WL-TR-91-1040

AD-A236 321

ADA COMPILER EVALUATION CAPABILITY
User's Guide, Release 2.0

Thomas Leavitt
Kermit Terrell

Boeing Military Airplanes
P. O. Box 7730
Wichita KS

May 1991

Interim Report

DTIC
ELECTE
JUN 0 4 1991
S B D

Approved for public release; distribution is unlimited.

AVIONICS DIRECTORATE
WRIGHT LABORATORY
AIR FORCE SYSTEMS COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OHIO 45433-6543

91-00931

91 5 31 010

NOTICE

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility or any obligation whatsoever. The fact that the government may have formulated or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication, or otherwise in any manner construed, as licensing the holder, or any other person or corporation; or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

This report is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.

This technical report has been reviewed and is approved for publication.

Raymond Szymanski
RAYMOND SZYMANSKI
Project Engineer

25 March 1991
Date

FOR THE COMMANDER

Charles H. Krueger
CHARLES H. KRUEGER, Chief
System Avionics Division
Avionics Directorate

3 Apr 91
Date

If your address has changed, if you wish to be removed from our mailing list, or if the addressee is no longer employed by your organization please notify WL/AAAF , WPAFB, OH 45433-6543 to help us maintain a current mailing list.

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

May 1991 Interim

Ada Compiler Evaluation Capability
User's Guide, Release 2.0

C-F33615-86-C-1059
PE-63756D
PR-2853
TA-01
WU-03

Thomas Leavitt
Kermit Terrell

Boeing Military Airplanes
Post Office Box 7730
Wichita KS

Raymond Szymanski
WL/AAAF-3 (513) 255-3947
WPAFB OH 45433-6543

WL-TR-91-1040

Approved for Public Release; Distribution is unlimited

The Ada Compiler Evaluation Capability (ACEC) is a set of over 1500 performance and usability tests used to assess the quality of Ada compilers. The ACEC also provides statistical analysis tools to assist in analyzing the results generated by the ACEC. The ACEC is documented through three major documents; the ACEC Reader's Guide, the ACEC User's Guide and the ACEC Version Description Document.

This document, the ACEC User's Guide, describes the details necessary to execute the ACEC test suite and the associated support tools.

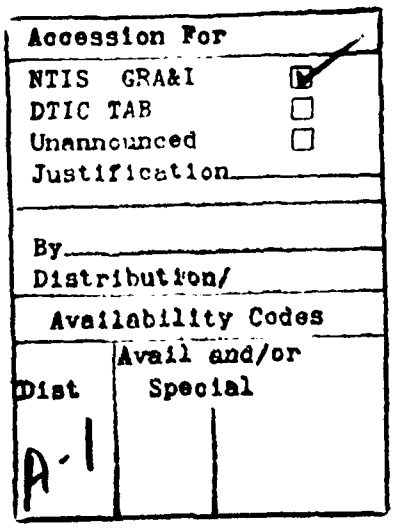
Ada, Compiler, Evaluation, ACEC
Metrics, Evaluation & Validation Project

109

UN UN UN UL

LIMITATIONS

This document is controlled by the Boeing Military Airplanes (BMA) Software and Languages Organization. All revisions to this document shall be approved by the above organization prior to release.



ABSTRACT

This document is the User's Guide for the Ada Compiler Evaluation Capability (ACEC) contract. The purpose of this document is to guide an ACEC user in running the ACEC benchmark test suite and supporting tools.

Contents

1 SCOPE **8**
1.1 IDENTIFICATION . . . 8
1.2 PURPOSE . . . 8
1.3 INTRODUCTION . . . 8
1.3.1 Functional Requirements for the ACEC Project . . . 8
1.3.2 "Potential" ACEC Users . . . 9
1.3.3 Philosophy and General Approach . . . 10
1.3.4 ACEC Classification Taxonomy . . . 16
1.4 EXECUTING THE ACEC . . . 18

2 APPLICABLE DOCUMENTS **25**
2.1 GOVERNMENT DOCUMENTS . . . 25
2.2 NON-GOVERNMENT DOCUMENTS . . . 25

3 TEST SUITE PREPARATION **26**
3.1 INSTALLATION . . . 26
3.1.1 VAX VMS . . . 28
3.1.2 UNIX . . . 28
3.2 PREPARATION . . . 28
3.2.1 Input / Output . . . 30
3.2.2 Global Package . . . 31
3.2.3 Timing Considerations . . . 32
3.2.3.1 Calendar . . . 32
3.2.3.2 CPU Time Considerations . . . 32
3.2.3.2.1 How Test Problems Are Measured . . . 35
3.2.4 Math Package . . . 38
3.2.4.1 Alternative methods for MATH . . . 43
3.2.4.1.1 Instantiate system provided NUMWG package . . . 43
3.2.4.1.2 Adapting to an existing non-NUMWG math library . . . 43
3.2.4.1.3 GEN MATH with portable MATH DEPENDENT . . . 45
3.2.4.1.4 GEN MATH with tailored MATH DEPENDENT . . . 47
3.2.4.2 DEPTEST . . . 49
3.2.4.3 MATHTEST . . . 51
3.2.5 Space Measurement . . . 52

3.2.5.1 Code expansion measurement 52
3.2.5.1.1 Using Label'ADDRESS 52
3.2.5.1.2 Using the GETADR assembly routine 52
3.2.5.2 RTS size 53
3.2.5.3 Assembly language procedure 56
3.2.6 System Parameters 56
3.2.7 Exceptional Tests 58
3.2.7.1 Interrupts 58

4 TEST SUITE COMPILATION 59
4.1 ORDER OF COMPILATION 59
4.2 SYSTEM DEPENDENT TESTS 59
4.3 USING INCLUDE 60
4.4 COMPILATION 63
4.4.1 Potential Compilation Problems 63
4.4.2 Example Compilation Routines 64
4.4.2.1 VAX/VMS 64
4.4.2.2 UNIX 65

5 LINKING/DOWNLOADING THE BENCHMARKS 67
5.1 MERGING PROGRAMS 67

6 RUNNING THE BENCHMARKS 69
6.1 POTENTIAL RUNTIME PROBLEMS 69
6.2 RUNNING A SUBSET 72

7 SYMBOLIC DEBUGGER ASSESSOR 74

8 PROGRAM LIBRARY ASSESSOR 78

9 DIAGNOSTIC MESSAGE ASSESSOR 81

10 ANALYSIS 83
10.1 PREPARING THE DATA 83
10.2 RUNNING MEDIAN 91
10.2.1 New Versions of the ACEC 91
10.2.1.1 Rerunning the tests 91
10.2.1.2 Reanalysis 91
10.3 INTERPRETING RESULTS 92
10.4 SSA 92
10.4.1 System Specification File 92

10.4.2 Input Data Files 97
10.4.3 Other Data Files 100
10.4.4 Measurement Units 100
10.4.5 Implementation Dependencies 100

11 CONSIDERATIONS FOR CODING ADDITIONAL TESTS 101

12 ACEC USER FEEDBACK 104
12.1 HOW TO SUBMIT A PROBLEM REPORT 104
12.2 HOW TO REQUEST CHANGES 107

13 NOTES 110
13.1 ABBREVIATIONS, ACRONYMS 110

List of Figures

1 VDD APPENDICES 12
2 THE ACEC COMMAND FILES 21
3 CPU_TIME_CLOCK FOR DEC ADA 34
4 TIMING LOOP TEMPLATE 36
5 USING INCLUDE 61
6 PROGRAM MERGING 68
7 CONSTRUCTING MED DATA 88
8 SINGLE SYSTEM ANALYSIS SPECIFICATION FILE SYNTAX 94
9 SINGLE SYSTEM ANALYSIS SPECIFICATION FILE EXAMPLE 96
10 SINGLE SYSTEM ANALYSIS DATA INPUT FILE FORMATS 99
11 SAMPLE TEST PROGRAM TEMPLATE 107

1 SCOPE

This section identifies the User's Guide, states its purpose, and summarizes the contents of this guide.

1.1 IDENTIFICATION

This document is the User's Guide for the Ada Compiler Evaluation Capability (ACEC) Software Product.

1.2 PURPOSE

The purpose of this document is to guide the user in running the ACEC benchmark programs and the assessors on a new system and to aid the user in preparing reports summarizing the results. For help in interpreting these findings, refer to the Reader's Guide.

1.3 INTRODUCTION

This guide will give ACEC users the information necessary to adapt and execute the ACEC Software Product. It explains how to use the support tools, and how to deal with the problems which can occur in the process of executing the ACEC *test suite*.

User comments and suggestions for enhancements are solicited and may be reflected in future releases of the product.

The ACEC shall provide a performance assessment capability for Ada compilation systems. The individual ACEC benchmarks provide the raw data for this assessment.

1.3.1 Functional Requirements for the ACEC Project

The ACEC Software Product consists of two Computer Software Configuration Items (CSCIs):

- The operational software, which consists of a suite of performance test programs, and sets of compilation units and script files to exercise and assess a compilation system's symbolic debugger, program library manager, and diagnostic messages.
- The support tools, which assist the ACEC user in preparing the test suite for compilation, in extracting data from the results of executing the test suite, and in analyzing the performance measurements obtained.

The high level requirements on the Operational Software are derived from the needs of the end user community which the ACEC is constructed to serve. The ACEC shall make it possible to:

- Compare the performance of several implementations. The Operational Software shall permit the determination of which is the better performing system for given expected Ada usage.
- Isolate the strong and weak points of a specific system, relative to others which have been tested. Weak points, once isolated, can be enhanced by implementors or avoided by programmers.
- Determine what significant changes were made between releases of a compilation system.
- Predict performance of alternate coding styles. For example, the performance of rendezvous may be such that designers will avoid tasking in their applications. The ACEC will provide information to permit users to make such decisions in an informed manner.
- Determine whether the functional capabilities of a symbolic debugger (if one is present) are sufficient to accomplish a set of predefined scenarios which represent slightly more than a minimal set of capabilities. Compilation systems have provided debuggers with very different user interfaces and functional capabilities. These scenarios will require adaptation to different systems.
- Determine whether the functional capabilities of a program library management system are sufficient to accomplish a set of predefined scenarios which represent more than a minimal set of capabilities. The LRM does not specify a user interface or a set of required capabilities. These scenarios will require adaptation to different systems.
- Evaluate the clarity and accuracy of a system's diagnostic messages. There are no standards for the format or content of diagnostic messages. The interpretation of the system response to these compilation units will require manual inspection and evaluation.

1.3.2 "Potential" ACEC Users

There are several types of users who might be interested in using the ACEC. The different types of users will look to the ACEC to provide different kinds of information.

- Compiler implementors will want to know the strong and weak points of their system(s). They will also want to be able to assess improvements in their system.
- Compiler selectors are interested in comparing performance across systems to choose the best system for their project.
- Compiler users will want to be able to predict the performance of design approaches. They will also want to be able to isolate the strong and weak points of the compilation system they are using and to monitor performance differences between releases of a compilation system.

1.3.3 Philosophy and General Approach

The ACEC user is expected to already know how to use the Ada compilation system being tested. While this is not always a realistic assumption, it is infeasible for the ACEC User's Guide to explain how *every* Ada compilation system which an ACEC user might encounter will operate. For details on how to use any particular Ada compilation system, the ACEC user is referred to the documentation on the compilation system. Similarly, for details of how a user can perform operations in the host operating system, the user must consult system documentation. In particular, an ACEC user is expected to know (or be able to find out from sources other than ACEC documentation) how to: read files from tape; use the text editor; construct a command file; compile, link and execute an Ada program; delete Ada compilation units from a program library; and in general, how to use the tools provided by the Ada system and the host operating system. Before starting with the ACEC proper, they need to perform any Ada program library creation which the compilation system they are using requires.

The Ada programs provided are generally portable, consistent with the requirement to test all major features of the Ada language. In some cases a feature is inherently implementation dependent and will have to be adapted to operate on each new system.

For a complete list of system dependent tests, refer to the Version Description Document (VDD) Appendix VII, "System Dependent Test Problems". The ACEC does not restrict itself to a subset of the language to try to increase its portability. For example, there are some test problems which use floating point types declared with 9 digits of precision, although there are several Ada implementations where this size exceeds SYSTEM.MAX_DIGITS and is not acceptable. There are some test programs which will not terminate when executed on systems which do not support pre-emptive priorities — these programs contain checking code which will output test problem failure code on systems which do not support pre-emption, but they are included in the test suite even though portability is thereby decreased slightly. The decision to include these test programs was made before the Ada Board and the Director of the Ada Joint Program Office ruled that valid Ada implementations must support pre-emptive priority scheduling, and the problems would have been retained in the ACEC even if the ruling had been that pre-emptive priority scheduling was implementation dependent, because many projects are concerned about performance where pre-emption is implemented.

Some test problems may fail to execute on a validated implementation when they violate system capabilities, such as exceeding a capacity limit or not supporting pragma PACK. Each project must decide for itself how serious it considers the failure of any test problem.

The ACEC contains a large number of test problems. Most individual problems are fairly small. Many address one language feature or present an example which is particularly well suited to the application of a specific optimization technique.

One focus of the ACEC analysis tools is on comparing performance data between different compilation systems. Another is on studying the results on one particular system. The analysis tool MEDIAN computes overall relative performance factors between systems and will isolate

test problems where any individual system is much slower (or faster) than expected, relative to the average performance of all systems on that problem and the average performance of the problem on all systems. ACEC users can review the MEDIAN report to isolate the weak and strong points of an implementation, by looking for common threads among the test problems which report exceptional performance data. ACEC users can also examine the results on a system in more detail by running the Single System Analysis program (SSA). MEDIAN and SSA are discussed further in Section 10, "ANALYSIS" of this document, or for detailed information, refer to the Reader's Guide, Sections "MEDIAN OUTPUT" and "SINGLE SYSTEM ANALYSIS".

To be successful any suite of tests must be satisfactory at each of three independent levels.

- The organization of the test suite and supporting tools.

  The topics of interest at this level are the reporting facilities provided and the ease with which a user can identify the test problems in the suite which address areas that the user is interested in.

  The ACEC comparative analysis program will compare performance data between systems. It will identify the test problems which show statistically "unusual" results.

  The ACEC Single System Analysis program will look at related test results and help isolate the strong and weak points of a system's performance.

  An extensive system of indexes and cross reference lists are provided in the Version Description Document's (VDD) appendices that accompany each release. Using these, the test problems which share a particular characteristic can easily be found and their results examined. The VDD appendices are:

Appendix	Name	Contents
I	Test Problem Descriptions	List of test problem names with a brief description of each. New or withdrawn tests are identified.
II	Test Problem to Source File Map	List of test problems and the source file they are contained in.
III	Tape Description	List of files on the delivery tape.
IV	Quarantined Test Problems	Cross reference of test problems observed to fail on some systems.
V	ACEC Keyword Index - 1	List of primary purposes (with LRM references) and their associated test problems, as well as secondary, and incidental purposes, and comparison tests.
VI	ACEC Keyword Index - 2	List of test problems with their primary purposes (which may be for comparison with other tests).
VII	System Dependent Test Problems	List of test problems which exercise system dependent features.
VIII	Optimization Techniques	List of optimization techniques and the benchmarks designed to test them.
IX	Withdrawn Test Problems	List of test problems which have been withdrawn.

Figure 1: VDD APPENDICES

- The relationships between sets of individual test problems in a test suite.

  The primary concern at this level is the breadth and depth of coverage the test problems in the suite provide.

  The functional requirements on the ACEC for breadth and depth of the coverage are discussed in the Reader's Guide, Section "SCOPE OF THE ACEC", and summarized here.

  - The test suite should contain problems which:
    - Address all major syntactic language features.
    - Demonstrate the presence (or absence) of particular compiler optimization techniques by comparing results among related test problems. For example, there are several sets of problems where one version is a "hand-optimized" variation of another. If the system executed both versions in the same time, then it was able to recognize the "more general" example. Test problem SS49 is the statement "ii := LL * 0;" and SS45 is the statement "ii := 0;". If both statements take the same time to execute, it is fair to assume that the compiler has recognized the algebraic simplification possible in SS49 and exploited it.
    - Representative problems from Ada practice. It is important to include examples of how Ada is actually being used. Practical problems are not designed to be optimized (or to be unoptimizable) — they are simply built to get the work done. Some examples are fairly large programs, and can give an estimate for the effects of program locality on cache memory usage which is not representative of very small programs. An example is the KALMAN program, which performs a digital space state filter operation.
    - Classical benchmark problems used in the comparison of other languages. These include programs such as Ackermann's function, Whetstone, Dhrystone, and sort programs. Results from these programs may be available for other languages.

    The ACEC test suite is designed to provide extensive coverage of language features and common constructions.
  - Test problems should not generally duplicate other test problems. For consistency checking, some duplication is desirable.
  - Test problems should differentiate between systems. A good test problem will run well on some target systems and poorly on others. Executing a test problem which all systems treat the same does not provide useful information about whether one system is better or worse than another.

    Because a limited number of systems were tested prior to this release, a test problem which all these systems treated similarly may show differences when run on

additional targets. When there is a *"reasonable" expectation that a problem might* show differences, it is prudent to retain it in the test suite.

The results of some test problems are of independent interest — feature tests such as rendezvous times; or exception propagation time; or procedure call time. Relative performance data is not always sufficient. A system may have a relatively fast rendezvous, compared to other Ada implementations, but the absolute time is also important. For example, a real time system may need to cycle every 20 milliseconds to satisfy the applications requirements. An implementation of that application which requires 100 rendezvous will not satisfy its performance requirements when the fastest entry call takes 2 milliseconds. If all other computations were instantaneous, the program would take 180 more milliseconds to perform the indicated synchronization than are available.

A particular test problem may be related to other problems to expose the presence of specific optimizations. One may be an unoptimizable version of another. If the first were removed from the suite it would be difficult to compare the tests and reveal the presence of the optimization — even when the first problem may not differentiate between systems.

If all the trial systems perform the specific optimization in a comparable manner, then the two test problems will be serially correlated on all systems tested. However, *adding the performance measurement results from* an implementation which does not perform the optimization will weaken the correlation between the two test problems, making the two test problems useful and non-redundant.

- The properties of individual test problems.

  At this level, the relevant question is whether or not the particular test problem is "well-written." The answer to this question depends on the intended purpose of the problem, in addition to the characteristics of the actual code comprising it.

  A test problem which can be optimized into a null statement is a poor problem if it was intended to expose the performance of addition operators, but may be a well written problem if the intention was to test for potential optimizations. For example, "XX := XX + 0;" would be a poor test for general addition of literal values, but is appropriate to test for algebraic simplification.

  Test problems which exhibit the characteristics described below are defined as "poorly written". These are further discussed in the Reader's Guide, Section "CORRECTNESS OF TEST PROBLEMS".

  - The problem could be erroneous Ada.

    This condition is sufficient to disqualify a problem. Erroneous test problems will be *withdrawn.*

- The performance of the test problem could be nonrepeatable.

  It might take a different computation path when it is repeatedly executed, falsifying the assumption that the timing loop can execute the problem many times, divide by the number of executions, and obtain a valid estimate for one execution. If the test problem takes a different amount of execution time on each iteration, a sequence of timing estimates may not converge.

- A problem intended to test one feature may actually test another.

  For example, a problem designed to test passing literal values to a simple function might be expanded inline and folded, making the test problem more a test of possible compile time optimization than a test of passing literal parameters. While it is important to test for the folding of inlined subprograms, it is also important to test for specifying literal actual parameters when inlining is neither requested nor possible.

Also of concern at this level is the accuracy of the timing measurements themselves. The steps the ACEC takes to insure that the timing loop code is accurate is discussed in depth in the Reader's Guide, Section "DETAILS OF TIMING MEASUREMENTS".

Each test problem in the ACEC has been compared against the criteria listed above.

For the purpose of running the MEDIAN analysis tool when a user has performance data from only one compilation system, the ACEC is distributing sample data derived from running MEDIAN on the performance data from the trial systems. This represents the "average" performance of the systems tested, and can be useful in detecting differences between the one system the ACEC user has data on and a hypothetical "typical" system.

There are three different but complementary ways an ACEC user can examine test results. These are discussed in the Reader's Guide, Section "HOW TO INTERPRET THE OUTPUT OF THE ACEC", and reviewed here.

- After running all the test programs on several systems, use MEDIAN to identify the test problems which have statistically unusual behavior on each system.
- Select a set of test problems which use a set of features of special interest and examine the results of these problems.
- Run the Single System Analysis program.

These approaches can be used in combination. An ACEC user could examine the set of test problems which MEDIAN flagged as having unusual performance, trying to discover if they share a common characteristic. After reading the test descriptions in the VDD Appendix I, "TEST PROBLEM DESCRIPTIONS", the user may speculate that slower than predicted performance may be due to subprogram linkages. The user could then refer to the feature cross reference

index in the VDD Appendix V, "ACEC KEYWORD INDEX - 1", to get a list of all test problems using subprogram linkages, and then examine the performance of this list of problems to see if the hypothesis of subprogram linkages seems justified (the SSA has several tables examining subprogram linkage variations). It may be that only some of the test problems which call on subprograms are unusually slow, but perhaps the user would notice that performance problems occur on subprograms which pass nonscalar objects as formal parameters. The VDD indexes and the source text itself provide a rich field for exploration. Depending on their interest, time, and the use they intend to put the results to, it is possible that a user may form very detailed hypotheses about combinations of language features which are responsible for particularly slow performance, and the user may construct unique test problems to verify those hypotheses. Many ACEC users will not be that interested or have that much time to spend. If the reason they are running the ACEC is to choose between two or three compilation systems for one target processor, they may not care if they find that there are not large differences in performance between the systems. The details of the differences may not be important to them.

The important point of the above discussion is that ACEC users will not have to understand the details of a thousand different test problems to obtain useful information from the results.

