ApTest

PowerPC EABI
TEST SUITE (PEATS)

Functional Specification

Version 1.0

April 8, 1996

Applied Testing and Technology, Inc.

Release	Date	Area	Modifications
1.0	April 8, 1996		Initial release

Applied Testing and Technology, Inc.
59 North Santa Cruz Avenue, Suite U
Los Gatos, CA 95030 USA

Voice: 408-399-1930
Fax: 408-399-1931
aptest@aptest.com

CONTENTS

List of Figures

1 Overview

Document objective

This document describes PEATS, the PowerPC Embedded ABI (EABI) Validation Test Suite.

Document scope

This document describes PEATS in terms of:

- architecture
- testing logic
- coverage
- user interface
- reports
- execution environment
- packaging
- documentation
- assertions
- expectations language

Associated references

- *PowerPC Embedded Application Binary Interface*, Version 1.0, Motorola, Inc., 1/10/95.
- *System V Application Binary Interface, PowerPC Processor Supplement*, Draft, Revision A, SunSoft, Part No. 802-3334-01, March 1995.
- *System V Application Binary Interface*, Novell, Inc., 1995.
- *DWARF Debugging Information Format*, Revision: 1.0.3, UNIX International Programming Languages SIG, (July 31, 1992).
- *The DejaGNU Testing Framework for DejaGNU Version 1.1*, Cygnus Support, Nov 1993.
- IEEE Std. 1003.3-1991 *Standard for Test Methods for Measuring Conformance to POSIX*. Institute of Electrical and Electronic Engineers, Inc., 1991.

2 Architecture

Overview of PEATS

PEATS is a complete package for installing, configuring, and running a large series of tests to validate compilers, linkers, and archivers for conformance to the EABI.

The deliverables include:

- installation and configuration aids
- logic to control test sequencing
- analysis programs that do the actual inspections and reporting on the operation of the tools and their conformance to standards
- test programs used to exercise the tools for evaluation purposes.

The major components of PEATS are shown in the following figure.

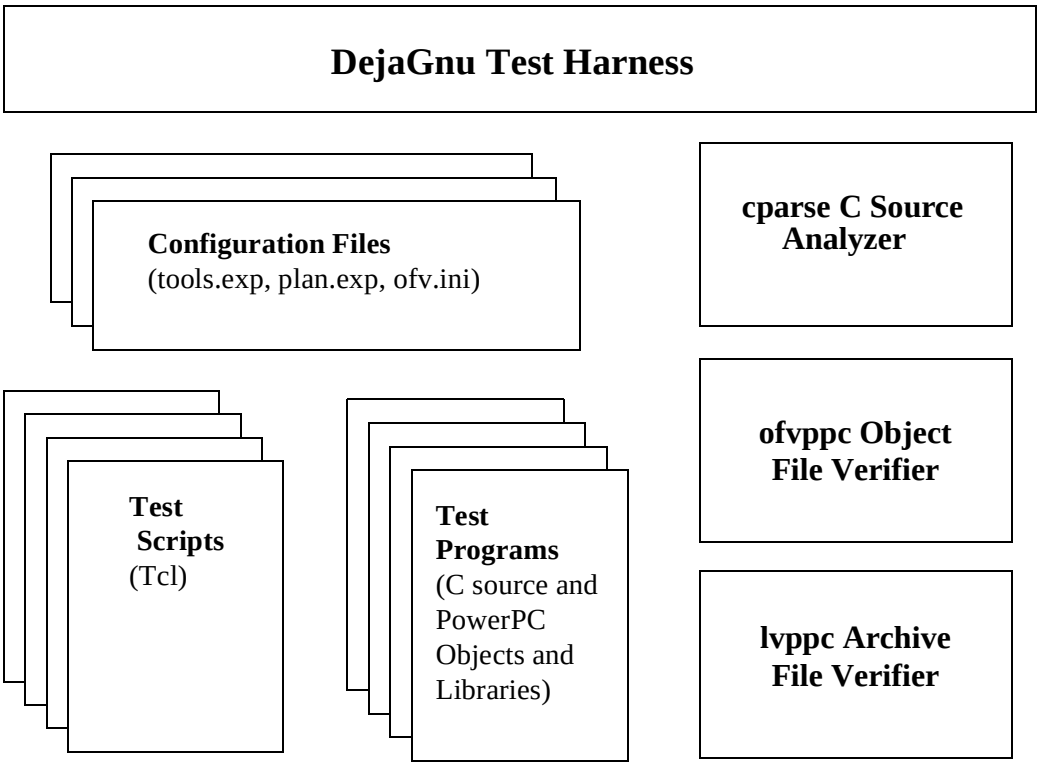

Figure 2-1 PEATS components

TCL test logic

The programs that contain the logic for running PEATS are written in the Tcl language. These programs include logic for:

- checking configuration information
- running the tools to be tested
- iterating through a series of test programs and variations
- running analysis programs to validate the output from the tools
- reporting the results

As delivered, PEATS has four "test variables" each with two possible values, the combinations of which create up to sixteen testable "variations" to be applied to every test program. These variables and their values are:

- debugging with or without debugging information (e.g., -g)
- dialect ISO or K&R variants of the C language
- optimization with or without optimization (e.g., -O)
- order big-endian or little-endian target byte order.

The Tcl programs are designed to be flexible; they are indifferent to the number of C test programs and to the exact definition of test variables and their values (i.e., indifferent to the number or meaning of the test variations), both of which can be extended by the PEATS user. The Tcl programs are also designed to make it easy to insert a new Tcl module to perform tests on some other tool (other than a compiler, linker, or archiver).

Analysis programs

PEATS validates compilers, linkers, and archivers by checking the correctness of object files generated by these tools. Checking is in terms of *assertions* of two types:

- *Syntactic* assertions are checks on the form and internal consistency of a single relocatable object file, or a linked object file and the object files which were input to the linker to produce the linked object file. Syntactic assertions do not require reference to C source programs.
- *Semantic* assertions are checks of consistency between C source files and corresponding object files.

PEATS includes the following programs (written in C) which are called by the Tcl programs to examine the inputs and outputs of the tools under test for the purpose of determining whether the tools conform to the applicable standards:

- source program analyzer (`cparse`) – analyzes a C source file and creates an ASCII file of *expectations* as to what should appear in the corresponding object file after compilation (semantic expectations)

- object file verifier (`ofvppc`) – analyzes ELF-DWARF object files to verify that syntactic and semantic expectations have been met, and in the case of linked files, that the input object files have been combined properly
- verifier for library archive files (`lvppc`) – analyzes archives to verify that syntactic and semantic expectations have been met, and that the archive is properly constructed from its component object files.

C test programs

C test programs are provided to exercise the tools being tested. The collection of test programs includes:

- single-module single-focus tests, such as a program containing many local variables, a program defining many types, etc.
- multi-module programs to exercise linkage capabilities such as sharing of definitions, external functions, and external variables
- a few programs that are intentionally quite large for the purpose of "stress testing" the tools by presenting them with a large number of types, functions, parameters, scopes, variables, etc.

Installation and configuration tools and methods

The PEATS distribution relies on the `tar` program in the UNIX® environment (PKZIP in the PC environment) to create required directories and to place program and data files into their proper locations. The distribution includes pre-built executable files for some commonly-used platforms (UNIX and Windows 95)and a `make` file to build or rebuild needed executable files on those and other platforms.

The distribution includes three configuration files that need to be adapted to a user's particular environment. Two files, `tools.exp` and `plan.exp`, are used to specify:

- the operating system commands for ordinary operations such as file copy, file move, etc.
- the names, standard options, and capabilities of the compiler, linker, and archiver to be tested
- the command-line options for selecting particular variations of language dialect, debugging information, optimization levels, and byte-ordering.

A third file, `ofv.ini`, conditions the object file verifier for the compiler and linker under test.

Use of such configuration files makes PEATS adaptable to a variety of different platforms without the need to change any of the delivered programs.

General framework for static testing of EABI files

Testing is highly automated so that one can start a large series of unattended tests with a single command. The major part of the testing is a static evaluation of the files produced by the compiler, linker, archiver, or similar tool in response to a known set of inputs and controls. In “Supplemental testing using run-time validation” on page 6 there is a discussion of additional testing that relies on run-time checking rather than static examination of program files.

DejaGNU framework

PEATS uses the DejaGNU test framework which was originally developed for testing GNU tools such as the GNU C compiler. This framework manages the automatic execution of a collection of tests and reports the results of those tests in summary and in detail.

Use of DejaGNU as the test framework offers a variety of benefits:

- compliance with POSIX standard 1003.3
- a uniform command-line interface to the testing process
- flexibility and extensibility to facilitate the development of new test cases
- portability across native and cross-development platforms.

Organization of directories

During installation of PEATS, a directory tree is created for the purpose of organizing the testing process. In the root directory of this tree are the configuration files for the testing process. The following are the major subdirectories:

- a subdirectory of Tcl programs containing the test logic for batch testing
- a subdirectory of test programs with branches (lower-level directories) for each individual test program – `test001`, `test002`, etc.
- a subdirectory for the analysis programs, with branches for the different analysis programs in source code form and a `bin` subdirectory for executables
- a “trusted” subdirectory for the outputs from a trusted tool set, with branches for each individual test program (see discussion of trusted objects below).

This organization makes it easy to find and modify the configuration files, add new test cases, or if necessary, modify or extend the test logic.

Use of trusted objects

Trusted object files are those produced by a “trusted” tool set when given PEATS’s standard input files (i.e., test programs) to process. The trusted tool set could be one that is known for its authority and reliability with respect to the ELF and DWARF

specifications, or it could be an earlier well-tested version of the same tool set on which PEATS is being run..

Use of trusted objects is an important part of the general framework for testing. Object files created by the tool set under test are linked with trusted object files (different components of the same program) to check that they can be successfully linked to form a complete program. Similarly, archives created by the tool set being tested are linked with trusted object files to check for interoperability.

Supplemental testing using run-time validation

Specialized run-time tests are provided to verify conformance in areas that are difficult to check statically. As an example, it is not easy to examine generated code and determine that function arguments are correctly evaluated and correctly passed. The most interesting cases for testing purposes may be those in which the argument expressions are quite complex and optimization of generated code obscures the connection between source code and generated code. As another example, correct data layout involving packing, alignment, and byte order is more easily validated by run-time tests. Such run-time tests require a suitable target environment.

The following checks rely on the use of well-understood test cases (source programs) and the examination of the resulting data structures in target memory and the behavior of test programs in execution:

- specific tests for proper caller-callee linkage, with verification of correct parameter passing and register preservation
- specific tests for correct data representation and layout

The supplemental run-time tests require a suitable target. See “Target Hardware” on page 25 for details. The target is connected to the host via an RS-232 serial communications link as shown below.

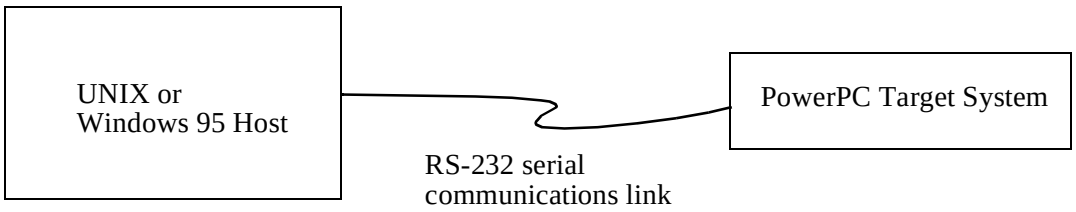

Figure 2-2 Host-target configuration for supplemental testing

Extensibility of PEATS

PEATS is designed for easy adaptation to the needs of particular users. There are several different directions users can take in extending PEATS.

Adding test programs

Users can provide additional test programs (C programs) to be run through PEATS. Such programs must conform to ISO standard C in order for the source analyzer to process them (certain pragmas and extensions for alignment and packing are accomodated, see "Editing configuration files" on page 13). A few simple steps will make a user's own test programs part of the static testing framework:

- each user program (one or more files) must be placed in its own test subdirectory
- each user program needs an accompanying Tcl control file in the same subdirectory with a single text line to invoke the test-sequencing logic.

Adding test variables

Users can modify or add test variables to create new variations in test runs. For example, instead of testing compilations with and without optimization, a user might want to test four different levels of optimization. Or a user might want to create a new "architecture" test variable to run test variations for the different target processor models supported by a tool set.

Testing other tools

PEATS as delivered is ready to test three tools, a compiler, a linker, and an archiver. However, the test logic is organized so that it is generally unaware of and indifferent to the exact number of tools to be tested. Instead, when a user asks for a "compiler" test run, an attempt is made to find a correspondingly-named Tcl module (`compiler.exp`). If the required Tcl module is found, then the testing can proceed. Following this pattern, a user could create a test for a new tool, say a "converter" tool, and by properly naming the test logic file for that tool (`converter.exp`), the new test will become part of the general framework for static testing and thereafter four tools will be testable.

3 Testing Logic

Framework

The framework for testing provides an iterative process that can perform a very large number of tests in a single run.

For example, if three tools are tested against 50 test programs with 16 variations of the test variables (with and without optimization, with and without debug, etc.) then there will be 2,400 (3x50x16) test variations run. Each single test, say a linker test, may involve many linker steps and verification of each link, so the full test sequence is highly combinatorial.

The user runs PEATS by invoking DejaGNU from the command line using the standard DejaGNU `runtest` command. Options to `runtest` allow selection of tests for specific tools or specific tests. Details are given later.

DejaGNU begins by reading the configuration files, including `plan.exp` which defines the test variables, e.g. debugging (with or without), byte-ordering (big- or little-endian), etc. The options on the command line are combined with the test variable definitions to determine what tests will be run. Some of the test programs carry out the supplemental testing for caller-callee usage and data layout. These tests require the target hardware, which must be connected and ready if these tests are to be run.

Once the configuration information is checked, DejaGNU cycles through all of the requested tests and produces summary and detail reports as requested.

Testing steps

In more detail,when PEATS is run the test logic proceeds as follows:

- CONFIGURATION: The configuration information is checked; configuration variables must be defined and have acceptable values.
- TEST PROGRAMS: Each test program in the sub-tree of test programs is completely processed before going on to the next test program.
- VARIATIONS: Each allowed variation of the test variables is fully applied before going on to the next variation. For example, testing is done first with the variation defined as the ISO language dialect, big-endian byte order, with debug information, and with optimization. For a given test program, each such variation is fully explored before going on to the next variation of the test variables.
- TOOLS TO BE TESTED: For a given test and variation, each tool is fully tested before going on to the next tool. For example, compiler checking is done first, then the linker is checked, and finally the archiver is checked.
- STEPS FOR EACH TOOL: As each tool is tested, the tool is applied and then its output is checked, then the tool is applied again and the output checked, until all steps for that tool have been completed. For example, the compiler is applied to

each source module in a test directory, whereas the linker is used repeatedly to link program modules in different groupings and sequences.

Figure 3-1 “Tool flow” below shows the tool flow for a single C source file.

- REPORTING: Every low-level step and its result (pass, fail, or unresolved) is listed in the detail report and tallied in the summary report.

Testing choices

Through settings in the configuration files and options on the DejaGNU `runtest` command line, a user can control the testing process without modifying any code.

By modifying the configuration file `plan.exp`, a user can disable any case or cases of any of the test variables. So, for example, if a user disables the little-endian byte ordering, all tests are run with the big-endian byte ordering and there are only half as many variations to be run for each test.

On the `runtest` command line, a user must specify which tools are to be tested. The standard choices are **compiler**, **linker**, **archiver**, or **all**. A user may also list which tests are to be used (e.g. `test001,test005,test020`); the default is to apply all tests. A mechanism is provided to exclude supplemental tests requiring target hardware.

The following figure shows the flow for a test of a single source file. When a test directory contains multiple source files, the same flow is carried out for each. If a linker test is requested, the object files in the test are linked to produce the file `test.out`, and this is then verified by `ofvppc`. If an archiver test is requested, the object files and archive are tested by `lvppc`.

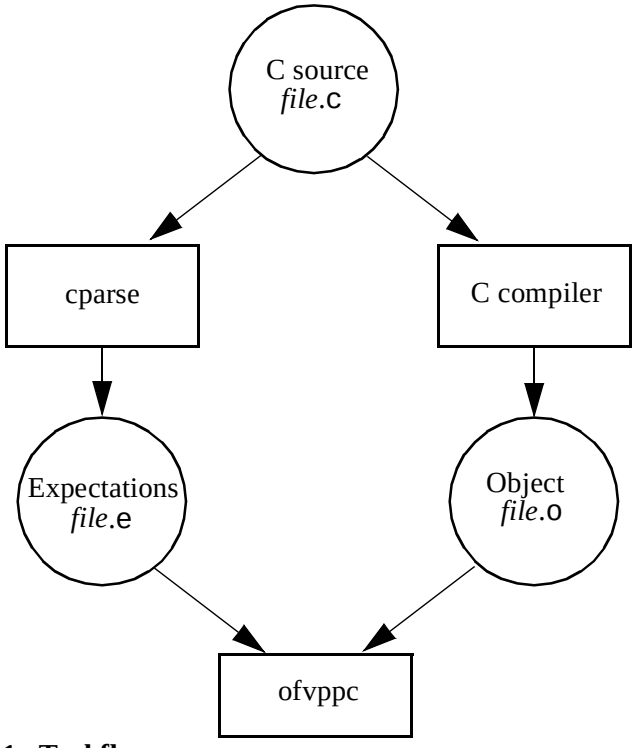

Figure 3-1 Tool flow

Exception handling

PEATS is prepared to handle several different kinds of exceptions that may arise during a run.

- At the start of each run, the test logic checks for the existence of certain needed files, the presence (definition) of certain needed variables, and the consistency of names which must correspond. If any of these prerequisite conditions is not satisfied, an error message is written and the run is abandoned.
- If the test logic encounters an internal inconsistency, resulting from a programming error in the distributed logic files or a user's added logic files, the test run is abandoned. If the error is not noticed internally but is noticed by the DejaGNU test framework, then the run will be halted by DejaGNU with a series of messages pinpointing the erroneous statement and its program context.
- If a tool under test writes any output messages, those are included in the test report. If the tool returns an error code or fails to produce its expected output file, then the attempt to validate the output from that tool is abandoned for that particular instance (the test is therefore "unresolved").
- Conformance errors detected by the analysis tools are written to the detailed report file and result in a score of "fail".
- If trusted object files are not found, then the tests which depend on those files are skipped.

4 Coverage

What is tested

The following are the general categories of checks that are made. For a detailed listing of assertions, see Appendix A, and for a list of all tests cases, see Appendix C.

- Object file syntactic checks
- Source-to-object semantic checks
- Object-to-object linker consistency checks
- Incremental linking
- Cross-linking (mixing optimized and unoptimized modules, etc.)
- Archive syntax checks
- Object-to-archive consistency checks
- Correct calculation of relocation expressions
- Archive usability checks
- Tool set interoperability (mixing with trusted objects)
- Run-time checks of caller-callee linkage, data representation, and data layout.

What is not tested

- Correct generation of caller-callee linkage, data representation, and relocation are all checked; instruction sequences are not otherwise checked.
- Provision is made for certain command line options, pragmas, and C extensions relating to alignment and packing. Other options and pragmas are ignored.
- Correct generation of debugging information for offsets of members within structures, including bit offsets for bit fields, is checked by examing applicable DWARF Location Descriptions. Location Descriptions are not otherwise checked.
- Frame information in `.tags` and related sections is checked for correct form and internal consistency, but is not checked semantically.
- Except for addresses, all aspects of DWARF Line Number Tables are checked, including existence of a statement for each entry.

Methods used to obtain broad coverage

For PEATS to be effective, it must be thorough in checking the many details of compliance with the standard (this is reflected in the many assertions listed in Appendix A). Furthermore, PEATS must be designed to ensure that its test runs will fully explore the behavior of the tools being tested so as to exercise all or almost all of the checking

that has been provided. The following strategies are used to force such broad coverage of the built-in checking capabilities.

- Many of the test programs are single-focus programs, designed to explore compliance with a particular aspect of the standard. Several careful readings of the standards have been used to look for additional items which can be tested through such focused testing.
- Several public domain programs are used to test the ability of the tools to handle large programs with a realistic mix of constructs; this is broad horizontal testing rather than focused testing.
- Testing is done across all defined variations of the test variables so as to exercise the tools in all supported combinations of relevant options. This checking seeks to expose problems that may be particular to optimization, debugging information, language dialect, etc.

In POSIX terminology, PEATS attempts to be EXHAUSTIVE in checking every syntactically testable aspect of every requirement in the standard. With respect to requirements that cannot be checked by syntactic examination alone, PEATS employs the following techniques.

- It analyzes source files and creates expectations that are used to enforce certain semantic requirements.
- It performs run-time checks to observe whether certain additional semantic requirements have been met.

Semantic checking is less than EXHAUSTIVE because not all possible checks are made. For example, no checks are made on program logic or computational expressions to ensure that semantics are preserved from source code to object code.

It should also be noted that PEATS evaluates only those objects and archive files that are derived from its test cases and those test cases are perforce only a sample of the possible source files that could be input to the tool set. In that sense, the entire Test Suite operates only at the THOROUGH level, providing sufficient testing to validate that the implementation under test matches the stated requirements by using representative data and testing boundary conditions.

Checks on coverage

In addition to the design and development steps taken to ensure broad coverage of the available checks, there is a a built-in mechanism for computer tracking of assertion coverage. Around every assertion in the verification programs there is code to record the exercise of that checking mechanism. By running the full suite of test programs and variations and reporting which of the assertion checks have been used, the set of unused checks can be identified. Supplementary test cases can then be provided to cover the previously assertions not previously reached. This method is used until all checks that reasonably can be exercised are in fact being exercised by the test programs.

