194 195 205 207 208

9.4: Semantics
6 11 12 14 56 57 58 185

9.5: Exceptions
7

Cross-reference between ANSI Standard and Test Programs (cont.)

Section 10: Control Statements

10.2: Syntax
5 15 17 18 19 20 46 88 166 176 177 178 179 180 181 206

10.4: Semantics
5 15 16 17 18 19 21 27 46 85 87 88 91 166

10.5: Exceptions
86 89 90 180 181

Section 11: For-Statements and Next-Statements

11.2: Syntax
44 45 46 47 48 49 50 51 166 182 183 184

11.4: Semantics
44 45 46 47 48 49 50 51 52 53 54 55 166

Section 12: The Print-Statement

12.2: Syntax
1 6 165 172 173 174 175 192 193 203 204

12.4: Semantics
1 6 7 9 10 11 12 13 14 165 192 193 203

12.5: Exceptions
8

Section 13: The Input-Statement

13.2: Syntax
107 108 109 110 113

13.4: Semantics
107 108 109 110 111 112

13.5: Exceptions
112

Section 14: The Data-, Read-, and Restore-Statements

14.2: Syntax
92 93 94 95 102 103 104 105 106

14.4: Semantics
92 93 94 95 96

14.5: Exceptions
97 98 99 100 101

Cross-reference between ANSI Standard and Test Programs (cont.)

Section 15: Array-Declarations

15.2: Syntax
56 57 58 62 65 66 67 68 69 70 71 72

15.4: Semantics
56 57 58 62 65 66 67 68 69 70 71 72 73 74 75
76 80 81 82 83 84

Section 16: User-Defined Functions

16.2: Syntax
151 152 155 157 158 159 164 171

16.4: Semantics
151 153 154 155 156 157 158 159 160 161 162 163 164 167

Section 17: The Randomize Statement

17.2: Syntax
131

17.4: Semantics
131

Section 18: The Remark-Statement

18.2: Syntax
15

18.4: Semantics
15

Appendix A

Differences between Versions 1 and 2 of the Minimal BASIC Test Programs

In the development of Version 2, we introduced a wide variety of changes in the test system. Some were substantive, some stylistic. Below is a list of the more significant differences.

1. Perhaps the most extensive change has to do with the more complete treatment of the errors and exceptions which must be detected and reported by a conforming processor. We've tried to make clear the distinction between the two and just what conformance entails in each case. Also, Version 2 tests a wider variety of anomalous conditions for the processor to handle. It is in this area of helpful recovery from programmer mistakes that the Minimal BASIC standard imposes stricter requirements than other language standards and the tests reflect this emphasis.

2. Version 2 differs significantly from Version 1 in its treatment of accuracy requirements. We abandoned any attempt to compute internal accuracy for the purpose of judging conformance as being too vulnerable to the problems of circularity. Rather we formulated a criterion of accuracy, and computed the required results outside the program itself. The programs therefore generally contain only simple IF statements comparing constants or variables (no lengthy expressions). Those test sections where we did attempt some internal computation of accuracy, e.g., the error measure and computation of accuracy of constants and variables, are informative only.

3. There are a number of new informative tests for the RND function. These are to help users whose applications are strongly dependent on a nearly patternless RND sequence.

4. The overall structure of the test system is more explicit. The group numbering should help to explain why testing of certain sections of the ANSI standard had to precede others. Also, it should be easier to isolate the programs relevant to the testing of a given section by referring to the group structure.

5. We tried to be especially careful to keep the printed output of the various tests as consistent as their subject matter would allow. In particular, we always made sure that the programs stated as explicitly as possible what was necessary for the test to pass or fail and that this message was surrounded by triple asterisks.

References

1. American National Standard for Minimal BASIC, X3.60-1978, American National Standards Institute, New York New York, January 1978.