1.3.4 ACEC Classification Taxonomy

The Ada Compiler Evaluation Capability will not answer every question a user might have about Ada compilers, for example, dollar cost is not addressed. The ACEC is concerned with evaluating systems, not just compilers. No attempt is made to "factor out" the contribution of hardware to overall performance. Some ACEC users will want to compare different target architectures, in addition to comparing different compilers for the same target. This taxonomy will help to place the ACEC tests in perspective and answer user questions about what assistance the ACEC will and will not provide.

- Covered
  - Execution Time Efficiency (see Reader's Guide, Section "EXECUTION TIME EFFICIENCY")

    This is the major emphasis of the ACEC. Users will want to be able to examine the results of the ACEC to study aspects of Ada performance. The ACEC helps the user in isolating the particular tests he may need by providing indexes which list tests by various criteria.
  - Code Size Efficiency (see Reader's Guide, Section "CODE SIZE EFFICIENCY")

    Code expansion size is an important area of interest for the ACEC. This is measured by using the label'ADDRESS attribute. On systems which do not support this feature, users can adapt the ACEC timing loop code to measure code expansion size in other ways, as discussed in Section 3.2.5.1.2.

The ACEC will gather size measurements for all of the tests along with execution time. While not a major thrust of the project, some tests are included which measure data space allocated to objects by using the X'SIZE attribute and comparing the size of the packed objects to the minimum size possible. Sequences of allocation and deallocation in a collection (LRM 13.2.b and 3.8) are included which will fail if space is not reclaimed. The tests are designed as performance tests but will demonstrate, as a side effect, whether storage reclamation takes place. These results, along with others, are reported in the ancillary data section of the SSA report.

- Compile Time Efficiency (see Reader's Guide, Section "COMPILE TIME EFFICIENCY")

  While measuring compile time efficiency is not the primary purpose of the ACEC, data is collected and can be analyzed on a compilation unit basis.

- Symbolic Debugger Assessor (see Section "SYMBOLIC DEBUGGER ASSESSOR")

  The ACEC provides a set of debugger assessor scenarios (programs and sequences of operation to perform) to enable users to evaluate a compilation system's debugger.

- Program Library Assessor (see Section "PROGRAM LIBRARY ASSESSOR")

  The ACEC provides a set of library assessor scenarios (programs and sequences of operation to perform) to enable users to evaluate a compilation system's library system.

- Diagnostic Message Assessor (see Section "DIAGNOSTIC MESSAGE ASSESSOR")

  The ACEC diagnostic assessor tests will determine whether a system's diagnostic messages clearly identify the condition and provide information to correct it, and whether warning messages are generated for various conditions.

- Not Explicitly Covered

  - Test for Existence of Language Features

    This is the charter of the ACVC (Ada Compiler Validation Capability) project. The ACEC test suite assumes that the full Ada language is supported and correctly implemented. The ACEC contains tests for the performance of representation clauses and implementation dependent features (LRM chapter 13 features). Some test problems will require modification to run on different systems (such as using the pragma INTERFACE and calling on a procedure written in assembler language) and may fail on some Ada implementations which do not support the full language.

  - Capacity Tests

    The ACEC is not designed to systematically probe the capacity limits of a system, but in the course of exercising the test suite, users may discover some of the system's

capacity limits. The Library Assessor does explicitly exercise a system's library capacities.

1.4 EXECUTING THE ACEC

This section provides an overview of the steps necessary to prepare, compile, link, and execute the ACEC test suite and to use the analysis tools to study the data generated by executing the test suite.

To execute the ACEC the user must perform the following steps:

- Before compiling any programs, the user should finish reading the User's Guide. Some potential problems have been anticipated, and a user can save time and effort by reviewing these before attempting to run any tests.

- Verify that the Ada system is ready to accept compilations. Some Ada systems require their users to explicitly create an Ada program library before performing any compilations. Users should consult the system documentation to see how this is done, if necessary.

- Read text files from the ACEC distribution tape.

  These will include the ACEC Operational Software and Support Tools and some command files which can be used as models for other systems.

- Compile the package GLOBAL. This package is WITHed by *all* other test programs and must be present in the library for successful compilation.

- Construct a working math package. There are several methods a user may use to accomplish this, these are listed in Section 3.2.4.

- Compile, link and execute TESTCAL1 and TESTCAL2. These problems test the accuracy of the runtime clock using the interface provided by the package CALENDAR.

- Compile and link INCLUDE. Test it on any of the test programs. This is the program which inserts the timing loop code.

- Adapt the command files CMP, CMP_CK, CMP_SP, and CMP DIFF NAMES. These command files INCLUDE, compile, and link an ACEC test program. They may need to be adapted to the host operating system and to invoke the Ada compilation system being tested. The versions provided assume that the host system supports parameters in command files. UNIX scripts provide a facility of comparable power.

  The difference between the four versions are as follows:

- CMP. This command file uses standard options to compile the program. It specifies the compiler options for optimization and suppression of checking for violations of predefined constraints. It also computes the elapsed time to compile and link a program.
- CMP_CK. This command file differs from CMP only in that it specifies a compiler option to not suppress checking for violations of predefined constraints. On a compilation system which fully supports PRAGMA SUPPRESS, it is not necessary to adapt this file and CMP can be used.

  There are many compilation systems which do not honor PRAGMA SUPPRESS in a source program, but which provide a comparable facility with a compiler directive. The CMP CK and CMP command procedures provide ways which will permit such systems to be evaluated fairly. Systems which honor source pragmas can use identical files for CMP_CK and CMP.
- CMP_SP. This command file differs from CMP only in that it specifies a compiler option to optimize space usage. On a compilation system which fully supports PRAGMA OPTIMIZE, it is not necessary to adapt this file and CMP can be used.
- CMP_DIFF_NAMES. This command file differs from CMP in that it compiles a test program into a different name than the name of the input file. It is used on test programs which commonly fail to compile, so that a dummy version of a test program can be created which will generate a well formed error message if executed. This dummy version will be overridden if the compilation of the actual program succeeds.

It is not necessary for a host system to provide a command file capability to execute the ACEC. It is, however, much more convenient to run the ACEC on a capable host. Command file processing permits:

- Repeatable operations — without worry about typing errors
- Automated collection of compilation time data
- Unattended operations — the user does not have to "babysit" a system
- Ease of use

- Compile and link all problems in the test suite, deleting units as they are no longer needed. Many compilers store a compiled Ada program's syntax trees on disk to facilitate linking and visibility to library units. To conserve resources, the ACEC library objects should be removed from the Ada program library as soon as they are no longer necessary (after the last step which used them).

- Execute the test suite, saving results into a file (if possible).

- Compile FORMAT. Run FORMAT on the file produced from the prior step.
- Perform all the above steps on all the Ada compilation systems of interest.
- Compile MED_DATA_CONSTRUCTOR. Run MED_DATA_CONSTRUCTOR using the output from FORMAT as input. The package MED_DATA will be produced as output.
- Compile and execute the comparison program MEDIAN.
- Compile and execute the analysis program SSA.

There is some flexibility in the order in which the above steps can be performed. The test programs in the test suite can be compiled and executed in any order. Many test programs can compile and execute without package MATH (or DBL_MATH). These could be run before MATH is transported. See the Version Description Document (VDD) Appendix VII, "SYSTEM DEPENDENT TEST PROBLEMS" for a list of those test programs that WITH package MATH.

There are two sets of command files on the distribution tape which can serve as models that an ACEC user can adapt to the host operating system and compiler. The samples are for DEC Ada under VMS, and for the MIPS compilation system running on a Silicon Graphics IRIS-4D under UNIX (this compilation system is based on the Verdix Ada Development System) — MIPS is **not** an acronym, it is the proper name of a company. The files with the suffix ".COM" on the distribution tape are the DEC Ada VMS version, and the files with the suffix ".UNX" are the Silicon Graphics version.

A user need not use these command files, or variations of them. The command files are provided to simplify the effort in executing the ACEC against a compilation system. If a compilation system has other methods for compiling and executing programs, they may be used instead.

There is **no** dial-up help for users who run into difficulties in porting the ACEC. Assistance is limited to the provision of this User's Guide, the Reader's Guide, and the Version Description Document.

If difficulties in compiling Ada programs are encountered, it may be appropriate to contact the vendor of the Ada compilation system, since the programs in the ACEC are "portable" Ada programs. Difficulties in compiling and executing them should be reported to the organization maintaining the Ada compilation system for explanation and correction. Such reporting should be consistent with the classification then current on the ACEC — in particular, if the ACEC Software Product is restricted for release within the government and government contractors, then sending the "test suite" to a vendor is **not** proper.

The following figure shows the command files which will perform the steps noted above for the DEC Ada compilation systems and can be modified for other compilation systems.

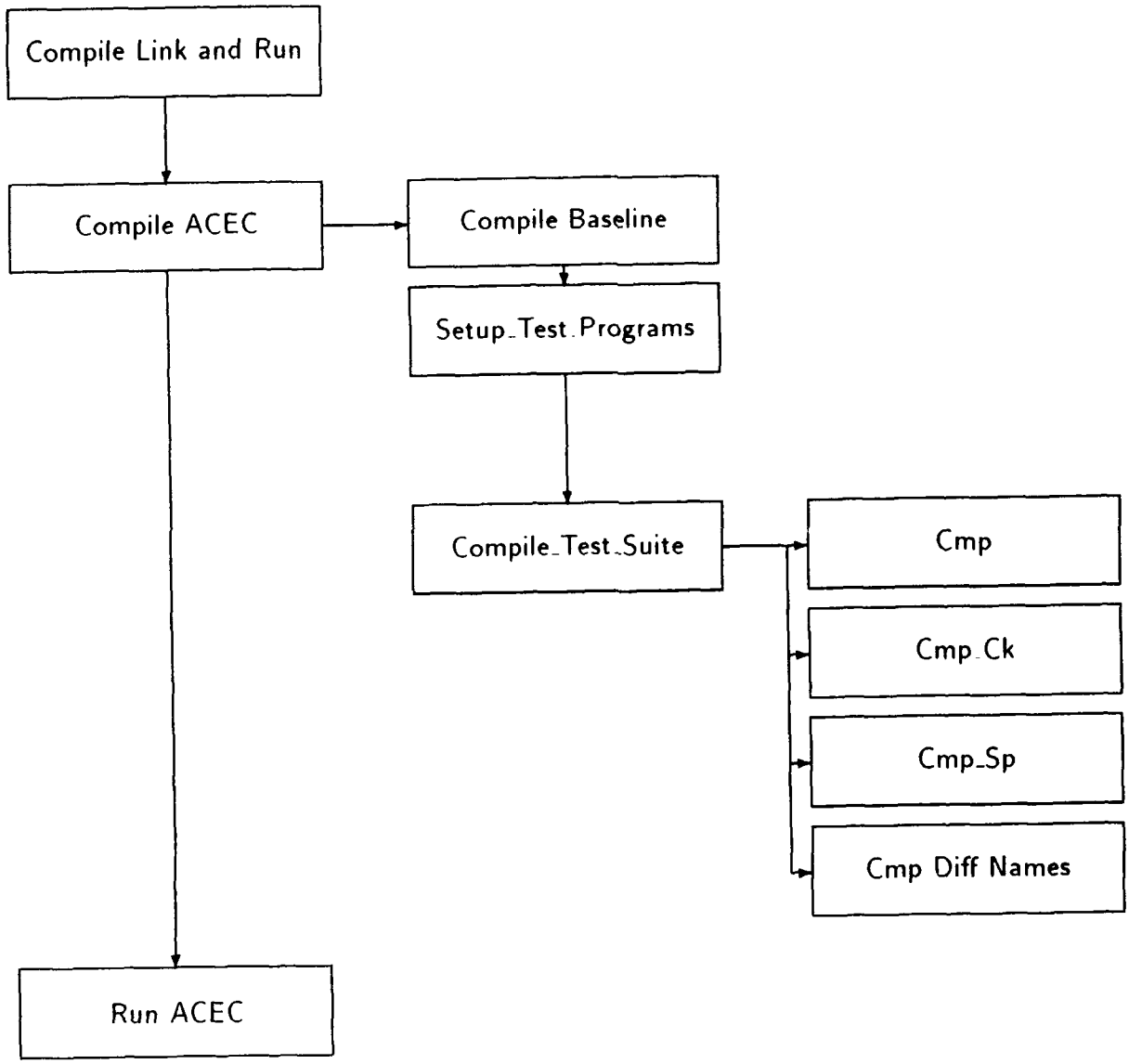


Figure 2: THE ACEC COMMAND FILES

Each of these steps will be elaborated later in this guide. The major purpose of each step is:

- Compile Baseline

  This file compiles the "foundation" or "baseline" of the ACEC Software Product. This compiles the packages GLOBAL, either RAN32 or RAN16 (depending on the capabilities of the target), and MATH and DBL_MATH which are math libraries for 6 and 9 digit floating point types respectively. In order to compile the math packages, depending on the options selected and the software support of the implementation, the user may have had to compile the packages MATH_DEPENDENT, DEPTEST, and GEN_MATH. Users have a choice between different versions of GLOBAL and different ways of constructing MATH. The command files must be adapted if the default choices are not what is desired. These alternatives are discussed in more depth later.

- Setup_Test_Programs

  This file compiles and links the programs MATHTEST and DBL_MATHTEST which verify the accuracy of the math library, and the programs TESTCAL1 and TESTCAL2 which verify the accuracy of the runtime clock using the interface provided by the package CALENDAR.

- Compile_Test_Suite

  This command file compiles and links all the test programs. It deletes intermediate objects from the program library as soon as they are no longer needed to conserve disk space.

- Run_ACEC

  This file executes all the test programs.

Many users may ask: How long is it going to take to compile and execute the ACEC test suite? This is hard to answer, since there are sources of variability in estimating time. Some of these variability sources are discussed below:

- Running the ACEC involves compiling all the test problems and support tools. The 334 performance test programs generate over a half million lines of source after INCLUDE has been executed. GEN_MATH and the math test programs add approximately 27K lines of source to be compiled and the support tools add another 20K. A system with a slow compiler will take more time to process this source than a system with a fast compiler.

- Some programs, in particular the math library, will need to be adapted to each target. It is difficult to predict the time required for this adaptation since it involves review and understanding of the system documentation (for example, finding out what the floating point

representation is) and modification of Ada source for the package MATH_DEPENDENT. The time required to make the adaptation also depends on the number of math packages that are or are not provided by the vendor. The ACEC user will need to learn about the supported facilities — in particular the Chapter 13 features, including record representation clauses.

- Users may uncover errors in the compilation system using the ACEC. Although for performance comparisons, an ACEC user could treat such problems as "failed" and essentially ignore them during cross system comparison, they may want to isolate and report errors for correction as they are found.

  The ACEC should be run only against validated implementations of Ada. However, even validated systems can contain errors which may be detected by the code in the ACEC. The ACEC does not include code intended to discover implementation errors — that is the charter of the Ada Compiler Validation Capability project. During development and testing of the ACEC, some of the test problems uncovered errors in implementations. Where implementation errors were found, and in a few other places where it is simple to include verification code, the test problems attempt to check that they have executed correctly. All test problems which contain verification code will report an error when an error is detected.

- It is quite possible that running the ACEC will be the first task an organization does with a new system. This may be the case when an organization has obtained a compilation system for evaluation. Programmers running the ACEC may be learning to use the Ada compilation system and the host operating system by running the ACEC. It is certain that a user experienced with the Ada compilation system and host operating system (including using text editors, building command files or scripts, understanding what cryptic diagnostic messages *really* mean,...) will be able to work through the preparation phase of the ACEC much faster than a user with no prior experience on the system.

- After being compiled and linked, each program must be executed. Before execution of the program can occur on a cross-compiled target, the executable code must be downloaded to the target. Depending on the available hardware (speed and error rate), downloading can take longer than the time spent compiling and executing the test suite combined.

For comparison purposes, a DEC VAX station 3100 (which is roughly a 2.7 MIP processor) with 16 megabytes of memory and a WORKING SET size of 4500 pages (1 page = 512 bytes) of real memory on one implementation can run through the files to setup, compile and link all the test problems in approximately 13 hours, and can execute the test suite in approximately 11 hours. These times were measured on an otherwise idle system.

The time to use the tools to extract the measurement data from the output of the ACEC execution and process it through the analysis tool is fairly small, and should take less than an hour on a system — although it does require working with system text editors and for a user who is unfamiliar with an editor the time can be highly variable.

The amount of difficulty in adapting the math library and isolating errors as they occur will be highly variable. Two weeks should be sufficient for most systems, but this will depend on the system documentation, facilities supported, system debugging aids, etc. The ACEC distribution tape includes samples of the package MATH_DEPENDENT which worked for the trial implementations. These can serve as models, or may be usable without further modification. If problems are encountered which require vendor support for correction, the time required to use the ACEC can grow considerably.

2 APPLICABLE DOCUMENTS

The following documents are referenced in this User's Guide.

2.1 GOVERNMENT DOCUMENTS

MIL-STD-1815A — Reference Manual for the Ada Programming Language (LRM) 1983

2.2 NON-GOVERNMENT DOCUMENTS

D500-12471-1 — Ada Compiler Evaluation Capability (ACEC)
Technical Operating Report (TOR)
Reader's Guide
Boeing Military Airplanes
P.O. Box 7730
Wichita, Kansas

D500-12472-1 — Ada Compiler Evaluation Capability (ACEC)
Version Description Document
Boeing Military Airplanes

Improving a Poor Random Number Generator,
C. Bays and S. D. Durham,
ACM Transactions on Mathematical Software,
Volume 2, Number 1, March 1976.

Introduction to the Theory of Statistics,
A. Mood and F. Graybill, McGraw Hill. 1963.

The Need for an Industry Standard of
Accuracy for Elementary-Function Programs,
C. Black, R. Burton, and T. Miller,
ACM Transactions on Mathematical Software,
Volume 10, Number 4, December 1984

3 TEST SUITE PREPARATION

Most of this guide deals with programs which do not require special equipment or operator intervention to run. The vast majority of the programs in the test suite fit in this category. However, there are a few "exceptional" programs which require special handling. Refer to Section 3.2.7 on Exceptional Tests for details.

3.1 INSTALLATION

This section will guide the user in installing the ACEC test suite. This job is system dependent, but some general guidance is possible and more detailed instructions are provided for two widely available operating systems: VAX VMS and UNIX.

Test problem file names are unique and limited to 8 characters, permitting the test suite and support tools to run on a host with a "flat" file system without a hierarchical directory structure. The only requirement is that, in addition to the storage space for the Ada compilation system being tested, there must be sufficient secondary storage space available for the ACEC. How much space is required varies greatly between software implementations due to the usage characteristics of the host system, but the following information on disk space requirements can be used for planning purposes:

- Disk usage for source files:

  The ACEC source requires approximately 9 megabytes of disk space (6.9 megabytes for the performance test source files, 1.1 megabytes for the ACEC documentation (Guides and VDD), and the rest for the support tools.) It is not necessary to keep all the source files online to run the ACEC. An ACEC user could run the performance tests by reading the source for test programs as needed and deleting them after they are compiled, greatly reducing the peak disk space usage.

- Disk space for Ada program library use for the performance tests:

  ACEC observed the disk space needed for peak usage to be in the range of 2 to almost 4 megabytes. Long term library storage is required for packages GLOBAL, MATH, DBL MATH, RAN and INCLUDE. The intermediate files for the performance test programs are deleted as soon as the programs are linked to minimize disk usage.

- Disk usage for executables:

  ACEC observed the total usage to be in the range of 13 to approximately 100 megabytes. By executing and then deleting a test program's executable, the peak disk usage is greatly reduced. Approximately 1.1 megabyte should be sufficient for the largest program to compile and execute.

- Disk usage for output data files:

  Typical sizes for the output data files are listed below:

  - COMPILE_ACEC_LOG .90 megabytes
  - COMPILE_TEST_SUITE .23 megabytes
  - RUN_ACEC_LOG .75 megabytes

- Memory usage for performance tests:

  The majority of the programs will run on a 1750A with 64K 16-bit words.

In addition to the disk space guidelines, the following information on time allotment to execute the ACEC product can also be used for preliminary planning:

- Preparation time:

  The time required to adapt the ACEC to a particular implementation varies with the characteristics of the compilation system and the experience of the ACEC user on the system being evaluated.

- Time for compilation of the performance tests:

  Essentially, it is the time to compile 500,000 lines of code. For a compiler operating at 500 lines per minute, this will take just under 17 hours.

- Time for execution of performance tests:

  This varies greatly between systems and compilation options selected. For an example, it took 17 hours to execute all the performance tests for one compilation system on a VAXstation 3100. The time may be MUCH slower for embedded targets due to downloading.

- Total time:

  As a rough estimate to aid preliminary planning, one programmer should be able to install and run the complete ACEC test suite and assessors in less than 4 weeks, if there are at least 20 megabytes of free disk space. The amount of time varies with the experience of the programmer, the reliability of the system being tested, and the amount of free disk space. Analysis of results and isolation of errors can be very time consuming and are not included in the time estimates. It is possible to run subsets of the ACEC tests in less time.

3.1.1 VAX VMS

This section explains how to install the ACEC test suite on one VAX VMS hosted system.

The procedure under VMS proceeds as outlined in Section 1.4. The user must adapt the *.COM files to reflect the directory names of their account. For compilers other than DEC Ada, the commands to invoke the compiler and manipulate the Ada program library system must be adapted to the demands of the compilation system being evaluated. It may be necessary to modify the procedure to satisfy special requirements of particular compilation systems.

On systems with limited available space it may not be possible to retain all the executable files created by COMPILE TEST SUITE.COM to execute them in one block after all performance tests have been compiled. On such systems, it may be necessary to modify COMPILE_TEST_SUITE.COM to execute each test program (preferably at least twice to observe variability) and then delete the executable file to conserve disk space.