5 User Interface

Installation and configuration

Installation of PEATS is not difficult. The product is delivered as a single "tar" file for UNIX platforms and as a "pkzip" file for Windows 95. The extraction process creates all necessary directories and places all files in the appropriate directories. Two additional steps are required to complete the installation:

- obtaining executable versions of the analysis programs (source code analyzer, object file verifier, and archive verifier) which can be run on the user's platform
- editing the configuration files to adapt PEATS to the user's environment – the platform and tools to be tested.

Building Test Suite components

PEATS is delivered with a Makefile for setting up the necessary executables.

For some commonly used platforms (listed in the *PEATS User's Guide*) PEATS includes pre-built executables. In such cases, a simple make command is all that is needed to install the executables. For example, on AIX, the command `make aix` copies the AIX executable into the `bin` directory where they are needed.

For other platforms, a user needs to review the Makefile and adapt it according to the instructions written in the Makefile. For example, a user will need to specify the name of the C compiler and the form of some options to be used. After adapting the Makefile, the user can run the `make` program to build the needed executables.

Editing configuration files

The files `tools.exp`, `plan.exp`, and `ofv.ini` in the root directory of PEATS contain configuration information that is a prerequisite to any testing. As distributed, these files contain sample values; these are not default values. The configuration files must be modified to match the environment in order to ensure correct operation of PEATS.

The file `tools.exp` contains settings that should only need to be configured once, during installation. These settings are expressed in the form of assignments to Tcl variables. A detailed list of these variables can be found in the *PEATS User's Guide*. Collectively, these variables provide the following information:

- invocation names (path specifications) of the tools to be tested – compiler, linker, and archiver
- standard option specifications to be used with the tools
- option designators to cause the tools to perform specific functions, e.g., the option needed to get the archiver to extract files from an archive

- names and options for operating system tools, e.g., make a directory, file copy, file move, and file remove
- standard extensions for source files, object files, and archive files.

In addition, `tools.exp` contains variables to indicate which cases of the test variables the tool set does or does not support.

The file `plan.exp` defines the test variables (*debug*, *dialect*, *optimization*, and *order*) the allowed cases of each variable, and the options used to effect those cases. It is the options that are of particular concern during installation and setup. For example, to include debugging information, there may be option designators for the compiler as well as for the linker; a user needs to specify these options correctly in order for PEATS to exercise the plan variations as intended.

`ofvppc` requires the file `ofv.ini` to condition it for the object files generated by the compiler and linker under test. Example settings in that file are:

- Prefix for architecture specific and vendor-supplied section names.
- Object file extension name.
- Type for a plain “char”.
- Type for a plain bit-field.
- Whether debugging information entries (DIEs) for function declarations for functions not referenced are required.
- Whether there shall be a lexical block DIE for the block representing the body of a function.
- Whether all nested lexical blocks shall have corresponding DIEs, or only those declaring names.
- Whether DIEs for unnamed bit-fields are required.
- Prefix and suffix attached to names of global and local functions and objects in the symbol table.

The file contains extensive comments and is self-documenting.

Runtest command line

To start PEATS, a user invokes the DejaGNU `runtest` program. The command line that invokes `runtest` determines which tools are tested, which test cases are used, and the detail reporting level. For additional flexibility, the command line also allows control of options used with particular tools during the run.

Basic syntax

The following is the general form of the `runtest` command line for starting PEATS:

`runtest --tool peats RUN=`*tool_name tests detail_level*

The specification `--tool peats` is required on every such command line – it specifies the directory and subdirectories containing the test programs of PEATS.

In the following example, compiler tests are applied to one test case (`test001`), and a verbose detail report file is written.

```
runtest --tool peats RUN=compiler test001.exp --verbose
```

Specifying the tools to be tested

On the `runtest` line, the phrase `RUN=` is used to indicate which of the tools is to be tested. In the sample above, the compiler is tested. The following *tool_name* keywords are recognized after the equal sign:

- **compiler** compiler tests (compile source files and verify object file)
- **linker** linker tests (link object files in different ways and verify object files)
- **archiver** archiver tests (build, verify, and use library files)
- **target** supplemental tests which execute on the target system
- **all** combination of compiler, linker, and archiver tests

Specifying the test cases to be used

On the `runtest` line, one can list the test programs to be used; if none are listed then all test cases in PEATS are used. Test programs must be located in the subdirectories under `peats`. In the default case, every file in the test directories of the form "*name*.exp" is assumed to contain the Tcl code to initiate a particular test case.

Controlling the detail reporting level

PEATS produces both a summary report and a detail report as described in Chapter "PEATS Reports" beginning on page 21. If a detail level is not specified on the `runtest` command line then the detail report will have the same information as the summary report.

Following DejaGNU conventions, a first level of detail is requested by adding `--verbose` to the `runtest` line. This provides detail about every testing variation and step in the run as well as information about every assertion that fails during the analysis of an object file or library file. For additional detail use the verbose modifier twice, e.g.,

```
runtest --tool peats RUN=all --verbose --verbose
```

The verbosity level also affects the details sent to the standard output during a run.

Passing options to specific tools

In special situations, it may be desirable to use the `runtest` command line to set certain options for particular tools. The following constructs can be used to add one or more options during a single test run:

CC_USING="*options*"	add options to invocations of the compiler
LD_USING="*options*"	add options to invocations of the linker
OFV_USING="*options*"	add options to invocations of the object verifier
LV_USING="*options*"	add options to invocation of the library verifier.

Options added in this way are inserted into the appropriate command lines immediately after those options specified in the configuration file `tools.exp` (see previous discussion in “Editing configuration files” on page 13).

As an example, the following command sets options for the compiler and the object verifier:

```
runtest --tool peats RUN=all CC_USING=-g OFV_USING=”-D -v2”
```

The quotes surrounding *options* may be omitted for a single simple option (one word, no spaces).

Command line syntax and options for the analysis tools

The analysis tools are invoked by PEATS with options from two possible sources:

- As specified in the configuration file `tools.exp`.
- From “*x*_USING” options on the `runtest` command line as described in “Passing options to specific tools” on page 16.

In addition, the analysis tools may be invoked from the command line to verify files manually.

The next three sections describe the analysis tools and their options. Use these descriptions to customize `tools.exp` or for manual invocation.

C source program analyzer

The program `cparse` analyzes a C source program and generates an *expectations file* used during semantic checking of an object file by `ofvppc`. The name of the expectations file is the same as that of the C source file but with the extension changed to “`.e`”. The following shows the usage of `cparse` (enter `cparse` without arguments to get this).

```
cparse (C File Parser for PEATS) version r.n mm-dd-yy
      Copyright (c) 1996 ApTest.
```

```
Generates expectations for object files based on C
      sourcefiles.

Usage:  cparse  [options]  <source_file>

Program options are:

   -o:filespec     use filespec for pathname of the output file
   -q              quiet mode (omit program identification)
   -r              make outer source file name relative (no
                   path)
```

For example, the command line:

```
cparse myprog.c
```

reads the C source file `myprog.c` and produces an expectation file called `myprog.e` which will later be used as input to the Object File Verifier.

Object File Verifier

The `ofvppc` program verifies the correctness of a single PowerPC object file. Correctness is verified in terms of assertions of two types: *syntactic* assertions which check form and consistency within the object file, and *semantic* assertions which check for consistency between the object file and its corresponding source file (see “Analysis programs” on page 3 for further discussion).

The following shows its command line interface (enter `ofvppc` without arguments to get this).

```
Usage: ofvppc [-abeDlMOPQSTV]
         [-C filename] [-c filename] [-d directory]
         [-p code|data|mixed] [-r attribute-report-filname]
         [-s area:assertion_id | :filename] [-u filename]
         [-v level] object-filename
```

- `-a` show all errors, do not stop after the first
- `-b` expect big-endian
- `-C` add to specified file all coverage messages (no files to be verified)
- `-c` add to specified file the coverage messages for assertions actually covered
- `-D` display file’s ELF/DWARF information
- `-e` verify ELF only
- `-l` do linked file analysis (the object-filname must be a linked output file

`-M`	suppress processing of semantic assertions for preprocessor directives
`-O`	do not check certain assertions likely to fail with optimized code
`-P`	print text of all assertions to standard output (no files to be verified)
`-p`	expect PIC, PID, or mixed (PIC and/or PID and/or plain) [may be repeated]
`-Q`	quiet mode - suppress final valid or invalid object file message
`-r`	update report of usage of attrbiutes-by-tag in specified file
`-S`	suppress processing of semantic assertions
`-s`	suppress specified assertions [may be repeated or be in specified file]
`-T`	do not dump attributes in the DWARF debugging information entries tree
`-u`	add unique assertions identifiers to the specified file
`-V`	print the version number
`-v`	print verbose error messages at the specified level

Notes:

1. The -s option suppresses an assertion and may be given in one of two forms:

   `-s <`*area*`>:<`*assertion_id*`>`
   `-s :<`*filename*`>`

   Each <*area*>:<*assertion_id*> combination identifies a single assertion. See Appendix A for a list of all valid *area* and *assertion_id* names. If the second form is used, the file should contain <*area*>:<*assertion_id*> pairs, one per line. Blank lines and lines beginning with a '/' character will be ignored and may be used to document the reason for suppressing assertions. There is a limit of 100 suppressed assertions.

   This option suppresses the *reporting* of the given assertion. However the assertion is still checked. In some cases, `ofvppc` may still fail if data related to the suppressed assertion is required by later code. Absent such a failure, an object file causing only suppressed assertions will be reported as valid.

Using ofvppc with linked files

The term *linked file* refers to an executable or relocatable (for incremental linking) file output by a linker. The files input to the linker to make a linked file are referred to as *contributing files*.

As the term is used here, "semantic" analysis for object files refers to validation of a single object file against the expectations generated by applying `cparse` to the corresponding C file (and any files it includes). If run without special options on a linked file (either executable or relocatable for incremental linking), `ofvppc` will assume it is to do semantic analsyis and will expect only one contributing file in the linked file.

There are a number of assertions specific to analysis of linked files ("Linked Objects" on page 114). For example, some of these linked-file assertions validate that all symbols

have been transformed correctly from the contributing files to the linked files, and that relocation expressions have been evaluated correctly. Use of these special assertions is called *linked-file analysis*.

Linked file analysis is requested by the -l options.

As noted in "Editing configuration files" on page 13, `ofvppc` also requires a file, `ofv.ini`, which provides the `ofvppc` program with information on certain defaults and extensions used by the compiler under test.

A sample `ofv.ini` file is delivered with PEATS and most be checked or modified by the user during installation to conform to the compiler under test.

Assertions

Assertions issued by `ofvppc` have a form illustrated by the following:

```
DBGPUNB: MISSING_PUBNAMES           A:Syn DWARF1 3.10.1
    A global object shall be represented by an offset/name
    pair in the ".debug_pubnames" section.
```

where:

`DBGPUBN`	The *assertion area*. The assertions are categorized into 13 areas, each of which typically relates to one type of section in the object file. Appendix A is organized by area, and the area name is used with the -s option to suppress individual assertions.
`MISSING_PUBNAME`	The *assertion-id*. These are unique within an *area* (see below), and are used with the -s option to suppress individual assertions.
`A`	One of two *assertion classes* as defined by POSIX 1003.3: **A** a base required assertion **C** a base conditional assertion
`Syn`	Indicates whether an assertion is syntactic ("Syn") or semantic ("Sem").
`DWARF1`	A short name that identifies the specification from which the assertion is derived. The -V option identifies the specifications supported by `ofvppc`.
`3.10.1`	A reference to the specification from which the assertion is derived. For DWARF assertions, the reference is a paragraph number; for other assertions, it is a section title.
`A global...`	The text of the assertion.

Library File Verifier

The `lvppc` program verifies the correctness of one or more PowerPC archive files (PEATS always invokes `ofvppc` with only one input object file on the command line).

While archives may contain files of any type, `lvppc` checks only object files within an archive. In all cases, it makes syntactic and semantic checks on the form of the archive, and consistency checks between the archive and the object files contained within it.

The `lvppc` command line interface is:

```
Usage: lvppc [-PQ] [-C filename][-c filename]
      [-s assertion_id | :filename] archive-filename
```

-C	add to specified file all coverage messages (no files to be verified)
-c	add to specified file the coverage messages for assertions actually covered
-P	print text of all assertions to standard output (no files to be verified)
-Q	quiet mode - suppress final valid or invalid object file message
-s	suppress specified assertions [may be repeated or be in specified file]
-V	print the version number

The format of `lvppc` assertions is similar to that of `ofvppc`.

6 PEATS Reports

DejaGnu reports

Two standard reports are produced by DejaGnu:

- A Summary Log containing the names of the tests run and their results.
- A Detailed Log showing the output generated by the tests.

Unless the `--outdir` option to `runtest` is used these files will be placed in the current working directory. The file names are *toolname*`.sum` and *toolname*`.log` where *toolname* is the name given to `runtest` with the `--tool` option. Thus, with PEATS, these are always `peats.sum` and `peats.log`.

Summary logs

The DejaGnu summary report provides a one-line summary of each result produced by PEATS. As each PEATS test generates and validates objects/archives multiple times from a single test program, multiple such lines will be included in the summary for each test. An excerpt from a sample summary log is shown in below.

```
Test Run By craig on Tue Oct 31 21:52:39 PST 1995
Native configuration is aix

                === peats tests ===

Running ./peats/test017/test017.exp ...
. . .
PASS:         verifying “tested/test017/nodebug/iso/no-opt/little/test017.o”
FAIL:         verifying “tested/test017/nodebug/iso/no-opt/little/test.out”
Running ./peats/test039/test039.exp ...
. . .
FAIL:         verifying “tested/test039/debug/iso/no-opt/little/root.out”
UNRESOLVED:   ERROR linking in directory
        “/usr/home/users/craig/work/tested/test039/debug/iso/no-opt/little”
WARNING: There were archiver messages for test.a
FAIL:         verifying “tested/test039/debug/iso/no-opt/little/test.out”
PASS:         verifying “tested/test039/nodebug/iso/opt/little/test039a.o”
. . .
                === peats Summary ===

# of expected passes 12
# of unexpected failures 18
# of unresolved testcases 6
```

Figure 6-1 Sample summary log

Detailed Logs

The content of the detailed log depends on the level of verbosity requested when PEATS is run. If the `--verbose` option to `runtest` is not used, the detailed log will be identical to the summary log.

If the `--verbose` option to `runtest` is used, the detailed log will be contain information about the test programs used and the details of failures: the assertion(s) failed and the reasons for the failure.

An extract from a detailed log for the C Compiler tests generated with `--verbose` on the `runtest` command and with the `-v2` option given to `ofvppc` follows.

```
Opening log files in .
Test Run By craig on Tue Oct 31 23:19:33 PST 1995
Native configuration is aix

. . .
=>=>=> TEST “test017” (in /usr/home/users/craig/work/ppc/test017)

=>=> VARIATION (With-Debug ISO No-Optimize Little-Endian)

=> STEP compiler-command -g test017.c

=> STEP tools/bin/ofvppc -a -Q -v0 -v2 test017.o
*** ERROR in verifying “tested/test017/debug/iso/no-opt/little/test017.o”
E_SHOFF_INVALID                          A Syntactic  ELF Header
    If the value of the ELF header’s e_shoff field does not equal zero the
    value shall be a multiple of 4.
    e_shoff= 0x42e7


                === peats Summary ===

# of expected passes 12
# of unexpected failures 18
# of unresolved testcases 6
```

Figure 6-2 Sample detail log

Reporting attribute usage based on semantic expectations

PEATS validates conformance of object files to standards with a goal of improving inter-operability of tools. Appendix 1 of the DWARF specification “enumerates the attributes that are most applicable to each type of debugging information entry.” However, it states that a “DWARF producer is free to generate any, all, or none of the attributes described in the text as being applicable to a given entry.”

The second quotation is an overstatement because many debugging information entries have attributes which are required by other sections of the specification. Nevertheless, except for such required attributes, PEATS cannot and does not report errors for an attribute listed for some entry in Appendix 1 but not present for some or all instances of that entry, even when it is clear that, for example, a debugger would need the attribute.

Thus, a compiler or linker not generating such attributes may still "conform" to the standard, even though its ability to inter-operate with other tools is compromised.

To help characterize the performance of the tools under test with respect to this debugging information, `ofvppc` can generate an "attribute usage" report showing the attributes used for each type of debugging information entry in the ".debug" section of an object file, and how this attribute usage compares to 1) the semantic expectations generated by the `cparse` analyzer, 2) Appendix 1 of the DWARF specification, and 3) usage by the "trusted" tool set.

The -r option to `ofvppc` specifies a file for generation of the attribute usage report. If the report file already exists, it is updated. As delivered, PEATS uses the -r option when invoking `ofvppc`, and deletes the file at the beginning of a run so that the report will show aggregate attribute usage for all tests made during the run.

An excerpt from an attribute usage report follows:

```
TAG_global_variable (fundamental type) 1388
   AT_sibling          ST      AT_fund_type        ST
   AT_name             ST      AT_location         ST

TAG_global_variable (modified fundamental type) 148/149
   AT_sibling          ST      AT_location         ST 147
   AT_name             ST      AT_start_scope      S  1/2
   AT_mod_fund_type    ST      AT_lo_user + 1         148
```

Each entry reports on one type of debugging information entry. The first line shows the tag or *sub-tag* (see below) naming the debugging information entry and a count of the form describing the tag's appearance in object files contributing to the report. The count is one of two forms:

n	The tag was expected to appear *n* times and did.
m / *n*	The tag was expected to appear the *n* number of times but actually appeared only the *m* number of times. This likely indicates an error.

For each attribute of each tag, the report shows:

- If present, the letter **S**, meaning that 1) the attribute appears in Appendix 1 of the DWARF specification (the "most applicable" attributes), and 2) that it is relevant for C programs and therefore "expected" by the `cparse` analyzer. A file containing the table of these expected attributes is delivered with PEATS.
- If present, the letter **T**, meaning that the attribute is used for this type of entry by the "trusted" tool set. A file giving this table is also delivered with the Tool Suite, and may be edited or replaced by the user if a new run of the trusted tool set is made.

  ➤ If neither an **S** nor a **T** is present, the attribute was generated by the tool under test but not listed in the DWARF Appendix 1 nor produced by the "trusted" tool set. The `AT_lo_user+1` attribute above is an example.

- If the attribute was expected and appeared the expected number of times, then no count is shown. Otherwise, one or two counts are shown, with meaning depending on the presence of the **S** and/or T letters (shown as *ST* in the table below):

ST	*m*	The attribute was expected to appear with every entry, but appeared only *m* times. This likely indicates an error.
ST	*m / n*	The attribute was expected *n* times but actually appeared only *m* times (where *n* < the number of appearances of the whole tag entry). The `AT_start_scope` attribute is an example. The case *m* = *n* is probably normal; *m* != *n* may represent anomalous behavior.
	m	The attribute was not expected by the `cparse` analyzer but appeared *m* times. The `AT_lo_user+1` attribute is an example.

The concept of *sub-tag* entries is used to differentiate expected variants of a type of debugging information entry. The example shown above shows two variants of the debugging information entry for global variables of a fundamental type and global variables of a modified fundamental type (two additional sub-tags for user-defined-type and modified-user-defined-type are not shown here). All of the first should have an `AT_fund_type` attribute while all of the second should have an `AT_mod_fund_type` attribute instead. If these were combined under a single entry, it would be necessary to use the *m*/*n* form for counts for this any perhaps related attributes, making it more difficult to spot anomalous behavior.

For entries which are present in either the DWARF specification or the “trusted” tool set, but which are not present in any occurrence of the tools under test covered by the report, only the first line of the entry is shown giving the entry tag or sub-tag name and a 0 count.

The report ends by characterizing the performance of the tools under test versus that of the DWARF specification and the “trusted” tool set. These characterizations show the number of unique debugging information entries (tags) and attributes detected for each of the three cases, and the total number of tag and attribute instances for the tools under test.

7 Execution Environment

UNIX

In order to run PEATS, a UNIX host environment needs to provide:

- A UNIX shell (if `ofvppc` and `lvppc` are be run stand-alone)
- Tools under test
- A native C development system (for building analysis tools — not necessary if using binaries shipped with the product, see Chapter "Packaging" beginning on page 26)
- DejaGnu (the distribution of which includes Tcl and Expect). DejaGnu must be installed on the implementation before PEATS is run. PEATS was developed and tested with DejaGnu version 1.2. This release of DejaGnu can be retrieved by anonymous ftp from the host `prep.ai.mit.edu`, file name `/pub/gnu/dejagnu-1.2.tar.gz`.
- The `make` command
- The `tar` command.

Windows 95

A Windows 95 host environment needs to provide:

- Tools under test
- The Microsoft Visual C/C++ development system (for building analysis tools — not necessary if using binaries shipped with the product)
- PKZIP for Windows. PKZIP is used to unpack the PEATS distribution, and also the DejaGnu and Tcl distributed with PEATS.

  PKZIP for Windows (as opposed to PKZIP for DOS) is required because DejaGnu includes files names not conforming to the DOS 8.3 standard. PKZIP for Windows is shareware and can be downloaded by pointing a World-Wide Web browser at `http://www.pkware.com`.

A modified version of DejaGnu, which includes Tcl, is shipped with PEATS for the Windows 95 environment.

Target Hardware

For the calling convention and data layout tests, the following target hardware and software, or their equivalent, are required:

<to be specified>

A serial RS-232 communications link from the host to the target is also required.

8 Packaging

PEATS is shipped in *tar* file format for UNIX platforms and *pkzip* format for the Windows 95 platform.

The UNIX tar file includes binaries for AIX systems.

The Window-95 pkzip file contains binaries for Intel or compatible microprocessors supporting Windows 95.

9 Documentation

PEATS includes two manuals. These documents are provided as postscript files.