2. J. A. Lee, A Candidate Standard for Fundamental BASIC, NBS-GCR 73-17, National Bureau of Standards, Washington DC, July 1973

3. T. R. Hopkins, PBASIC - A Verifier for BASIC, Software - Practice and Experience, Vol. 10, 175-181 (1980)

4. D. E. Knuth, The Art of Computer Programming, Vol. 2, Addison-Wesley Publishing Company, Reading Massachusetts (1969)

NBS-114A (REV. 2-8C)

U.S. DEPT. OF COMM. BIBLIOGRAPHIC DATA SHEET (See instructions)	1. PUBLICATION OR REPORT NO. NBS SP 500-70/1	2. Performing Organ. Report No.	3. Publication Date November 1980

4. TITLE AND SUBTITLE Computer Science and Technology

NBS Minimal BASIC Test Programs - Version 2 - User's Manual, Volume 1 - Documentation

5. AUTHOR(S)

John V. Cugini, Joan S. Bowden, Mark W. Skall

6. PERFORMING ORGANIZATION (If joint or other than NBS, see instructions)

NATIONAL BUREAU OF STANDARDS
DEPARTMENT OF COMMERCE
WASHINGTON, D.C. 20234

7. Contract/Grant No.

8. Type of Report & Period Covered

Final

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS (Street, City, State, ZIP)

Same as item 6.

10. SUPPLEMENTARY NOTES

Library of Congress Catalog Card Number: 80-600163

☐ Document describes a computer program; SF-185, FIPS Software Summary, is attached.

11. ABSTRACT (A 200-word or less factual summary of most significant information. If document includes a significant bibliography or literature survey, mention it here)

This publication describes the set of programs developed by NBS for the purpose of testing conformance of implementations of the computer language BASIC to the American National Standard for Minimal BASIC, ANSI X3.60-1978. The Department of Commerce has adopted this ANSI standard as Federal Information Processing Standard 68. By submitting the programs to a candidate implementation, the user can test the various features which an implementation must support in order to conform to the standard. While some programs can determine whether or not a given feature is correctly implemented, others produce output which the user must then interpret to some degree. This manual describes how the programs should be used so as to interpret correctly the results of the tests. Such interpretation depends strongly on a solid understanding of the conformance rules laid down in the standard, and there is a brief discussion of these rules and how they relate to the test programs and to the various ways in which the language may be implemented.

12. KEY WORDS (Six to twelve entries; alphabetical order; capitalize only proper names; and separate key words by semicolons)

Basic; language processor testing; minimal basic; programming language standards; software standards; software testing

13. AVAILABILITY

☒ Unlimited

☐ For Official Distribution. Do Not Release to NTIS

☒ Order From Superintendent of Documents, U.S. Government Printing Office, Washington, D C 20402.

☐ Order From National Technical Information Service (NTIS), Springfield, VA. 22161

14. NO. OF PRINTED PAGES

79

15. Price

$4.00

U.S. GOVERNMENT PRINTING OFFICE: 1980-331-021/6715

USCOMM-DC 6043-P80

NBS TECHNICAL PUBLICATIONS

PERIODICALS

JOURNAL OF RESEARCH—The Journal of Research of the National Bureau of Standards reports NBS research and development in those disciplines of the physical and engineering sciences in which the Bureau is active. These include physics, chemistry, engineering, mathematics, and computer sciences. Papers cover a broad range of subjects, with major emphasis on measurement methodology and the basic technology underlying standardization. Also included from time to time are survey articles on topics closely related to the Bureau's technical and scientific programs. As a special service to subscribers each issue contains complete citations to all recent Bureau publications in both NBS and non-NBS media. Issued six times a year. Annual subscription: domestic $13; foreign $16.25. Single copy, $3 domestic; $3.75 foreign.

NOTE: The Journal was formerly published in two sections: Section A "Physics and Chemistry" and Section B "Mathematical Sciences."

DIMENSIONS/NBS—This monthly magazine is published to inform scientists, engineers, business and industry leaders, teachers, students, and consumers of the latest advances in science and technology, with primary emphasis on work at NBS. The magazine highlights and reviews such issues as energy research, fire protection, building technology, metric conversion, pollution abatement, health and safety, and consumer product performance. In addition, it reports the results of Bureau programs in measurement standards and techniques, properties of matter and materials, engineering standards and services, instrumentation, and automatic data processing. Annual subscription: domestic $11; foreign $13.75.