3.1.2 UNIX

This section tells how to install the ACEC test suite on a UNIX hosted system. Bourne shell scripts are presented which compile and execute the ACEC on one UNIX hosted system.

The procedure under UNIX will also proceed as outlined in Section 1.4. The ACEC must adapt the script files to reflect the directory names on the system being tested. The commands to invoke the compiler and manipulate the Ada program library system must be adapted to the compilation system being evaluated. ACEC users should expect to adapt the scripts to their UNIX implementations — the sample UNIX scripts use straightforward Bourne shell operations analogous to the VMS command files, but this has not been sufficient to avoid implementation dependencies. Bourne shell scripts were used rather than "C shell" scripts to try to increase the portability of the scripts to other implementations. All UNIX implementations are reported to support the Bourne shell, but not all support the "C shell."

3.2 PREPARATION

This section discusses the steps which should be performed prior to compiling the ACEC test suite to insure that meaningful results are obtained. It also discusses some of the problems which may be encountered in transporting the ACEC to a new compilation system. The documentation for the Ada compilation system being tested should be consulted for specific details.

The ACEC Software Product was developed under the working assumption that the entire Ada language is supported. The ACEC source code does not deliberately restrict itself to a subset of Ada in either the test suite or the support tools. Some test programs, and some tools, may fail to compile (or to execute properly) on systems not supporting full Ada. For example, some tasking test programs (DELAYS, TASK44, and TASK45) will not terminate normally if

the underlying task scheduler does not recognize pre-emptive priorities. Some test problems use floating point types with 9 digits of precision, which not all implementations will support (GAMM2, KALMAN, LOOP3, S0301T15, S0316T30). Some test programs use PRAGMA PACK to the bit level and unchecked conversions between integer types and packed boolean array types to perform bit manipulation on integer types (DES7 and S0500T12). In these cases, the failure of an implementation to support the feature may result in a few test problems not being able to execute. On the other hand, if an implementation provided no support for tasking (including omitting the DELAY statement) then none of the test suite would run, since a DELAY is used in the timing loop code. It may be possible to work around some limitations of an implementation. This is discussed in the following subsections.

Areas of Ada where the ACEC problems may need user adaptation are:

- Definition of type GLOBAL.BIGINT is derived from the predefined type INTEGER. On systems where INTEGER does not have sufficient range (more than 16 bits) it will be necessary to modify the definition in GLOBAL to derive the type from LONG_INTEGER.

- The variables GLOBAL.EXCESSIVE_TIME_WARNING and GLOBAL.EXCESSIVE_TIME are used to limit test problem execution time and restrict the total time used to execute an individual test problem. The defaults are that the system will print a message after five minutes when the timing loop gains control and the problem is not terminating (so that users can tell that the problem is not in an infinite loop) and will terminate execution after thirty minutes. These numeric values can be adapted by users as desired.

  Setting these values will not provide a fine grain limitation on each test problem, because the elapsed time is only checked after control flow passes normally into the timing loop code. If a test problem executes for twenty minutes, when it returns after the first execution, it will print one warning message, and after the second execution (that is, after forty minutes) it will stop.

  Refer to the Reader's Guide, Section "DETAILS OF TIMING MEASUREMENTS", for a discussion of the timing loop.

- An ACEC user wanting to measure CPU time will have to adapt the function CPU_TIME_CLOCK. Refer to Section 3.2.3.2 for more discussion.

- The function GETADR may need to be adapted. Refer to Section 3.2.5.1 for more discussion.

- The variable GLOBAL.TIMER_TOLERANCE specifies the requested accuracy of the execution time measurements. It is defaulted to five per cent, but can be changed if desired.

- The array GLOBAL.T_VALUE specifies the statistical confidence levels. These can be adapted by a user if desired. The adapting user will require access to statistical tables.

- Tying tasks to interrupts.

  Here the LRM permits each implementation to define the meaning of the value in the **USE AT** clause.

- Using programmer specified task scheduling.

  Although not required by the LRM, many implementations provide a way for users to specify how equal priority tasks should be scheduled; whether to use run-till-blocked scheduling or time-slicing, or to specify a value for the time-slice quantum. The ACEC test programs DELAY1 3 and DELAY4 6 direct the user to modify the program and/or linkage procedure (if possible) to specify a particular scheduling discipline. The distributed source contains a deliberate error so that it will not compile without the user explicitly reviewing and modifying the test program. The test suite provides tests for this implementation dependent feature because:

  - Knowing whether a system permits programmers to specify scheduling discipline is a fact of independent interest to some ACEC users.
  - By constructing alternate versions of some programs which specify different scheduling disciplines, it is possible to derive information about the task scheduling overhead, which would not otherwise be available. It is possible to calculate the overhead for a task switch when time-slicing between equal priority tasks is supported and a user can request an alternative task scheduling discipline for comparison. This information is of interest to some ACEC users.

  Different implementations which support the capability provide different ways for a programmer to request it — some use an implementation defined pragma, some a linker directive, some a call on an implementation defined runtime system routine.

In all of these examples, an ACEC user may chose not to perform the adaptation. They may not be interested in the features being tested, may be unable to perform the operations, or may not have time to adapt the programs. Where results are unavailable, the user will record these test problems as implementation dependent.

3.2.1 Input / Output

The ACEC outputs strings containing numeric results of performance tests. If TEXT IO and FLOAT_IO are not supported, the results of the tests cannot be displayed without modification to the timing loop code. The user trying to run Ada programs on a new system should run a "Hello, World" program to insure that simple string output is working. This is a program which uses TEXT IO to output a simple string such as "Hello, World". If FLOAT IO is not directly supported, users will have to develop workarounds. Some alternatives have been developed on other systems.

If the Ada system supports a complete version of the Text_IO package, this requirement will cause no problems. However, some Ada compilers targeted to real-time systems (where the hardware is limited) may limit their I/O facilities. A user may have to write a floating point to a text-string conversion routine.

For bare machine implementations of Ada, the effort required to get TEXT_IO to work well enough to output results of the timing and sizing measurements on a console can be large. Portions of the Ada runtime library may need to be modified, I/O device drivers may need to be written and tested. It would be possible (but painful) to use a debugger to extract the performance measurement data. A user could provide stubbed versions of the I/O subprograms, and then set a breakpoint in these dummy subprograms and then use the debugger commands to examine what would have been output if the "real" I/O subprogram were present.

3.2.2 Global Package

There may be modifications necessary to the package GLOBAL to adapt it to another implementation. The sample command file COMPILE_BASELINE indicates where changes should be made.

- The command file COMPILE_BASELINE is distributed with three options for the timing loop code, corresponding to the CLOCK, CPU, and SIZ alternatives, two of which are "commented out." These example alternatives in COMPILE_BASELINE.COM help in adapting to another compilation system.

- To measure CPU time, it will be necessary to use the version of GLOBAL and the timing loop code (INITTIME, STARTIME, STOPTIME0, and STOPTIME2) with the suffix ".CPU" This is accomplished by compiling GLOBAL and overwriting the timing loop code files with suffix *.TXT with the corresponding *.CPU versions. The timing loop code files with suffix *.CLOCK contain the code to measure elapsed time. The *.CPU versions will collect performance measurements using calls to a user defined function GLOBAL.CPU_TIME_CLOCK. The user must replace the function CPU_TIME_CLOCK (in the source file GLOBAL.CPU) with one specific to the implementation they are testing which returns the job elapsed CPU time as a value of type CALENDAR.DURATION. The version distributed with the ACEC may be used as a model for constructing such a function for other systems.

- If the system does not support an integer type with at least 32 bits of precision, the declaration of the type "BIGINT" and "BIGNAT" will not compile and must be removed. Also, the test problems which use this type will fail at compile time.

- If the system does not support a floating point type with 9 digits of precision, the declaration of the type "DOUBLE" will not compile and must be removed. Also, the test problems which use this type will fail at compile time.

- The function "ADDRESS_TO_INT" converts a value of type SYSTEM.ADDRESS to an integer type. This function is used to compute the code expansion sizes by subtracting the address values of two label'ADDRESS attributes (or of two type ADDRESS variables obtained by a GETADR function). On different systems, SYSTEM.ADDRESS'SIZE will differ, forcing a modification to the return type of this function.

3.2.3 Timing Considerations

The INCLUDE tool is used to insert the timing loop code into the test suite so that any change(s), if necessary, only have to be made in one place. For more information, refer to Section 4.3.

An ACEC user may decide to collect timing measurements either using CALENDAR to measure elapsed time or an implementation dependent function to measure CPU time. These options are discussed below.

3.2.3.1 Calendar

The elapsed timing measurements are performed using the function CLOCK in the predefined package CALENDAR. CALENDAR must work accurately for the timing loop code to function. It is tested with the programs TESTCAL1 and TESTCAL2. These programs print out a count of elapsed minutes whenever the minute changes for 15 minutes (or until the process is aborted by the user). The ACEC user should verify that a line is generated every 60 seconds using a (stop)watch. Some error is tolerable, but most systems should show no discernible error. A one second drift in two minutes is less than a 1% error. While this magnitude of error is small enough to permit the ACEC to collect reliable measurements, it would be intolerably large for many ultimate applications.

If CALENDAR.CLOCK doesn't work, the ACEC user should get it fixed before proceeding. The first thing to check is that the Ada system has been properly installed.

During development of the ACEC, several systems were observed which had gross errors in CALENDAR — one had a 70 second minute, and one measured minutes passing as fast as the system could print. On one 1750A system, the clock never advanced. This turned out to be due to a hardware error which was corrected by modification of the system initialization code.

The ACEC Software Product is set up to measure elapsed time as a default. The ACEC user can modify this to measure CPU time, if desired, as described in Section 3.2.3.2, "CPU Time Considerations."

3.2.3.2 CPU Time Considerations

An ACEC user can choose to run the timing loops using CPU time rather than elapsed time. To do so, the version of the included files INITTIME.CPU, STARTIME.CPU, STOP-

TIME0.CPU, and STOPTIME2.CPU should be renamed with a .TXT suffix and GLOBAL.CPU should be compiled instead of GLOBAL.CLOCK. The separate function CPU_TIME_CLOCK will need to be replaced with code which queries the Ada runtime environment and returns the CPU time as a value of the predefined and system dependent type CALENDAR.DURATION. To use CPU time measurements, it is necessary that the target environment maintain job CPU — many general purpose, multiuser operating systems do provide for the collection of job CPU time data for billing purposes. Few bare machine targets do so. It will not be possible to collect CPU time measurements when the underlying environment does not track CPU time.

Using CPU time will permit the collection of measurements on multiprogramming target systems without having to shut the system down to eliminate contending jobs. CPU time measurements will not be (greatly) affected by the presence of concurrent users, as elapsed time measurements are. If CPU times are desired, it is possible to execute the ACEC test suite without requiring the exclusive access that elapsed time measurements require. For most test problems, a CPU time measurement is a good approximation of the elapsed time it would take to execute the test problem on an unloaded system. When this is true, using CPU time measurement is a data collection technique which attempts to simplify the gathering of performance measurements. However, for several classes of programming constructions of particular importance to mission critical computer resource applications, CPU time is not comparable to elapsed time on an unloaded system. During I/O operations and **DELAY** statements the CPU can be idle before the execution of the statement is completed. A real time system may need to do an I/O operation and wait until it is completed to proceed. The CPU time may be much less than the elapsed time, but operators waiting on the application must wait in real time. Similarly, a real time control system is unconcerned with CPU time.

Operating systems which maintain CPU time information have traditionally done so for the purpose of accounting systems and charging. It is not necessary for these purposes to allocate time with extreme precision. For example, some systems may consider all process scheduling operations to be system overhead. The CPU timer may charge all scheduling operations to overhead or to the job which was executing when the event which triggered the scheduling occurred (I/O interrupt, delay expiration, ...). An Ada implementation which assigns each Ada task to an operating system process will not accurately reflect the "true" cost of tasking on the system.

Measurements for CPU time and elapsed time for **DELAY** statements and I/O operations should not be directly compared since they are measurements of fundamentally different quantities. CPU time measurements of tasking problems may be suspect.

On bare machine targets, elapsed time measurements are the appropriate metric to collect.

The following figure is an example of a function which will access CPU time.

```
- This function was developed by the ACM-SIGAda.PIWG (Association for Computing
- Machinery, Special Interest Group on Ada, Performance Issues Working Group)
- It is their program A000012. This version is compatible with DEC VAX Ada,
- calling on the VMS System Service routine "$GETJPI". Refer to VAX/VMS
- System Services Reference Manual, Order No. AA-Z502C-TE for more information.
-
- The Ada function has a return type of DURATION.
-
- A common implementation technique introduced errors in using CPU time for
- timing measurements. One field in the Task Control Block (TCB) will
- represent cumulative CPU time, but is only updated on task scheduling.
- A system call which returns the field from the TCB will ignore the time
- the task has expended in the current quantum (that is, since last scheduled).
- This would appear to a program as a clock which "stutters", keeping the
- same value for a relatively long time and then updating itself by several
- "ticks" at one time. Such a clock can keep long term accuracy, but programs
- using it must accommodate substantial amounts of jitter. To compute current
- CPU time, the time since last initiation of the task should be added to the
- value stored in the TCB. If the built in system call does not do this,
- a user can. If not done, the ACEC timing loop will compute a larger than
- otherwise necessary value for the jitter compensation time, and the
- time to execute the test suite will be longer than it needs to be.
- Accuracy should not be seriously degraded. The VMS system call performs the
- desired compensation.

with SYSTEM; use SYSTEM;
with CONDITION_HANDLING; use CONDITION_HANDLING;
with STARLET; use STARLET;

separate ( Global )

function CPU_TIME_CLOCK return DURATION is

    CPUTIM : INTEGER;
    pragma VOLATILE ( CPUTIM );
    JPI_STATUS : COND_VALUE_TYPE;
    JPI_ITEM_LIST : constant ITEM_LIST_TYPE :=
                                ( ( 4 , JPI_CPUTIM , CPUTIM'ADDRESS , ADDRESS_ZERO ) ,
                                ( 0 , 0 , ADDRESS_ZERO , ADDRESS_ZERO ) );

    CPU_TIME_AS_DURATION : DURATION;
begin

- Call GETJPI to set CPUTIM to total accumulated CPU time
- (in 10-millisecond tics)

    GETJPI ( STATUS => JPI_STATUS , ITMLST => JPI_ITEM_LIST );
    CPU_TIME_AS_DURATION := DURATION ( LONG_FLOAT ( CPUTIM ) / 100.0 );
return CPU_TIME_AS_DURATION;

end CPU_TIME_CLOCK;
```

Figure 3: CPU_TIME_CLOCK FOR DEC ADA

3.2.3.2.1 How Test Problems Are Measured

An ACEC user will not generally have to understand the details of the techniques used to obtain timing measurements. However, it is possible that on a new system, the timing loop code will fail and the ACEC user must understand it enough to fix it. The Reader's Guide, Section "REQUIREMENTS OF THE TIMING LOOP CODE", contains an additional discussion of the design requirements for the timing loop.

Each test problem is measured by "plugging it into" a template which will, when executed, measure and report on the execution time and size of the test problem contained within it.

The timing loop code consists of four (4) code files (INITTIME, STARTIME, STOPTIME0, and STOPTIME2) which are incorporated into the source by a preprocessor (INCLUDE) which performs text inclusion. The body "INITTIME" is included once per program and contains code to initialize the timing loop variables (as discussed later in this section). The other code files bracket each test problem to be timed, provide a place for writing an identification of the test, compute the execution time and code expansion size of the test problem they enclose, and output the measurements obtained.

The general form for all test programs is:

```
.
with global; use global;
with calendar; use calendar;
                                        -- declarations
      .
      .
      .
begin
      .                                 -- initializations can go here
pragma include ("inittime");            -- once per program
      .                                 -- or initializations can go here too
      .
      .
pragma include ("startime");            -- first test problem
      ...                               -- test problem code goes here
pragma include ("stoptime0");
      put("...");                       -- name and description goes here
pragma include ("stoptime2");
      .
      .
      .
pragma include ("startime");            -- second test problem
      ...                               -- test problem code goes here
pragma include ("stoptime0");
      put("...");                       -- name and description goes here
pragma include ("stoptime2");
      ...                               -- additional test problems enclosed by
      ...                               -- startime / stoptime0 / stoptime2 follow
```

Figure 4: TIMING LOOP TEMPLATE

The "..." after the PRAGMA INCLUDE("inittime") is replaced by the initialization code for the problem (if any); the "..."after the PRAGMA INCLUDE("startime") is replaced by the test problem to be timed; and the "..." after the PRAGMA INCLUDE("stoptime0") would be replaced by a PUT (or a sequence of **PUT_LINE** and **PUT** statements) which prints the problem name and an English description of the problem. It is appropriate to include lists of related tests and the purpose of the test.

The purpose of each of the included files is as follows:

- INITTIME. This code initializes the timing loop variables and computes the execution time and code expansion space for the null loop. It is executed once per program. If it is included in a program twice, there is likely to be a compiler error because it defines labels which cannot be duplicated within an Ada program.
- STARTIME. This code is the beginning of the timing loop.
- STOPTIME0. This code is the end of the timing loop.
- STOPTIME2. This code outputs the measurements collected on the current test problem. This is a separate piece of code to permit the including of problem specific output text.

There are two conditions where an ACEC user could consider modifying the timing loop code for better system performance, relative to the clock vernier (refer to the Reader's Guide, Section "CLOCK VERNIER", for discussion). On target systems with highly precise system clocks (on the same order as an instruction time), the vernier is superfluous and will simply make the test programs larger to compile without increasing the accuracy of the measurements; in fact some error will be introduced by the vernier on these systems. Also, on target systems where CALENDAR.CLOCK (or the CPU_TIME_CLOCK function when the user is measuring CPU time rather than elapsed time) is so slow that the system clock will tick many times in the time required to execute the function, the vernier computation will not increase the accuracy of the timing measurements. On such target systems, the use of the vernier makes the ACEC test programs larger and take more time to compile and execute. Measurements on such targets would be better with a simpler timing loop. The ACEC does not distribute a version of the timing loop with this modification for several reasons:

- Such targets are not currently in common use.
- Distributing modified timing loop would complicate the ACEC instructions.
- The relative cost advantage gained by not using a vernier on such targets is not large. Simplifying the timing loop will reduce the amount of code presented to an Ada compiler, making compilation times between different systems less directly comparable. This would be important for small test programs where the timing loop code is larger than the rest of the program.

- An ACEC user who wants to modify the timing loop to avoid the vernier should be able to do so after reviewing the discussions in the Reader's Guide, Section "CLOCK VERNIER", and the source code. If desired, an ACEC user could replace the entire timing loop with another mechanism. The use of INCLUDE to insert the timing loop is intended to simplify such modifications if needed.

There are three versions of the timing loop files with different suffixes. The differences between them are listed below:

- .CLOCK.

  This is the version for collecting elapsed time measurements.

- .CPU

  This is the version for collecting CPU time measurements.

- .SIZ

  This is the version which includes a call on a GETADR routine to measure code expansion sizes on systems which do not support the label'ADDRESS clause. It uses elapsed clock time for measuring time.

The command file COMPILE_BASELINE is set up to copy a set of timing loop files from *.CLOCK files to *.TXT files. Commented out options will copy other sets of timing files to *.TXT files. To select another option, the user should make the obvious adaptations. INCLUDE will insert the files named *.TXT into the test program.

3.2.4 Math Package

The ACEC test suite and analysis tools require a math library of elementary functions. Many MCCR applications will use elementary math functions, as do the ACEC analysis tools. A validated Ada system is not required by the LRM to provide a math package. The ACEC provides a portable implementation of a math library to permit execution on compilation systems which do not provide any math library support.

For the second release the ACEC test suite and tools use a compatible subset of the Association for Computing Machinery, Special Interest Group on Ada, Numerics Working Group (ACM SIGAda NUMWG) proposed specifications for an elementary math function library. The math library used in the second release differs from the math library used in the first release in the spellings of function names, the use of default parameters, the treatment of exceptions, and the removal of some functions in GEN_MATH which are not defined in NUMWG.

There are four approaches to adapting MATH, listed in decreasing order of preference. Each will be discussed in more detail in the following sections. The command file COMPILE_BASELINE.COM contains these four options, with three being "commented out." This

command file shows which units need to be compiled (and tested if appropriate) to produce a version of MATH for a new compilation system. ACEC users will need to tailor one of these options if they choose to use the vendor supplied math library. The version of MATH created by COMPILE_BASELINE will be tested by the MATHTEST program, which is compiled in the SETUP TEST PROGRAMS command file. The differences in the MATH packages can affect performance.

The ACEC uses the NUMWG package specification and recommends using the vendor's math library where it is available. The ACEC provides a portable version of a math library for use on systems which either do not provide a math library or where the provided math library is not adaptable. For example, the functions in the provided library may not be sufficiently accurate for project use. The performance degradation resulting from not using an optimized target specific math library can be considered a performance penalty applied against a system for **not** providing a math library.

- Where possible, instantiate the system provided version of the NUMWG package GENERIC_ELEMENTARY_FUNCTIONS.

  Where a compilation system provides a NUMWG package, it can be directly instantiated. Most projects would be expected to use this alternative where it is available. Where this alternative is available, execution times can be fast because the body of the package can be tailored to the target hardware.

- Where possible, interface with an implementation provided non-NUMWG math library.

  Where a compilation system provides a math library which is not compatible with NUMWG recommendations, adapt MATH and DBL MATH by providing bodies for the functions which "pass-through" calls to the provided non-NUMWG library.

  Where this alternative is available, execution times can be fast because the procedures might be implemented as interfaces to highly optimized routines which are tailored to the target hardware.

  Because there is no Ada language requirement to support a NUMWG package, a system which does not support it should not necessarily be downgraded unless the ACEC users have other requirements which suggest that support for a NUMWG package is desirable.

- Use GEN_MATH with MATH_DEPENDENT_PORT.

  The ACEC has developed a portable math package for use on systems which do not provide a usable math library.