- The *PEATS User's Guide* provides information on operating PEATS including:
  - Introduction, Overview, and Architecture
  - Installation and Configuration
  - Execution
  - Reports
  - Stand-alone use of the analysis tools
  - Debugging and trouble shooting
  - A complete list of all assertions
  - A complete list of all tests provided with PEATS.
- The *PEATS Programmer's Guide* provides information on maintaining and evolving PEATS, including:
  - Structure of PEATS, especially all directories.
  - Details of the testing process and instructions for modifying it and for modifying or adding tests.
  - Theory of operation of the analysis programs.

The *PEATS Programmer's Guide* is supplemented with in-line documentation of the PEATS source code.

Each release of PEATS also contains *Release Notes* documenting:

- Changes since the last release
- Known problems and work arounds

A Assertions

See “Assertions” on page 19 for details on the format of an assertion and the meaning its various fields.

OFVPPC Assertions

```
ofvppc supporting specifications:

SVR4 ABI
   System V Application Binary Interface
   Fourth Edition
   Novell, Inc.
   122 East 1700 South; Provo, UT 84606

SVR4 ABI PPC
   System V Application Binary Interface PowerPC Processor Supplement
   Revision A, March 1995
   SunSoft
   2550 Garcia Ave.; Mountain View, CA 94043
   Part No: 802-3334-01

PPC EABI
   PowerPC Embedded Application Binary Interface
   Version 1.0, 1/10/95
   Motorola, Inc
   6501 William Cannon Drive West; Austin, TX 78735
   Contact: Microcontroller Technologies Group

DWARF 1
   DWARF Debugging Information Format
   Revision: 1.0.3 (July 31, 1992)
   UNIX International
   Waterview Corporate Center; 20 Waterview Blvd.; Parsippany, NJ 07054
   Contact: Vice President of Marketing
```

ELF Header

```
HEADER:  EF_PPC_EMB_NOT_SET                A: Syn
         PPC EABI: 4. Machine Information
   The ELF header’s e_flags member shall have bit EF_PPC_EMB (0x80000000) set.

HEADER:  ET_TYPE_NOT_ET_EXEC               A: Sem  SVR4 ABI: 4. Introduction
   The ELF header’s e_type member shall equal ET_EXEC for an executable file.

HEADER:  ET_TYPE_NOT_ET_REL                A: Sem  SVR4 ABI: 4. Introduction
   The ELF header’s e_type member shall equal ET_REL for a relocatable file.

HEADER:  E_EHSIZE_INVALID                  A: Syn  SVR4 ABI: 4. Header
   The ELF header’s e_ehsize member shall equal sizeof(Elf32_Ehdr) = 52.
```

```
HEADER:  E_IDENT_EI_CLASS_INVALID          A: Syn
         SVR4 ABI: 4. ELF Identification
   The fifth byte of the ELF header's e_ident member (e_ident [EI_CLASS])
   shall equal ELFCLASS32 (1).

HEADER:  E_IDENT_EI_DATA_INVALID           A: Syn
         SVR4 ABI: 4. ELF Identification
   The sixth byte of the ELF header's e_ident member (e_ident [EI_DATA])
   shall be one of the following:

             Name           Value
             -----------    -----
             ELFDATA2LSB    1
             ELFDATA2MSB    2

HEADER:  E_IDENT_EI_DATA_WRONG             A: Sem
         SVR4 ABI: 4. Machine Information
   The ELF header's e_ident [EI_DATA] member shall be ELFDATA2MSB for a
   big-endian object file.

HEADER:  E_IDENT_EI_DATA_WRONG             A: Sem
         SVR4 ABI: 4. Machine Information
   The ELF header's e_ident [EI_DATA] member shall be ELFDATA2LSB for a
   little-endian object file.

HEADER:  E_IDENT_EI_MAG0_INVALID           A: Syn
         SVR4 ABI: 4. ELF Identification
   The first byte of the ELF header's e_ident member (e_ident [EI_MAG0])
   shall equal 0x7f.

HEADER:  E_IDENT_EI_MAG1_INVALID           A: Syn
         SVR4 ABI: 4. ELF Identification
   The second byte of the ELF header's e_ident member (e_ident [EI_MAG1])
   shall equal 0x45.

HEADER:  E_IDENT_EI_MAG2_INVALID           A: Syn
         SVR4 ABI: 4. ELF Identification
   The third byte of the ELF header's e_ident member (e_ident [EI_MAG2])
   shall equal 0x4c.

HEADER:  E_IDENT_EI_MAG3_INVALID           A: Syn
         SVR4 ABI: 4. ELF Identification
   The fourth byte of the ELF header's e_ident member (e_ident [EI_MAG3])
   shall equal 0x46.

HEADER:  E_IDENT_EI_VERSION_INVALID        A: Syn
         SVR4 ABI: 4. ELF Identification
   The seventh byte of the ELF header's e_ident member (e_ident[EI_VERSION])
   shall equal EV_CURRENT (1).

HEADER:  E_IDENT_PADDING_INVALID           A: Syn
```

```
         SVR4 ABI: 4. ELF Identification
   The eighth through sixteenth bytes of the ELF header's e_ident member
   (e_ident[EI_PAD] through e_ident[EI_NIDENT-1]) shall equal zero.

HEADER:  E_PHENTSIZE_TOO_SMALL             A: Syn  SVR4 ABI: 4. Header
   The e_phentsize member shall be either zero or at least sizeof
   (Elf32_Phdr) = 32.

HEADER:  E_PHENTSIZE_ZERO                  A: Syn  SVR4 ABI: 4. Header
   If the ELF header's e_phoff member does not equal zero its e_phentsize
   member shall not equal zero.

HEADER:  E_PHNUM_NON_ZERO                  A: Syn  SVR4 ABI: 4. Header
   If the ELF header's e_phoff member equals zero its e_phnum member shall
   equal zero.

HEADER:  E_PHNUM_ZERO                      A: Syn  SVR4 ABI: 4. Header
   If the ELF header's e_phoff member does not equal zero its e_phnum member
   shall not equal zero.

HEADER:  E_PHOFF_BEYOND_EOF                A: Syn  SVR4 ABI: 4. Header
   The ELF header's e_phoff member plus the minimum program header length of
   sizeof (Elf32_Phdr) = 32 shall be less than the size, in bytes, of the
   file.

HEADER:  E_PHOFF_MISALIGNED                A: Syn
         SVR4 ABI: 4. Data Representation
   The ELF header's e_phoff member shall equal 0 or a multiple of 4.

HEADER:  E_PHOFF_SHOULD_BE_NON_ZERO        A: Syn  SVR4 ABI: 4. Header
   If the ELF header's e_type member equals ET_EXEC (2) its e_phoff member
   shall not equal zero.

HEADER:  E_PHOFF_TOO_SMALL                 A: Syn  SVR4 ABI: 4. Header
   The ELF header's e_phoff member shall equal zero or a value greater than
   or equal to its e_ehsize member.

HEADER:  E_SHENTSIZE_ZERO                  A: Syn  SVR4 ABI: 4. Header
   If the ELF header's e_shoff member does not equal zero its e_shentsize
   member shall not equal zero.

HEADER:  E_SHNUM_NON_ZERO                  A: Syn  SVR4 ABI: 4. Header
   If the ELF header's e_shoff member equals zero its e_shnum member shall
   equal zero.

HEADER:  E_SHNUM_ZERO                      A: Syn  SVR4 ABI: 4. Header
   If the ELF header's e_shoff member does not equal zero its e_shnum member
   shall not equal zero.

HEADER:  E_SHOFF_MISALIGNED                A: Syn
         SVR4 ABI: 4. Data Representation
   The ELF header's e_shoff member shall be 0 or a multiple of 4.
```

```
HEADER:  E_SHOFF_TOO_SMALL                  A: Syn  SVR4 ABI: 4. Header
   The ELF header's e_shoff member shall equal zero or a value greater than
   or equal to its e_ehsize member.

HEADER:  E_SHOFF_ZERO_FOR_FOR_ET_REL        A: Syn
         SVR4 ABI: 4. Introduction: File Format
   If the ELF header's e_type member equals ET_REL (1) the value of its
   e_shoff member shall not equal zero.

HEADER:  E_SHSTRNDX_TOO_BIG                 A: Syn  SVR4 ABI: 4. Header
   If the ELF file header's e_shstrndx member is not SHN_UNDEF it shall be <=
   the ELF file header's e_shnum member.

HEADER:  E_TYPE_INVALID                     A: Syn  SVR4 ABI: 4. Header
   The ELF header's e_type member shall be one of the following:

            Name         Value
            ---------    ------
            ET_NONE           0
            ET_REL            1
            ET_EXEC           2
            ET_CORE           4
            ET_LOPROC    0xff00
            ...
            ET_HIPROC    0xffff

HEADER:  E_VERSION_INVALID                  A: Syn  SVR4 ABI: 4. Header
   The ELF header's e_version member shall equal EV_CURRENT (1).

HEADER:  PROGRAM_HEADER_BEYOND_EOF          A: Syn  SVR4 ABI: 4. Header
   If the ELF header's e_phoff member does not equal zero the result of
   (e_phoff + (e_phnum * e_phentsize)) shall equal a value not greater than
   the size, in bytes, of the file.

HEADER:  SECTION_HEADER_BEYOND_EOF          A: Syn  SVR4 ABI: 4. Header
   If the ELF header's e_shoff member does not equal zero the result of
   (e_shoff + (e_shnum * e_shentsize)) shall equal a value not greater than
   the size, in bytes, of the file.
```

ELF Program Header

```
PROGHDR: PT_LOAD_SEGMENTS_OUT_OF_ORDER      A: Syn  SVR4 ABI: 5. Program Header
   Segment entries in the program header with p_type equal PT_LOAD shall
   appear in ascending order, sorted on the p_vaddr member.

PROGHDR: PT_PHDR_E_PHOFF_INVALID            A: Syn  SVR4 ABI: 5. Program Header
   If a program header entry's p_type member equals PT_PHDR (6) the entry
   shall specify the segment containing the program header table.

PROGHDR: PT_PHDR_FOLLOWS_PT_LOAD            A: Syn  SVR4 ABI: 5. Program Header
```

A program header entry whose p_type member equals PT_PHDR (3) shall precede all other entries with p_type member equal to PT_LOAD (1).

PROGHDR: PT_PHDR_MORE_THAN_ONE A: Syn SVR4 ABI: 5. Program Header
A program header shall contain no more than one entry whose p_type member equals PT_PHDR (3).

PROGHDR: P_ALIGN_INVALID A: Syn SVR4 ABI: 5. Program Header
A program header entry's p_align member shall equal 0 or 1, or a positive integral power of 2.

PROGHDR: P_FILESZ_P_MEMSZ_INCONSISTENT A: Syn SVR4 ABI: 5. Program Header
If a program header entry's p_type member equals PT_LOAD (1) the entry's p_filesz member shall be less than or equal to its p_memsz member.

PROGHDR: P_FLAGS_INVALID A: Syn SVR4 ABI: 5. Program Header
Bits 3 through 27 of a program header entry's p_flags member shall equal zero.

PROGHDR: P_OFFSET_TOO_SMALL A: Syn SVR4 ABI: 5. Program Header
A program header entry's p_offset member shall equal a value greater than or equal to the ELF header's e_ehsize member.

PROGHDR: P_TYPE_INVALID A: Syn SVR4 ABI: 5. Program Header
The ELF program header's p_type member shall be one of the following:

Name	Value
PT_NULL	0
PT_LOAD	1
PT_NOTE	4
PT_PHDR	6
PT_LOPROC	0x70000000
...	
PT_HIPROC	0x7fffffff

PROGHDR: P_VADDR_OR_P_OFFSET_INVALID A: Syn SVR4 ABI: 5. Program Header
If its p_align member is greater than 1 the value of a program header entry's p_vaddr member shall equal its p_offset member modulo its p_align member.

PROGHDR: SEGMENT_BEYOND_EOF A: Syn SVR4 ABI: 5. Program Header
A program header entry's (p_filesz + p_offset) members shall be less than or equal to the size of the file in bytes.

ELF ".rela*" Section

RELOC: REL_TAGS_R_OFFSET_INVALID A: Syn
SVR4 ABI PPC: 4. Special Sections
The r_offset field of each entry in the ".rel.tags" section shall be divisible bythe size of each entry in the associated ".tags" section = 8.

RELOC: REL_TAGS_R_OFFSET_TOO_BIG A: Syn
SVR4 ABI PPC: 4. Special Sections
The r_offset field of each entry in the ".rel.tags" section shall not be larger than the size of the associated ".tags" section.

RELOC: R_OFFSET_TOO_BIG_RELOCATABLE A: Syn SVR4 ABI: 4. Relocation
The r_offset member of a relocation record in a relocatable file (Elf_Hdr.e_type equal ET_REL) shall be less than the size in bytes of the segment to which the relocation applies.

RELOC: R_SYM_TOO_BIG A: Syn SVR4 ABI: 4. Relocation
The ELF32_R_SYM value (the most significant 24 bits of a relocation table entry's r_info member) shall equal a value less than the number of entries in the corresponding symbol table.

RELOC: R_TYPE_INVALID A: Syn
PPC EABI: 4. Relocation Types
The following relocation types shall be supported:

Name	Value
R_PPC_NONE	0
R_PPC_ADDR32	1
R_PPC_ADDR24	2
R_PPC_ADDR16	3
R_PPC_ADDR16_LO	4
R_PPC_ADDR16_HI	5
R_PPC_ADDR16_HA	6
R_PPC_ADDR14	7
R_PPC_ADDR14_BRTAKEN	8
R_PPC_ADDR14_BRNTAKEN	9
R_PPC_REL24	10
R_PPC_REL14	11
R_PPC_REL14_BRTAKEN	12
R_PPC_REL14_BRNTAKEN	13
R_PPC_RELATIVE	22
R_PPC_UADDR32	24
R_PPC_UADDR16	25
R_PPC_REL32	26
R_PPC_SDAREL16	32
R_PPC_SECTOFF	33
R_PPC_SECTOFF_LO	34
R_PPC_SECTOFF_HI	35
R_PPC_SECTOFF_HA	36
R_PPC_EMB_NADDR32	101
R_PPC_EMB_NADDR	102
R_PPC_EMB_NADDR16_LO	103
R_PPC_EMB_NADDR16_HI	104
R_PPC_EMB_NADDR16_HA	105
R_PPC_EMB_SDAI16	106
R_PPC_EMB_SDA2I16	107
R_PPC_EMB_SDA2REL	108

```
R_PPC_EMB_SDA21          109
R_PPC_EMB_MRKREF         110
R_PPC_EMB_RELSEC16       111
R_PPC_EMB_RELST_LO       112
R_PPC_EMB_RELST_HI       113
R_PPC_EMB_RELST_HA       114
R_PPC_EMB_BIT_FLD        115
R_PPC_EMB_RELSDA         116
```

ELF Section Header Table

```
SECTBL:  E_SHSTRNDX_NEEDS_SHT_STRTAB       A: Syn  SVR4 ABI: 4. Sections
   The ELF header's e_shstrndx member shall contain zero or a valid section
   header index to a section whose sh_type member equals SHT_STRTAB (3).

SECTBL:  INITIAL_HEADER_NON_ZERO           A: Syn  SVR4 ABI: 4. Sections
   Each member in the initial section header table entry shall equal zero.

SECTBL:  SECTION_OVERLAP                   A: Syn  SVR4 ABI: 4. Sections
   Sections shall not overlap (sections with sh_type member equal to
   SHT_NOBITS (8) or with an sh_size value of 0 are ignored), nor shall any
   section overlap the ELF header, the section table, or the program header
   (if present).

SECTBL:  SECTION_OVERLAPS                  A: Syn  SVR4 ABI: 4. Sections
   Sections shall not overlap (sections with sh_type member equal to
   SHT_NOBITS (8) or with an sh_size value of 0 are ignored).

SECTBL:  SECTION_OVERLAPS_ELF_HEADER       A: Syn  SVR4 ABI: 4. Sections
   A section shall not overlap the ELF Header.

SECTBL:  SECTION_OVERLAPS_PROGRAM_HEADER   A: Syn  SVR4 ABI: 4. Sections
   A section shall not overlap the Program Header.

SECTBL:  SECTION_OVERLAPS_SECTION_TABLE    A: Syn  SVR4 ABI: 4. Sections
   A section shall not overlap the Section Table.

SECTBL:  SECTION_TABLE_BEYOND_EOF          A: Syn  SVR4 ABI: 4: Sections
   A section header table entry's (sh_offset + sh_size)  members shall
   contain a value less than the size of the file in bytes.

SECTBL:  SHT_RELA_MISALIGNED               A: Syn
         SVR4 ABI: 4. Data Representation
   If an ELF section header table entry's sh_type member equals or SHT_RELA
   (4) the entry's sh_offset member shall be a multiple of 4.

SECTBL:  SHT_RELA_SH_ENTSIZE_INVALID       A: Syn  SVR4 ABI: 4. Sections
   If a section header table entry's sh_type member equals SHT_RELA (4) the
   entry's sh_entsize member shall equal sizeof (Elf32_Rela) = 12.

SECTBL:  SHT_RELA_SH_INFO_INVALID          A: Syn  SVR4 ABI: 4. Sections
```

If a section header table entry's sh_type member equals SHT_RELA (4) the entry's sh_info member shall contain a valid section header table index.

SECTBL: SHT_RELA_SH_LINK_INVALID A: Syn SVR4 ABI: 4. Sections
If a section header table entry's sh_type member equals SHT_RELA (4) the entry's sh_link member shall contain a valid section header table index to a section header table entry whose sh_type member equals SHT_SYMTAB (2).

SECTBL: SHT_REL_UNEXPECTED A: Sem SVR4 ABI: 4. Sections
An executable file shall not contain a section table entry with an sh_type field equal to SHT_RELA.

SECTBL: SHT_SYMTAB_MISALIGNED A: Syn
SVR4 ABI: 4. Data Representation
If a section header table entry's sh_type member equals SHT_SYMTAB (2) and the entry's sh_offset member does not equal zero the value of its sh_offset member shall be a multiple of 4.

SECTBL: SHT_SYMTAB_MORE_THAN_ONE A: Syn SVR4 ABI: 4. Sections
An ELF object file shall have no more than one section header table entry whose sh_type member equals SHT_SYMTAB (2).

SECTBL: SHT_SYMTAB_SH_ENTSIZE_INVALID A: Syn SVR4 ABI: 4: Sections
If a section header table entry's sh_type member equals SHT_SYMTAB the value of its sh_entsize member shall be sizeof (Elf32_Sym) = 16

SECTBL: SHT_SYMTAB_WITH_NO_SHT_STRTAB A: Syn SVR4 ABI: 4. Sections
If a section header table entry's sh_type member equals SHT_SYMTAB (2) the entry's sh_link member shall equal a valid section header table index to a section header table entry whose sh_type member equals SHT_STRTAB (3).

SECTBL: SH_ADDRALIGN_INVALID A: Syn SVR4 ABI: 4: Sections
A section header table entry's sh_addralign member shall equal 0 or 1, or a positive integral power of 2.

SECTBL: SH_ENTSIZE_NOT_ZERO A: Syn SVR4 ABI: 4: Sections
A section header table entry's sh_entsize member shall equal zero unless the sh_type member equals one of the following:

Name	Value
SHT_RELA	4
SHT_SYMTAB	2
SHT_ORDERED	0x7fffffff

SECTBL: SH_FLAGS_HAS_INVALID_BITS A: Syn SVR4 ABI: 4: Sections
Bits 3 through 27 of a section header table entry's sh_flags member shall equal zero.

SECTBL: SH_INFO_NOT_ZERO A: Syn SVR4 ABI: 4: Sections
A section header table entry's sh_info member shall equal zero unless the sh_type member equals one of the following:

Name	Value
SHT_RELA	4
SHT_SYMTAB	2
SHT_ORDERED	0x7fffffff

SECTBL: SH_LINK_NOT_ZERO A: Syn SVR4 ABI: 4: Sections
A section header table entry's sh_link member shall equal zero unless the sh_type member equals one of the following:

Name	Value
SHT_RELA	4
SHT_SYMTAB	2
SHT_ORDERED	0x7fffffff

SECTBL: SH_NAME_TOO_BIG A: Syn SVR4 ABI: 4: Sections
If the file contains a section name string table (the ELF header's e_shstrndx is non-zero) then the sh_name member of a section header table entry shall be <= the size of the section name string table as given by the sh_size member of section header table entry for the section name string table.

SECTBL: SH_OFFSET_ZERO_WITH_SH_SIZE A: Syn SVR4 ABI: 4: Sections
If a section header table entry's sh_type member does not equal SHT_NOBITS (8) and the entry's sh_size member does not equal zero its sh_offset member shall not equal zero.