NONPERIODICALS

Monographs—Major contributions to the technical literature on various subjects related to the Bureau's scientific and technical activities.

Handbooks—Recommended codes of engineering and industrial practice (including safety codes) developed in cooperation with interested industries, professional organizations, and regulatory bodies.

Special Publications—Include proceedings of conferences sponsored by NBS, NBS annual reports, and other special publications appropriate to this grouping such as wall charts, pocket cards, and bibliographies.

Applied Mathematics Series—Mathematical tables, manuals, and studies of special interest to physicists, engineers, chemists, biologists, mathematicians, computer programmers, and others engaged in scientific and technical work.

National Standard Reference Data Series—Provides quantitative data on the physical and chemical properties of materials, compiled from the world's literature and critically evaluated. Developed under a worldwide program coordinated by NBS under the authority of the National Standard Data Act (Public Law 90-396).

NOTE: The principal publication outlet for the foregoing data is the Journal of Physical and Chemical Reference Data (JPCRD) published quarterly for NBS by the American Chemical Society (ACS) and the American Institute of Physics (AIP). Subscriptions, reprints, and supplements available from ACS, 1155 Sixteenth St., NW, Washington, DC 20056.

Building Science Series—Disseminates technical information developed at the Bureau on building materials, components, systems, and whole structures. The series presents research results, test methods, and performance criteria related to the structural and environmental functions and the durability and safety characteristics of building elements and systems.

Technical Notes—Studies or reports which are complete in themselves but restrictive in their treatment of a subject. Analogous to monographs but not so comprehensive in scope or definitive in treatment of the subject area. Often serve as a vehicle for final reports of work performed at NBS under the sponsorship of other government agencies.

Voluntary Product Standards—Developed under procedures published by the Department of Commerce in Part 10, Title 15, of the Code of Federal Regulations. The standards establish nationally recognized requirements for products, and provide all concerned interests with a basis for common understanding of the characteristics of the products. NBS administers this program as a supplement to the activities of the private sector standardizing organizations.

Consumer Information Series—Practical information, based on NBS research and experience, covering areas of interest to the consumer. Easily understandable language and illustrations provide useful background knowledge for shopping in today's technological marketplace.

Order the **above** *NBS publications from: Superintendent of Documents, Government Printing Office, Washington, DC 20402.*

Order the **following** *NBS publications—FIPS and NBSIR's—from the National Technical Information Services, Springfield, VA 22161.*

Federal Information Processing Standards Publications (FIPS PUB)—Publications in this series collectively constitute the Federal Information Processing Standards Register. The Register serves as the official source of information in the Federal Government regarding standards issued by NBS pursuant to the Federal Property and Administrative Services Act of 1949 as amended, Public Law 89-306 (79 Stat. 1127), and as implemented by Executive Order 11717 (38 FR 12315, dated May 11, 1973) and Part 6 of Title 15 CFR (Code of Federal Regulations).

NBS Interagency Reports (NBSIR)—A special series of interim or final reports on work performed by NBS for outside sponsors (both government and non-government). In general, initial distribution is handled by the sponsor; public distribution is by the National Technical Information Services, Springfield, VA 22161, in paper copy or microfiche form.

BIBLIOGRAPHIC SUBSCRIPTION SERVICES

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

Cryogenic Data Center Current Awareness Service. A literature survey issued biweekly. Annual subscription: domestic $35; foreign $45.

Liquefied Natural Gas. A literature survey issued quarterly. Annual subscription: $30.

Superconducting Devices and Materials. A literature survey issued quarterly. Annual subscription: $45. Please send subscription orders and remittances for the preceding bibliographic services to the National Bureau of Standards, Cryogenic Data Center (736) Boulder, CO 80303.

U.S. DEPARTMENT OF COMMERCE
National Bureau of Standards
Washington, D.C. 20234

OFFICIAL BUSINESS

Penalty for Private Use, $300

POSTAGE AND FEES PAID
U.S. DEPARTMENT OF COMMERCE
COM-215

SPECIAL FOURTH-CLASS RATE
BOOK