  Instantiating this package to implement MATH should involve the least user effort on systems which do not implement the NUMWG recommendations. The performance of this option will probably be slower than for systems where the implementors have provided math packages tailored to target hardware.

Before compiling MATH.PORT, the user should verify that the MATH_DEPENDENT package is working by running the package DEPTEST.ADA. This package is discussed in more detail in the next section.

- Use GEN_MATH with a version of MATH_DEPENDENT tailored to the target system.

  Tailor a target specific version of MATH DEPENDENT. The program DEPTEST can be used to verify correct execution. Then use MATH DEPENDENT to compile the generic package GEN_MATH (which depends on MATH_DEPENDENT) and use GEN_MATH.ADA to compile the packages MATH.PORT and DBL_MATH.PORT — these units instantiate GEN MATH.

  GEN_MATH contains functions which will deliver accurate results for floating point base types where GLOBAL.DOUBLE'MACHINE MANTISSA is as large as 60 bits. ACEC users evaluating systems where DOUBLE'MACHINE MANTISSA is larger than this will either have to use vendor supplied libraries or adapt GEN_MATH and provide higher accuracy implementations of the functions. It is not expected that this will be a significant problem — there are systems which provide floating point types with more than 60 bits of mantissa, however typically they support several intermediate length types so that an Ada type declared with nine decimal digits of precision will be mapped to a hardware type with 60 or fewer bits of mantissa which is all the ACEC demands. The ACEC math library is not designed to provide support for any type with more than 60 bits of precision.

There are several versions of MATH (and DBL MATH) compilation units, distinguished by their suffix, as follows:

- MATH.ADA — version assuming NUMWG support.
- MATH.DEC — version which a user will adapt to pass-through function calls to an implementor provided non-NUMWG math library. The sample implementations provide interfaces to a FORTRAN math library and can be easily adapted to other systems which access a similar library.
- MATH.PORT — version using the ACEC provided GEN_MATH package.

The command file COMPILE_BASELINE.COM contains alternatives which compile MATH and DBL MATH using the different approaches. One is used and the others are presented as comments. An ACEC user adapting to another compilation system could use any of the alternatives as a model.

In each case, after developing a version of MATH and DBL_MATH, the user should verify the correctness of these packages by executing the programs MATHTEST and DBL MATHTEST. These programs report the numeric accuracy of the math functions. If the results of these tests indicate significant loss of accuracy, the user should investigate — the adaptation of

MATH_DEPENDENT may be incorrect, or the compiler may contain errors which should be isolated before proceeding with an evaluation.

This verification is important even when a vendor math library is used without modification. As reported in "The Need for an Industry Standard of Accuracy for Elementary-Function Programs," by C. Black, R. Burton, and T. Miller in ACM Transactions on Mathematical Software, Volume 10, Number 4, December 1984, there are inaccuracies in the subprograms for the elementary mathematical functions provided by computer manufacturers. This is a serious problem because many computer users assume that they are working with reliable routines and simply accept supplied packages without testing. This topic is discussed in Section 3.2.4.3.

The ACEC test suite and tools do not use all the functions and features of the NUMWG specifications. These functions are not provided in the portable package GEN_MATH. The NUMWG functions which are not required are:

- The hyperbolic (and inverse hyperbolic) functions
- The trigonometric (and inverse) functions with a cycle parameter
- The logarithm function with a base parameter
- The COT function

If the GEN_MATH package is instantiated with a constrained type, it will not operate as a NUMWG conformant would — it may raise an error when any intermediate variable in a computation is out-of-range, rather than only when initial or final values are out-of-range.

The specifications for the GEN_MATH package have been changed from the first release of the ACEC to make it compatible with the NUMWG recommendations. Users who have built personal test problems using MATH or DBL_MATH will have to adapt these problems to use the modified package GEN_MATH. The changes made to GEN_MATH (and the programs which call on it) were:

- The spellings of the following function names were changed as indicated:

```
LN     to  LOG
ASIN   to  ARCSIN
ACOS   to  ARCCOS
ATAN2  to  ARCTAN
ATAN   to  ARCTAN
             (ARCTAN has two parameters, the second
              parameter has a default value of 1.0,
              making it serve for both ATAN and ATAN2)
```

- The functions MIN, MAX, and SGN were removed from GEN_MATH because they are not part of the NUMWG specifications.

  Versions of these functions are now defined in GLOBAL for the types GLOBAL.REAL, GLOBAL.DOUBLE, GLOBAL.INT, and GLOBAL.BIGINT for the use of the ACEC programs which need them.

- GEN_MATH was modified to raise the NUMWG defined exception ARGUMENT_ERROR where appropriate, instead of NUMERIC_ERROR as it had done.

- GEN_MATH was modified to follow NUMWG recommendations with respect to behavior of functions at boundary conditions. For example, following the NUMWG recommendations, the evaluation of LOG(0.0) should raise ARGUMENT ERROR; in the first release of the ACEC, GEN_MATH returned the largest negative value for this input.

The functions required by the ACEC test suite are:

- "**" The power function, raising a real number to a real exponent.
- ARCCOS. The trigonometric arc cosine.
- ARCSIN. The trigonometric arc sine.
- ARCTAN. The trigonometric arc tangent.
- COS. The trigonometric cosine.
- EXP. The exponential function.
- LOG. The natural logarithm.
- SIN. The trigonometric sine function.
- SQRT. The square root function.
- TAN. The trigonometric tangent function.

The ACEC provides two programs (MATHTEST and DBL_MATHTEST) to test the accuracy of the math packages. These should be run to insure that the math libraries are performing correctly. These test programs might reveal accuracy flaws in a vendor library, or flaws in adapting the ACEC generic math package to a new target. For the ACEC test problems, it is sufficient if the math library is not grossly inaccurate — say not losing more than 10 bits of accuracy. However, the ACEC user should carefully examine the accuracy requirements of their applications before using a math library which is not essentially accurate to target machine precision. The NUMWG has recommended accuracy standards for the elementary

math functions in terms of permissible maximal errors over various ranges — the MATHTEST and DBL MATHTEST programs compare the observed errors against this standard and report if it is exceeded. The NUMWG also recommends that the value of functions for some specific values (usually for zero) be precise — the MATHTEST and DBL_MATHTEST programs compare the calculated results for these values to the recommend values.

The remaining subsections will discuss in more detail:

- Alternative methods to construct MATH and DBL_MATH packages.
- Testing of MATH and DBL MATH packages.

3.2.4.1 Alternative methods for MATH

The following sections discuss details of the alternative methods of providing a MATH package.

3.2.4.1.1 Instantiate system provided NUMWG package

Where provided, a NUMWG package should be straightforward to instantiate and efficient to use.

3.2.4.1.2 Adapting to an existing non-NUMWG math library

This alternative involves providing a package which "passes-through" a function call by interfacing to an implementor provided routine, mapping name changes, exception processing, and argument definitions as required. The following code shows how this would appear for the ARCTAN function on DEC Ada.

```
with global; use global;
package math IS

argument_error : exception ;

...

function arctan (y: real; x: real := 1.0 ) return real;

...

end math;
```

```
-----------------------------------------------

with math_lib;
package body math is

package vms_math_lib is new math_lib( real ) ;

...

  function arctan (y: real; x: real := 1.0 ) return real is
  begin
    if x = 1.0 then
      return vms_math_lib.atan(y);
    else
      return vms_math_lib.atan2(y,x);
    end if;
  exception
  when others => raise argument_error;
  end arctan;

...

end math;
```

It might appear that by promoting the use of an implementor provided math library (whose body is probably not written in Ada) that the ACEC test problems which call on math functions would not be performing comparable tasks on different systems. In general, a fair test problem would execute the same Ada source on all systems being compared. However, it has been traditional for languages which have specified a math library (such as FORTRAN) for implementors to provide users with interfaces to a math library which is not always written in the source language — there is often one optimized version of a math library which all languages reference. It is anticipated that Ada implementors will follow this tradition and provide math libraries. If a standard package specification is adopted by the different implementors, it will be simple to transport applications If and when this happens, applications which use elementary math functions and are concerned with performance will use the provided math libraries (without regard for what language the bodies of the math packages are written in) and the performance of a system executing a version of math functions coded in portable Ada would be of little intrinsic interest. Even now when not all implementations provide access to math libraries from Ada, for those which do, it is the performance of these libraries which is of

most interest to users because they are the libraries which will be used in performance sensitive applications. For applications concerned with portability, there will have to be a simple way to interface with the provided libraries and to verify their accuracy.

For a compilation system which provides a math library which does not contain all the functions the ACEC test suite requires, an ACEC user might: adapt the ACEC portable math library to provide the missing functions (and access the implementor library for the functions which are provided); use only the portable math library; or use the implementor library and not run the test programs which use the unsupported functions. An implementor provided math library might not handle exception conditions in a comparable manner — rather than raise an exception for an invalid argument (as the ACEC portable math library does) it might crash the program. Such behavior complicates the task of interfacing the ACEC test suite to an external library and can make the MATHTEST program, for one, impossible to execute without source code modifications.

An ACEC user can construct a version of MATH (or DBL_MATH) with interfaces to a vendor provided math library by writing a package MATH (or DBL_MATH) which specifies the functions and provides bodies for each which return the value of a call on the appropriate vendor library function.

An example of such an adaptation for the DEC Ada compilation system is provided on the distribution tape in the files MATH.DEC and DBL_MATH.DEC.

If an implementation provides an elementary function library in a package named MATH, it will be awkward to use this approach because defining the package MATH to be used in the test suite will override the definition of MATH provided. A user may be able to copy the source of the provided math library and rename it, or may be forced to modify the source of the ACEC programs which reference the package MATH (to name a package with a different spelling, permitting MATH to be reserved by the implementation).

3.2.4.1.3 GEN_MATH with portable MATH_DEPENDENT

To provide a math library for target systems which do not provide one, the ACEC distributes a portable version of a generic math package which can be readily adapted to additional targets.

The math package provided is based on the book Software Manual for the Elementary Functions by William J. Cody, Jr., and William Waite, published by Prentice-Hall in 1980.

The ACEC math packages MATH and DBL MATH depend on two generic packages:

- MATH_DEPENDENT:

  This is a representation dependent generic package which provides functions which permit the manipulation of fields of floating point numbers. It is instantiated using the declared types.

  It is discussed in more detail later.

- GEN_MATH:

  This is a generic package which instantiates MATH_DEPENDENT and contains the algorithms for the supported elementary math functions.

The ACEC math package refers to a package MATH DEPENDENT which provides access to the following three attributes of a real number:

- IntExp (x) which returns the integer representation of the exponent in the normalized representation of its floating-point number parameter. For example, IntExp (3.0) = 2 on binary machines because 3.0 = 0.75 * (2**2).

- Adx (x, n) which adds N to the integer exponent in the floating-point representation of X, thus scaling X by the N-th power of the radix. For example, Adx (1.0, 2) = 4.0 on binary machines because 1.0 = 0.5 * (2.0**1) and 4.0 = 0.5 * (2.0**3).

- SetExp (x, n) which returns the floating-point representation of a number whose significand is the significand of the floating-point number x, and whose exponent is the integer n. For example, SetExp (1.0, 3) = 4.0 on binary machines because 1.0 = 0.5 * 2.0**1 and 4.0 = 0.5 * (2.0**3).

There are two different approaches to implementing MATH DEPENDENT: one which is representation dependent and which directly manipulates the bit fields within a floating point number (this must be tailored to each target since it depends on the floating point number representation); and one which implements the attribute functions without "bit tweaking" using only the values of floating point number. The latter is the representation independent approach discussed in this section.

The portable version of MATH_DEPENDENT is coded by using operations on the floating point values without directly manipulating the bit patterns used to represent the values. To see how this is possible, consider the three functions exported by MATH_DEPENDENT in turn.

- Function ADX:

  This function is straightforward to implement by multiplying or dividing by an appropriate power of two. It can be tolerably efficient using a precomputed array containing the powers of two.

- Function SETEXP(X,N):

  Using the function INTEXP to determine the power of two of a floating point value, the result of SETEXP can be computed as

```
return ( X / 2.0 ** INTEXP(x) ) * 2.0 ** N ;
```

which is representation independent. This formulation is intended for clarity and would be rather slow if coded as shown. The exponentiation can be efficiently calculated using an array of the powers of two.

- Function INTEXP:

  The implementation of this function is the key to the proposed representation independent implementation. It is possible to directly determine the largest power of two greater than or equal to a floating point value (which will be the value of the binary exponent of the value) by searching an array containing powers of two instead of manipulating the bits of the floating point number representation. This will not be as efficient as a direct "bit manipulation" approach, but it is independent of where exponent fields are located in floating point numbers.

The functions must be coded carefully to avoid numeric overflow or underflow.

3.2.4.1.4 GEN_MATH with tailored MATH_DEPENDENT

The ACEC Software Product distribution tape contains the following versions of the file MATH_DEPENDENT.

.PORT	portable version	
.DEC	DEC Ada	VAX HOST/VAX TARGET

The Ada model numbers are defined in terms of binary radix. For a target machine with a non-binary radix, the error bounds produced by using the Ada model numbers will not be as tight as they are with binary radix targets. This will be most apparent in MATHTEST and DBL_MATHTEST which verify the accuracy of the math library. These programs use attributes MACHINE_MANTISSA, MACHINE_EMAX, and MACHINE_EMIN to obtain the properties of the target machine used to calculate tolerable error bounds. For a non-binary radix machine representation, the value of these attributes will be the smallest binary value which is consistent with the actual representation. Using this definition in MATHTEST will permit (slightly) more numeric errors in the implementation of the math functions before errors are reported.

In the following sections, two separate points are discussed. The first discusses the types of modification which may be necessary to MATH_DEPENDENT, and the second discusses how various types of errors in the implementation of MATH_DEPENDENT would show up in DEPTEST results.

The package MATH DEPENDENT must be adapted to reflect both the characteristics of the target machine floating point hardware and the facilities which the Ada compilation system provides to manipulate bit fields in floating point variables.

The size and location of the sign, exponent, and mantissa of a floating point number are critical, as are other representation details such as the encoding of the exponent field (biased, sign magnitude, or complement number representation). This information should be extracted from the documentation on the target machine. It is often included in Appendix F.

Once information on the floating point representation is determined, there may still be a problem in coding MATH DEPENDENT. The fundamental reason is that Ada is designed to be portable and system dependent operations are not universally supported.

It is possible for an ACEC user to implement a tailored version of MATH_DEPENDENT by interfacing with an assembler coded routine. This might produce the fastest execution speeds.

There are several approaches to adapting MATH_DEPENDENT to a target system.

- The cleanest approach is the use of record representation clauses to treat the fields of a floating point number as integer (sub)types. Adaptation will involve modification to reflect the target representation. Remember that different compilers for the same target hardware may choose to number the bytes in a record differently (left-to-right vs right-to-left). Record representation clauses are a Chapter 13 feature which is not universally supported.

- To isolate fields in a floating point value, it is necessary to sidestep normal Ada type rules. This can be done by:

  - Defining several access types which point to integer and floating point objects and arranging for them all to point to the *same* actual objects. That is, instantiating UNCHECKED_CONVERSION between the access types (pointers to integers and pointers to float) and as part of system setup, initializing all the pointers to the same actual location.

  - Using instantiations of UNCHECKED_CONVERSION, either between floating point types and integer types, or between scalar types and record types. This approach was used for the DEC Ada version of MATH_DEPENDENT.

  Bit field extraction can then be coded using integer arithmetic:

  - divide and MOD to extract fields (however, a divide of a negative value in a machine using two's complement integer arithmetic is **not** a shift);
  - multiply by powers of two to shift left;
  - add to OR fields (after insuring that the field in one operand is zero);
  - negation to complement bits.

  All these methods have disadvantages. The first is the most straightforward, but may not compile when the sizes are different (even though the conversion between different

sized objects occurs in a piece of code which will not be executed when the sizes differ). The second alternative relies on the PRAGMA SUPPRESS being honored; however, a compiler which determines at compile time that a constraint violation would occur when a statement is executed may generate a compile time error (warning) and generate code which would raise the CONSTRAINT ERROR exception at execution time. SUPPRESS grants a compiler permission to omit checking, but the LRM explicitly allows compilers to ignore a pragma SUPPRESS (LRM 11.7, paragraph 20).

3.2.4.2 DEPTEST

DEPTEST is a program which tests the functions in MATH DEPENDENT with a range of arguments which will expose many of the potential errors in the implementation of these functions. The program contains a series of statements which call on the functions and compare the results returned to the correct answer. In the DEPTEST output file, the discrepancies are flagged with a string "<<< ERROR >>>" starting in column 65, making them easy to detect.

Below are listed several symptoms of errors which might be reported by DEPTEST, and some candidates for what the underlying source of the errors might be:

- If IntExp returns a constant for all values, the function is probably not extracting the right bit field from the number. Perhaps parameters are grossly wrong (e.g. using record representation clauses, the bit numbering is backwards) or there are byte ordering problems (machine architectures can number bytes from the high end or the low end; and when the target orders differently than the programmer expected it will appear that the bytes have been interchanged).

- If IntExp returns a value which is a constant power of two off, the exponent bias is probably not set correctly.

- If SetExp or AdX modify any bits of the mantissa, the computations to adjust the exponent field are wrong. When the computed results are not some power of two different from the expected results, the exponent field is not being isolated properly. Check for byte-interchange, bit numbering and field sizes.

- If AdX returns a value which is a constant multiple of the correct value, the exponent field location may be a bit or two off.

- If the result of SetExp is off by a constant power of two, the exponent bias may be wrong. When AdX works properly an invalid bias is a likely cause of a constant factor error.

- Negative values, of either the floating point value or of the exponent field are given special processing. If results for negative values are wrong, the code for processing negative values needs to be reviewed.

- Optimizing compilers may do strange things with these functions. Consider the following example which occurred on one compiler during development. The compiler noticed that the SetExp function assigned to a float (through an access type), did an UNCHECKED_CONVERSION of the access to float to another access type and performed some manipulations on that second type, and finally returned the float value pointed to by the first access type. The compiler performed flow analysis and decided that there were no modifications to the value pointed to by the first access type before it was returned and so the load of the modified result could be "optimized" away as invariant. This transformed the SetExp function into an identity function and made it useless. The compiler vendor agreed after inspection that the UNCHECKED_CONVERSION should have made the flow optimizer aware an alias had been created which could modify the values of the object pointed by the access type, and has modified the compiler to be aware of this fact. The immediate workaround to this problem was to insert an external procedure call between the assignment to the access type variables, which worked, but increased the execution time of the functions.

  A similar condition may arise in other compilation systems and being warned about the possibility, users may not be totally frustrated in developing workarounds.

If DEPTEST does not initially work correctly, and the errors observed do not fit one of the patterns described above, the first thing a user should try is to compile the package MATH DEPENDENT with no optimizations. On some systems, requesting support for a debugging option is a good way to suppress optimizations. The package may work then. If it does, the ACEC user may decide to: isolate the difference optimization makes, perhaps by examining the listing of the machine code generated; or refer the package to the compilation system maintainers for correction; or simply not use any optimization options on that compilation system.

These functions must work properly for GEN_MATH to work. It is futile to try to verify that GEN MATH is correct by running MATHTEST until MATH_DEPENDENT has been verified with DEPTEST. It is much simpler and faster to isolate and correct errors in the MATH_DEPENDENT function using DEPTEST than using GEN_MATH. It is much easier to debug a function when the expected results are checked by a test program, than when a programmer must observe that a complex function (such as LN) sometimes returns a wrong result and isolate the problem in that function to an error in a low level function which it calls upon. Most of the problems uncovered while transporting GEN_MATH onto new compilation systems by MATHTEST have been due to errors in the functions in the MATH DEPENDENT package.

3.2.4.3 MATHTEST

Once MATH and DBL_MATH have been adapted to the target system, they should be tested to verify correct operation. The programs MATHTEST and DBL_MATHTEST are provided to do this. These programs test various identities and special cases for the elementary math functions, and output the number of bits in error in the computation of the function. There are several groups of tests, covering ranges of the functions, and over each range, the program computes 2,000 sample points distributed at random (usually dividing the range into 2,000 intervals and selecting a point within each interval using a uniform random distribution). On each range, the program displays both the maximum relative error and the root-mean-square error in terms of the number of bits of precision lost. The root-mean-square is the square root of the sum of the squares of all the errors — it is commonly used as a measure of the "average" error in the set of numbers.

MATHTEST and DBL_MATHTEST will write an error message whenever the maximum error is larger than what the NUMWG specifications recommend, whenever some of the specific identities that the NUMWG recommends fail, and when selected examples which should (or should not) raise an exception, based on the NUMWG specifications, do (or do not).

These programs are an adaptation of the work of Cody and Waite. The interested reader is referred to their book, cited in Section 2.2, for details.

MATHTEST requires the package RANDOM, which contains a random number generator. There are two versions of RANDOM, one using 16-bit integers (RAN16) and another using 32-bit integers (RAN32). For systems which support 32 bit integer types, RAN32 should be used. This package uses a linear congruence pseudo-random number generator and should be fairly fast. For compilation systems which do not support integer types with that range, RAN16 must be used. That package uses a Tausworth random number generator with a shuffling technique as described in "Improving a Poor Random Number Generator," by C. Bays and S. D. Durham, ACM Transactions on Mathematical Software, Volume 2, Number 1, March 1976. RAN16 should be fairly portable, although it assumes that it can perform an UNCHECKED_CONVERSION between a packed boolean array of 16 elements and an integer type. Since the only purpose of either generator is to provide a source of random values for testing the math library, the quality of the generator required is not as strict as would be necessary for more exacting purposes. MATHTEST results should show very few bad tests, and the loss of significant bits should not be larger than the NUMWG recommendations.