SECTBL: SH_TYPE_INVALID A: Syn SVR4 ABI: 4. Sections
A section header table entry's sh_type member shall equal one of the following values:

Name	Value
SHT_NULL	0
SHT_PROGBITS	1
SHT_SYMTAB	2
SHT_STRTAB	3
SHT_RELA	4
SHT_NOTE	7
SHT_NOBITS	8
SHT_LOPROC	0x70000000
...	
SHT_ORDERED	0x7fffffff
SHT_HIPROC	0x7fffffff
SHT_LOUSER	0x80000000
...	
SHT_HIUSER	0xffffffff

ELF Section Header Table - Special Sections

```
SPECSEC: DWARF_MISSING                    A: Sem
         PPC EABI: 3: DWARF Definition
   A file which is compiled with debugging information shall contain DWARF
   1.0.3 sections.

SPECSEC: NOTE_SH_SIZE_INVALID             A: Syn  SVR4 ABI: 4. Sections
   A section header table entry having an sh_name member referencing a string
   equal to ‘.note’ shall have an sh_size member divisible by four.

SPECSEC: PCC_EMB_SECTIONS_TOO_BIG         A: Syn
         PPC EABI: 4. Special Sections
   The sum of the sh_size members of the sections with section header table
   entries having sh_name members referencing strings equal to
   ".PPC.EMB.sbss0" and ".PPC.EMB.sdata0" shall not exceed 64K bytes.

SPECSEC: PPC_EMB_SBSS0_MORE_THAN_ONE      A: Syn
         PPC EABI: 4. Special Sections
   A file shall not contain more than one section header table entry having
   an sh_name member referencing a string equal to ".PPC.EMB.sbss0".

SPECSEC: PPC_EMB_SDATA0_MORE_THAN_ONE     A: Syn
         PPC EABI: 4. Special Sections
   A file shall not contain more than one section header table entry having
   an sh_name member referencing a string equal to ".PPC.EMB.sdata0".

SPECSEC: PPC_EMB_SEGINFO_MISALIGNED       A: Syn  SVR4 ABI PPC: 4. Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".PPC.EMB.seginfo" shall have an sh_addralign member equal to 0.

SPECSEC: REL_TAGS_SH_INFO_NOT_TAGS        A: Syn  SVR4 ABI PPC: 4. Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".rel.tags" shall have an sh_info member containing the section
   index of the associated ".tags" section.

SPECSEC: REL_TAGS_SH_LINK_NOT_TAGSYM      A: Syn  SVR4 ABI PPC: 4. Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".rel.tags" shall have an sh_link member containing the section
   index of the associated ".tagsym" section.

SPECSEC: SBSS2_MORE_THAN_ONE              A: Syn
         PPC EABI: 4. Special Sections
   A file shall not contain more than one section header table entry having
   an sh_name member referencing a string equal to ".sbss2".

SPECSEC: SBSS2_PLUS_SDATA2_TOO_BIG        A: Syn
         PPC EABI: 4. Special Sections
   The sum of the sh_size members of the sections with section header table
   entries having sh_name members referencing strings equal to ".sbss2" and
   ".sdata2" shall not exceed 64K bytes.
```

```
SPECSEC: SDATA2_MORE_THAN_ONE              A: Syn
         PPC EABI: 4. Special Sections
   A file shall not contain more than one section header table entry having
   an sh_name member referencing a string equal to ".sdata2".

SPECSEC: SHT_SYMTAB_MISSING                A: Sem  PPC EABI: DWARF Definition
   A file which is compiled with debugging information shall contain a
   section table entry with an sh_type value equal SHT_SYMTAB.

SPECSEC: SH_FLAGS_FOR_BSS                  A: Syn
         SVR4 ABI: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".bss" shall have standard attributes (sh_flags & ~SHF_MASKPROC)
   equal to SHF_ALLOC (2) + SHF_WRITE (1).

SPECSEC: SH_FLAGS_FOR_COMMENT              A: Syn
         SVR4 ABI: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".comment" shall have standard attributes (sh_flags &
   ~SHF_MASKPROC) equal to zero.

SPECSEC: SH_FLAGS_FOR_DATA                 A: Syn
         SVR4 ABI: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".data" shall have standard attributes (sh_flags & ~SHF_MASKPROC)
   equal to SHF_ALLOC (2) + SHF_WRITE (1).

SPECSEC: SH_FLAGS_FOR_DATA1                A: Syn
         SVR4 ABI: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".data1" shall have standard attributes (sh_flags &
   ~SHF_MASKPROC) equal to SHF_ALLOC (2) + SHF_WRITE (1).

SPECSEC: SH_FLAGS_FOR_DEBUG                A: Syn
         SVR4 ABI: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".debug" shall have standard attributes (sh_flags &
   ~SHF_MASKPROC) equal to zero

SPECSEC: SH_FLAGS_FOR_DEBUG_ARANGES        A: Syn
         SVR4 ABI: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".debug_aranges" shall have standard attributes (sh_flags &
   ~SHF_MASKPROC) equal to zero

SPECSEC: SH_FLAGS_FOR_DEBUG_PUBNAMES       A: Syn
         SVR4 ABI: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".debug_pubnames" shall have standard attributes (sh_flags &
   ~SHF_MASKPROC) equal to zero

SPECSEC: SH_FLAGS_FOR_FINI                 A: Syn
```

```
         SVR4 ABI: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".fini" shall have standard attributes (sh_flags & ~SHF_MASKPROC)
   equal to SHF_ALLOC (2) + SHF_EXECINSTR (4).

SPECSEC: SH_FLAGS_FOR_INIT                A: Syn
         SVR4 ABI: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".init" shall have standard attributes (sh_flags & ~SHF_MASKPROC)
   equal to SHF_ALLOC (2) + SHF_EXECINSTR (4).

SPECSEC: SH_FLAGS_FOR_NOTE                A: Syn
         SVR4 ABI: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".note" shall have standard attributes (sh_flags & ~SHF_MASKPROC)
   equal to zero.

SPECSEC: SH_FLAGS_FOR_PPC_EMB_SBSS0       A: Syn
         SVR4 ABI PPC: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".PPC.EMB.sbss0" shall have standard attributes (sh_flags &
   ~SHF_MASKPROC) equal to SHF_ALLOC (2) + SHF_WRITE (1).

SPECSEC: SH_FLAGS_FOR_PPC_EMB_SDATA0      A: Syn
         SVR4 ABI PPC: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".PPC.EMB.sdata0" shall have standard attributes (sh_flags &
   ~SHF_MASKPROC) equal to SHF_ALLOC (2) + SHF_WRITE (1).

SPECSEC: SH_FLAGS_FOR_PPC_EMB_SEGINFO     A: Syn
         SVR4 ABI PPC: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".PPC.EMB.seginfo" shall have standard attributes (sh_flags &
   ~SHF_MASKPROC) equal to zero.

SPECSEC: SH_FLAGS_FOR_RELA                A: Syn
         SVR4 ABI: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".rela*" shall have standard attributes (sh_flags &
   ~SHF_MASKPROC) that include the SHF_ALLOC (2) if the file has a loadable
   segment that includes relocation, otherwise the bit will be off.

SPECSEC: SH_FLAGS_FOR_REL_TAGS            A: Syn
         SVR4 ABI PPC: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".rel.tags" shall have standard attributes (sh_flags &
   ~SHF_MASKPROC) equal to SHF_EXCLUDE (0x80000000).

SPECSEC: SH_FLAGS_FOR_RODATA              A: Syn
         SVR4 ABI: 4. Special Sections
```

```
   A section header table entry having an sh_name member referencing a string
   equal to ".rodata" shall have standard attributes (sh_flags &
   ~SHF_MASKPROC) equal to SHF_ALLOC (2).

SPECSEC: SH_FLAGS_FOR_RODATA1              A: Syn
         SVR4 ABI: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".rodata1" shall have standard attributes (sh_flags &
   ~SHF_MASKPROC) equal to SHF_ALLOC (2).

SPECSEC: SH_FLAGS_FOR_SBSS                 A: Syn
         SVR4 ABI PPC: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".sbss" shall have standard attributes (sh_flags & ~SHF_MASKPROC)
   equal to SHF_ALLOC (2) + SHF_WRITE (1).

SPECSEC: SH_FLAGS_FOR_SBSS2                A: Syn
         SVR4 ABI PPC: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".sbss2" shall have standard attributes (sh_flags &
   ~SHF_MASKPROC) equal to SHF_ALLOC (2) + SHF_WRITE (1).

SPECSEC: SH_FLAGS_FOR_SDATA                A: Syn
         SVR4 ABI PPC: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".sdata" shall have standard attributes (sh_flags &
   ~SHF_MASKPROC) equal to SHF_ALLOC (2) + SHF_WRITE (1).

SPECSEC: SH_FLAGS_FOR_SDATA2               A: Syn
         SVR4 ABI PPC: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".sdata2" shall have standard attributes (sh_flags &
   ~SHF_MASKPROC) equal to SHF_ALLOC (2) or SHF_ALLOC (2) + SHF_WRITE (1).

SPECSEC: SH_FLAGS_FOR_SHSTRTAB             A: Syn
         SVR4 ABI: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".shstrtab" shall have standard attributes (sh_flags &
   ~SHF_MASKPROC) equal to zero.

SPECSEC: SH_FLAGS_FOR_STRTAB               A: Syn
         SVR4 ABI: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".strtab" shall have standard attributes (sh_flags &
   ~SHF_MASKPROC) that include the SHF_ALLOC (2) if the file has a loadable
   segment that includes the symbol string table, otherwise the bit will be
   off.

SPECSEC: SH_FLAGS_FOR_SYMTAB               A: Syn
         SVR4 ABI: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".symtab" shall have standard attributes (sh_flags &
```

```
   ~SHF_MASKPROC) that include the SHF_ALLOC (2) if the file has a loadable
   segment that includes the symbol table, otherwise the bit will be off.

SPECSEC: SH_FLAGS_FOR_TAGS                A: Syn
         SVR4 ABI PPC: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".tags" shall have standard attributes (sh_flags & ~SHF_MASKPROC)
   equal to SHF_ALLOC (2).

SPECSEC: SH_FLAGS_FOR_TAGSYM              A: Syn
         SVR4 ABI PPC: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".tagsym" shall have standard attributes (sh_flags &
   ~SHF_MASKPROC) equal to SHF_EXCLUDE (0x80000000).

SPECSEC: SH_FLAGS_FOR_TEXT                A: Syn
         SVR4 ABI: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".text" shall have standard attributes (sh_flags & ~SHF_MASKPROC)
   equal to SHF_ALLOC (2) + SHF_EXECINSTR (4).

SPECSEC: SH_TYPE_FOR_BSS                  A: Syn
         SVR4 ABI: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".bss" shall have an sh_type member equal to SHT_NOBITS (8).

SPECSEC: SH_TYPE_FOR_COMMENT              A: Syn
         SVR4 ABI: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".comment" shall have an sh_type member equal to SHT_PROGBITS (1).

SPECSEC: SH_TYPE_FOR_DATA                 A: Syn
         SVR4 ABI: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".data" shall have an sh_type member equal to SHT_PROGBITS (1).

SPECSEC: SH_TYPE_FOR_DATA1                A: Syn
         SVR4 ABI: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".data1" shall have an sh_type member equal to SHT_PROGBITS (1).

SPECSEC: SH_TYPE_FOR_DEBUG                A: Syn
         SVR4 ABI: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".debug" shall have an sh_type member equal to SHT_PROGBITS (1).

SPECSEC: SH_TYPE_FOR_DEBUG_ARANGES        A: Syn
         SVR4 ABI: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".debug_aranges" shall have an sh_type member equal to
   SHT_PROGBITS (1).
```

```
SPECSEC: SH_TYPE_FOR_DEBUG_PUBNAMES        A: Syn
         SVR4 ABI: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".debug_pubnames" shall have an sh_type member equal to
   SHT_PROGBITS (1).

SPECSEC: SH_TYPE_FOR_FINI                  A: Syn
         SVR4 ABI: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".fini" shall have an sh_type member equal to SHT_PROGBITS (1).

SPECSEC: SH_TYPE_FOR_INIT                  A: Syn
         SVR4 ABI: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".init" shall have an sh_type member equal to SHT_PROGBITS (1).

SPECSEC: SH_TYPE_FOR_LINE                  A: Syn
         SVR4 ABI: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".line" shall have an sh_type member equal to SHT_PROGBITS (1).

SPECSEC: SH_TYPE_FOR_NOTE                  A: Syn
         SVR4 ABI: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".note" shall have an sh_type member equal to SHT_NOTE (7).

SPECSEC: SH_TYPE_FOR_PPC_EMB_SBSS0         A: Syn
         SVR4 ABI: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".PPC.EMB.sbss0" shall have an sh_type member equal to SHT_NOBITS
   (8).

SPECSEC: SH_TYPE_FOR_PPC_EMB_SDATA0        A: Syn
         SVR4 ABI: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".PPC.EMB.sdata0" shall have an sh_type member equal to
   SHT_PROGBITS (1).

SPECSEC: SH_TYPE_FOR_PPC_EMB_SEGINFO       A: Syn
         SVR4 ABI: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".PPC.EMB.seginfo" shall have an sh_type member equal to
   SHT_PROGBITS (1).

SPECSEC: SH_TYPE_FOR_RELA                  A: Syn
         SVR4 ABI: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".rela" shall have an sh_type member equal to SHT_RELA (4).

SPECSEC: SH_TYPE_FOR_REL_TAGS              A: Syn
         SVR4 ABI: 4. Special Sections
```

```
   A section header table entry having an sh_name member referencing a string
   equal to ".rel.tags" shall have an sh_type member equal to SHT_REL (9).

SPECSEC: SH_TYPE_FOR_RODATA                A: Syn
         SVR4 ABI: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".rodata" shall have an sh_type member equal to SHT_PROGBITS (1).

SPECSEC: SH_TYPE_FOR_RODATA1               A: Syn
         SVR4 ABI: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".rodata1" shall have an sh_type member equal to SHT_PROGBITS (1).

SPECSEC: SH_TYPE_FOR_SBSS                  A: Syn
         SVR4 ABI: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".sbss" shall have an sh_type member equal to SHT_NOBITS (8).

SPECSEC: SH_TYPE_FOR_SBSS2                 A: Syn
         SVR4 ABI: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".sbss2" shall have an sh_type member equal to SHT_NOBITS (8).

SPECSEC: SH_TYPE_FOR_SDATA                 A: Syn
         SVR4 ABI: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".sdata" shall have an sh_type member equal to SHT_PROGBITS (1).

SPECSEC: SH_TYPE_FOR_SDATA2                A: Syn
         SVR4 ABI: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".sdata2" shall have an sh_type member equal to SHT_PROGBITS (1).

SPECSEC: SH_TYPE_FOR_SHSTRTAB              A: Syn
         SVR4 ABI: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".shstrtab" shall have an sh_type member equal to SHT_STRTAB (3).

SPECSEC: SH_TYPE_FOR_STRTAB                A: Syn
         SVR4 ABI: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".strtab" shall have an sh_type member equal to SHT_STRTAB (3).

SPECSEC: SH_TYPE_FOR_SYMTAB                A: Syn
         SVR4 ABI: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".symtab" shall have an sh_type member equal to SHT_SYMTAB (2).

SPECSEC: SH_TYPE_FOR_TAGS                  A: Syn
         SVR4 ABI: 4. Special Sections
```

```
   A section header table entry having an sh_name member referencing a string
   equal to ".tags" shall have an sh_type member equal to SHT_ORDERED
   (0x7fffffff).

SPECSEC: SH_TYPE_FOR_TAGSYM                A: Syn
         SVR4 ABI: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".tagsym" shall have an sh_type member equal to SHT_SYMTAB (2).

SPECSEC: SH_TYPE_FOR_TEXT                  A: Syn
         SVR4 ABI: 4. Special Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".text" shall have an sh_type member equal to SHT_PROGBITS (1).

SPECSEC: TAGSYM_IN_WRONG_FILE_TYPE         A: Syn  SVR4 ABI PPC: 4. Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".tagsym" shall appear only in an object file (a file with an ELF
   header e_type member equal to ET_REL (1)).

SPECSEC: TAGS_MISALIGNED                   A: Syn  SVR4 ABI PPC: 4. Sections
   A section header table entry having an sh_name member referencing a string
   equal to ".tags" shall have an sh_addralign member equal to 4.
```

ELF ".strtab" Section

```
STRTBL:  FIRST_BYTE_NULL                   A: Syn  SVR4 ABI: 4. String Table
   The first byte of a string table section shall equal a null character.

STRTBL:  STRING_TABLE_MISALIGNED           A: Syn  SVR4 ABI: 4. String Table
   If a section header table entry's sh_type member equals SHT_STRTAB (3) and
   the entry's sh_offset member does not equal zero the value of its
   sh_offset member shall be a multiple of 4.

STRTBL:  UNTERMINATED_STRING               A: Syn  SVR4 ABI: 4. String Table
   The last byte of a string table section (at an offset in the file equal to
   the section header table entry's sh_offset member + the entry's sh_size
   member - 1) shall equal a null character.
```

ELF ".symtab" Section

```
SYMTBL:  BINDING_WRONG                     A: Sem  SVR4 ABI: 4. Symbol Table
   The binding of an object in the ELF symbol table, as given by its
   ELF32_ST_BIND value shall match that of the type of the corresponding
   object in the C source -- an STB_GLOBAL or STB_WEAK symbol shall
   correspond to a C source object which is externally visible outside the
   source file and an STB_LOCAL symbol shall correspond to a C source object
   which is not externally visible (is "static" or local).

SYMTBL:  FILE_NAME_WRONG                   A: Sem  SVR4 ABI: 4. Symbol Table
```

The name given by a symbol table entry having an ELF32_ST_TYPE value (the lower four bits of its st_info member) equal to STT_FILE (4) shall be that of the source file associated with the object file.

SYMTBL: FIRST_ENTRY_NON_ZERO A: Syn SVR4 ABI: 4. Symbol Table
Each member of the initial entry in the symbol table entry shall equal zero.

SYMTBL: FUNCTION_UNEPXECTED A: Sem SVR4 ABI: 4. Symbol Table
An entry in the symbol table having an ELF32_ST_TYPE value equal to STT_FUNC shall correspond to a function included in the corresponding C source code with a name which matches after prepending one underscore to the name from the source.

SYMTBL: GLOBAL_FUNCTION_TYPE_WRONG A: Sem SVR4 ABI: 4. Symbol Table
A global function in the source shall have a corresponding entry in the symbol table with an ELF32_ST_BIND value of STB_FUNC.

SYMTBL: GLOBAL_NAME_MISSING A: Sem SVR4 ABI: 4. Symbol Table
A global name in the source shall have a corresponding entry in the symbol table.

SYMTBL: GLOBAL_OBJECT_TYPE_WRONG A: Sem SVR4 ABI: 4. Symbol Table
A global data object in the source shall have a corresponding entry in the symbol table with an ELF32_ST_BIND value of STB_GLOBAL or STB_WEAK.

SYMTBL: LOCAL_NAME_MISSING A: Sem SVR4 ABI: 4. Symbol Table
A function or statically allocated data object which is not externally visible outside a module shall have a corresponding entry in the symbol table with an ELF32_ST_BIND value of STB_LOCAL.

SYMTBL: OBJECT_UNEPXECTED A: Sem SVR4 ABI: 4. Symbol Table
An entry in the symbol table having an ELF32_ST_TYPE value equal to STT_OBJECT shall correspond to a data object included in the corresponding C source code with a name which matches after prepending one underscore to the name from the source.

SYMTBL: SHT_SYMTAB_SH_INFO_INVALID A: Syn SVR4 ABI: 4. Sections
If a section header table entry’s sh_type member equals SHT_SYMTAB (2) the entry’s sh_info member shall be one greater than the symbol table index of the last local symbol (binding STB_LOCAL).

SYMTBL: SIZE_WRONG A: Sem SVR4 ABI: 4. Symbol Table
If the size of an object in the C source code can be determined, then the st_size member of the entry in the symbol table for that object shall equal that value.

SYMTBL: STB_LOCAL_FOLLOWS_STB_GLOBAL A: Syn SVR4 ABI: 4. Symbol Table
A symbol table entry having its ELF32_ST_BIND value (the upper four bits of its st_info member) equal to STB_LOCAL (0) shall precede every symbol table entry with an ELF32_ST_BIND value equal STB_WEAK (2) or STB_GLOBAL (1).

SYMTBL: STT_FILE_HAS_WRONG_ST_BIND A: Syn SVR4 ABI: 4. Symbol Table
If the ELF32_ST_TYPE value of an ELF symbol table entry (the lower four bits of its st_info member) equal STT_FILE (4), then its ELF32_ST_BIND value (the upper four bits of its st_info member) shall equal STB_LOCAL (0).

SYMTBL: STT_FILE_HAS_WRONG_ST_SHNDX A: Syn SVR4 ABI: 4. Symbol Table
If the ELF32_ST_TYPE value of an ELF symbol table entry (the lower four bits of its st_info member) equal STT_FILE (4), then its st_shndx member shall equal SHN_ABS (0xfff1).

SYMTBL: ST_BIND_INVALID A: Syn SVR4 ABI: 4. Symbol Table
The ELF32_ST_BIND value of a symbol table entry (the upper four bits of its st_info member) shall be one of the following:

Name	Value
STB_LOCAL	0
STB_GLOBAL	1
STB_WEAK	2
STB_LOPROC	13
...	
STB_HIPROC	15

SYMTBL: ST_OTHER_INVALID A: Syn SVR4 ABI: 4. Symbol Table
A symbol table entry’s st_other member shall equal zero.

SYMTBL: ST_SHNDX_TOO_BIG A: Syn SVR4 ABI: 4. Symbol Table
If a symbol table entry’s st_shndx member is less than SHN_LORESERVE (0xff00) its st_shndx member shall be less than the ELF header’s sh_shnum member.