An ACEC user may be presented with a choice between using a fast implementation provided math library on which MATHTEST detects errors, and a GEN_MATH based version which is slower but with smaller errors and which processes exceptions as the NUMWG specifications recommend. This is not a trivial choice. The ACEC test suite and support tools do not strongly rely on the NUMWG recommended exception processing and most math libraries not developed specifically to conform to the NUMWG recommendations will not conform with these recommendations. If the treatment of exceptions is the only discrepancy reported

by MATHTEST and DBL_MATHTEST in testing an adaptation of a vendor provided non-NUMWG math library, users may decide applications they develop would use the vendor library. That is, easy portability to other NUMWG based systems may not be a concern. Such users would properly not consider testing using anything but the supplied math library. If MATHTEST and DBL MATHTEST detect large numeric errors, an ACEC user must decide, based on the expected usage of the math library, which math library to use for testing.

3.2.5 Space Measurement

Users may be concerned with two measures of code size: (1) code expansion and (2) Run Time System (RTS) size.

3.2.5.1 Code expansion measurement

Depending on the system, there are two alternative methods available for measuring code size expansion. Both are discussed below.

3.2.5.1.1 Using Label'ADDRESS

If the system under investigation supports the ADDRESS attribute, then this measurement will be straightforward. This attribute is a Chapter 13 feature which has not been required for validation. Some systems used during ACEC development accepted the attribute, but always returned zero for a value.

3.2.5.1.2 Using the GETADR assembly routine

The ACEC uses the label'ADDRESS attribute to collect code expansion size measurements. Not all implementations support this attribute correctly. Some compilers may generate an error or warning message when processing the label'ADDRESS attribute in the source, and some accept the syntax and return random results. The last case can be detected by the printing of random values for sizes. An ACEC user should review documentation on the Ada compilation system being tested to see if the attribute is supported.

If the ADDRESS attribute is not supported, an assembly language procedure can be written which returns the address of its caller. The details of such an assembler routine are system dependent. The following example works on DEC Ada Version 2.0 under VMS.

```
.TITLE      GETADR
;
;           This procedure returns the value of the calling
;           modules call address in R0.
```

```
;
        .PSECT    CODE        PIC, SHR, NOWRT, LONG
;
;
        .ENTRY    GETADR    ^M<>
;
;       Move the PC contents to R0
;
        MOVL      16(SP), R0
        RET
        .END
```

This program is on the distribution tape named GETADR.MAR.

To use the GETADR function there is a set of files on the distribution tape with the suffix ".SIZ" which incorporate this modification for collecting code expansion size and elapsed time measurements on DEC Ada. In versions 1.5 and earlier of DEC Ada, the label'ADDRESS attribute always returned the value zero, making it useless for measuring code expansion sizes.

It may be necessary to modify the linker commands to specify the library where the assembler object exists. Details of adapting this to other implementations depend strongly on the provisions for calling assembler routines from Ada programs and are highly system dependent.

An ACEC user may decide that code expansion size measurements are not of primary concern. Such a user may ignore size measurements and simply report zero for all labels.

To compute the code expansion size, the INCLUDE tool inserts a unique label after the last line of the insertion for STARTIME; another unique label before the first line of STOPTIME0; and generates an assignment statement setting the variable GLOBAL.EXPANSION_SIZE to the difference between the addresses of these two labels for STOPTIME2. If the label'ADDRESS clause does not work, the user must modify these statements before compiling them. The INITTIME file contains a usage of the label'ADDRESS clause to compute the null loop code expansion size, and this statement may also need to be modified. It is possible to modify these statements by editing the expanded code files, but it is much less effort to modify the INCLUDE program and the INITTIME file to avoid use of the label'ADDRESS clause.

3.2.5.2 RTS size

This information may be available from the load map produced by the linker. If not, the user will need to get in touch with the compiler vendor. The size of the run-time system will no doubt vary depending on the features used. For example, tasking programs will require many run-time facilities not otherwise needed.

Some operating systems provide a tool (for example, the SIZE command on some UNIX implementations) to examine a relocatable or executable file and report the size of the code sections. If such a tool is available, it could be used to obtain size information.

The following is a (truncated) sample from a link map generated by the TeleSoft VAX hosted/VAX targeted compiler.

```
REED                                                    15-APR-1988 15:36       VAX-11 Linker V04-00

                                +-------------------------+
                                ! Object Module Synopsis !
                                +-------------------------+

Module Name      Ident              Bytes      File                                    Creation Date      Creator
-----------      -----              -----      -----                                   -------------      -------
 TSADARTL                                0 [TELEGEN2.TSADA315]TSADARTL.EXE;1      7-OCT-1987 20:26  VAX-11 Linker
V04-00
 LIBRTL          V04-001                 0 SYS$COMMON:[SYSLIB]LIBRTL.EXE;2       22-MAY-1987 23:12  VAX-11 Linker
V04-00
 SCRSHR          X-1                     0 SYS$COMMON:[SYSLIB]SCRSHR.EXE;1       22-MAY-1987 23:16  VAX-11 Linker
V04-00
  .MAIN.                               828 USER2:[LEAVITT.TELESOFT]REED.OBM;1    15-Apr-1988 15:35  TeleSoft Ada
 REED                                19360 USER2:[LEAVITT.TELESOFT]MYLIB.OLB;2  15-Apr-1988 15:31  TeleSoft Ada
 ECC                                  1628 USER2:[LEAVITT.TELESOFT]MYLIB.OLB;2  15-Apr-1988 15:29  TeleSoft Ada
 GLOBAL.CPU_TIME_CLOCK
                                       140 USER2:[LEAVITT.TELESOFT]MYLIB.OLB;2  12-Apr-1988 15:07  TeleSoft Ada
 GLOBAL                               7616 USER2:[LEAVITT.TELESOFT]MYLIB.OLB;2  12-Apr-1988 15:06  TeleSoft Ada
 CGSMAIN.OBJ     V1.0                   21 [TELEGEN2.TSADA315]CGSMAIN.SAV;1      9-SEP-1987 17:22  VAX/VMS Macro
V04-00

                                +---------------------------+
                                ! Program Section Synopsis !
                                +---------------------------+

 Psect Name      Module Name      Base     End          Length          Align          Attributes
 ----------      -----------      ----     ---          ------          -----          ----------
 $ADATA                        00000200 0000086F 00000670 (       1648.) LONG 2   PIC,USR,CON,REL,LCL,NOSHR,NOEXE,
RD,  WRT,NOVEC
                 .MAIN.        00000200 0000029B 0000009C (        156.) LONG 2
                 REED          0000029C 0000029F 00000004 (          4.) LONG 2
                 ECC           000002A0 0000030F 00000070 (        112.) LONG 2
                 GLOBAL        00000310 0000086F 00000560 (       1376.) LONG 2
 $ACODE                        00000A00 00007713 00006D14 (      27924.) LONG 2   PIC,USR,CON,REL,LCL,  SHR,  EXE,
RD,NOWRT,NOVEC
                 .MAIN.        00000A00 00000C9F 000002A0 (        672.) LONG 2
                 REED          00000CA0 0000583B 00004B9C (      19356.) LONG 2
                 ECC           0000583C 00005E27 000005EC (       1516.) LONG 2
                 GLOBAL.CPU_TIME_CLOCK
                               00005E28 00005EB3 0000008C (        140.) LONG 2
                 GLOBAL        00005EB4 00007713 00001860 (       6240.) LONG 2
 $CGSIN                        00007714 00007728 00000015 (         21.) LONG 2   PIC,USR,CON,REL,LCL,  SHR,  EXE,
RD,NOWRT,NOVEC
                 CGSMAIN.OBJ   00007714 00007728 00000015 (         21.) LONG 2

                                +------------------+
                                ! Symbols By Name !
                                +------------------+
```

This portion of the map displays information on the size of the program (which is 828 + 19,360 + 1,628 + 140 + 7,616 + 21 bytes). The runtime library (RTL) size is zero because it is a shareable image. Not all link maps are this helpful.

3.2.5.3 Assembly language procedure

There is one test problem, SS747 in program S0747T47, which uses PRAGMA INTERFACE to call on an assembly language procedure (which then does a simple return). This problem will have to be adapted to each implementation, using the assembly language of the target.

Below is an example of an assembly language procedure in MACRO, the assembly language for the VAX, which complies with the linkage conventions for the DEC Ada compiler.

```
.TITLE      NULL PRG
.PSECT      CODE        PIC, SHR, NOWRT, LONG
;
.ENTRY      ASMNUL
            RET
.END        ASMNUL
```

Not all implementations will support the interface to procedures coded in assembler. Even when supported, if a project has determined that it will **not** be ir :luding assembler code procedures in its applications, this problem will not be of importance or interest and may be skipped.

When linking to routines coded in assembler, it will often be required to specify additional information to the linker. This is *implementation dependent*.

3.2.6 System Parameters

System parameters are variables, named numbers, and compiler switches which an ACEC user may modify.

There are several groups of such parameters which impact the ACEC timing routines, the compilation system, and the runtime system. The parameters *available for the second and* third categories are implementation dependent, but some general advice is given.

Users should not have to vary any of these settings.

- There is a set of constants which control the number of executions of the timing loop a problem will perform. The named numbers BASIC ITERATION COUNT, MIN ITERATION COUNT, and MAX ITERATION COUNT in the package GLOBAL limit the timing loop. The first limits the *maximum number of iterations* of the inner timing loop. The next two set lower and upper bounds on the outer timing loop. Setting small values for these constants will limit the total amount of time the execution of the test suite takes; however, it may result in increased error in the measurements (and an increase in the number of test problems flagged as not being within the requested statistical confidence levels). A full discussion of the timing loop and the use of these constants is contained in the Reader's Guide, Section "CLOCK VERNIER".

GLOBAL.TIMER_TOLERANCE is the requested error tolerance, initially set at 5%. The statistical confidence level is set in the array GLOBAL.TIMER_CONFIDENCE_LEVEL, initially set for 80% and the associated constant array GLOBAL.T_VALUE which contains entries for the t-distribution. To modify the confidence level, a user will have to replace the values of this array with values from the t-distribution for other confidence levels which can be obtained from any standard statistical text. See the Reader's Guide, Section "CLOCK VERNIER", for more details. It is not expected that many ACEC users will modify the confidence levels, although some instructions for doing this are contained as comments in the source code for the package GLOBAL.

- On some systems, directives are given to the compiler by options specified at the invocation of the compiler. These are implementation dependent and inherently non-portable. A user may ask what settings are appropriate for a new system. Such options have included: optimization levels; requests to perform automatic inline subprogram expansion; information that subprograms are nonrecursive (permitting alternative and faster linkages); requests for cross reference listings; program debug support; the name of the program library to be used; etc. In general, the options specified should be those which produce the faster execution, consistent with the values of the pragma SUPPRESS options given in the test programs. The primary purpose of several test problems is to explore the performance of constraint checking and exception processing, or to observe the effect of requesting PRAGMA OPTIMIZE(SPACE). Compiler options should not overwrite these. Note that some of the test problems are recursive, and specifying to a compiler that all subprograms are non-recursive will introduce errors and is not permissible.

- The default stack space may be inadequate, for either the main program or user defined tasks. While user defined tasks can specify a T'STORAGE SIZE for a task activation, there is no syntax in the language to specify a size for the main program. Many systems provide a method, often in the linker, to reset a default size for tasks and for the main program. Adjustment of this parameter may be necessary to get some of the programs to run. An ACEC user may choose to mark a problem as failed when the default settings are not sufficient. This is acceptable, but readers should understand that when a problem is marked as failed due to capacity limitations, that may simply mean the tester did not find a setting which permitted execution — such a setting may *exist*, but requires experimentation to find. Each ACEC user must decide how much effort to invest exploring adaptation parameters when an implementation's default sizes are insufficient to permit execution. In some cases the problem with memory space is not that one individual task overflows its allocation, but that the total memory requirements of the active tasks exceed available memory. It may be possible to reduce default task sizes to permit all concurrent tasks to fit in memory and still have sufficient memory allocated to each task so that they do not overflow.

3.2.7 Exceptional Tests

3.2.7.1 Interrupts

To execute the interrupt tests (named INT_0, INT_1, ...) the compilation system must support tying tasks to interrupts, as in LRM 13.5.1. These problems tie task entries to hardware interrupts and use LOW_LEVEL_IO to trigger the interrupts. To execute these programs, the target system must support both LOW_LEVEL_IO and tying task entries to interrupts. Many systems will not support either.

On implementations which support tying tasks to interrupts but do not support LOW LEVEL IO, the problems may be adapted to use a routine coded in assembler language which will trigger the interrupt when the assembler routine is called.

The LRM states that the conventions that define the interpretation of a value of the type SYSTEM.ADDRESS (occurring in the address clause) is implementation-dependent. One natural interpretation for the argument of the **USE AT** clause is as an interrupt level.

Many Ada implementations on multiprogramming targets will not support interrupt entries. Programs which directly manipulate hardware are generally privileged processes on current operating systems. When such a program malfunctions, it can crash the operating system. Many operating systems force such operations to go through operating system calls so that the operating system can maintain control and protect its own integrity.

4 TEST SUITE COMPILATION

The ACEC test suite contains a number of individual test programs which must be compiled, linked, and executed.

4.1 ORDER OF COMPILATION

The ACEC test suite is organized as one executable program per file. Some programs contain more than one test, but each program is independent of the other programs. If more than one compilation unit is required, these are included in the source file as separate compilation units, in valid compilation order. For example, the source file for a program which uses a unique package will contain both the source for the package and the source for the program.

There are two exceptions: MATH and GLOBAL. Instructions for modifying and compiling MATH were presented above. GLOBAL may also need to be modified before it is compiled. The package GLOBAL is used by all of the benchmark programs. It must be compiled before any of the timing programs can be compiled. Depending on system problems, GLOBAL may need to be changed. If the system does not support 32-bit integers then the declarations involving "BIGINT" should be commented out. If the system does not support real numbers longer than 32-bits the declarations involving "double" should be excluded. MATH uses GLOBAL, so GLOBAL must be compiled first.

The use of the Ada rules for combining source files with multiple compilation units simplifies the construction of the test suite and the steps necessary to transport it to new systems. In particular, the source files containing the test programs are essentially independent, and can be compiled, and recompiled, in any order which is convenient. Information about compiling a package or library unit before the other units which WITH it is implicit in the ordering of the units in the source file. A user will not have to modify a command file to reflect a particular order of compilation.

4.2 SYSTEM DEPENDENT TESTS

Some tests use system dependent features, such as TEXT_IO, or Chapter 13 features which may not be available on all target systems. Other tests require extended precision floating point numbers. The Version Description Document (VDD) Appendix VII, "SYSTEM DEPENDENT TEST PROBLEMS", provides a list of tests which use various system dependent features. If a feature is not supported on your system, you need not try to run the tests that utilize it, and should expect failure if the tests are attempted.

4.3 USING INCLUDE

After the system dependent modifications are finished, the INCLUDE tool will insert the (possibly modified) timing code into the benchmark programs. INCLUDE assumes that "inputFile" and "outputFile" are defined as logical names prior to its execution. The suffix of "inputFile" is assumed to be ".a"; the suffix of "outputFile" is assumed to be ".ada". This use of suffixes makes it easy to see if a text file has been INCLUDEd or not. On systems which do not support file suffixes, the user must adapt whatever conventions the host system supports. Some UNIX based Ada compilers require the input to the compiler have a suffix ".a" instead of ".ada." Testing on such a compiler will require the suffixes be adapted to match the requirements of the compiler. There are Ada compilation systems running under operating systems which do not support the concept of a file suffix (one example is the Rational R2000). On such a system more extensive modifications are required to INCLUDE: the appending of the ".txt" default suffix in the body of INCLUDE must be removed. There are operating systems which do not support the concept of "logical names," including some versions of UNIX. On these systems, while it would be possible to modify INCLUDE to prompt the user for the names of the input and output files, the sample command files provided for the UNIX systems are sufficient to make INCLUDE work without modifying the source for INCLUDE. The command file to execute INCLUDE uses UNIX pipes to copy the UNIX standard input file into a known name so that the Ada program can read it, and copies the output from the INCLUDE program from a known name into the UNIX standard output pipe. Another modification would have been to make the INCLUDE Ada program read from Ada STANDARD_INPUT and write Ada STANDARD_OUTPUT which should be mapped to the UNIX system's STANDARD_INPUT and STANDARD OUTPUT; however, this would raise the problem of forcing the INCLUDE program's error messages (which are currently written to Ada's STANDARD OUTPUT) to appear within the output file which would be awkward. ACEC users who are more experienced in UNIX shell programming may be able to design more elegant solutions to this problem. The method used has the performance disadvantage of copying the input and output files, but it does accomplish the intended task without modifying the source text of the INCLUDE program.

The INCLUDE program reads the source text, looking for lines that begin with PRAGMA INCLUDE. These lines are copied as comment lines, the file specified in the PRAGMA INCLUDE is inserted, and the rest of the source text is copied as is into the new file. The program should now be ready to compile, with timing code in place. This is shown below:

Figure 5: USING INCLUDE

In the figure, the source for a test program "P.A" is run through the INCLUDE process, generating an expanded text file "P.ADA" which is then passed through the compiler (and linker), generating an executable file "P.EXE".

The program INCLUDE also inserts unique labels between the beginning and end of the timing loop (at the end of STARTIME and at the beginning of STOPTIME0) and inserts an assignment statement setting EXPANSION SIZE to the difference between these labels (using the label'ADDRESS attribute) and converting the result to type GLOBAL.INT. Several implementations tested do not support the label'ADDRESS attribute, and the presence of this assignment on these systems can generate a compilation error, preventing execution of the program.

Users have two basic choices.

- They could decide not to collect code expansion measurements at all. This is certainly simple to do and some classes of users are not very interested in code expansion (or at least would only be interested in code expansion sizes when the execution time measurements indicate an acceptable performance level).

  To ignore code expansion sizes, two changes need to be made. First, modify INCLUDE so that the assignment statement will set CODE_EXPANSION to zero, rather than to an expression. Second, modify INITTIME so that the null loop code expansion size is printed as a constant zero and does not use the label'ADDRESS clause.

- The user can collect code expansion size by writing an assembler language function which returns as its value the address of the caller. This was mentioned in Section 3.2.5.1.2.

  After writing the function, the users can incorporate it into the timing loop code by modifying the included files: INITTIME, STARTIME, STOPTIME0, and the package GLOBAL.

  - INITTIME.

    Include in the null loop computation the determination of the NULL LOOP SIZE.

  - STARTIME.

    Include as the last statement in the file an assignment of GETADR to START_ADDRESS.

  - STOPTIME0.

    Include as the first statement in the file an assignment of GETADR to STOP_ADDRESS.

  - GLOBAL.

    Add declarations for the variables: START ADDRESS and STOP ADDRESS (of type SYSTEM.ADDRESS), and NULL_LOOP_SIZE (of type INT).

    Add a declaration for the function GETADR.

Modify the procedure STOPTIME2 to compute CODE_EXPANSION_SIZE prior to printing it by subtracting START ADDRESS from STOP ADDRESS.

Timing measurements obtained when the GETADR function is used will be more variable, since including two calls on an external function in the null timing loop will result in a longer running null loop time. This will tend to make it harder to measure the times of quickly executing test problems.

The distribution tape will include versions of GLOBAL, INITTIME, STARTIME, and STOPTIME0 with a suffix of ".SIZ" which is a version of the timing loop code which will collect code expansion measurements using the GETADR function for the DEC Ada compilation system. This can be used as a model for implementations which do not support the label'ADDRESS clause.

4.4 COMPILATION

The user is now ready to compile the test suite. This section will discuss problems which might be encountered in compiling the test suite and present some examples which show how the compilation of the test suite can be accomplished.

4.4.1 Potential Compilation Problems

This section delineates some of the potential problems which may arise and suggests workarounds when available. A complete list of tests with compilation difficulties on some systems is available in the VDD (Version Description Document) Appendix IV, "QUARANTINED TEST PROBLEMS".

Some test problems are implementation dependent. Some are tests for implementation dependent language features (such as tying tasks to interrupts) and some are tests which may exceed implementation defined capacity limits (such as problems using floating point types declared with 9 digits of precision). For the ACEC to span the major features, there are test problems using features not all implementations will support. These include the use of packing, UNCHECKED_CONVERSION, UNCHECKED_DEALLOCATION, record representation clauses, tying tasks to interrupts, length clauses, etc.

It is possible that running any new program through a validated Ada compilation system will uncover an error in that system. This is not expected to be a common occurrence, and the ACEC is not designed to try to find errors in implementations. If such errors are observed, a problem report against the Ada compilation system should be submitted.

It is possible that an Ada program, even though it has executed as expected on multiple validated Ada compilers, is erroneous. For example, a program may work as expected when library packages are elaborated in some particular order, but which fail when they are elaborated in one order which is permitted by the LRM. All the systems the program was run against with

satisfactory results may have succeeded because the implementation happened to select a fortuitous (but not guaranteed) order. If this is observed, an error report against the ACEC should be submitted.

Some Ada systems put arbitrary limits on the size of a procedure. If this difficulty is encountered while compiling a test program which contains multiple test problems (in particular, the SIMPLE test programs named SnnnnTmm, which contain up to 15 separate test problems per program), the program can be split into smaller files, taking care to copy any global declarations to each file.

Several compilation systems tested during development had the "user belligerent" (as contrasted to "user friendly") property of corrupting the program library system when a user aborted the compiler. The systems did not always inform users in an understandable way that the library was inconsistent, and sometimes simply acted strangely, refusing to accept programs which compiled without difficulties before the library crashed. This forced the creation of a new library and the recompilation of all prior units. Users who are testing a compilation system which they do not know to be robust may avoid problems by not aborting the compiler, or at least remember that aborting a compilation is a potentially hazardous operation which may cause strange behavior later.

4.4.2 Example Compilation Routines

Two examples of the control procedures necessary to accomplish these tasks will be available: one for VAX VMS and one for UNIX.