SYMTBL: ST_TYPE_INVALID A: Syn SVR4 ABI: 4. Symbol Table
The ELF32_ST_TYPE value of a symbol table entry (the lower four bits of its st_info member) shall be one of the following:

Name	Value
STT_NOTYPE	0
STT_OBJECT	1
STT_FUNC	2
STT_SECTION	3
STT_FILE	4
STT_LOPROC	13
...	
STT_HIPROC	15

SYMTBL: SYMBOL_TABLE_MISALIGNED A: Syn
SVR4 ABI: 4. Data Representation

```
   If a section header table entry's sh_type member equals SHT_SYMTAB (2) and
   the entry's sh_offset member does not equal zero the value of its
   sh_offset member shall be a multiple of 4.

SYMTBL:  TAGSYM_ST_BIND_WRONG              A: Syn
         SVR4 ABI PPC: 4: Special Sections
   The ELF32_ST_BIND value of an entry in the ".tagsym" section shall be
   STB_LOCAL.

SYMTBL:  TAGSYM_ST_TYPE_WRONG              A: Syn
         SVR4 ABI PPC: 4: Special Sections
   The ELF32_ST_TYPE value of an entry in the ".tagsym" section shall be
   ST_NOTYPE.

SYMTBL:  _SDA2_BASE_MISSING                A: Syn
         PPC EABI: 4. Special Sections
   There shall be a symbol table entry for the symbol "_SDA2_BASE_".

SYMTBL:  _SDA2_BASE_NON_ZERO               A: Syn
         PPC EABI: 4. Special Sections
   A file which does not have a section header table entry having an sh_name
   member referencing strings equal to either ".sbss2" or ".sdata2" shall
   have a symbol table entry with an st_name member referencing a string
   equal to "_SDA2_BASE_" and with an st_value member of zero.

SYMTBL:  _SDA2_BASE_TOO_FAR_AWAY           A: Syn
         PPC EABI: 4. Special Sections
   A file which has section header table entries having sh_name members
   referencing strings equal to either ".sbss2" or ".sdata2" shall have a
   symbol table entry with an st_name member referencing a string equal to
   "_SDA2_BASE_" and with st_value such that the address of any byte in
   ".sbss2" or ".sdata2" is within a 16-bit signed offset of "_SDA2_BASE_".
```

ELF ".tags" Section

```
   <To be supplied>
```

".debug" Section

```
DEBUG:   ABSTRACT_PARAMETER_HAS_LOCATION   A: Syn  DWARF 1: 3.3.6
   The children which describe the parameters of a TAG_global_subroutine or
   TAG_subroutine entry which is an abstract declaration of an inlined
   subroutine shall not have location descriptions.

DEBUG:   ARRAY_BYTE_SIZE_WRONG             C: Sem  DWARF 1: 3.8.3
   If a TAG_array_type debugging information entry has a AT_byte_size
   attribute, then the value of that attribute shall be the size of the
   entire array as determined statically at compile-time.

DEBUG:   ARRAY_DIMENSION_MISSING           A: Syn  DWARF 1: 3.8.3
```

The AT_subscr_data attribute in a TAG_array_type debugging information entry shall contain a data item specifying a dimension for each dimension of the array as specified in the source code.

DEBUG: ARRAY_DIMENSION_WRONG A: Sem DWARF 1: 3.8.3
The upper bound part of the data item in an AT_subcr_data attribute describing a dimension of an array shall match the dimension given in the source code.

DEBUG: ARRAY_ELEMENT_FORMAT_CODE_BAD A: Syn DWARF 1: 3.8.3
The format specifier for the element type of a TAG_array_type debugging information entry shall be FMT_ET (0x8).

DEBUG: ARRAY_ELEMENT_TYPE_WRONG A: Sem DWARF 1: 3.8.3
The type attribute for the element type of a TAG_array_type debugging information entry shall specify a type matching the element type specified in the source code.

DEBUG: ARRAY_FORMAT_SPECIFIER_BAD_CODE A: Syn DWARF 1: 3.8.8
The format specifier contained in the data element of an AT_subscr_data attribute describing an array dimension shall be one of the following:

Name	Value
FMT_FT_C_C	0x0
FMT_FT_C_X	0x1
FMT_FT_X_C	0x2
FMT_FT_X_X	0x3
FMT_UT_C_C	0x4
FMT_UT_C_X	0x5
FMT_UT_X_C	0x6
FMT_UT_X_X	0x7
FMT_ET	0x8

DEBUG: ARRAY_LOWER_BOUND_NON_ZERO C: Syn DWARF 1: 3.8.3
An AT_subscr_data attribute specifying a lower bound for an array shall have the value zero.

DEBUG: ARRAY_STRIDE_SIZE_MISSING A: Sem DWARF 1: 3.8.3
If the amount of storage allocated to hold each element of an array is different from the amount of storage that is normally allocated to hold an object of the indicated element type, then the array type entry shall have an AT_stride_size attribute to provide the size in bits of each element of the array.

DEBUG: ARRAY_STRIDE_SIZE_WRONG A: Sem DWARF 1: 3.8.3
If the amount of storage allocated to hold each element of an array is different from the amount of storage that is normally allocated to hold an object of the indicated element type, then the array type entry shall have an AT_stride_size attribute with a constant value equal to the size in bits of each element of the array.

```
DEBUG:   ARRAY_SUBSCRIPTS_MISSING          A: Syn  DWARF 1: 3.8.3
   Every TAG_array_type debugging information entry shall have an
   AT_subscr_data attribute to specify the array subscripts and element data
   type.

DEBUG:   ARRAY_TYPE_MISSING                A: Syn  DWARF 1: 3.8.1
   A TAG_array_type debugging information entry shall have a AT_subscr_data
   attribute to specify its dimensions and the type of its elements.

DEBUG:   AT_BIT_OFFSET_FORM_BAD            A: Syn  DWARF 1: 4.4
   Attribute AT_bit_offset shall have format encoding FORM_DATA2 (0x5).

DEBUG:   AT_BIT_SIZE_FORM_BAD              A: Syn  DWARF 1: 4.4
   Attribute AT_bit_size shall have format encoding FORM_DATA4 (0x6).

DEBUG:   AT_BYTE_SIZE_FORM_BAD             A: Syn  DWARF 1: 4.4
   Attribute AT_byte_size shall have format encoding FORM_DATA4 (0x6).

DEBUG:   AT_COMMON_REFERENCE_FORM_BAD      A: Syn  DWARF 1: 4.4
   Attribute AT_common_reference shall be have form encoding FORM_REF (0x2).

DEBUG:   AT_COMP_DIR_FORM_BAD              A: Syn  DWARF 1: 4.4
   Attribute AT_comp_dir shall have format encoding FORM_STRING (0x8).

DEBUG:   AT_CONST_VALUE_FORM_BAD           A: Syn  DWARF 1: 4.4
   Attribute AT_const_value shall have format encoding FORM_STRING (0x8),
   BLOCK2 (0x3), FORM_BLOCK4 (0x4), FORM_DATA2 (0x5), FORM_DATA4 (0x6), or
   FORM_DATA8 (0x7).

DEBUG:   AT_CONTAINING_TYPE_FORM_BAD       A: Syn  DWARF 1: 4.4
   Attribute AT_containing_type shall have format encoding FORM_REF (0x2).

DEBUG:   AT_DEFAULT_VALUE_FORM_BAD         A: Syn  DWARF 1: 4.4
   Attribute AT_default_value shall have format encoding FORM_ADDR (0x10),
   FORM_DATA2 (0x4), FORM_DATA8 (0x7), or FORM_STRING (0x8).

DEBUG:   AT_DISCR_FORM_BAD                 A: Syn  DWARF 1: 4.4
   Attribute AT_discr shall have format encoding FORM_REF (0x2).

DEBUG:   AT_DISCR_VALUE_FORM_BAD           A: Syn  DWARF 1: 4.4
   Attribute AT_discr_value shall have input FORM_BLOCK2 (0x3).

DEBUG:   AT_ELEMENT_LIST_TYPE_FORM_BAD     A: Syn  DWARF 1: 4.4
   Attribute AT_element_list_type shall have format encoding FORM_BLOCK4 (0x4)

DEBUG:   AT_FRIENDS_FORM_BAD               A: Syn  DWARF 1: 4.4
   Attribute AT_friends shall have format encoding FORM_BLOCK2 (0x3).

DEBUG:   AT_FUND_TYPE_FORM_BAD             A: Syn  DWARF 1: 4.4
   Attribute AT_fund_type shall have format encoding FORM_DATA2 (0x5)

DEBUG:   AT_HIGH_PC_FORM_BAD               A: Syn  DWARF 1: 4.4
```

```
   Attribute AT_high_pc shall have format encoding FORM_ADDR (0x1)

DEBUG:   AT_INLINE_FORM_BAD               A: Syn  DWARF 1: 4.4
   Attribute AT_inline shall have format encoding FORM_STRING (0x8).

DEBUG:   AT_IS_OPTIONAL_FORM_BAD          A: Syn  DWARF 1: 4.4
   Attribute AT_is_optional shall have format encoding FORM_STRING (0x8).

DEBUG:   AT_LANGUAGE_BAD                  C: Syn  DWARF 1: 3.1: 5.
   If a TAG_compile_unit entry has an AT_language attribute then that
   attribute shall have the value LANG_C or LANG_C89.

DEBUG:   AT_LANGUAGE_FORM_BAD             A: Syn  DWARF 1: 4.4
   Attribute AT_language shall have format encoding FORM_DATA4 (0x6).

DEBUG:   AT_LOCATION_FORM_BAD             A: Syn  DWARF 1: 4.4
   Attribute AT_location shall have format encoding FORM_BLOCK2 (0x3).

DEBUG:   AT_LOWER_BOUND_FORM_BAD          A: Syn  DWARF 1: 4.4
   Attribute AT_lower_bound shall have format encoding FORM_DATA2 (0x5),
   FORM_DATA4 (0x6), FORM_DATA8 (0x7), or FORM_REF (0x2).

DEBUG:   AT_LOW_PC_FORM_BAD               A: Syn  DWARF 1: 4.4
   Attribute AT_low_pc shall have format encoding FORM_ADDR (0x1)

DEBUG:   AT_MEMBER_FORM_BAD               A: Syn  DWARF 1: 4.4
   Attribute AT_member shall have format encoding FORM_REF (0x2)

DEBUG:   AT_MOD_FUND_TYPE_FORM_BAD        A: Syn  DWARF 1: 4.4
   Attribute AT_mod_fund_type shall have format encoding FORM_BLOCK2 (0x3)

DEBUG:   AT_MOD_U_D_TYPE_FORM_BAD         A: Syn  DWARF 1: 4.4
   Attribute AT_mod_u_d_type shall have format encoding FORM_BLOCK2 (0x3)

DEBUG:   AT_NAME_FORM_BAD                 A: Syn  DWARF 1: 4.4
   Attribute AT_name shall have format encoding FORM_STRING (0x8)

DEBUG:   AT_ORDERING_FORM_BAD             A: Syn  DWARF 1: 4.4
   Attribute AT_ordering shall have format encoding FORM_DATA2 (0x5).

DEBUG:   AT_PRIVATE_FORM_BAD              A: Syn  DWARF 1: 4.4
   Attribute AT_private shall have format encoding FORM_STRING (0x8).

DEBUG:   AT_PRODUCER_FORM_BAD             A: Syn  DWARF 1: 4.4
   Attribute AT_producer shall have format encoding FORM_STRING (0x8).

DEBUG:   AT_PROGRAM_FORM_BAD              A: Syn  DWARF 1: 4.4
   Attribute AT_program shall have format encoding FORM_STRING (0x8).

DEBUG:   AT_PROTECTED_FORM_BAD            A: Syn  DWARF 1: 4.4
   Attribute AT_protected shall have format encoding FORM_STRING (0x8).
```

```
DEBUG:   AT_PROTOTYPED_FORM_BAD            A: Syn  DWARF 1: 4.4
   Attribute AT_prototyped shall have format encoding FORM_STRING (0x8).

DEBUG:   AT_PUBLIC_FORM_BAD                A: Syn  DWARF 1: 4.4
   Attribute AT_public shall have format encoding FORM_STRING (0x8).

DEBUG:   AT_PURE_VIRTUAL_FORM_BAD          A: Syn  DWARF 1: 4.4
   Attribute AT_pure_virtual shall have format encoding FORM_STRING (0x8).

DEBUG:   AT_RETURN_ADDR_FORM_BAD           A: Syn  DWARF 1: 4.4
   Attribute AT_return_addr shall have format encoding FORM_BLOCK2 (0x3).

DEBUG:   AT_SIBLING_FORM_BAD               A: Syn  DWARF 1: 4.4
   Attribute AT_sibling shall have format encoding FORM_REF (0x2).

DEBUG:   AT_SPECIFICATION_ADDR_FORM_BAD    A: Syn  DWARF 1: 4.4
   Attribute AT_specification_addr shall have format encoding FORM_REF (0x2).

DEBUG:   AT_START_SCOPE_FORM_BAD           A: Syn  DWARF 1: 4.4
   Attribute AT_start_scope shall have format encoding FORM_DATA4 (0x6).

DEBUG:   AT_STMT_LIST_FORM_BAD             A: Syn  DWARF 1: 4.4
   Attribute AT_stmt_list shall have format encoding FORM_DATA4 (0x6).

DEBUG:   AT_STRIDE_SIZE_FORM_BAD           A: Syn  DWARF 1: 4.4
   Attribute AT_stride_size shall have format encoding FORM_DATA4 (0x6).

DEBUG:   AT_STRING_LENGTH_FORM_BAD         A: Syn  DWARF 1: 4.4
   Attribute AT_string_length shall have input FORM_BLOCK2 (0x3).

DEBUG:   AT_SUBSCR_DATA_TYPE_FORM_BAD      A: Syn  DWARF 1: 4.4
   Attribute AT_subscr_data_type shall have format encoding FORM_BLOCK2 (0x3)

DEBUG:   AT_UPPER_BOUND_FORM_BAD           A: Syn  DWARF 1: 4.4
   Attribute AT_upper_bound shall have format encoding FORM_DATA2 (0x5),
   FORM_DATA4 (0x6), FORM_DATA8 (0x7), or FORM_REF (0x2).

DEBUG:   AT_USER_DEF_TYPE_FORM_BAD         A: Syn  DWARF 1: 4.4
   Attribute AT_user_def_type shall have format encoding FORM_REF (0x2)

DEBUG:   AT_VIRTUAL_FORM_BAD               A: Syn  DWARF 1: 4.4
   Attribute AT_virtual shall have format encoding FORM_STRING (0x8).

DEBUG:   BIT_FIELD_BIT_OFFSET_WRONG        A: Sem  DWARF 1: 3.8.4.2
   The AT_bit_offset attribute of a TAG_member entry representing a bit field
   of a structure or union shall have a constant value equal to the number of
   bits to the left of the leftmost (most significant) bit of that bit field.

DEBUG:   BIT_FIELD_BIT_SIZE_WRONG          A: Sem  DWARF 1: 3.8.4.2
   The AT_bit_size attribute of a TAG_member debugging information entry
   representing a bit field of a structure or union shall have a constant
   value equal to the number of bits occupied by the bit field.
```

```
DEBUG:   BIT_FIELD_BYTE_SIZE_WRONG         A: Sem  DWARF 1: 3.8.4.2
   If a TAG_member debugging information entry for a bit field has an
   AT_byte_size attribute, then the value of that attribute shall be a
   constant equal to the number of bytes needed to contain the bit field
   (including any necessary padding bits).

DEBUG:   BIT_FIELD_REQUIRED_ATTRS          A: Syn  DWARF 1: 3.8.4.2
   If a TAG_member entry for a structure or union member contains either an
   AT_bit_offset attribute or an AT_bit_size attribute then the member
   represented is a bit field and both attributes shall be present.

DEBUG:   CU_MISSING                        A: Syn  DWARF 1: 3.1
   At the end of all debugging information entries owned by a compilation
   unit shall come either the end of the ".debug" section or another
   TAG_compile_unit debugging information entry.

DEBUG:   DEBUG_INFO_NOT_TERMINATED         A: Syn  DWARF 1: 2.3
   The ".debug" section shall terminate each sibling chain with a null entry,
   thereby completing the tree represented by the debugging information
   entries.

DEBUG:   DIE_TAG_MISSING                   A: Syn  DWARF 1: 3.1
   A debugging information entry shall begin with a tag code that ends before
   the end of the section.

DEBUG:   ENUMERATION_NAME_WRONG            A: Sem  DWARF 1: 3.8.5
   The TAG_enumeration_type debugging information entry for an enumeration
   type having a tag name shall have an AT_name attribute containing the
   enumeration tag name as it appears in the source program.

DEBUG:   ENUMERATION_SIZE_MISSING          A: Syn  DWARF 1: 3.8.5
   A TAG_enumeration_type debugging information entry shall have an
   AT_byte_size attribute.

DEBUG:   ENUMERATOR_NAME_MISMATCH          A: Sem  DWARF 1: 3.8.5
   The TAG_enumerator debugging information entries representing the members
   of an enumeration shall appear in the same order with the same names (in
   the AT_name attribute) as the declarations of the enumeration members in
   the source program.

DEBUG:   ENUMERATOR_NAME_MISSING           A: Syn  DWARF 1: 3.8.5
   A DW_TAG_enumerator entry representing an enumeration member shall have a
   DW_AT_name attribute.

DEBUG:   ENUMERATOR_PARENT_WRONG           A: Syn  DWARF 1: 3.8.5
   The TAG_enumerator debugging information entry representing an enumeration
   member shall be a child of a TAG_enumeration_type entry representing the
   enumeration to which the member belongs.

DEBUG:   ENUMERATOR_VALUE_MISSING          A: Syn  DWARF 1: 3.8.5
```

A TAG_enumerator entry representing an enumeration member shall have an AT_const_value attribute.

DEBUG: ENUMERATOR_VALUE_WRONG A: Sem DWARF 1: 3.8.5
The TAG_enumerator entry representing an enumeration member shall have an AT_const_value attribute whose value is the actual numeric value of the enumerator as represented on the target system.

DEBUG: EXTERNAL_PUBNAME_MISSING A: Syn DWARF 1: 3.6, 3.10.1
A TAG_global_variable debugging information entry shall have an entry in the ".debug_pubnames" section.

DEBUG: FORM2_MISSING A: Syn DWARF 1: 4.4
If an attribute in a debugging information entry has a form of FORM_DATA2, the corresponding value shall be two bytes ending before the end of the compilation unit’s information.

DEBUG: FORM4_MISSING A: Syn DWARF 1: 4.4
If an attribute in a debugging information entry has a form of FORM_ADDR, FORM_DATA4,FORM_REF4 the corresponding value shall be four bytes ending before the end of the compilation unit’s information.

DEBUG: FORM8_MISSING A: Syn DWARF 1: 4.4
If an attribute in a debugging information entry has a form of FORM_DATA8, value shall be two bytes ending before the end of the compilation unit’s information.

DEBUG: FORMAL_PARAMETER_MISSING A: Sem DWARF 1: 3.8.6
A parameter to a function that is a single argument of a specified type shall be represented by a TAG_formal_parameter debugging information entry.

DEBUG: FORMAL_PARAMETER_TYPE_MISSING A: Syn DWARF 1: 3.8.6
A TAG_formal_parameter debugging information entry owned by a TAG_subroutine_type debugging information entry shall have one of the four type attributes (fundamental type, modified fundamental type, user-defined type, or modified user-defined type).

DEBUG: FORMAL_PARAMETER_TYPE_WRONG A: Sem DWARF 1: 3.8.6
A TAG_formal_parameter debugging information entry shall have an AT_type attribute matching the type specified for the corresponding parameter in the source code.

DEBUG: FORM_BLOCK_2_LEN_MISSING A: Syn DWARF 1: 4.4
If an attribute in a debugging information entry has a form of DW_FORM_BLOCK2, the corresponding value shall begin with a block length that is itself contained in two bytes appearing before the end of the compilation unit’s information.

DEBUG: FORM_BLOCK_2_TOO_BIG A: Syn DWARF 1: 4.4
If an attribute in a debugging information entry has a form of DW_FORM_BLOCK2, the corresponding value shall begin with a block length

such that the block is completely contained within the compilation unit's information.

DEBUG: FORM_BLOCK_4_LEN_MISSING A: Syn DWARF 1: 4.4
If an attribute in a debugging information entry has a form of DW_FORM_BLOCK4, the corresponding value shall begin with a block length that is itself contained in four bytes appearing before the end of the compilation unit's information.

DEBUG: FORM_BLOCK_4_TOO_BIG A: Syn DWARF 1: 4.4
If an attribute in a debugging information entry has a form of DW_FORM_BLOCK4, the corresponding value shall begin with a block length such that the block is completely contained within the compilation unit's information.

DEBUG: FORM_STRING_NULL_MISSING A: Syn DWARF 1: 4.4
If an attribute in a debugging information entry has a form of FORM_STRING, the corresponding value shall be a null-terminated string ending before the end of the compilation unit's information.

DEBUG: FUNDAMENTAL_TYPE_BAD_ENCODING A: Syn DWARF 1: 4.8
A fundamental type attribute shall have a value which is one of the following:

Name	Value
FT_char	0x1
FT_signed_char	0x2
FT_unsigned_char	0x3
FT_short	0x4
FT_signed_short	0x5
FT_unsigned_short	0x6
FT_integer	0x7
FT_signed_integer	0x8
FT_unsigned_integer	0x9
FT_long	0xa
FT_signed_long	0xb
FT_unsigned_long	0xc
FT_pointer	0xd
FT_float	0xe
FT_void	0x14
FT_lo_user	0x8000
...	
FT_hi_user	0xffff

DEBUG: INLINED_SPECIFICATION_MISSING A: Syn DWARF 1: 3.3.6
A TAG_inlined_subroutine debugging information entry shall have an AT_specification attribute whose value is a reference to the debugging information entry representing the declaration or out of line instance of the subroutine.

DEBUG: LABEL_LOW_PC_MISSING A: Syn DWARF 1: 3.5

```
   A TAG_label entry shall have a AT_low_pc attribute.

DEBUG:   LABEL_MISSING                    A: Sem  DWARF 1: 3.5
   Every label in a C source program shall have a corresponding TAG_label
   debugging information entry at the appropriate scope level in the object
   file, with an AT_name attribute matching the label name in the source file.

DEBUG:   LABEL_NAME_MISSING               A: Syn  DWARF 1: 3.5
   A TAG_label entry shall have a AT_name attribute.

DEBUG:   LABEL_UNEXPECTED                 A: Sem  DWARF 1: 3.5
   Every TAG_label debugging information entry shall match in name and scope
   a label defined in the corresponding C source code.

DEBUG:   LEXICAL_BLOCK_HIGH_PC_MISSING    A: Syn  DWARF 1: 3.4
   A TAG_lexical_block entry shall have a AT_high_pc attribute.

DEBUG:   LEXICAL_BLOCK_LOW_PC_MISSING     A: Syn  DWARF 1: 3.4
   A TAG_lexical_block entry shall have a AT_low_pc attribute.

DEBUG:   MEMBER_NAME_WRONG                A: Sem  DWARF 1: 3.8.4.2
   The TAG_member debugging information entry for a member of a structure or
   union having a name in the source program shall have an AT_name attribute
   matching the member name as it appears in the source program.

DEBUG:   MEMBER_PARENT_WRONG              A: Syn  DWARF 1: 3.8.4.1
   A TAG_member entry shall be owned by a TAG_structure_type or
   TAG_union_type entry.

DEBUG:   MEMBER_TYPE_MISSING              A: Syn  DWARF 1: 3.8.4.2
   A TAG_member entry for a structure or union data member entry shall have
   an AT_type attribute.

DEBUG:   MEMBER_TYPE_WRONG                A: Sem  DWARF 1: 3.8.4.2
   The AT_type attribute of a TAG_member debugging information entry shall
   describe the type of the member.

DEBUG:   SIBLING_MISSING                  A: Syn  DWARF 1: 2.3
   Every debugging information entry, except the special entry whose tag is
   TAG_padding, shall have a sibling attribute.

DEBUG:   STMT_LIST_BAD                    C: Syn  DWARF 1: 3.1: 6.
   If a TAG_compile_unit debugging information entry has an AT_stmt_list
   attribute, the attribute shall have a value that is an offset into the
   ".debug_line" section.

DEBUG:   STRUCT_BYTE_SIZE_WRONG           A: Sem  DWARF 1: 3.8.4.1
   If the size of a structure can be determined at compile time, then the
   TAG_structure_type debugging information entry for that structure shall
   include an AT_byte_size attribute containing a constant value equal to the
   length of the structure (including any necessary padding bytes).
```

```
DEBUG:   STRUCT_INCOMPLETE                A: Sem  DWARF 1: 3.8.4.1
   The TAG_structure type debugging information entry for an incomplete
   struct type shall not have an AT_byte_size attribute.

DEBUG:   STRUCT_MEMBERS                   A: Sem  DWARF 1: 3.8.4.1
   The members of a struct shall be represented by debugging information
   entries that are owned by the corresponding TAG_structure_type entry;
   those struct members shall be represented in the same order as the
   corresponding list of members in the source program.

DEBUG:   STRUCT_MEMBER_LOCATION_MISSING   A: Syn  DWARF 1: 3.8.4.2
   The TAG_member entry for a member of a structure shall have an
   AT_data_member_location attribute

DEBUG:   STRUCT_MEMBER_OFFSET_WRONG       A: Sem  DWARF 1: 3.8.4.2
   A TAG_member debugging information entry for a structure member shall have
   a AT_location containing the relative offset of that member from the base
   of the innermost structure containing the member.

DEBUG:   STRUCT_NAME_MISSING              A: Sem  DWARF 1: 3.8.4.1
   The TAG_structure_type debugging information entry for a struct having a
   tag name shall contain an AT_name attribute matching the tag name in the
   source code.

DEBUG:   SUBPROGRAM_GLOBAL_MISSING        A: Sem  DWARF 1: 3.3
   A function in a C source program that is not declared "static" shall be
   described by a TAG_global_subroutine debugging information entry.

DEBUG:   SUBPROGRAM_HIGH_PC_MISSING       A: Syn  DWARF 1: 3.3.3
   The TAG_global_subroutine or TAG_subroutine debugging information entry
   for the definition of a function shall have an AT_high_pc attribute
   (except for the entry describing an in-lined function for which no
   out-of-line instance was generated).

DEBUG:   SUBPROGRAM_LOW_PC_MISSING        A: Syn  DWARF 1: 3.3.3
   The TAG_global_subroutine or TAG_subroutine debugging information entry
   for the definition of a function shall have an AT_low_pc attribute (except
   for the entry describing an in-lined function for which no out-of-line
   instance was generated).

DEBUG:   SUBPROGRAM_MISSING               A: Sem  DWARF 1: 3.3.1
   Every function in a C source file shall be represented by a debugging
   information entry with a tag of TAG_global_subroutine, TAG_subroutine, or
   TAG_inlined_subroutine which has a AT_name  attribute matching the
   function name used in the source code.

DEBUG:   SUBPROGRAM_NAME_MISSING          A: Syn  DWARF 1: 3.3.1
   A TAG_global_subroutine, TAG_subroutine, or TAG_inlined_subroutine
   debugging information entry shall have an AT_name attribute

DEBUG:   SUBPROGRAM_PARAMETERS_ORDER      A: Sem  DWARF 1: 3.3.4, 3.6: 1, 3
```

Every formal parameter in a C source function definition or declaration shall be represented by a TAG_formal_parameter entry which (1) is a child of the TAG_global_subroutine, TAG_subroutine or TAG_inlined_subroutine representing the function, (2) has a type attribute which matches the type of the formal parameter, and (3) if the parameter is named, has a AT_name attribute matching the given name.

DEBUG: SUBPROGRAM_PARAMETERS_OWNER A: Syn DWARF 1: 3.3.4, 3.6
TAG_formal_parameter and/or DW_TAG_unspecified_parameters entries shall be owned by a TAG_global_subroutine, TAG_subroutine, TAG_inlined_subroutine, or TAG_subroutine_type debugging information entry.

DEBUG: SUBPROGRAM_PROTOTYPE_NO A: Sem DWARF 1: 3.3.2
A TAG_subroutine or TAG_global_subroutine debugging information entry representing a function that was not declared with a function prototype style shall not have an AT_prototype attribute.

DEBUG: SUBPROGRAM_TYPE_MISSING A: Sem DWARF 1: 3.3.2
A C function which is not "void" shall be represented by a TAG_global_subroutine or TAG_subroutine debugging information entry having one of the four type attributes (fundamental type, modified fundamental type, user-defined type, or modified user-defined type).

DEBUG: SUBPROGRAM_TYPE_WRONG A: Sem DWARF 1: 3.3.2
The type attribute in a TAG_global_subroutine or TAG_subroutine debugging information entry shall match the type returned by the function as declared in the C source code.

DEBUG: SUBPROGRAM_UNEXPECTED A: Sem DWARF 1: 3.3.1
The AT_name attribute of every TAG_global_subroutine, TAG_subroutine, and TAG_inlined_subroutine debugging information entry shall match in name a function defined in the corresponding C source code.

DEBUG: SUBPROGRAM_UNEXPECTED_EXTERNAL A: Sem DWARF 1: 3.3.1
A "static" function in a C source program shall be represented by a TAG_subroutine debugging information entry.

DEBUG: SUBPROGRAM_UNEXPECTED_PC A: Sem DWARF 1: 3.3.3
A TAG_global_subroutine or TAG_subroutine entry for which no out-of-line instance has been generated shall not have an AT_low_pc or AT_high_pc attribute.

DEBUG: SUBPROGRAM_UNEXPECTED_TYPE A: Sem DWARF 1: 3.3.2
A "void" C function is be represented by a TAG_global_subroutine or TAG_subprogram debugging information entry which shall not have an attribute for the return type.

DEBUG: SUBROUTINE_PROTOTYPE_NO A: Sem DWARF 1: 3.8.6
A TAG_subroutine_type debugging information entry representing a function type that was not declared with a function prototype style, shall not contain an AT_prototyped attribute.

```
DEBUG:   SUBROUTINE_TYPE_MISNAMED          A: Sem  DWARF 1: 3.8.6
   The TAG_subroutine_type debugging information entry for a function type
   having a name shall have an AT_name attribute matching the function type
   name as it appears in the source program.

DEBUG:   SUBROUTINE_TYPE_WRONG             A: Sem  DWARF 1: 3.8.6
   The TAG_subroutine_type debugging information entry for a function type
   returning a value (not "void") shall have a type (fundamental, modified
   fundamental, user defined, or modified user defined) attribute that
   correctly denotes the type returned by the function as indicated in the
   source code.

DEBUG:   SUBROUTINE_UNEXPECTED             A: Sem  DWARF 1: 3.8.6
   Every TAG_subroutine_type entry shall match in name a function type
   defined in the corresponding C source code.

DEBUG:   SUBR_TYPE_CHILD_BAD_TAG           A: Syn  DWARF 1: 3.8.6
   Any child entries of a TAG_subroutine_type shall describe the type(s) of
   the argument(s) to the function and shall be of type TAG_formal_parameter,
   TAG_unspecified_parameters, [[or TAG_enumeration (for a parameter type
   specified by an enumeration definition)]]. (Note: allowing enumerations is
   an extension to the DWARF specification.)

DEBUG:   TYPEDEF_MISSING                   A: Sem  DWARF 1: 3.8.1
   Every named typedef in a source file shall be represented by a TAG_typedef
   debugging information entry with the same name in the same (corresponding)
   scope.

DEBUG:   TYPEDEF_NAME_MISSING              A: Syn  DWARF 1: 3.8.1
   A TAG_typedef entry shall have a AT_name attribute.

DEBUG:   TYPEDEF_TYPE_MISSING              A: Syn  DWARF 1: 3.8.1
   A TAG_typedef debugging information entry shall contain one of the four
   type attributes (fundamental type, modified fundamental type, user-defined
   type, or modified user-defined type).

DEBUG:   TYPEDEF_UNEXPECTED                A: Sem  DWARF 1: 3.8.1
   Every TAG_typedef debugging information entry with an AT_name attribute
   shall match in name and scope a typedef in the corresponding C source code.

DEBUG:   TYPE_MODIFIER_CONST_MISSING       A: Sem  DWARF 1: 2.5.3
   A "const" type in a source program shall be represented by a type
   containing the MOD_const modifier.

DEBUG:   TYPE_MODIFIER_POINTER_MISSING     A: Sem  DWARF 1: 2.5.3
   A pointer type in a source program shall be represented by a type
   containing the MOD_pointer_to modifier.

DEBUG:   TYPE_MODIFIER_VOLATILE_MISSING    A: Sem  DWARF 1: 2.5.3
   A "volatile" type in a source program shall be represented by a type
   containing the MOD_volatile modifier.
```

DEBUG: UNION_BYTE_SIZE_WRONG A: Sem DWARF 1: 3.8.4.1
If the size of a union can be determined at compile time, then the TAG_union_type entry for that union shall include an AT_byte_size attribute containing a constant value equal to the length of the union (including any necessary padding bytes).

DEBUG: UNION_INCOMPLETE A: Sem DWARF 1: 3.8.4.1
The TAG_union type debugging information entry for an instance of an incomplete union shall not have an AT_byte_size attribute.

DEBUG: UNION_MEMBERS_MISSING A: Sem DWARF 1: 3.8.4.1
The members of a union shall be represented by debugging information entries that are owned by the corresponding TAG_union; those union members shall be represented in the same order as the corresponding list of members in the source program.

DEBUG: UNION_MEMBER_OFFSET_WRONG A: Sem DWARF 1: 3.8.4.2
The debugging information entry for a union member shall have an AT_location attribute containing the relative offset of that member from the base of the innermost union containing the member.

DEBUG: UNION_NAME_MISSING A: Sem DWARF 1: 3.8.4.1
The TAG_union_type debugging information entry for a union having a tag name shall have an AT_name attribute matching the tag name in the source code.

DEBUG: VAR_EXTERNAL_MISSING A: Sem DWARF 1: 4.1:2
A file-level variable in a C source program that is not declared "static" shall be described by a TAG_global_variable debugging information entry.

DEBUG: VAR_GLOBAL_MISSING A: Sem DWARF 1: 4.1
Every global variable in a source file shall be represented by a TAG_global_variable debugging information entry at the appropriate scope level in the object file, with a AT_name attribute matching the name used in the source file.

DEBUG: VAR_GLOBAL_UNEXPECTED A: Sem DWARF 1: 3.6
Every TAG_global_variable debugging information entry shall match in name and scope a variable defined in the corresponding C source code.

DEBUG: VAR_LOCAL_MISSING A: Sem DWARF 1: 4.1
Every local variable (including module-static variables) in a source file shall be represented by a TAG_variable debugging information entry at the appropriate scope level in the object file, with a AT_name attribute matching the name used in the source file.

DEBUG: VAR_LOCAL_UNEXPECTED A: Sem DWARF 1: 3.6
Every TAG_variable debugging information entry shall match in name and scope a variable defined in the corresponding C source code.

DEBUG: VAR_TYPE_MISSING A: Syn DWARF 1: 3.6: 3

A TAG_global_variable or TAG_local_variable debugging information entry shall have one of the four type attributes (fundamental type, modified fundamental type, user-defined type, or modified user-defined type).

DEBUG: VAR_TYPE_WRONG A: Sem DWARF 1: 3.6: 3
A TAG_global_variable or TAG_local_variable debugging information entry shall have type attribute that matches the type of the corresponding variable in the C source code.

DEBUG: VAR_UNEXPECTED_EXTERNAL A: Sem DWARF 1: 3.6
A variable in a C source program that is declared "static" or has local scope shall be described by a TAG_variable debugging information entry.

".debug_aranges" Section

DBGARAN: DEBUG_INFO_OFFSET_BAD A: Syn DWARF 1: 3.10.2
In a ".debug_aranges" set header the third field shall be the offset into the ".debug_info" section of the compilation unit entry referenced by the set.

DBGARAN: DEBUG_INFO_OFFSET_TOO_BIG A: Syn DWARF 1: 4.15
In a ".debug_aranges" set header the third field shall be an offset into the ".debug_info" section and shall therefore have a value that is less than the size of the ".debug_info" section.

DBGARAN: SET_LENGTH_MISSING A: Syn DWARF 1: 4.15
At the end of a set of address ranges shall come either the end of the ".debug_aranges" section, or the four byte length field of a new set header.

DBGARAN: SET_LENGTH_TOO_BIG A: Syn DWARF 1: 4.15
The first header field of a set of address ranges shall give the length, counting itself, of the contribution to the ".debug_aranges" section for a compilation unit such that that contribution is completely contained within the section.

DBGARAN: SET_LENGTH_TOO_SMALL A: Syn DWARF 1: 4.15
In a ".debug_aranges" set header the first field shall give the length of the set of entries for a compilation unit and shall be a value of sufficient size to hold at least the set header.

DBGARAN: UNTERMINATED_SECTION A: Syn DWARF 1: 4.15
In the ".debug_aranges" section, the final address range descriptor in a set of entries for a compilation unit shall have a 0 address and a 0 length, and this terminating entry shall coincide with the end of the section as determined by its length.

DBGARAN: VERSION_BAD A: Syn DWARF 1: 4.15
In a ".debug_aranges" set header the second field shall be the version number with a value of 2.

DWARF 1 Location Descriptions

```
DBGLOC:  BASEREG_MISSING                    A: Syn  DWARF 1: 2.4.1, 4.7
   The operation OP_BASEREG shall have a single operand value of size FT_long
   ending before the end of the location description.

DBGLOC:  CONST_MISSING                      A: Syn  DWARF 1: 2.4.1, 4.7
   The operation OP_CONST shall have a single operand value of size FT_long
   ending before the end of the location description.

DBGLOC:  OPCODE_INVALID                     A: Syn  DWARF 1: 2.4
   A location description consists of zero or more location operations, each
   beginning with one of the following valid operation codes:

            Operation     Code   Type of operand
            ----------    ----   ---------------
            OP_REG        0x01   number
            OP_BASEREG    0x02   number
            OP_ADDR       0x03   address
            OP_CONST      0x04   number
            OP_DEREF2     0x05
            OP_DEREF      0x06
            OP_ADD        0x07
            OP_lo_user    0xe0
            ...
            OP_hi_user    0xff

DBGLOC:  REG_MISSING                        A: Syn  DWARF 1: 2.4.1, 4.7
   The operation OP_REG shall have a single operand value of size FT_long
   ending before the end of the location description.

DBGLOC:  TARGET_ADDRESS_MISSING             A: Syn  DWARF 1: 2.4.1, 4.7
   The operation OP_ADDR shall have a single target_address operand value
   ending before the end of the location description
```

".debug_pubnames" Section

```
DBGPUBN: DEBUG_INFO_LENGTH_BAD              A: Syn  DWARF 1: 3.10.1
   In a ".debug_pubnames" set header, the third field shall be the length in
   bytes of the contents of the ".debug" section generated to represent the
   compilation unit corresponding to that set of "pubnames", and shall
   therefore match the value implied by the first field of the header for
   that compilation unit.

DBGPUBN: DEBUG_INFO_OFFSET_BAD              A: Syn  DWARF 1: 3.10.1
   In a ".debug_pubnames" set header the third field shall be the offset into
   the ".ebug" section of the compilation unit entry referenced by the set.

DBGPUBN: DEBUG_INFO_OFFSET_TOO_BIG          A: Syn  DWARF 1: 4.14
```

In a ".debug_pubnames" set header the third field shall be an offset into the ".ebug" section and shall therefore have a value that is less than the size of the ".ebug" section.

DBGPUBN: NAME_BAD A: Syn DWARF 1: 3.10.1
The name in an offset name pair in the ".debug_pubnames" section shall match the name given by the AT_name attribute of the corresponding debugging information entry in the ".debug" section.

DBGPUBN: NAME_TOO_LONG A: Syn DWARF 1: 4.14
The second part of each offset/name pair in the ".debug_pubnames" section shall be a null-terminated string entirely contained within the set.

DBGPUBN: OFFSET_BAD A: Syn DWARF 1: 3.10.1
Each offset/name pair in the ".debug_pubnames" section shall begin with a 4-byte offset (relative to the third field of the set's header) which references a debugging information entry in the ".debug" section for the object represented by the pair.

DBGPUBN: OFFSET_MISSING A: Syn DWARF 1: 4.14
Each offset/name pair in the ".debug_pubnames" section shall begin with a 4-byte offset entirely contained within the set.

DBGPUBN: OFFSET_TOO_BIG A: Syn DWARF 1: 4.14
Each offset/name pair in the ".debug_pubnames" section shall begin with a 4-byte offset to a debugging information entry in the ".ebug" section for the object represented by the pair, and shall therefore have a value less then the length of the debugging information for the compilation unit to which that debugging information entry belongs.

DBGPUBN: PUBNAME_MISSING A: Syn DWARF 1: 3.10.1
A global object shall be represented by an offset/name pair in the ".debug_pubnames" section.

DBGPUBN: SET_LENGTH_MISSING A: Syn DWARF 1: 4.14
In a ".debug_pubnames" section, at the end of a set of entries shall come either the end of the section, or the 4-byte length field of a new set header.

DBGPUBN: SET_LENGTH_TOO_BIG A: Syn DWARF 1: 4.14
In a ".debug_pubnames" set header the first field shall give the length, not counting itself, of the contribution to the ".debug_pubnames" section for a compilation unit such that that contribution is completely contained within the section.

DBGPUBN: SET_LENGTH_TOO_SMALL A: Syn DWARF 1: 4.14
In a ".debug_pubnames" set header the first field shall give the length of the set of entries for a compilation unit and shall be a value of sufficient size to hold at least the set header.

DBGPUBN: UNTERMINATED_SECTION A: Syn DWARF 1: 4.14

In the ".debug_pubnames" section, the final offset/name pair in a set of entries shall be followed by a final 4-byte offset only containing the value 0, and this terminating entry shall appear at the position implied by the length of the set of entries as given by the first field of the header for the set.

DBGPUBN: VERSION_BAD A: Syn DWARF 1: 4.14
In a ".debug_pubnames" set header the second field shall be the version number with a value of 2.

DBGPUBN: WRONG_TYPE_OF_DIE A: Syn DWARF 1: 3.3, 3.6, 3.10.1
An object In the ".debug_pubnames" section shall have a corresponding debugging information entry with the same name of type TAG_global_subroutine or TAG_global_variable.

".line" Section

DBGLINE: LAST_LINE_NUMBER_NOT_ZERO A: Syn DWARF 1: 3.11
The final entry of a source statement table shall have a line number equal to zero.

DBGLINE: LENGTH_NOT_MULTIPLE_OF_ENTRY A: Syn DWARF 1: 4.16
The length field beginning a new source statement table shall have a value which is a multiple of the source statement entry size.

DBGLINE: LENGTH_TOO_BIG A: Syn DWARF 1: 3.11
The length field beginning a new source statement table shall give the length, including itself, of the table such that the entire table is contained within a single ".line" section.

DBGLINE: LENGTH_TOO_SMALL A: Syn DWARF 1: 3.11
The length field beginning a new source statement table shall have a value at least equal to the minimum size of a source statement table (20 bytes).

DBGLINE: LINE_ENTRY_MISSING A: Sem DWARF 1: 3.11
Every C source statement shall be represented by an entry in the corresponding source statement table in the ".line" section.

DBGLINE: LINE_ENTRY_UNEXPECTED A: Sem DWARF 1: 3.11
Every entry in a source statement table in the ".line" section shall give the line number of an executable C statement in the corresponding source file.

DBGLINE: SECTION_TOO_SMALL A: Syn DWARF 1: 3.11
A non-null ".line" section shall be of sufficient size to hold at least one source statement table.

DBGLINE: STATEMENT_COLUMN_MISSING A: Sem DWARF 1: 3.11
Every C source statement shall be represented by an entry in the corresponding source statement table having a column number which is either SOURCE_NO_POS (0xffff) or which matches the first column number of that source statement.