4.4.2.1 VAX/VMS

The following VAX/VMS command file (Compile_Acec.com) may be used as a model for other compilation systems. The command files called by this file are on the distribution tape.

```
$ set verify
$ set def       [leavitt.test]
$!-------------------------------------------------------------------!
$!  Rename the directory [leavitt.test] here and in all the other .COM !
$!  files to a local directory usable for testing.  This set of files  !
$!  for DEC Ada assumes that the users LOGIN.COM file includes an      !
$!  appropriate ACS SET LIBRARY command.
$!
$!  Compile the "baseline" of the ACEC Test Suite.  The "baseline" of  !
$!  the suite consists of Global, Ran, Gen_Math, Math, and Dbl_Math.   !
$!  These will need to be compiled before anything else, in the order  !
$!  given.                                                             !
$!-------------------------------------------------------------------!
$!
$ @Compile_Baseline
$!
$!-------------------------------------------------------------------!
$!  Next, compile the files necessary for the "Setup" tests of the     !
$!  ACEC.  These tests include "MathTest" which verifies the accuracy  !
```

```
$!  of your Math package on your system, and "TestCal" which verifies  !
$!  the accuracy of your system clock.                                  !
$!-------------------------------------------------------------------!
$!
$! @Setup_Test_Programs
$!
$!-------------------------------------------------------------------!
$!  Setup INCLUDE                                                    !
$!-------------------------------------------------------------------!
$!
$ @compile_tools
$!
$!-------------------------------------------------------------------!
$!  Compile the files that comprise the ACEC test suite.             !
$!-------------------------------------------------------------------!
$! @Compile_Test_Suite
$!
$ show time
$!
$ Dir/since=yesterday ACEC_Lib_Dir
$!
```

4.4.2.2 UNIX

The following UNIX Bourne shell script (Cmp_Acec.unx) may be used as a model for other UNIX based systems. Interested users can review the complete scripts as they are presented on the distribution tape.

```
#
# CMP_ACEC
#

PATH=$PATH:/usr/vads/bin     # set path to include VADS directory

#!---------------------------------------------------------------------!
#!  Rename the directories test_suite_dir, tools_dir, and work_dir     !
#!  here and in all the other .UNX files to local directories usable   !
#!  for testing.  This set of files assumes that the appropriate       !
#!  directories and Ada library have been created.                     !
#!---------------------------------------------------------------------!

test_suite_dir=/usr/people/lindsey/fqtlib
tools_dir=/usr/people/lindsey/fqtlib/tools_dir
work_dir=/usr/people/lindsey/fqtlib/test_work

cd ${test_suite_dir}

echo 'Cmp_ACEC:  Compile the ACEC: ps=' $$ >> ${work_dir}/cmp_acec.log
#!---------------------------------------------------------------------!
#!  Compile the "baseline" of the ACEC Test Suite.  The "baseline" of  !
#!  the suite consists of Global, Ran, Gen_Math, Math, and Dbl_Math.   !
#!  These will need to be compiled before anything else, in the order  !
#!  given.                                                             !
#!---------------------------------------------------------------------!
  cmp_base

#!---------------------------------------------------------------------!
#!  Next, compile the files necessary for the "Setup" tests of the     !
#!  ACEC.  These tests include "MathTest" which verifies the accuracy  !
#!  of your Math package on your system, and "TestCal" which verifies  !
#!  the accuracy of your system clock.                                 !
#!---------------------------------------------------------------------!
  cmp_tst_pr

#!---------------------------------------------------------------------!
#!  Compile the ACEC support files.                                    !
#!---------------------------------------------------------------------!
  cmp_tools

#!---------------------------------------------------------------------!
#!  Compile the files that comprise the ACEC test suite.               !
#!---------------------------------------------------------------------!
  cmp_ts
```

5 LINKING/DOWNLOADING THE BENCHMARKS

Linking the programs should be straightforward. The name of the main program is the name of the source file containing the test problem. This may need to be specified in the link commands to each system. Users should request a link map (sometimes called a memory map) so they can determine the size of the runtime system.

For self-targeted systems, loading a program is equivalent to running the executable file produced by the linker. The only problem expected is when the size of the load module exceeds memory capacity. Some capacity limits may be impossible to work around, such as when a program declares an array which is larger than the address space of the target machine. In other cases, it may be possible to adjust parameters to permit the program to execute. On some multiprogramming host systems, it may be sufficient to request a larger than default partition. On other systems, it may be possible to get the program to run by adjusting values associated with the default sizes for task activations and/or the main program, or with the size of the global heap.

Other complications arise with cross compilers. The file format of the load module may need to be modified to match that required by the target hardware. For example, on the 1750A several formats exist. This is system dependent and not peculiar to the ACEC test suite. Contact your hardware system implementor for help.

Potential difficulties include code files that are too large, particularly for targets with limited memory. For the simple tests, which include more than one test per executable unit, a workaround is available: divide the program into two or more executables. For the other tests, they will have to be marked as failed.

Since the time to download programs into target systems can take longer than the time to compile and execute them, considerable user time is saved by combining several test programs into one executable main program.

5.1 MERGING PROGRAMS

Where downloading is a very time consuming process, it may be possible to speed up the execution of the test suite by combining several test programs into one large program. The first thought on hearing this is that it will not help, since if each program takes five minutes to download, loading a program twice as large will take twice as long and the end result is not a gain. However, this is not always true. The Ada runtime system should only be loaded once for each program, and the size of the runtime system is often a large factor in the size of the executable code of a test problem. Also, the procedures to download and execute a program often take a considerable amount of operator time for each individual program.

Consider the following program template.

Using this template, several test programs can be combined into a larger program. As long

```
with program 1;
with program_2;
...
procedure mergetest is
begin
    program_1;
    program_2;
...
end mergetest;
```

Figure 6: PROGRAM MERGING

as the resulting program will fit within the capacity limits of the target system, the results of combining the programs should be the same; however, the total download time may be significantly smaller. This setup does not require *any* modifications to the programs to be merged. However, it will enter each program into the library system. This approach will work where the individual programs are compiled separately into the program library. In this case the mergetest program source file will not contain the source text for the individual programs, but only the "with" and the body of the main program.

A reasonable question to ask is whether using the method recommended here will result in different code being generated for the test problems. Each individual program need not be linked as a main program, and the construction of a merged main program will involve a separate link command. Good systems should not load multiple copies of the TEXT_IO package, even though it is "WITH"ed and instantiated in each of the smaller compilation units. The code executed for each merged test problem should be the same as the code executed when the problems are not merged. Targets where the execution time can vary with the memory location (for example, those with lower speed memories at high addresses) can observe different execution times when the test problems are run in a linked mode. This should be anticipated by users executing on such targets. Even without merging tests they should expect to see occasional anomalous results due to memory allocation — some problems may be large enough to force the allocation of some code into "slow" memory.

6 RUNNING THE BENCHMARKS

Ordinarily, running the benchmarks should be a straightforward process. All of these tests ran successfully on the five trial systems, except those that have been flagged in the Version Description Document Appendix IV, "QUARANTINED TEST PROBLEMS". If you are compiling and running on a self-targeted system, you may be able to set up an "exec" or "com" file which will include, compile, link, and run all of the normal benchmark tests without further effort.

6.1 POTENTIAL RUNTIME PROBLEMS

This section delineates some of the potential problems which may arise at runtime and suggests workarounds when available. A complete list of tests that failed while executing is available in the VDD (Version Description Document) Appendix IV "QUARANTINED TEST PROBLEMS" for the current version of the ACEC.

The following is a list of potential runtime problems:

- The range of valid task priorities is implementation defined, as stated in LRM 9.8 and 13.7. Several tests will not operate as designed if an implementation does not support priorities in the range 1 .. 3.

- Several tasking tests will fail if run on a runtime system which does not support pre-emptive priority scheduling. These include DELAYS, TASK44, and TASK45. These programs contain some checks and will generate a failure message if they detect that the scheduler is not pre-emptive.

- If the timing loop initialization code, INITTIME.TXT, finds that the value of SYSTEM.TICK is not consistent with observed clock measurements, it will report a discrepancy which should be examined.

- No output at all means that something fundamental is wrong. The time for the null loop should always be produced. If not, the timing loop is not working and the CALENDAR package should be suspected.

- No output for a particular test problem means failure. You may be able to isolate the problem. Almost certainly the system you are testing is at fault.

- Nonsense times, either very large or very small. These will often be flagged in the analysis stage by the MEDIAN program (see the Reader's Guide, Section "RANDOM ERRORS", for more information).

- Programs may fail and produce an Ada exception message, or a system failure message. While exceptions are raised in the course of running some of the tests, there are handlers for these "expected" errors. A program running on one of the smaller systems may run out of memory and raise a STORAGE_ERROR. Some of these failures may be rectified by changing the stack size or by specifying a length clause; others in the tasking group may be run by setting the allocated storage with T'STORAGE_SIZE. Contact your compiler vendor for more guidance.

- If a program fails at execution time for no apparent reason, the first step in isolating the problem will typically be to recompile it requesting both no optimization and no constraint check suppression. If the program ran without incident when compiled with these options, the user should suspect a problem with the compiler's optimization routines. The ACVC tests are typically run without suppression and using the default setting for optimization, so it is possible that there are errors in the compiler which would have been detected by the ACVC if a different set of compiler options had been specified. It is not the objective of the ACEC to search for errors in compilation systems, but only to test performance.

- During development of the ACEC test suite, two systems were found on which a few test problems ran for excessively long times and appeared to be in infinite loops. After detailed investigation, it was determined that the programs were *not* in an infinite loop and would have eventually terminated, although it would have taken days of computing. The source of the difficulty was that the performance of the test problem was not stable. The timing loop code in use then, in an effort to achieve a measurement within the requested error bounds and confidence levels, repeatedly increased the repetition and cycle count. Although this approach is generally proper, in this case it did not help since the measurements were varying widely and apparently randomly. These problems would have eventually stopped when the maximum iteration count and maximum cycle count were reached, but this would have taken many hours, and when it finished, it would still not have a valid measurement within the requested confidence level.

  The timing loop code was modified to:

  - Test for failure to converge and not continue to increase the repetition and cycle counts when the test problem has executed for a significant amount of time and shows highly variable performance measurements. In particular, when the time spent executing the test problem (less null loop overhead) is greater than one second and is varying between iterations of the inner timing loop by a factor of two, the test problem is stopped (if the minimum number of cycles has been reached) and a message informing the user of the variable performance is output to the results file. Normally, a variation of a factor of two in the performance estimate of a test problem would trigger an increase in the inner timing loop repetition count: this is

what was causing the *very* long execution times in the original version of the timing loop code.

- Output a message after a test problem executes for 5 minutes (when the timing loop gains control) so that the user will know that the test problem is **not** in an infinite loop.
- After a test problem has executed for 30 minutes (and the timing loop gains control) it will stop and report the current best estimate of performance of the test problem.

As the timing loop has been modified, test problems similar to the above will be marked as unreliable. The result file may also contain additional lines of informative output stating that the problem has executed for more than five minutes without converging. It may also contain an additional line indicating that the test problem was prematurely terminated (did not complete the maximum number of cycles which would otherwise be performed) due to apparent instability in the measurements.

- There are several conditions which might result in the execution time measurement of a test problem being considered unreliable.
  - The timing loop may terminate by reaching a maximum limit rather than having the measurements satisfy the confidence level for small percentage errors. For test problems translating into a few machine instructions, a small absolute error in measuring a test problem may correspond to a large relative error: consider a test problem translated into a null statement. Since the value being measured will be close to zero, dividing by this value can generate large relative errors corresponding to errors whose absolute value can be arbitrarily small.

    Some test problems which are translated into nulls may be measured as taking a small amount of execution time due to noise in the measurement process. On target systems which are measuring code expansion, it is appropriate to consider test problems with small unreliable times and zero space as being measurement noise and actually being zero time test problems.
  - The test problem may perform operations with variable or inconsistent speeds. For example, the time to perform a read from a disk file can vary based on the mechanical properties of the disk, including: seek times; rotational latency; automatic soft error recovery involving application of error correcting codes; retrying the physical I/O read if it fails (only reporting a hard error to an application when the automatic error recovery procedure fails); and disk cache buffering, whether in the operating system or in the disk controller, will introduce very large variations in performance.

    It is possible that an implementation may do something which makes the time to execute a test problem vary with the number of times it is executed. This is not

expected, and is certainly not typical, but it is possible. For example, a system which kept a journal of I/O operations and checked it for duplicates would build an ever growing list of operations which it checked each time. If a system always logged each operation, then the added overhead would be essentially constant and the problem would be repeatable.

- Contending jobs on the system can prevent stable measurements. This source of errors may disappear if the test program is rerun at a different time.

In general, a test problem with an unreliable time measurement should be re-executed. Since the conditions producing unreliable measurements are often transitory, simply re-running the program will often be sufficient to produce a reliable measurement.

A user should not consider an unreliable measurement as a failure of the system under test. In most cases, the test problem has been successfully translated. What fault exists is in the ACEC timing measurement code not being able to determine a consistent measurement.

The first steps an ACEC user can take to isolate an execution time error, assuming that the error diagnostics are not self explanatory, are presented below:

- Recompile the program requesting no optimization and no suppression of checking. If this modified version works without problems, it suggests an error in the compiler, although it is possible that the program contains an implementation dependency, and the compilation system changes the interpretation it makes based on the compilation options specified.

- If the test doesn't terminate (and doesn't generate any messages indicating that it is not in an infinite loop), then the user should check that the system can execute one iteration of the test problem. Modify the program to remove the timing loop code and execute the test problem once. This may be checked with a symbolic debugger, if available.

  If a test problem fails consistently the first time it is executed, the debugging should be fairly straightforward. On the other hand, if the program executes several cycles before failing, the fault will be harder to isolate.

6.2 RUNNING A SUBSET

An ACEC user does not have to run the full benchmark suite. They should select the test programs they wish to run and then compile and execute only those programs. If they are using the provided command files as models, they can modify them by not compiling or executing the programs not of interest. After the preparation step of compiling GLOBAL and MATH (and DBL_MATH), an ACEC user can essentially compile (or recompile) the test suite in any order desired. Test programs using multiple compilation units are organized as one text file

with multiple compilation units (presented in a valid compilation order). This packaging of compilation units into files simplifies the procedures for compiling programs. The user simply submits the source file to the compiler.

The FORMAT tool and the MED_DATA CONSTRUCTOR tool work without modification on a subset of problems. FORMAT simply creates a smaller aggregate. The MED DATA CONSTRUCTOR creates a MED DATA package containing only those problems appearing in at least one input aggregate. Any problem which appears in some input aggregates but not in others will be assigned an "err_no_data" value when missing.

Some ACEC users may be interested in the performance of one area, such as the tasking. The user can find the test problems associated with tasking by looking under task in the VDD Appendix V, "ACEC KEYWORD INDEX-1". The test programs which these test problems are contained in can be found by referencing the VDD Appendix II, "TEST PROBLEM TO SOURCE FILE MAP". The ACEC usually follows the convention that the file name containing a test program has the same name as the main program. This makes it trivial to find the source file containing a test program given the name of the program.

It is not true that systems have the same relative performance on all test problems. Some systems do some problems much better, or much worse, than they do other problems. Running a small subset of test problems means that an ACEC user may be surprised that some language feature runs much slower than the others — and it could turn out to be a feature that is widely used in their ultimate application area and was not included in the subset of the ACEC selected for execution because they felt that "all systems will do it approximately the same." An example may be file I/O, where performance differences of 100 times between systems are not uncommon. Many users may initially feel that "if the program has to read and write, the speed of the reads and writes will be determined by the speed of physical devices and should be about the same on the similar hardware."

7 SYMBOLIC DEBUGGER ASSESSOR

The LRM does not require a compilation system to provide any type of symbolic debugger. However, a good debugger can improve programmer productivity and enhance the desirability of a compilation system. Many of the popular Ada compilation systems include a symbolic debugger. The ACEC provides a set of debugger assessor scenarios (programs and sequences of operations to perform) to enable users to evaluate debuggers. The scenarios emphasize the determination of functional capabilities and not the elegance or efficiency of the user interface. They include tests for capabilities which are commonly provided by debuggers and for capabilities which are not widely supported. They include both Ada and non-language specific capabilities.

The effort required to adapt this assessor can be fairly large. It will require the user to understand enough about the system being evaluated to know whether certain capabilities are provided by the system. An acceptable system can provide the requested capabilities in a different fashion than presented in the ACEC sample systems. To decide that some capability is not present will require careful review of the documentation on a system's debugger because the capability may be present in an unexpected form — what is a separate command on one system could be a qualifier to a command on another, and different names are given to similar capabilities in different systems. If a user has to look up information in the implementation reference manual before issuing each command, the adaptation effort should be expected to take longer than if the user were already familiar with the system being evaluated. Completing the debugger assessor on a system new to the evaluator should be expected to take at least a week, although the tester will learn more about the capabilities and usability of the system debugger than they might otherwise learn in several weeks of experimentation.

An ACEC evaluation need not always include the execution of each assessor. If the organization performing the evaluation is not concerned with debuggers, it might not perform the evaluation. In a source selection environment, if analysis of performance tests results or library assessor results show that a system is unacceptable, it will not be necessary to complete the debugger assessor for that system.

Some systems have a compile time option which must be specified to permit the use of a debugger. Programs not compiled using this switch may not be runnable under the debugger, or may have access to a severely restricted set of capabilities. For the purposes of the ACEC debugger assessor, a system with and without the option should be considered as different products. It may appear to be unnecessary to know what debugger capabilities are available for programs compiled with a switch requesting no debug support. However, it may be difficult to use a debugger to isolate a problem in a program which only fails when the debug option is not set.

Each debugger scenario tests a debugger capability, such as breakpoints. Each scenario contains one or two programs and a set of debugging operations to perform. The ACEC user

must adapt the commands to compile and run the program, and the debugger commands, to match the target system. The distribution tape provides the programs, named DBG*.ADA, and a set of sample debugger command files for the VMS debugger, named DBG*.COM. Comments in the programs and command files describe the purpose of each scenario and the operations to be performed.

The operational characteristics of each debugger scenario are similar. The program will be compiled, linked, and invoked under the debugger. The user will then set a breakpoint on the function DESCRIBE whose source text explains the sequence of debugger operations which are to be performed. Comments throughout the code will lead the user through the scenario. The user will have to adapt the sequence of debugger operations for each system; the commands should be recorded so that the results can be replicated if desired. It may be possible to have the system log the operations. The log file can then be used to verify the results, and it may also be runnable as a debugger command file, so that the sequence of commands can be easily repeated.

The ACEC debugger assessor provides a template for recording results. To fill in the template, the user will invoke a system text editor and replace the blanks in the template with YES/NO indicators, or print a hard copy and fill in the blanks. The user may use a text editor to insert comments into the report template. The user should note any operations which are particularly awkward. The user may also wish to comment on the quality of the user interface, the quality of the documentation, the ease of learning, availability of additional features, and the relative importance of each scenario. The interpretation of the report is discussed in the Reader's Guide, Section "SYMBOLIC DEBUGGER ASSESSOR".

To run the debugger assessor, adapt the sample command files to the compilation system under test and perform the following steps in order:

- PREPARATION

  Create directories and insert directory names into the command files SETUP_DBG.COM and CMP_1_DBG.COM. The *.COM files refer to examples for DEC Ada on VMS. The example command files use the following four directories:

TEST_SUITE_DIR	Contains the test suite and debugger command files.
TOOLS_DIR	Contains tools.
WORK_DIR	Used for compiling and running test programs.
ACEC_LIB_DIR	Ada library directory.

- SETUP

  Run SETUP DBG.COM to compile GLOBAL and the baseline files and prepare the work directory. SETUP_DBG.COM invokes three command files:

COMPILE_BASELINE	Compiles the ACEC baseline files. (The baseline files do not need to be compiled again if they were compiled before performance testing.)
SETUP_TEST_PROGRAMS	Compiles the ACEC setup tests. (The setup tests do not need to be run again if they were run before performance testing.)
PREPARE DBG DIR	Deletes all files in the work directory except the log files, and copies CMP_1_DBG and the timing loop code into the work directory.

  Most of the debugger scenarios do not require MATH and the timing loop files. If a subset of the debugger programs will be run, compiling GLOBAL and running PREPARE DBG DIR may be all the preparation that is necessary.

- COMPILE AND RUN

  Compile and run each debugger program individually. Invoke CMP_1_DBG with the program name, then invoke the program under the debugger.

CMP 1 DBG < program name >	Copies one debugger program to the work directory and compiles it.
RUN < program name >	VMS command to invoke program under the debugger.

  Once in the debugger, issue the debugger commands needed to complete the scenario. Record the results on the report template which is found in file DBG_TEMPLATE.TXT. While executing the debugger scenarios, it may be helpful to have a listing of the program

and the sample debugger command file on hand. The program and command file contain extensive comments to aid the user in executing each scenario.

- CLEANUP

  After all tests have been executed, run CLEANUP DBG FILES.COM to delete miscellaneous files and program library units which are no longer needed.

CLEANUP_DBG_FILES	Deletes unneeded files.

Assumptions made in the debugger scenarios and command files:

- It is assumed that breakpoints occur before execution of the line they are set on.
- Line numbers given are for the actual line number of the file (blank lines and comment lines are numbered).
- The example command files compile the programs with no optimization. Some debuggers may not function on optimized code, or may perform poorly. It is suggested that users run the debugger tests on unoptimized code. If desired, another evaluation may be performed on the optimized tests. Behavior with and without optimization is generally so different that optimization and no-optimization can be considered to represent different compilation systems.

8 PROGRAM LIBRARY ASSESSOR

The LRM levies only minimal requirements on a program library system — after a unit is compiled into a library it shall be possible to subsequently compile units which reference it. The LRM suggests (Section 10.4 paragraph 4) that a programming environment provide commands for creating the program library of a given program or of a given family of programs and commands for interrogating the status of the units of a program library. The form of these commands is not specified by the LRM.