Linked Objects

```
LINKED:  LINKED_ATTRIBUTE_CHANGED          A: Syn
         SVR4 ABI: 4: Sections (implicit)
   A linker shall not change the value of any attribute of a debugging
   information entry (except for address values which may undergo relocation).

LINKED:  LINKED_ATTRIBUTE_MISSING          A: Syn
         SVR4 ABI: 4: Sections (implicit)
   All attributes of a given debugging information entry input to a linker
   shall be present in the corresponding debugging information entry output
   by the linker.

LINKED:  LINKED_ATTRIBUTE_UNEXPECTED       A: Syn
         SVR4 ABI: 4: Sections (implicit)
   A linker shall not add attributes to debugging information entries.

LINKED:  LINKED_BAD_SHN_ABS                A: Syn
         SVR4 ABI: 4: Sections (implicit)
   A linker shall preserve a section index value of SHN_ABS (that is, if the
   st_shndx value of an input symbol table entry is SHN_ABS then the st_shndx
   value of the output symbol table entry having the same name shall be
   SHN_ABS).

LINKED:  LINKED_BAD_SHN_COMMON             A: Syn  SVR4 ABI: 4: Symbol Table
   A linker shall allocate and assign to a section a data object having a
   section index of SHN_COMMON (that is, if the st_shndx value of an input
   symbol table entry is SHN_COMMON then the st_shndx value of the output
   symbol table entry having the same name shall be that of some actual
   section not SHN_COMMON).

LINKED:  LINKED_BAD_ST_BIND                A: Syn
         SVR4 ABI: 4: Symbol Table (implicit)
   A linker shall preserve the public/private state of a symbol (that is, if
   the ELF32_ST_BIND value of an input symbol table entry is STB_GLOBAL or
   STB_WEAK then the ELF32_ST_BIND value of the output symbol table entry
   with the same name shall be STB_GLOBAL or STB_WEAK, or if the input
   binding is STB_LOCAL the output binding shall be STB_LOCAL).

LINKED:  LINKED_BAD_ST_OTHER               A: Syn
         SVR4 ABI: 4: Symbol Table (implicit)
   A linker shall preserve the st_other value of a symbol (that is, the
   st_other values shall be the same for input and output symbol table
   entries having the same name).

LINKED:  LINKED_BAD_ST_SIZE                A: Syn
         SVR4 ABI: 4: Symbol Table (implicit)
   A linker shall preserve the size of a public data object with non-zero
   size (that is, the st_size values shall be the same for input and output
   symbol table entries having the same name and being of type STT_OBJECT,
   with binding STB_GLOBAL or STB_WEAK, and with non-zero st_size value).
```

```
LINKED:  LINKED_BAD_ST_TYPE                A: Syn
         SVR4 ABI: 4: Symbol Table (implicit)
   A linker shall preserve the type of a symbol (that is, the ELF32_ST_TYPE
   values shall be the same for input and output symbol table entries having
   the same name).

LINKED:  LINKED_DUPLICATE_SYMBOL           A: Syn
         SVR4 ABI: 4: Symbol Table (implicit)
   An entry, as identified by its name, shall appear only once in a symbol
   table.

LINKED:  LINKED_SECTION_MISSING            A: Syn
         SVR4 ABI: 4: Sections (implicit)
   A linker shall output all sections found in its input.

LINKED:  LINKED_SECTION_UNEXPECTED         A: Syn
         SVR4 ABI: 4: Sections (implicit)
   A linker shall not create sections not found in its input.

LINKED:  LINKED_SYMBOL_MISSING             A: Syn
         SVR4 ABI: 4: Symbol Table (implicit)
   A linker shall output all symbol table entries found in its input.

LINKED:  LINKED_SYMBOL_UNEXPECTED          A: Syn
         SVR4 ABI: 4: Sections (implicit)
   A linker shall not create symbol table entries not found in its input.

LINKED:  SECTION_CONCATENATION_GAPS        A: Syn
         SVR4 ABI: 4: Sections (implicit)
   All sections in the input object files having the same name shall be
   combined in arrival order into a single section by that name in the linked
   output file with no gaps within the section except as required by
   alignment constraints.

LINKED:  SECTION_CONCATENATION_SIZE        A: Syn
         SVR4 ABI: 4: Sections (implicit)
   All sections in the input object files having the same name shall be
   combined into a single section by that name in the linked output file and
   the size of that output section shall be the sum of the sizes of those
   input sections plus the sizes of any gaps required by alignment
   constraints.
```

lvppc Assertions

```
ARCHIVE: ARCHIVE_SYMBOL_MISSING   A: Syn  SVR4 ABI: 7. Archive File
   An external symbol in an archive object file member shall be present in
   the archive symbol table.

ARCHIVE: ARCHIVE_SYMBOL_UNEXPECTED   A: Syn  SVR4 ABI: 7. Archive File
   A symbol shall not be present in the archive file symbol table unless it
   is an external symbol in some object file member in the archive file.
```

```
ARCHIVE: ARMAG_WRONG   A: Syn  SVR4 ABI: 7. Archive File
   An archive file shall begin with the characters '!<elf_>\n'.

ARCHIVE: EXCESS_BYTE_IN_FILE   A: Syn  SVR4 ABI: 7. Archive File
   No bytes in the file shall not be contained in a member.

ARCHIVE: AR_DATE_NOT_DECIMAL   A: Syn  SVR4 ABI: 7. Archive File
   Each archive header ar_date field shall be 12 bytes in length and contain
   the decimal representation of the modification date of the file at the
   time of its insertion into the archive, in a system dependent format.

ARCHIVE: AR_FMAG_WRONG   A: Syn  SVR4 ABI: 7. Archive File
   Each archive header ar_fmag field shall be 2 bytes in length and contains
   ar_fmag[0] = ''', ar_fmag[1] = '\n'.

ARCHIVE: AR_GID_NOT_DECIMAL   A: Syn  SVR4 ABI: 7. Archive File
   Each archive header ar_gid field shall be 6 bytes in length and contain
   the decimal representation of the member's group id.

ARCHIVE: AR_MODE_NOT_OCTAL   A: Syn  SVR4 ABI: 7. Archive File
   Each archive header ar_mode field shall be 8 bytes in length and contain
   the octal representation of the member's file system mode.

ARCHIVE: AR_NAME_LENGTH_WRONG   A: Syn  SVR4 ABI: 7. Archive File
   The archive header field ar_name shall be 16 bytes in length.

ARCHIVE: AR_NAME_IN_15_WRONG   A: Syn  SVR4 ABI: 7. Archive File
   If an archive member's name is 15 bytes or less in length the ar_name
   field of its archive header shall contain the member's file name
   terminated with '/' and padded with blanks on the right.

ARCHIVE: AR_NAME_OVER_15_WRONG   A: Syn  SVR4 ABI: 7. Archive File
   If an archive member's name is 16 bytes or more in length the ar_name
   field of its archive header shall contain a '/' followed by a zero-based
   offset of the member's name in the archive string table padded with blanks
   on the right.

ARCHIVE: AR_SIZE_DECIMAL   A: Syn  SVR4 ABI: 7. Archive File
   Each archive header ar_size field shall be 10 bytes in length and contain
   the decimal representation of the member's size in bytes.

ARCHIVE: AR_UID_NOT_DECIMAL   A: Syn  SVR4 ABI: 7. Archive File
   Each archive header ar_uid field shall be 6 bytes in length and contain
   the decimal representation of the member's user id.

ARCHIVE: MEMBER_CONTENTS_CHANGED   A: Syn  SVR4 ABI: 7. Archive File
   Each archive member shall consist of an archive header followed by the
   unchanged contents of the archived file.

ARCHIVE: MEMBER_MISALIGNED   A: Syn  SVR4 ABI: 7. Archive File
```

Each archive member shall begin on an even byte boundary with a newline used as padding if necessary.

ARCHIVE: STRING_TABLE_AR_NAME_WRONG A: Syn SVR4 ABI: 7. Archive File
If an archive string table is present its ar_name field shall consist of two ‘//’ characters followed by 14 blanks.

ARCHIVE: STRING_TABLE_MISSING A: Syn SVR4 ABI: 7. Archive File
If any archive member’s name is more than 15 bytes long, a string table shall precede all normal archive members, following the archive symbol table if it exists.

ARCHIVE: STRING_TABLE_WRONG A: Syn SVR4 ABI: 7. Archive File
If an archive string table is present it shall contain an array of the file names of the archive members, each followed by ‘/’ and a ‘\n’.

ARCHIVE: SYMBOL_TABLE_ARRAY_TOO_SHORT A: Syn SVR4 ABI: 7. Archive File
If an archive symbol table is present, the word after that containing the number of symbols in the table shall begin an array of words in big-endian byte order containing the offset within the archive of the archive header of the member in which each symbol is defined.

ARCHIVE: SYMBOL_TABLE_AR_NAME_WRONG A: Syn SVR4 ABI: 7. Archive File
If an archive symbol table is present its ar_name field shall contain ‘/’ followed by 15 blanks.

ARCHIVE: SYMBOL_TABLE_AR_SIZE_WRONG A: Syn SVR4 ABI: 7. Archive File
If an archive symbol table is present, the field ar_size for its archive header shall indicate the size of the sum of the word containing the number of symbols in the table, its file offset array, and its symbol name array.

ARCHIVE: SYMBOL_TABLE_MISSING A: Syn SVR4 ABI: 7. Archive File
If an archive file has one or more object file members with external symbols, its first member shall be an archive symbol table.

ARCHIVE: SYMBOL_TABLE_NAME_UNTERMINATED A: Syn SVR4 ABI: 7. Archive File
If an archive symbol table is present, the word following the file offset array shall begin an array of null terminated names for each symbol.

ARCHIVE: SYMBOL_TABLE_OUT_OF_ORDER A: Syn SVR4 ABI: 7. Archive File
If an archive symbol table is present, entries in its file offset and symbol name arrays shall be in the order of the archive members.

ARCHIVE: SYMBOL_TABLE_TOO_SHORT A: Syn SVR4 ABI: 7. Archive File
If an archive symbol table is present, its first 4 bytes shall be a word in big-endian byte order containing the total number of symbols in the symbol table.

Run-time assertions

Run-time alignment assertions

```
ALGNSEC: ALGN_BIG_ENDIAN_ORDER            A: Sem
               PowerPC ABI: 3. Data Representation
   Big-endian byte order means that the most significant byte of a halfword-
   or word-sized storage unit resides at the lowest byte address.

ALGNSEC: ALGN_LITTLE_ENDIAN_ORDER         A: Sem
               PowerPC ABI: 3. Data Representation
   Little-endian byte order means that the least significant byte of a
   halfword- or word-sized storage unit resides at the lowest byte address.

ALGNSEC: ALGN_TYPE_SIZES                  A: Sem
               PowerPC ABI: 3. Data Representation
   Fundamental types are of the correct size.  This includes char, unsigned
   char, signed char, short, signed short, unsigned short, int, signed int,
   long int, signed long, enum, unsigned int, unsigned long, pointer *,
   function (*)(), float, double, long double, and if supported, long long.

ALGNSEC: ALGN_TYPE_ALIGNMENT              A: Sem
               PowerPC ABI: 3. Data Representation
   Fundamental types are of the correct alignment.  This includes char,
   unsigned char, signed char, short, signed short, unsigned short, int,
   signed int, long int, signed long, enum, unsigned int, unsigned long,
   pointer *, function (*)(), float, double, long double, and if supported,
   long long.

ALGNSEC: ALGN_EABI_LONG_DBL               A: Sem
               PowerPC EABI: Data Representation
   The alignment of long double shall be 8 bytes (doubleword), although the
   size of long double shall be 16 bytes.

ALGNSEC: ALGN_EABI_AGGREGATE_W_LONG_DBL   A: Sem
               PowerPC EABI: Data Representation
   An array, structure or union containing a long double shall start aligned
   on an 8 byte boundary.

ALGNSEC: ALGN_ARRAY_ALIGNMENT             A: Sem
               PowerPC ABI: 3. Data Representation
   An array uses the same alignment as its elements.
ALGNSEC: ALGN_STRUCT_MAX_ALIGNMENT        A: Sem
               PowerPC ABI: 3. Data Representation
   A structure or union is aligned on the same boundary as its most strictly
   aligned component (i.e. the component with the maximum alignment).

ALGNSEC: ALGN_INTERNAL_PADDING            A: Sem
               PowerPC ABI: 3. Data Representation
```

```
   Each member is assigned the lowest available offset with the appropriate
   alignment.  This may require internal padding, depending on the previous
   member.

ALGNSEC: ALGN_AGGREGATE_MULT_OF_ALIGN     A: Sem
               PowerPC ABI: 3. Data Representation
   The size of any object, including aggregates and unions, is always a
   multiple of the object's alignment.

ALGNSEC: ALGN_BIT_STRUCT_MAX_ALIGNMENT    A: Sem
               PowerPC ABI: 3. Data Representation
   A structure or union with bit-fields is aligned on the same boundary as
   its most strictly aligned component (i.e. the component with the maximum
   alignment).

ALGNSEC: ALGN_BIT_INTERNAL_PADDING        A: Sem
               PowerPC ABI: 3. Data Representation
   Each bit-field member is assigned the lowest available offset with the
   appropriate alignment.  This may require internal padding, depending on
   the previous member.

ALGNSEC: ALGN_BIT_AGGREGATE_MULT_OF_ALIGN A: Sem
               PowerPC ABI: 3. Data Representation
   The size of any object, including aggregates and unions, with bit-fields
   is always a multiple of the object's alignment.

ALGNSEC: ALGN_BIT_FIELD_RANGES            A: Sem
               PowerPC ABI: 3. Data Representation
   Bit-fields have the appropriate ranges for the declared types including
   signed char, char, unsigned char, signed short, short, unsigned short,
   signed int, int, enum, unsigned int, signed long, long, and unsigned long.

ALGNSEC: ALGN_BIT_PLAIN_UNSIGNED          A: Sem
               PowerPC ABI: 3. Data Representation
   "Plain" bit-fields (that is, those neither signed no unsigned) always have
   non-negative values and have the same range as bit-fields of the same size
   with the corresponding unsigned type.

ALGNSEC: ALGN_BIT_ALLOCATION_ORDER        A: Sem
               PowerPC ABI: 3. Data Representation
   Bit-fields are allocated from least- to most-significant in little-endian
   implementations and most- to least-significant in big-endian
   implementations.

ALGNSEC: ALGN_BIT_CANNOT_CROSS_STOR_UNIT  A: Sem
               PowerPC ABI: 3. Data Representation
   A bit-field must entirely reside in a storage unit appropriate for its
   declared type.

ALGNSEC: ALGN_BIT_STOR_UNIT_SHARING       A: Sem
               PowerPC ABI: 3. Data Representation
```

```
   Bit-fields must share a storage unit with other structure and union
   members (either bit-field or non-bit-field) if and only if there is
   sufficient space within the storage unit.

ALGNSEC: ALGN_BIT_UNNAMED_BIT_FIELD_ALGN  A: Sem
               PowerPC ABI: 3. Data Representation
   Unnamed bit-fields' types do not affect the alignment of a structure or
   union, although an individual bit-field's member offsets obey the
   alignment constraints.

ALGNSEC: ALGN_BIT_UNNAMED_BIT_FIELD_0_LEN A: Sem
               PowerPC ABI: 3. Data Representation
   An unnamed, zero-width bit-field shall prevent any further bit-field or
   other member from residing in the storage unit corresponding to the type
   of the zero-width bit-field.
```

Run-time call assertions

Registers

```
CALLSEC: CALL_NONVOLTILE_REGS             A: Sem
               PowerPC ABI: 3. Function Calling Conventions: Registers
   Registers r1, r14-r31, f14-f31 are nonvolatile to the calling function.

CALLSEC: CALL_EABI_R2_ELF                 A: Sem
               PowerPC EABI: 3. Function Calling Conventions
   GPR2 shall contain the base of the ELF sections named .sdata2 and .sbss2,
   if either section exists in an object file.

CALLSEC: CALL_EABI_R2_ELF_16_BIT_OFF      A: Sem
               PowerPC EABI: 3. Function Calling Conventions
   If GPR2 contains the base of the ELF sections named .sdata2 and .sbss2,
   the base is an address such that every byte in the section is within a
   signed 16-bit offset of that address.

CALLSEC: CALL_EABI_R2_ELF_SHARED_OBJ      A: Sem
               PowerPC EABI: 3. Function Calling Conventions
   A routine in an ELF shared object file shall not use GPR2.

CALLSEC: CALL_R13_SDA_BASE                A: Sem
               PowerPC ABI: 3. Function Calling Conventions: Registers
   Register r13 is the small data area pointer (the loader defined symbol
   _SDA_BASE_).

CALLSEC: CALL_R31_ENVIRONMENT_PTR         A: Sem
               PowerPC ABI: 3. Function Calling Conventions: Registers
   Languages that require environment pointers shall use r31 for that purpose.

CALLSEC: CALL_COND_REGS_NONVOLATILE       A: Sem
               PowerPC ABI: 3. Function Calling Conventions: Registers
```

```
   Fields CR2, CR3, and CR4 of the condition register are nonvolatile.

CALLSEC: CALL_STACK_16_BYTE_ALIGN          A: Sem
               PowerPC ABI: 3. Function Calling Conventions: Registers
   The stack pointer (stored in r1) shall maintain 16-byte alignment.

CALLSEC: CALL_STACK_LOWEST_ALLOC_FRAME     A: Sem
               PowerPC ABI: 3. Function Calling Conventions: Registers
   The stack pointer shall point to the lowest allocated, valid stack frame.

CALLSEC: CALL_STACK_GROWS_TO_LOW_ADDRESSES A: Sem
               PowerPC ABI: 3. Function Calling Conventions: Registers
   The stack pointer shall grow towards low addresses.

CALLSEC: CALL_STACK_PTR_TO_PREV_FRAME      A: Sem
               PowerPC ABI: 3. Function Calling Conventions: Registers
   The contents of the word at the address pointed to by the stack pointer
   shall point to the previously allocated stack frame.

CALLSEC: CALL_VARARGS_COND_REG             A: Sem
               PowerPC ABI: 3. Function Calling Conventions: Variable
               Argument Lists
   CR bit 6 (CR1, floating-point invalid exception) shall be set by the
   caller of a variable argument list function.

CALLSEC: CALL_LR                           A: Sem
               PowerPC ABI: 3. Function Calling Conventions: Registers
   The LR register contains the address to which a called function normally
   returns.

CALLSEC: CALL_SIG_HANDLING                 A: Sem
               PowerPC ABI: 3. Function Calling Conventions: Registers
   If a signal handling function returns, the process resumes its original
   execution path with all registers restored to their original values.
```

Stack Frame

```
CALLSEC: CALL_EABI_STACK_8_BYTE_ALIGN      A: Sem
               PowerPC EABI: Function Calling Sequence
   The stack pointer (GR1) shall maintain 8-byte alignment, from
   initialization through all routine calls and dynamic stack space
   allocation.

CALLSEC: CALL_STACK_FIRST_FRAME            A: Sem
               PowerPC ABI: 3. Function Calling Conventions: Registers
   The contents of the word at the address pointed to by the first stack
   pointer shall point to 0 (NULL).

CALLSEC: CALL_STACK_PTR_RESTORED           A: Sem
               PowerPC ABI: 3. Function Calling Conventions: The Stack Frame
   The stack pointer shall be decremented by the called function in its
   prologue, if required, and restored prior to return.
```

```
CALLSEC: CALL_STACK_STORE_WORD_W_UPDATE    A: Sem
              PowerPC ABI: 3. Function Calling Conventions: The Stack Frame
   The stack pointer shall be decremented and the back chain updated
   atomically using one of the Store Word with Update instructions, so that
   the stack pointer always points to the beginning of a linked list of stack
   frames.

CALLSEC: CALL_STACK_PARM_AREA_CALLER_ALLOC A: Sem
              PowerPC ABI: 3. Function Calling Conventions: The Stack Frame
   The parameter list area shall be allocated by the caller and shall be
   large enough to contain the arguments that the caller stores in it.

CALLSEC: CALL_SAVE_FR_REG                  A: Sem
              PowerPC ABI: 3. Function Calling Conventions: The Stack Frame
   Before a function changes the value in any nonvolatile floating-point
   register, frn, it shall save the value in frn in the double word in the
   floating-point register save area 8*(32-n) bytes before the back chain
   word of the previous frame.

CALLSEC: CALL_SAVE_GR_REG                  A: Sem
              PowerPC ABI: 3. Function Calling Conventions: The Stack Frame
   Before a function changes the value in any nonvolatile general register,
   rn, it shall save the value in rn in the word in the general register save
   area 4*(32-n) bytes before the low-addressed end of the floating-point
   register save area.

CALLSEC: CALL_SAVE_CR_REG                  A: Sem
              PowerPC ABI: 3. Function Calling Conventions: The Stack Frame
   Before a function changes the value in any nonvolatile field in the
   condition register, it shall save the values in all the nonvolatile fields
   of the condition register at the time of entry to the function the CR save
   area.

CALLSEC: CALL_STACK_HAS_BACK_CHAIN_LR      A: Sem
              PowerPC ABI: 3. Function Calling Conventions: The Stack Frame
   The stack frame header consists of the back chain word and the LR save
   word.

CALLSEC: CALL_PADDING_IN_LOC_VAR_AREA      A: Sem
              PowerPC ABI: 3. Function Calling Conventions: The Stack Frame
   Any padding of the frame as a whole shall be within the local variable
   area.

CALLSEC: CALL_PARM_LIST_FOLLOWS_HEADER     A: Sem
              PowerPC ABI: 3. Function Calling Conventions: The Stack Frame
   The parameter list area shall immediately follow the stack frame header.

CALLSEC: CALL_REG_SAV_AREA_NO_PAD          A: Sem
              PowerPC ABI: 3. Function Calling Conventions: The Stack Frame
   The register save areas shall contain no padding.
```