The effort required to adapt this assessor can be fairly large. It will require the user to understand enough about the system being evaluated to know whether certain capabilities are provided in a system. An acceptable system can provide the requested capabilities in a different fashion than presented in the ACEC sample systems. If a user has to look up information in the implementation reference manual every time they try to adapt a scenario, the adaptation effort should be expected to take longer than if they were already familiar with the system being evaluated. Completing the program library assessor on a system new to the evaluator may require a week, although when completed the tester will have learned more about the capabilities and usability of the system's library than they might otherwise have learned in several weeks of casual use.

An ACEC evaluation need not necessarily include the execution of each assessor. In a source selection environment, if an analysis of results of the performance tests or the debugger assessor show that a system is unacceptable, it will not be necessary to complete the library assessor for that system.

The ACEC program library assessor capability is designed to determine whether a system supports selected capabilities. It also provides for the collection of some performance measurements — elapsed time and disk space usage. It contains a set of scenarios to be performed on a system and instructions for evaluating system responses.

The scenarios consist of compilation units and sequences of operations to perform on a program library. ACEC users must adapt the scenarios to each target system. The scenarios emphasize functional capabilities and this emphasis will minimize the role of subjective biases — the questions are not whether the evaluator *likes* the way the system provides a capability, but simply whether it is supported.

It is possible that based on a review of the system documentation an evaluator may (incorrectly) conclude that a capability is not supported when it actually is just not described in the manuals where the evaluator expected it to be. Different systems can provide the same capability in different ways. One system may use a separate command for something which another system provides by supplying a specific set of parameters to a more general command. ACEC users should carefully review system documentation (and perhaps contact the vendor) before deciding that a capability is not supported.

The scenarios should be equally applicable to a "point-and-shoot" graphic based user inter-

face as they are to a command-line based user interface. The sample DEC/VMS command files distributed with the ACEC provide a command-line interface to a program library manager, and contain comments discussing both the operations to be performed and the expected results.

Some library systems may have features not exercised by the set of scenarios. The library assessor template provides a space for user comments which can be used to report the presence of additional capabilities.

In some compilation systems, the mechanisms which provide the capabilities the scenarios test will not be isolated in a set of "Ada program library utilities." For example, a system may rely on file naming conventions to map from Ada compilation unit names to host operating system file names and permit the use of "normal" host operating system tools to provide some capabilities — such as reporting the disk space used by a compilation unit. Since the ACEC library assessor is concerned with the presence of a capability, this would satisfy the ACEC assessor, assuming that the technique was reliable. On a system which prefixes the names of subunits with the names of their parent units, it may be possible to use wild-card options in the operating system tools to manipulate a unit and all of its subunits as a group (for example, to determine the size of them all). However, if the host operating system limits file name lengths, it may map "long" Ada unit names using a different algorithm and this would defeat the unit/subunit grouping. Because this would make the technique of grouping unreliable, an ACEC library assessor should **not** consider such a system to provide the extended capability. On such a system, the name mapping is implementation dependent and users need to verify that the actual mapping is as simple as it appears to be when only short names are used. The ACEC library assessor scenarios provide some examples which might stress a system, but users must be aware of potential implementation dependent restrictions.

Each library scenario contains comments which describe what it does (or attempts to do). Most will compile provided compilation units into a library and then use the provided tools (library and host operating system commands) to perform some operation. They may request that the user record elapsed time and disk space, or verify that the requested operation has been performed.

The distribution tape contains the referenced compilation units and a set of command files to be adapted to different systems.

Capacity limits may be exceeded when running some of the library assessor tests. A user should attempt to determine whether a limitation is "hard" (for example, no Ada library can contain more than 1024 units) or "soft" (the system ran out of disk space during the testing).

The ACEC library system provides a simple template for collecting results. Users may print the form and fill it in by hand, or edit an on-line version. There are provisions for users to add descriptive comments about individual test problems, general comments about system operations, and subjective opinions.

The library assessor is organized as follows:

- A set of compilation units.

These units are named LIB*.ADA. These Ada source units are compiled in the library assessor scenarios, as directed by the command files.

- The command files LIB*.COM.

  These files will create and manipulate the program library structure, including compiling the LIB*.ADA units into it. Users must adapt the command files to the compilation system being evaluated. The command files are provided for the DEC Ada (VAX VMS host/target) compilation system. The commands for this system can serve as models in porting the library assessor to other compilation systems.

  The file LIB.COM is the main driver for the library assessors which will execute all the assessors in turn.

  These files contain detailed discussion of the operations to be performed and the evaluation of system responses.

- A file LIB_TEMPLATE.TXT.

  This file is a template which will be filled in by a user to report findings from individual scenarios. The evaluator will fill in:

  - Yes / No answers to questions on the presence of individual capabilities.
    Guidance is provided in the command file comments in areas where judgment calls may be necessary.
  - Some scenarios call for the recording of execution time in the template.
  - Some scenarios call for the recording of disk space usage in the template.
  - Provisions are made for general comments.
    A user may want to report that a special case of a capability is provided, but not the full generality requested for in the library assessor scenario. Capabilities not tested for in the assessor scenarios can be recorded as comments in the template for later review.

9 DIAGNOSTIC MESSAGE ASSESSOR

The LRM requires a compilation system to reject *illegal* programs, although the definition of *illegal* does not include all programs containing violations of the standard. The LRM does not specify the form or content of error or warning messages. The purpose of the ACEC diagnostic system assessor is to:

- Determine whether the diagnostic messages clearly identify the condition and provide information to correct it.
- Determine whether warning messages are generated for various conditions.

The ACEC diagnostic assessor tests include examples of both illegal programs and programs containing "suspicious" conditions which the ACEC user will pass through the compilation system. The generated diagnostic messages (if any) will then be manually compared, using the information provided with each problem, to see whether the message provides a set of specific information. A template will be completed based on this comparison.

Some compilation systems have flags which affect the number and type of messages produced. The printing of warning messages may be suppressible — and this may be the default condition. For the diagnostic assessor tests, the ACEC user should specify the compilation system options which provide the most thorough checking. Several systems provide an option for syntax checking which executes quickly but only performs a context-free parse of the source program and so will miss many error conditions which require context sensitive information to detect — this is **not** the option which should be used for the diagnostic assessor testing.

Each diagnostic problem contains comments which describe the condition and point out any specific pieces of information to look for in the generated message. For example, one diagnostic problem ends a subprogram before ending an "if" statement within it — the user is asked to see whether the diagnostic message identifies the location of the non-terminated "if" statement.

The distribution tape contains a set of compilation units named DIA*.Ada and a set of command files to compile and link the programs which will have to be adapted to different compilation systems. These command files are:

- DIAGCOMP.COM — This compiles the units which test for compilation errors.
- DIAGLINK.COM — This compiles and links the units which test for linker errors.
- DIAGNOS.COM — This is the driver file which compiles all the diagnostic compilation units.
- DIAGFILL.COM — This will generate a file large enough to fill up a disk. It is used if there is no facility to set a disk quota.

The ACEC diagnostic assessor provides a template for collecting results and a tool to analyze a completed template and produce a summary report based on it. To fill in the template, the user will invoke a system text editor and replace the blanks in the template with YES/NO indicators. The analysis tool DIAGREAD.ADA is an Ada program which reads a template from standard input and writes a report to standard output. The interpretation of the report is discussed in the Reader's Guide, Section "DIAGNOSTIC ASSESSOR".

There are several programs which should be run interactively, including DIA_R01A, which will run indefinitely or until the user decides to interrupt it.

The test, DIA L04A, is designed to explore what happens on a system which runs out of disk space. The user has two choices here:

- to set a disk quota to a low enough limit to exhaust disk space during tne compile and link process
- or, to use a com file (DIAGFILL.COM) which is designed to fill up your disk (on a VAX VMS system). Use this alternative with care.

10 ANALYSIS

There are two analysis tools provided with the ACEC: MEDIAN and SSA (Single System Analysis).

The MEDIAN analysis program compares sets of performance data from different compilation systems, each of which has executed the same release of the ACEC. It assumes that the performance data can be modeled as the product of a factor for each Ada compilation system and a factor for each test problem. MEDIAN computes these factors, using a statistically robust data analysis approach. It then calculates the residual factors for each test problem on each compilation system. A small residual factor indicates that the actual timing for the test problem is faster than would be predicted, based on the average performance of the system over all problems and the average performance of the problem over all systems. Residuals which are either small or large point out places where a system is performing better (or worse) than expected. MEDIAN uses heuristics to place indicators, drawing attention to residual factors which are exceptionally small (or large). Small residual values may indicate places where an optimization has been particularly worthwhile, or where the underlying target machine hardware is particularly well suited to executing the test problem efficiently.

The MEDIAN program produces additional data showing goodness-of-fit of the data model and the variation in the performance measurements. This additional output and its interpretation are described in depth in the Reader's Guide, Section "MEDIAN OUTPUT".

The SSA program analyzes relationships between test problems executed on one system, where there is some a priori relationship between results expected, or where inferences can be drawn about a system based on comparing results of related test problems. For example, some test problems are optimizable and some are hand optimized versions of the same logical operations; if the execution time for both versions is the same, it can be inferred that the system is either performing the intended optimization or another optimization of comparable effectiveness.

The rest of this section in the User's Guide is concerned with the steps necessary to format the performance data so that MEDIAN and SSA can analyze it.

10.1 PREPARING THE DATA

The MEDIAN analysis tool is an Ada program which processes a two dimensional array (TIME) of performance data, represented as a floating point type. The dimensions of this array are two enumerated types: the first is SYSTEMS, which declares an entry for each Ada compilation system being compared; the second is PROBLEM, which declares an entry for each test problem. These enumerated types, and the array of performance data reflecting the measurements obtained by executing the ACEC test suite on each different compilation system, are declared in the Ada package, MED_DATA.

To compare the performance results between several systems, the ACEC user will run the FORMAT tool and the MED_DATA CONSTRUCTOR tool to produce a variation of the MED_DATA package. The package will contain:

- A declaration of type SYSTEMS assigning unique identifiers to each system being compared.
- A declaration of type PROBLEM assigning unique identifiers to each test problem being considered. Some users may choose to add additional test problems of their own construction, or to exclude some test problems. If tests are added, the user must modify the enumeration type PROBLEM in the MED DATA CONSTRUCTOR program to include the new test problem names and the enumeration type CMP_UNITS to include the names of the files containing the new tests. No modifications are needed to run the MED_DATA CONSTRUCTOR on a subset of problem results.
- A declaration of the array TIME, initialized to contain the performance measurements obtained from executing each problem on each system. Each row of the array TIME represents the performance data from a compilation system. The performance data for each test problem is specified by using a named association giving the test problem name and corresponding performance measurement (or coded value indicating data is not available).

  Test problems for which results are not available are indicated in the array TIME by coded values. Any non-negative value is assumed to be a valid measurement; negative values are used to encode various types of missing data.

The FORMAT tool accepts as input a file containing the output of each ACEC test program (where this is available) and generates as output two files. Each file contains an initialized array aggregate specifying the test problem name and performance measurement for each ACEC test problem which reached normal completion and output performance measurements. One output file contains an aggregate of execution time data, and the other contains an aggregate of code expansion size data. The aggregates created by running FORMAT on the output of the test suite can be combined into versions of the MED_DATA package by running the MED DATA CONSTRUCTOR program.

An ACEC user could use a text editor to format the test results into aggregates to be input to the MED_DATA CONSTRUCTOR. This could be tedious and time consuming, and provides opportunities for introducing transcription errors. When each test problem is executed, the timing loop code writes, to Ada STANDARD OUTPUT, the performance measurements for that test problem. On target systems where STANDARD_OUTPUT can be saved in a file for later processing, it is possible to automate the extraction of performance data from this file. The FORMAT program does this.

One simple way to save STANDARD_OUTPUT in a data file on VAX VMS is to run the test suite as a batch job; STANDARD OUTPUT will then be written directly into the log file. On a UNIX system, STANDARD_OUTPUT is redirected to a file name of your choice. There are a few problems which must be run interactively — primarily to measure console output performance (CIO.A and IOTEST4.A), and to test whether a task waiting for console input blocks progress of other tasks in the system (ASYNC2.A and ASYNC4.A). These programs cannot be run as a batch job. Terminology is somewhat operating system specific, but the concepts of batch jobs and log files are understandable to most programmers, if not to all operating systems. FORMAT reads the log file and creates data aggregates specifying (using name association) the test problem name and the performance measurement for that test problem. It creates an aggregate for execution time and for code expansion size, each in a separate output file. It can do this because it knows the format that the test problems use for generating output (the format is column specific. Refer to the Reader's Guide, Section "OPERATIONAL SOFTWARE OUTPUT", for details). FORMAT reads one input file to create its output. If the user has executed the ACEC test suite as several separate jobs, then the individual log files should be combined into one file before presenting the file as input to FORMAT. Alternately, the user could run FORMAT on each piece and merge together the data aggregates generated from FORMAT. Using FORMAT will speed up extraction of the performance data, and will remove the possibility of transcription errors.

Not all bare machine targets will support a file system. If an ACEC user cannot save STANDARD OUTPUT in a file, it will be necessary to manually edit the data aggregates with the values of the performance measurements obtained. Even if the target system supports only a console for output, it may be possible to replace the physical console electronic connection with a connection to a computer which is programmed to operate as a simple terminal and also save all the output in a file. To simplify processing, it is recommended that the user who has to manually transcribe performance measurements construct the same type of data aggregates for each system that FORMAT would.

The next step in analyzing the data is to construct the MED DATA package, which is used to pass the data to MEDIAN. The execution time, code expansion size, and compile time data are analyzed in separate runs of the MEDIAN program, and a MED_DATA package must be constructed for each type of data to be analyzed. The MED_DATA package is produced by running the MED DATA CONSTRUCTOR program with the data aggregates produced by FORMAT as input. Before running the MED_DATA CONSTRUCTOR program, the user must perform the following steps:

- For each system to be compared, the user must prepare one data file containing an aggregate initialized with execution time, code expansion size, or compile time data. FORMAT produces 2 files, one for execution time data, and one for code expansion size data. The collection of compilation times is not automated. If compilation times are to be compared, the user must manually construct an aggregate for compile time data, or

write a program to extract the times from the log file.

- The user must prepare the SYS_NAMES file containing the type of measurement represented by the data values (execution time, code expansion size, or compile time), the names of the systems to be compared, and the names of the files containing the system aggregates. The format of the file must be as in the following example:

```
time – – or "size", or "compile", depending on measurement
system_1_name
– – Up to 20 lines of System One comments and description,
– – including version number of compiler and operating system,
– – processor configuration, etc.
system_1_file_name
system 2 name
– – Up to 20 lines of System Two comments and description.
system_2_file_name
.
.
.
```

- The user must construct a command file to assign the input file name to the logical name "sysnames" and call the MED_DATA_CONSTRUCTOR program. Two model command files are provided: MED DATA CONSTRUCTOR.COM for DEC Ada under VMS, and MED_DATA_CONSTRUCTOR.UNX for MIPS Ada under UNIX.

The output of the MED_DATA CONSTRUCTOR is a MED_DATA file containing the Ada package MED_DATA. The file name extension will indicate the type of data in the file (.tim for execution time, .siz for code expansion size, or .cmp for compile time). This file should be compilable with no modifications.

The MED_DATA CONSTRUCTOR tool is expected to run on any system which will run MEDIAN. Size may be a problem on ... ie systems, however, because the program uses a large internal data structure. If the user is unable to run the MED DATA CONSTRUCTOR program, the user may still be able to run MEDIAN with a small MED_DATA package. The user can manually construct a MED_DATA package by editing the dummy MED_DATA file from the ACEC distribution tape to include the systems the user wishes to compare. In editing the package, the user must modify the declarations of the enumeration type SYSTEMS to include a unique identification for each system being compared. The initialization of each row of the array TIME is accomplished by including text of the form:

SYSTEM X = >

{ *aggregate for SYSTEM X from FORMAT* },

for each system.

There are several sets of sample data (aggregates generated by FORMAT) included on the distribution tape which can be included for comparison purposes. These sets of data reflect results obtained during the development of the ACEC Software Product. They are:

DATA.ADA	Average of all targeted systems
DATA.VAX	Average of VAX targeted systems

During development, the ACEC was executed on 5 trial systems to demonstrate portability. The first file is an aggregate reflecting the problem factors for these systems. The other file reflects the average problem factors for the VAX. There are two compilation systems for this target. A comparison of different compilers for the same target machine permits a direct comparison of performance of the different compilers, since the machine idioms will be the same for all compilers for the same target. Where an ACEC user is only testing one system, the sample data will be the only comparative information available.

After MED_DATA is constructed reflecting the different systems to be compared, it is then compiled, and then the program MEDIAN is compiled. When MEDIAN is linked and executed, it will generate as output the comparative report. For details of the report, refer to the Reader's Guide, Section "MEDIAN OUTPUT".

This process is shown graphically below.

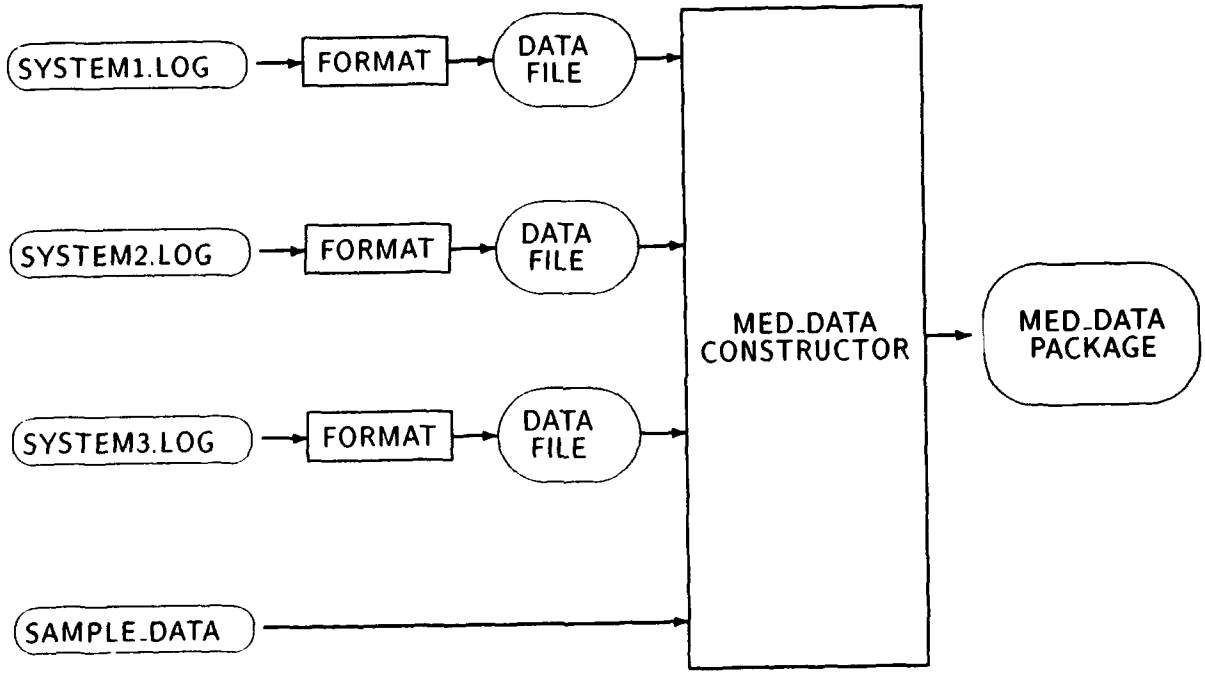


Figure 7: CONSTRUCTING MED_DATA

The MEDIAN program assumes that all performance measurements are non-negative floating point values. Negative values are used to encode error conditions. Different error conditions use the following mapping:

```
problem_name =>numeric_value,
    where
        numeric value is a nonnegative measurement
```

or, in the case where an error was detected,

```
problem_name  => XXX,
    where
        XXX is a negative numeric value with one of the following interpretation
            err at compilation time
                -1.0 =>error in problem during compile time
            err_at_execution_time
                -2.0 =>error in problem during execution time
            err no data
                -3.0 =>test not attempted; may work, but no data
                       available
            err dependent test
                -4.0 =>test not appropriate—system dependent optional
                       feature
            err_packaging
                -5.0 =>test not available due to packaging. It may
                       work, but was included in another test which
                       failed.
            err_unreliable_time
                -6.0 =>test ran, data considered unreliable
            err_withdrawn_test
                -7.0 =>test problem withdrawn
```

These numeric values are declared as named numbers in the package GLOBAL.

Test problems which failed at execution time during development of the ACEC often have exception handlers (or test the results of computations) and write an error code to STANDARD OUTPUT if they detect an execution time failure. FORMAT tests the confidence level indicator and outputs an error code for unreliable measurements when this indicator is present. Other failing test problems will have to have their error code value specified. If a test fails due to a compilation error, the user can edit the aggregates produced by FORMAT to include the err at compilation time code for that test.

FORMAT is coded so that when the indicator is set reflecting that the requested confidence level in timing measurements was not achieved it will write an "err_unreliable_time" code to STANDARD_OUTPUT for that problem. A user may wish to proceed using the measurements obtained. Such a user has two choices:

- They may edit the output of FORMAT, replacing the error code with the measure of time. For a small number of unreliable measurements, this is straightforward and effective.

- They may modify FORMAT so that it does not generate an err_unreliable_time error code when the confidence level indicator is set, but always outputs whatever actual measurements it finds. This is marked in the program text — there is only one place in the procedure FORMAT which examines the input file for the bad data indicator and replaces the "TIME" variable with the error indicator when it is set.

The MED_DATA package produced by MED_DATA_CONSTRUCTOR will fail to compile if the system names specified in the SYS_NAMES file are not unique and legal as enumeration literals. It should not be necessary to edit the MED_DATA package. The user need not specify values missing from the aggregates, and need not remove duplicate results. Missing values will be assigned an "err_no_data" value, and all but the last instance of a duplicate test result will be commented out in the output. If the user does not wish to use the last measurement obtained for a test which was run more than once, the aggregate may be edited.

Having users compile a package which declares initialized data for input to the main program is a bit unusual. It was selected to simplify the processing of performance data from bare machines which have no disk file processing capability. The expectation is that these systems *will* support console I/O, permitting Ada programs to output text, if only for ease in validation. It may not be possible to extract machine readable output files from such a system. To facilitate the comparison of performance data from such systems, it is important to have a method of entering data which is human readable and intuitive. Using named aggregates is a method which provides a simple operating procedure. After each test program is executed, the data aggregate for the system will be filled in with the results from the test problems which executed. The test programs can be run, or rerun, in any order, and the data does not have to be entered in a specific order (if positional association were to be used, then the risk of entering data for one test problem which accidentally is interpreted as applying to another test problem would be very large with over a thousand test problems).

If the ACEC were to assume that all target systems on which the test programs were executed supported a file system, it would be easy to design a more automated and "user friendlier" method of extracting performance data and presenting it to the analysis program. The ACEC project chose not to simplify operations on "friendly" targets at the expense of unduly complicated processing on bare machines.

10.2 RUNNING MEDIAN

This section gives instructions on the use of MEDIAN.

The report produced by MEDIAN is tabular. If more systems are compared than will fit across one line of output, the program will output additional pages as needed. MEDIAN will not write more than 80 columns on a line unless the globally defined variable "max line length" in MEDIAN is changed. Increasing the line length (to say 132) would print an extra column of data on each output page. This modification permits the production of more compact results when suitable output devices are available. The choice of 80 columns permits easy display on most terminals.

To use MEDIAN to compare code expansion sizes, the user will run the MED_DATA_CONSTRUCTOR program using the code expansion size aggregates generated from FORMAT, rather than the execution time aggregates. The processing required to compare expansion sizes is identical to the execution time analysis. For test problems which are detected to fail at execution time, a size of zero is always generated. This is not intended to be meaningful. No changes to the code in MEDIAN will be needed. The code expansion sizes are reported in bits and the "raw" data is directly comparable between systems having different values for SYSTEM.STORAGE_UNIT.

To use MEDIAN to compare compile times, the user will collect and format compile time data for input to the MED_DATA_CONSTRUCTOR. Problem names in the compile time aggregates will be the names of compilation units rather than the names of individual test problems.

10.2.1 New Versions of the ACEC

This section guides the user who has already run the ACEC test suite once, but now has an updated release of the ACEC.

10.2.1.1 Rerunning the tests

Tests with the same name will remain the same from release to release. New tests may be added and old tests may be removed. If it becomes necessary to modify a test problem (it was determined to contain invalid Ada code), the original problem will be withdrawn and a new problem with a different name entered to reflect the corrected test problem. A test problem with the same name between releases will have essentially the same code.

10.2.1.2 Reanalysis

The MEDIAN program compares sets of data. For it to be meaningful, the individual test problems on each target system must represent the same problem on all targets.

The ACEC is intended to track performance of systems over time, and the test problems from the first release will be retained in subsequent releases, unless error reports force them to

be dropped. Measurement data of a compilation system against the first release of the ACEC can be directly compared with data from that system on the second release of the ACEC — test problems not present in the first release should be marked as unavailable to MEDIAN ("err_no_data").

When a new release of a compilation system is made, the identity of the test problem will not change (the ACEC has not been rereleased), but the test suite needs to be rerun. The measurements on all problems may have changed, and bug fixes may make problems which previously failed now work (or make previously working problems fail).

10.3 INTERPRETING RESULTS

The Reader's Guide, Section "HOW TO INTERPRET THE OUTPUT OF THE ACEC", provides detailed instructions for interpreting the results produced by executing the test suite and analysis tools.

10.4 SSA

The ACEC single system analysis (SSA) tool, which will provide ACEC users with information on the performance characteristics of a single system, is similar in operation to the MEDIAN tool. It processes performance data aggregates as generated by the MED DATA CONSTRUCTOR tool.

The SSA tool works by comparing the results from related test problems, such as where one problem is a variation of another: for example, with and without a pragma specified; or an optimizable and a hand-optimized version of the same computation; or different coding styles to accomplish the same goals.

As with MEDIAN, it is possible to apply the SSA tool to a subset of test problems. In cases where there is no data available for any of the related problems in a set, the SSA tool will not produce a main report -- although it will generate a missing data file (unless suppressed) which lists all the test problems for which there were no results available.

The actual report consists of four sections (main report, table of contents, missing data report, and summary), which are discussed in the Reader's Guide, Section "SINGLE SYSTEM ANALYSIS". The output formats assume 80 column lines and 66 line pages; no special formatting is done.

10.4.1 System Specification File

The SSA tool begins by reading from a system specification file which provides file names containing the performance data aggregates and the SSA options. The name of this file is coded into the procedure readSpec as "ssa.txt". The **only** required line in this file is the

system name designation. All other information can be inferred from this name. The format of this file ("ssa.txt") is described below:

s*ystem name	—	[name] - (REQUIRED)
i*nput file names	—	[time] [size] [compile]
- - default		*.tim, *.siz, *.cmp
m*issing report	—	True | False
- - default	—	True
o*utput file names		[report] [missing] [contents] [summary]
- - default	—	*.rep, *.mis, *.con *.sum

Figure 8: SINGLE SYSTEM ANALYSIS SPECIFICATION FILE SYNTAX

The format is generally free form following these rules:

- Lines must be less than or equal to 80 characters.
- Lines that do not begin with 's', 'i', 'm' or 'o' are ignored, except for continuation lines. Capital letters are acceptable.
- Lines that begin with 'i' or 'o' call for special handling.
  - All file names must be provided, if any are provided. However, 'dummy' names may be used and the program will work. (If all input names are dummies, no data will be read, and only a missing data report will be produced.)
  - Continuation lines: file names may be up to 80 characters, so they may be placed on a separate line. There is no continuation line symbol. Processing continues until all expected names are found.
  - File names must be separated by a space, a comma, or a new line. A name must be contained on a single line.

This is perhaps more clearly explained by an example.

```
s = vax
i =vax.time,vax.size, vax.compile
m = false
o= vax.rep, vax.mis,
        vax.contents vax.sum
```

Figure 9: SINGLE SYSTEM ANALYSIS SPECIFICATION FILE EXAMPLE

This file directs the SSA program to read in the execution time data from a file named "vax.time", the size data from a file called "vax size", and the compile speed data from a file titled "vax compile". If the line beginning with "i=" were excluded, then the default values for these file names would be: "vax.tim", "vax.siz", "vax.cmp". The value for the missing data flag is false, the default value would have been true. In this example the output files are explicitly set to the same names that would have been given by default, except for the name of the contents file. Notice that all file names must be provided, if any are given. The program reads names in a set order for the main report, the missing data report, the contents file, and for the summary file. Furthermore, note that a missing data file name is required, even if the missing data flag is set to false. This file name will not be used, and no missing data file will be opened in this case.

10.4.2 Input Data Files

The SSA program expects the three input data files to possess certain essential characteristics. Any data files produced by MED DATA CONSTRUCTOR follow this format, and should not require any modification. However, users may have prepared their own files. Actually, if they prepare files like FORMAT's output, they can run MED DATA CONSTRUCTOR. They still might have to prepare their own if they cannot run MED DATA CONSTRUCTOR.

The procedures which read this input data are named readTime, readSize, and readCompile. All three of these procedures call the procedure findStart to skip the preliminary lines and find the proper starting point. The essential characteristics expected by findStart are:

- Lines must be less than or equal to 80 characters.
- Strings to be found must be the first nonblank characters on the line.
- The system name must be repeated twice. The match is case insensitive.
- The literal "-- -" is found.
- The next line is skipped.

The data is an Ada aggregate, with some limitations on the format. The following three line pattern, with the second two lines optional, is required. An ACEC problem name must always be the first nonblank string on the line it appears on. Certain Ada comments signal the presence of additional information in the execution time file. All other Ada comments are ignored.

- ACEC problem name floating point number
- --<<< additional numeric data, from ACEC output

- -->>> ancillary data, from ACEC output
- -- All other Ada comments are ignored
- Each problem name must be the first nonblank string on a line.

The data is read, until a line beginning with the stop signal ."))", is found

```
                    - whatever : ignored -
[ system name ]                                  - ( REQUIRED )
                    - whatever : ignored -
[ system name ] . . .                            - ( REQUIRED )
                    - whatever : ignored -
"= ." . . .                                      - ( REQUIRED )
"(" . . .                                        - ( REQUIRED )
- data, in the following pattern; the second two lines are optional -
[ ACEC problem name ] =  [ floating point number ]
--<<< additional numeric data, from ACEC output
-->>> ancillary data, from ACEC output

")" . . .                                        - ( REQUIRED )
                    - whatever : ignored -
```

Figure 10: SINGLE SYSTEM ANALYSIS DATA INPUT FILE FORMATS

10.4.3 Other Data Files

The SSA program is contained in one file, SSA.Ada. However, there are several additional data files which must be present. These files are all of the form *.SSA: LF.SSA, OPT.SSA, RTS.SSA, and STYLE.SSA. These files contain the templates for most of the tables. The files are standard text files, therefore, they are in human readable form, but editing might cause the SSA program to fail.

10.4.4 Measurement Units

The SSA program assumes that execution times are in microseconds, which are the units used in the standard ACEC timing loop (although a change of unit here would make no difference to the generation of the report). Program size is assumed to be in bits, which again is the standard unit generated by the ACEC timing loop code, but results are presented in bytes. Compile speed is assumed to be measured in seconds, but results are presented in minutes. A conversion is performed when the compile speed data is read into the SSA program. Since the user must gather and format the compile speed data, it may or may not be convenient to provide the data in seconds as the SSA program expects. Regardless of the units, the SSA program expects these times to be converted into minutes when the data is read. To facilitate this conversion, there is a conversion constant in the input procedure readCompile which must be modified appropriately. The relevant lines of code are given below. The compile time is read "as is" and then multiplied by the constant "convert" to change the units to minutes.

```
           -----------
PROCEDURE readCompile IS
           -----------

   .
   convert    : CONSTANT := 1.0 / 60.0 ; -- convert seconds to minutes
BEGIN
   .
   compileArray ( fileN ).time := compileArray ( fileN ).time * convert ;
   .
END readCompile ;
```

10.4.5 Implementation Dependencies

The SSA program does use large data aggregates, but it should run if MEDIAN runs. In addition, the SSA program at one point needs an integer type larger than 16 bits. This type (integer 32) is used for computing total line counts in the procedure compilationReport. This could be worked around on a system which only supports 16 bit integers by using real numbers for the totals.

11 CONSIDERATIONS FOR CODING ADDITIONAL TESTS

Each test problem is measured by inserting it into a template which will, when executed, measure and report on the execution time and code expansion size of the test problem contained within it, as discussed in Section 3.2.3.2.1.

There is an extensive discussion in the Reader's Guide, Section "CORRECTNESS OF TEST PROBLEMS", on things which might make a potential test problem invalid. The basic point to remember is that the test problem must be constructed to meet the following guidelines:

- The problem can be repetitively executed, and will follow the same path on each execution. In particular, the same control paths should be taken, and the repeated execution of arithmetic assignments should not produce numeric overflow (a test problem which increments an integer variable on each repetition is a mistake since it will eventually raise a CONSTRAINT ERROR).

- An optimizing compiler cannot "unduly" optimize the problem. In particular, it should not be able to detect that the test problem is invariant with respect to the timing loop code and only execute it once. Most test problems will need to have variables initialized to insure proper execution (for example, to prevent numeric overflow or other constraint errors), and if this initialization code is incorporated into the test problem using literal assignments, an optimizing compiler may be able to fold successive statements, essentially performing the intended test problem at compile time. While tests for folding are important, if that is not the purpose of the test problem being developed, it should be avoided.

- The test problem should be valid Ada. It should not rely on implementation dependent features of a system, unless the purpose of the problem is to test a feature which is *inherently* implementation dependent. For example, values of variables should be defined before being referenced, even though an ACEC user may be developing potential test problems on a system which assigns uninitialized variables to zero, as is permitted by the LRM. Ada programs can be written which depend on the order of evaluation of library packages. A good test problem will work with any valid (as defined by the LRM) order of elaboration, and not just the one adopted by the system used to originally develop the test problem.

  A test problem which calls assembler language procedures is implementation dependent, and so may require modification for each target system, but is important to test. Similarly, some implementations may impose various restrictions on capacities or on Chapter 13 features (such as representation clauses) or the tying of tasks to interrupts. Although

implementation dependent, it is important to include tests for these features in a general test suite.

- Test problems should avoid implementation dependencies. For *portability, test problems* should not use the predefined types INTEGER, and FLOAT, since their range is implementation dependent. In particular, in the LRM 3.6.1, it states that a discrete range where both bounds are of type *universal integer* will be implicitly converted to the predefined, and implementation dependent, type INTEGER. This should be avoided, since INTEGER'SIZE may vary *on the same target* based on implementation decisions in the compiler. In particular, a test problem should not contain a code fragment such as

```
FOR i in 1..10 LOOP ... END LOOP;
```

where the type of the **FOR** loop index "i" will then be the predefined implementation dependent type INTEGER. It is preferred in such cases to use a code fragment such as:

```
FOR i in global'int (1) .. global'int(10) LOOP ... END LOOP;
```

or

```
FOR i in global'bigint(1) .. global'bigint(10) LOOP ... END LOOP;
```

which will effectively request a specific size. In the example cited, even when no syntax errors are introduced by the use of one type or the other, there may be considerably different timings and implications on register usage between the two versions. When a discrete range in an array declaration is used, the difference in performance could force the use of an unnecessary integer type for all index computations.

The simplest way to comply with this directive is to use the type defined in the package GLOBAL. But, if user programs derive their own types from the universal types, that is also acceptable.

- To use MEDIAN to compare results, including those on user defined test problems, it is necessary to modify *only* the program MED DATA CONSTRUCTOR. The program MEDIAN itself need not be changed to include additional problems. The changes to MED_DATA_CONSTRUCTOR are straightforward. The user must include in the definition of the enumerated type PROBLEM the names for the user defined test problems. The user must include in the definition of the enumerated type CMP UNITS the names of the new programs containing the new tests. Then the MED_DATA_CONSTRUCTOR program must be recompiled

- Naming conflicts with the timing loop variables have to be avoided. A user can insure this by studying the package GLOBAL. The simplest way is to define their test problem as a procedure with no parameters and compile it into the Ada program library. A

standardized driver program which includes the timing loop code can then call on this procedure

A form for submitting a change request and a sample template is included in Section 12.2. The ACEC Reader's Guide, Section "CORRECTNESS OF TEST PROBLEMS", contains additional discussions on constructing test problems

Candidate test problems should minimize the use of obvious implementation dependencies, such as calls on host operating system routines, and specification of FORM parameters to I/O statements. Programs including UNCHECKED DEALLOCATION and UNCHECKED CONVERSION are not prohibited, but some uses of UNCHECKED CONVERSION will be more portable than others. A program performing boolean operations on integers may try to convert INTEGER to BOOLEAN, perform AND, OR, XOR, or NOT, operators and convert back to INTEGER. This may work on systems where BOOLEAN and INTEGER have the same size. A more portable approach would be to define an array type of packed BOOLEAN with range 0 .. INT'SIZE (where INT is a type derived from INTEGER with a range sufficient for the integer variables being used in the program), and converting between this array type and INT. The latter approach requires that packed boolean arrays are supported, but if this is done, it should work correctly on systems which have different default sizes for BOOLEAN and INTEGER.

12 ACEC USER FEEDBACK

ACEC users have two formal paths to provide feedback to influence future ACEC development. They can submit written problem reports and they can write change requests. No telephone support is provided. Written problem reports and change requests will be accepted, as directed below.

12.1 HOW TO SUBMIT A PROBLEM REPORT

Not every problem an ACEC user encounters with the test suite will be appropriate to report through the ACEC problem reporting system. If an ACEC program uncovers a clear error in a compilation system, this should simply be reported to the organization maintaining the compiler for resolution. Not all ACEC programs will be portable to all systems, since the test suite includes test problems to explore the performance of some implementation dependent features, and may have some programs which test features not supported on all targets. For example: tying tasks to interrupts; file I/O operations; operations on extended precision floating point types; interface to assembler routines; and some large programs which may exceed the capacity of some systems (at either execution time or compile time).

Failure of a test program is not sufficient reason to write an ACEC problem report unless the user believes that the failure is due to an error in the test problem itself, and is neither a reflection of an implementation error, nor a (legally) unsupported feature, nor a capacity limitation. Alternately, a test program may compile and execute without errors on an Ada implementation, but a user could still believe that the program is erroneous and submit an ACEC error report. This might occur when:

- The program illegally uses implementation dependent features, even though it worked on all the systems tested until the problem was discovered.
- A test problem is unexpectedly optimizable into something much different than the original "intent" of the test problem, as stated in the purpose. An example would be one where an optimizing compiler determines that the initialization code for a test problem can be folded into the body of the test problem, resulting in the test problem simplifying into a literal assignment, when the stated purpose of the problem is not to check for folding. The test problem as distributed may be a valid test problem to detect the presence of the "unexpected" optimization, but the purpose is wrong and the original intent may not be adequately tested for in the suite.
- A test problem which does not perform essentially the same computations on each repetition of the timing loop is invalid and should be corrected. If such a case is detected, it should be reported.

After completing the form on the following page, it should be mailed to:

R. Szymanski
Ada Compiler Evaluation Capability, Error Report
WRDC AAAF
Wright-Patterson AFB OH 45433-6543

ADA COMPILER EVALUATION CAPABILITY SOFTWARE PROBLEM REPORT

ORIGINATOR IDENTIFICATION

Originator's Name _____
Organization _____
Address _____
Telephone _____
Date _____

SYSTEM IDENTIFICATION

ACEC VERSION _____
Compilation System Version _____
Host Operating System Version _____
Target Operating System Version _____
Hardware Identification _____

PROBLEM DESCRIPTION

Source File with Problem _____
Explanation: _____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
(attach more pages if necessary)

12.2 HOW TO REQUEST CHANGES

There may be several types of change requests:

- They may consist of a program to be incorporated into the test suite. The simplest way for a user to submit a test problem, without needing to study the internals of the ACEC packages is to make their test programs match the following template:

```
procedure TESTPROC is
...
end TESTPROC;


with global; use global;
with calendar; use calendar;
with text_io; use text_io;
with TESTPROC;
procedure SAMPLE is
package int_io is new integer_io ( int ); use int_io;
package flt_io is new float_io ( real ); use flt_io;

begin
pragma include ("inittime");
pragma include ("startime");
      TESTPROC;
pragma include ("stoptime0");
      put("TESTPROC ...");          -- name and description goes here
pragma include ("stoptime2");
end SAMPLE;
```

Figure 11: SAMPLE TEST PROGRAM TEMPLATE

The procedure TESTPROC is a separate compilation unit containing the code to be measured. Separating it from the procedure SAMPLE avoids possible naming conflicts with variables in the ACEC package GLOBAL. Submitters should try to follow the guidelines for test problem construction discussed in the Section "Considerations for Coding

Additional Tests". They should verify that the names of the compilation units submitted do not duplicate names of units in the ACEC. They should include a filled out comment template.

- They may request that areas of the language, or various application problem domains, be studied for the creation of additional test problems. An example of such a request might be, "Include more testing of arithmetic on fixed point types or complex record structures," or "Include more examples of operations on **limited** and **limited private** types," or "Include an example of the simplex linear programming method."

- They may request enhancements to or modification of support tools.

- They may request modification to test procedures.

- They may request modifications to work around restrictions imposed by some compilation systems which will permit a variation of an existing test program to work under more systems.

- They may request a test problem be modified to work around a restriction on a system being tested. If the test problem was invalid Ada, then an error report, rather than a change request would be appropriate. A modification to an existing problem which makes it more portable would be a proper subject for a change request.

The depth of detail of a change request may vary. The more specific a request is, the easier it will be to respond to.

Change requests will be logged, evaluated and resolved.

After completing the form from the next page, it should be mailed to:

R. Szymanski
Ada Compiler Evaluation Capability, Change Request
WRDC/AAAF
Wright-Patterson AFB OH 45433-6543

ADA COMPILER EVALUATION CAPABILITY CHANGE REQUEST

ORIGINATOR IDENTIFICATION

Originator's Name _____
Organization _____
Address _____
Telephone _____
Date _____

SYSTEM IDENTIFICATION

ACEC VERSION _____
Compilation System Version _____
Host Operating System Version _____
Target Operating System Versior _____
Hardware Identification _____
(if a test program is submitted for incorporation into the ACEC, identify where it has been tested)

CHANGE DESCRIPTION AND JUSTIFICATION

(attach more pages if necessary)

13 NOTES

This section contains information only and is not contractually binding.

13.1 ABBREVIATIONS, ACRONYMS

ACEC	Ada Compiler Evaluation Capability
ACM	Association for Computing Machinery
BMA	Boeing Military Airplanes
CPU	Central Processing Unit
CSCI	Computer Software Configuration Item
DEC	Digital Equipment Corporation
I/O	Input / Output
LRM	(Ada) Language Reference Manual (MIL-STD-1815A)
MCCR	Mission Critical Computer Resource
NUMWG	Numerics Working Group (ACM SIGAda organization)
RTS	RunTime System
SIGAda	Special Interest Group on Ada (ACM sponsored organization)
SSA	Single System Analysis (proper name of an ACEC analysis tool)
VAX	Virtual Address eXtension (DEC family of processors)
VDD	Version Description Document
VMS	Virtual Memory System (DEC operating system for VAX processors)