Parameter Passing

```
CALLSEC: CALL_8_GR_FOR_PARM                A: Sem
              PowerPC ABI: 3. Function Calling Conventions: Parameter Passing
   Up to eight words are passed in general purpose registers, loaded
   sequentially into general purpose registers r3 through r10.

CALLSEC: CALL_8_FR_FOR_PARM                A: Sem
              PowerPC ABI: 3. Function Calling Conventions: Parameter Passing
   Up to eight words are passed in floating-point registers, loaded
   sequentially into floating-point registers f1 through f8.

CALLSEC: CALL_DBL_PARM_IN_FR               A: Sem
              PowerPC ABI: 3. Function Calling Conventions
   Double or float arguments are placed into the floating-point registers
   when available.

CALLSEC: CALL_SIMPLE_ARG_IN_GR             A: Sem
              PowerPC ABI: 3. Function Calling Conventions
   Arguments of type char, short, int, long, enum, and pointers an object are
   placed into general registers when available.

CALLSEC: CALL_STRUCT_UNION_DBL_IN_GR       A: Sem
              PowerPC ABI: 3. Function Calling Conventions
   An argument of type struct, union, or long double, any of which shall be
   treated as a pointer to the object placed into general registers when
   available.

CALLSEC: CALL_STRUCT_UNION_DBL_COPIES      A: Sem
              PowerPC ABI: 3. Function Calling Conventions: Parameter passing
   An argument of type struct, union, or long double, any of which shall be
   treated as a pointer to a copy of the object when call-by-value semantics
   are required.

CALLSEC: CALL_STRUCT_UNION_DBL_ORIG        A: Sem
              PowerPC ABI: 3. Function Calling Conventions: Parameter passing
   An argument of type struct, union, or long double, any of which shall be
   treated as a pointer the object may pass a pointer to the object itself
   only if the caller can ascertain that the object is constant.

CALLSEC: CALL_STACK_PARMS                  A: Sem
              PowerPC ABI: 3. Function Calling Conventions: Parameter Passing
   Arguments not otherwise handled above are passed in the parameter words of
   the caller's stack frame.

CALLSEC: CALL_LONG_LONG_PARMS_ALIGN        A: Sem
              PowerPC ABI: 3. Function Calling Conventions
   If an implementation supports long long data types, the argument is
   aligned on an even word boundary.

CALLSEC: CALL_LONG_LONG_PARM_ORDER         A: Sem
              PowerPC ABI: 3. Function Calling Conventions
```

```
   If an implementation supports long long data types, load the
   lower-addressed word of the long long into the general register and the
   higher-addressed word into general register + 1.

CALLSEC: CALL_SIMPLE_ARGS_ALIGN            A: Sem
               PowerPC ABI: 3. Function Calling Conventions
   Arguments of type char, short, int, long, enum, pointer to an object of
   any type, and pointers to a struct, union, or long double passed on the
   stack are considered to have 4-byte size and alignment, with simple
   integer types shorter than 32 bits sign- or zero-extended (conceptually)
   to 32 bits.

CALLSEC: CALL_FLOAT_ARGS_ALIGN             A: Sem
               PowerPC ABI: 3. Function Calling Conventions
   Float, long long (where implemented), and double arguments passed on the
   stack are considered to have 8-byte size and alignment, with float
   arguments converted to double representation.

CALLSEC: CALL_ARG_ALIGN                    A: Sem
               PowerPC ABI: 3. Function Calling Conventions
   Arguments passed upon the stack are aligned on a stack address that is a
   multiple of the alignment requirement of the argument.

CALLSEC: CALL_ARG_COPYING                  A: Sem
               PowerPC ABI: 3. Function Calling Conventions
   Arguments passed upon the stack are copied byte-for-byte, beginning with
   its lowest addressed byte, into increasing stack addresses.
```

Return Values

```
CALLSEC: CALL_RET_DBL                      A: Sem
               PowerPC ABI: 3. Function Calling Conventions: Return Values
   Functions shall return double values in f1.

CALLSEC: CALL_RET_FLOAT                    A: Sem
               PowerPC ABI: 3. Function Calling Conventions: Return Values
   Functions shall return float values rounded to single precision in f1.

CALLSEC: CALL_RET_WORD                     A: Sem
               PowerPC ABI: 3. Function Calling Conventions: Return Values
   Functions shall return values of type int, long, enum, short, and char, or
   a pointer to any type as unsigned or signed integers as appropriate, zero-
   or sign-extended to 32 bits if necessary, in r3.

CALLSEC: CALL_RET_SMALL_STRUCT_UNION       A: Sem
               PowerPC ABI: 3. Function Calling Conventions: Return Values
   A structure or union whose size is less than or equal to 8 bytes shall be
   returned in r3 and r4, as if it were stored in an 8-byte aligned memory
   area and then the low-addressed word were loaded into r3 and the
   high-addressed word into r4.

CALLSEC: CALL_RET_LONG_LONG                A: Sem
```

```
                PowerPC ABI: 3. Function Calling Conventions: Return Values
   Values of type long long and unsigned long long, where supported, shall be
   returned with the lower addressed word in r3 and the higher in r4.

CALLSEC: CALL_RET_LONG_DBL_STRUCT          A: Sem
                PowerPC ABI: 3. Function Calling Conventions: Return Values
   Values of type long double and struct that do not meet the requirements
   for being returned in registers are returned in a storage buffer allocated
   by the caller.

CALLSEC: CALL_RET_UNDEF_STRUCT_BITS        A: Sem
                PowerPC ABI: 3. Function Calling Conventions
   Bits beyond the last member of the structure or union returned in a
   storage buffer allocated by the caller, are not defined.

CALLSEC: CALL_RET_ADDR_BUFFER              A: Sem
                PowerPC ABI: 3. Function Calling Conventions: Return Values
   The address of a buffer used for returning large return types is passed as
   a hidden argument in r3 as if it were the first argument, causing the
   first available general register for argument passing to be r4 instead of
   r3.
```

B Expectations Language

The Expectations Language produced by the `cparse` program is intended to be both human-readable and easily-parsed by a program. The following are the basic concepts of this language.

- The language is written as ASCII text.
- Every "statement" in the language is contained within a single line of text, even if that line must be very long.
- Every statement in the language has a fixed set of fields, all present (except possibly the last field), so that once recognized, most statements can be parsed by a call to *sscanf*. Fields are separated by white space.
- Every statement begins with a three-letter abbreviation indicating the subject of the statement (BLO for block, TYP for type, etc.), followed by a single character denoting a subkind or modifier. For cases in which no modifier is required, an asterisk is used as a place holder. The modifier **"**<" means beginning as in beginning-of-block and the modifier **"**>" means" ending as in end-of-block.
- The modifier is followed by a *source_location* in the form of two integer fields specifying the line and column number in the source where the subject of the statement appears. Additional details about *source_location* are given in the discussion of Syntax later in this chapter.
- When statements include identifiers or strings (e.g. the name of a function), these longer fields come at the end of the statement.

Statement subjects and modifiers

The following are the subject abbreviations in the language, i.e., the recognized beginnings of statement lines, and their associated modifiers.

➤ Some of these expectation statements provide information not currently used by DWARF1.

BAS	*	definition of a base type
BIT	*	bit-field member of a **struct** or **union**
BLO	<	block begin
BLO	>	block end
COM	C	beginning of a compilation unit, C language
DEF	d	preprocessor definition
DIM	*	dimension of an array
ENU	*	enumeration member
FUN	*f*	function

INC	*	include file (SRC to follow)
LAB	*	label
LIN	*	new name for source file
MEM	*	member of a struct or union (other than a bit field)
NAM	*	name given to a type by a **typedef**
OLD	*	current function definition used old-style header
PAR	*p*	parameter (of a function)
REF	*r*	reference to (use of) a function or function pointer
SRC	<	source file begin
SRC	>	source file end
STM	*	executable statement (other than block begin or block end)
TYP	*k*	type definition (the possible values of *k* are given below)
TYP	>	end of a complex type (explained below)
UND	*	preprocessor undefinition (**#undef**)
VAR	*s*	variable (the possible values of *s* are given below).

Definition subkind

The vlaues of the subkind *d* for a preprocessor #define are:

+ for a repeated #define (same name and definition) without an intervening #undef
* all other uses of #define

Function forms

The values of the form *f* for functions are:

D extern declaration

d **static** (private) declaration

F public definition

f **static** (private) definition

Kinds of types

The values of the subkind *k* for types are listed below. Those marked “complex type” require multiple statements to fully specify the type, e.g., a struct requires nested MEM or BIT statements to enumerate the members of the struct.

A	array	complex type, followed by exactly one dimension (DIM statement)

C	const	const form of a previously known type
E	**enum**	complex type, followed by enumeration members (ENU statements)
F	function	complex type, followed by parameters (PAR statements)
P	pointer	
S	**struct**	complex type, followed by members (MEM statements)
U	**union**	complex type, followed by members (MEM statements)
V	**volatile**	volatile form of a previously known type

Parameter kind

The values of the parameter subkind *p* for function parameters is:

r	register storage
.	variable parameters (...)
*	all other parameters

Reference kind

The value of the reference subkind *r* for function references is:

F	for reference to a declared function
P	for reference to a declared function pointer
U	for reference to an undeclared function

Storage classes for variables

The values of the storage subkind *s* for variables are:

A	automatic
E	extern
P	public
R	register
S	static

Statement syntax

Listed below is the syntax for the various statements expressed in terms of the required sequence of fields.

BAS	*	*loc*	*new_ordinal*	*byte_size*			*base_type_name*
BIT	*	*loc*	*type_ref*	*byte_offset*	*bit_offset*	*bit_size*	*[name]*

BLO	*b*	*loc*			
COM	C	*loc*			*source-file_name*
DEF	*d*	*loc*			*name [definition]*
DIM	*	*loc*	*low_bound*	*high_bound*	
ENU	*	*loc*	*value*		*name*
FUN	*f*	*loc*	*type_ref*		*name*
INC	*	*loc*			*file_spec*
LAB	*	*loc*			*name*
LIN	*	*loc*			*new_name_of_source_file*
MEM	*	*loc*	*type_ref*	*byte-offset*	*[name]*
NAM	*	*loc*	*type_ref*		*name*
OLD	*				
PAR	*p*	*loc*	*type_ref*		*name*
REF	*r*	*loc*			*name*
SRC	*b*	*loc*			*file_name*
STM	*	*loc*			
TYP	*k*	*loc*	*new_ordinal*	*type_ref*	*[tag_name*]
TYP	>	*loc*	*new_ordinal*	*total_bytes*	
UND	*	*loc*			*name*
VAR	*s*	*loc*	*type_ref*		*name*

The following are the syntactic abbreviations used above:

- *b* is an angle bracket, either a less-than sign (begin) or greater-than sign (end).
- *f* is a function form as previously defined
- *k* is a type kind (subkind) as previously defined.
- *loc* is an abbreviation for source_location (two integer fields).
- *s* is a storage class as previously defined.

Also, the following semantic definitions and rules apply to the linguistic variables in the statement syntax above:

base_name — the name of a base type; it need not be a simple identifier, e.g. **signed int** is a valid base name

bit_offset	the offset of a bit field within its underlying type (**int** or **unsigned**); bit offsets depend on the byte-ordering in effect (big-endian or little-endian)
byte_offset	the offset of the left-most byte of a member within a structure or union; in the case of a bit field, this is the offset of the left-most byte of the field's container within the structure or union
byte_size	byte size of a base type or of a structure or union member; in the case of a bit field, this is the size of the bit field plus any padding bits required
high_bound	the high bound of a dimension of an array; in C, the number of elements
language	**C** for the C language
low_bound	the low bound of a dimension of an array; in C, always 0
new_ordinal	the ordinal assigned to a type that is being defined
producer	string giving the name of the producer of the compiler
source-filename	string giving the name of the source file from which this compilation unit was derived (exactly, with or without a path, as specified in the compilation command)
type_ref	the ordinal of a previously defined type
total_bytes	the total size of a struct, union, or array
work-directory	string giving the working directory at the time of the compilation command that produced this compilation unit

Source location

In the DWARF specification there is a concept called *Declaration Coordinates* defined to be the file name, line number, and column number of the first character of an identifier. The Expectation Language uses *source_location* for a similar purpose.

A *source_location* is made up of two integers (separated by white space) specifying the line and column number in the source where the subject of a statement appears. For identifiers, the location is that of the first character; for other constructs, the syntactic position designated by the *source_location* is defined below.

Line numbers and column numbers begin with 1 to mean first line or first column. A value of 0 means "undefined".

The following are the specific interpretations of source location for each kind of subject.

BAS		undefined
BIT		location of the bit field identifier
BLO	<	location of the opening left brace that begins the block
BLO	<	location of the closing right brace that ends the block

COM		not applicable
DEF		location of the identifier defined by #define
DIM		undefined
ENU		location of the enumeration member identifier
FUN		location of the function name
FUN	>	undefined
INC		line number of the #include, with column number undefined
LAB		location of the first character of the label
MEM		location of the member identifier, if named, otherwise undefined
NAM		location of the typedef identifier
PAR		location of the parameter identifier
SRC	<	location of the first visible character that is not part of a comment
SRC	>	location of the last visible character that is not part of a comment
STM		location of the first visible character of the executable statement
TYP		location of the tag if a tag is given (for **enum**, **struct**, or **union**)
TYP	>	undefined
UND		location of the identifier being undefined
VAR		location of the variable identifier.

Type ordinal

When a type is first introduced, it is assigned a new type-ordinal, and it is that ordinal number which is used for later reference in derived types, variables, functions, etc.

The term *type-ref* is used in the statement syntax to refer to an underlying type from which a type is derived, e.g.,when **const int** is introduced, **int** is its base type. and similarly for **int** [] the base type is **int**. When there is no underlying type, e.g., for a struct, zero is used as a placeholder in the *type_ref* field.

Sample statements

Consider the following C source program, with line numbers shown at the start of each line.

```
1   /* src001.c */
2
3   #define d000      0
4   #define void
5
6   typedef enum    e001    {member001 =   1}       type001;
```

```
7   typedef enum    e002    {member002 =   2}       type002;
8
9   struct  s
10      {
11          int         i001;
12          unsigned    u001:5;
13      }
14       v;
15
16
17  static int compute ()
18  {
19      int         sum;
20      struct s    w [2];
21      {
22          type001     v001    = member001;
23          type002     v002    = member002;
24
25          sum = v001 + v002 + (w [0].i001 = v.i001);
26      }
27      return sum;
28  }
29
30
31  int main (int argc, char * argv [])
32  {
33      return compute ();
34
35  }
36
37  #undef  d000
```

The expectations file generated from this C file is:

```
COM C     0    0                          src001.c
BAS *     0    0     1     1              char
BAS *     0    0     2     4              float
BAS *     0    0     3     8              double
BAS *     0    0     4    16              long double
BAS *     0    0     5     1              signed char
BAS *     0    0     6     4              signed int
BAS *     0    0     7     4              signed long
BAS *     0    0     8     2              signed short
BAS *     0    0     9     1              unsigned char
BAS *     0    0    10     4              unsigned int
BAS *     0    0    11     4              unsigned long
BAS *     0    0    12     2              unsigned short
BAS *     0    0    13     0              void
BAS *     0    0    14     4              int   (bit-field)
BAS *     0    0    15     4              long  (bit-field)
BAS *     0    0    16     2              short (bit-field)
```

```
SRC <     3   16                          src001.c
DEF *     3    2                          d000 0
DEF *     4    2                          void
TYP E     6   14    17     0              e001
ENU *     6   20     1                     member001
TYP >     6   35    17     4              e001
NAM *     6  289    17                    type001
TYP E     7   14    18     0              e002
ENU *     7   20     2                     member002
TYP >     7   35    18     4              e002
NAM *     7   38    18                    type002
TYP S     9    8    19     0              s
MEM *    11    9     6     0              i001
BIT *    12   12    10     4     0     5 u001
TYP >    13    2    19     8              s
VAR p    14    3    19                    v
TYP F     0    0    20     6
TYP >     0    0    20     0
FUN f    17   12    20                    compute
OLD *     0    0
BLO <    18    1
VAR a    19    8     6                    sum
TYP A    20    9    21    19
DIM *     0    0     0     2
TYP >     0    0    21    16
VAR a    20   11    21                    w
BLO <    21    2
STM *    22   12
VAR A    22   12    17                    v001
STM *    23   12
VAR A    23   12    18                    v002
STM *    25    3
BLO >    26    2
STM *    27    2
BLO >    28    1
FUN >     0    0
TYP P     0    0    22     1
TYP P     0    0    23    22
TYP F     0    0    24     6
PAR *     0    0     6
PAR *     0    0    23
TYP >     0    0    24     0
FUN F    31    5    24                    main
PAR *    31   15     6                    argc
PAR *    31   28    23                    argv
BLO <    32    1
STM *    33    2
REF F    33    9                          compute
BLO >    35    1
FUN >     0    0
```

```
UND *    37    2                    d000
SRC >    35    1                    src001.c
```

C PEATS TESTS

The following is a list of tests supplied with PEATS. Each test is in its own directory, and is used for compiling and linking. Directories with more than one module are also used for archiving. See “Steps taken for each kind of run” on page 32 for details on how the modules in a test directory participate in compiling, linking, and archiving tests.

Table 3-1 Test cases written by ApTest

Directory	Focus of test
test001	long variable names
test002	long function names
test003	long label names
test004	long tag names and long member names
test005	long typedef names
test006	many type names
test007	many member names within one type
test008	many #define names
test009	many function names
test010	many parameters within one function
test011	a single statement spanning many source lines
test012	many executable statements on one source line
test013	arrays with many dimensions
test014	many include files
test015	many #line directives
test016	many labels in one function
test017	a structure with many members
test018	a union with many members
test019	a deeply-nested pointer type
test020	a deeply-nested structure type
test021	a deeply-nested union type
test022	deeply-nested code blocks with declarations
test023	deeply-nested single struct definition
test024	construction of a complex type in a parameter list
test025	an enumeration with many assigned values

Table 3-1 Test cases written by ApTest*(continued)*

Directory	Focus of test
test026	a complex function return type
test027	a define with many parameters
test028	ISO types constructed with “const” and “volatile”
test029	many local variables
test030	many register variables
test031	many local static variables
test032	use of like names in different name spaces
test033	many function parameters designated “register”
test034	many local variables in a nested block
test035	many labels in nested blocks
test036	function pointers
test037	bit fields
test038	many public functions referenced across compilands
test039	five modules with functions that depend on each other
test040	nested structs and unions with anonymous bit fields
test041	incomplete array declarations
test046	large program combining many of the constructs from other test cases
test047	multiple levels of include files
test048	use of an include file with executable statements
test049	use of standard include files
test050	preprocessor #define and #undefine
test051	complex #define macros
test052	#line directives
test053	all of the base types
test054	all storage classes
test055	permuted type specifiers
test056	multiple declarators within a declaration
test057	complex declarators
test058	typedefs
test059	multi-dimensional array types
test060	application of type qualifiers to array types

Table 3-1 Test cases written by ApTest*(continued)*

Directory	Focus of test
test061	enumeration constants with complex value assignments
test062	definition of an enum type inside a constant expression
test063	tagged and untagged structs and unions
test064	signed, unsigned, and “plain int” bit fields
test065	alignment and sizing
test066	alignment and sizing of bit fields
test067	type qualifiers
test068	function types and function pointers
test069	old-style parameter lists
test070	referenced and unreferenced public variables
test071	adjustment of array parameters
test072	adjustment of function parameters
test073	equivalence of types differing only in array completions
test074	abstract parameter lists
test075	type casts
test076	scope of enumerated constants
test077	locally declared externs
test078	expressions using the sizeof operator

Table 3-2 Public domain test programs

Directory	Focus of test
test042	Fourteen `.c` files and `.h` files
test043	One `.c` file
test 044	One `.c` file
test.045	Ten `.c` files and `.h` files

Table 3-3 Supplemental run-time tests written by ApTest

Directory	Focus of test
	Alignment tests
testa001	Endian ordering (Big)
testa002	Endian ordering (Little)
testa003	Size of fundamental types
testa004	Alignment of fundamental types
testa005	EABI long double alignment
testa006	EABI aggregates with long doubles
testa007	Array alignment
testa008	Maximum alignment rule
testa009	Internal padding
testa010	Aggregates are a multiple of the object's alignment
testa011	Maximum alignment rule for bit-fields
testa012	Internal padding for bit fields
testa013	Aggregates w/bit-fields are a multiple of the obejct's alignment
testa014	Bit-field ranges
testa015	Plain bit-fields behave as unsigned
testa016	Bit-field allocation order
testa017	Bit-field can cross byte but not storage unit boundaries
testa018	Bit-fields may share storage unit with other struct/union members
testa019	Unamed bit-field alignment
testa020	Zero-length unnamed bit-fields
	Calling convention tests
testc002	EABI r2 for ELF .sdata2 and sbss2 sections
testc003	EABI r2 can reach all 16-bit offsets in ELF sdata2 and sbss2 sections
testc004	EABI r2 not used in ELF shared objects
testc005	r13 contains _SDA_BASE
testc006	r31 contains environment pointer
testc007	Nonvolatile condition registers
testc008	16-byte alignment of stack

Table 3-3 Supplemental run-time tests written by ApTest *(continued)*

Directory	Focus of test
testc009	Stack points to lowest allocated frame
testc010	Stack grows to low addresses
testc011	Back chain points to previous frame
testc012	Condition register setting when using varargs
testc013	LR registers
testc014	Signal handling handling and registers
testc015	EABI 8-byte aligned stack
testc016	First frame points to NULL
testc017	Stack pointer restored prior to return
testc018	Stack pointed updated iwth store word with update calls
testc019	Parameter list area caller allocated
testc020	Saving floating-point registers
testc021	Saving general registers
testc022	Saving condition registers
testc023	Stack has back chain and LR word
testc024	Padding in local variable area
testc025	Parameter list follows header
testc026	Register save area has no padding
testc027	8 general registers for parameters
testc028	8 floating-point registers for parameters
testc029	double parameter in floating-point register
testc030	Simple arguments in general register
testc031	struct, union, double arguments in general register
testc032	Pointers to copies of struct, union, double arguments
testc033	Pointers to original struct, union, double arguments
testc034	Parameters passed on stack
testc035	long long parameter alignment
testc036	long long parameter order
testc037	Simple argument alignment
testc038	Float-point argument alignment
testc039	Argument alignment

Table 3-3 Supplemental run-time tests written by ApTest *(continued)*

Directory	Focus of test
testc040	Argument copying
testc041	Return double
testc042	Return float
testc043	Return word
testc044	Return small struct union
testc045	Return long long
testc046	Return long double struct
testc047	Return undefined structure bits
testc048	Return address buffer