

This is an **Archive IEEE Standard**. It has been withdrawn or superseded by a later version of the standard.

ARCHIVE STANDARDS MAY HAVE VALUE AS HISTORICAL DOCUMENTS, BUT THEY ARE NOT VALID OR APPROVED IEEE STANDARDS. THE IEEE SHALL NOT BE LIABLE FOR ANY DAMAGES RESULTING FROM THE SUBSCRIBER'S OR USERS' PRACTICE OF THIS UNAPPROVED, ARCHIVE IEEE STANDARD.

For more information about this or other current, revised or withdrawn standards, please consult the IEEE Standards Status report on-line at <http://standards.ieee.org/db/status/>.

IEEE Standard for Conformance Test Requirements for Futurebus+

Sponsor
**Bus Architecture Standards Committee
of the
IEEE Computer Society**

Approved September 15, 1993
IEEE Standards Board

Abstract: A suite of abstract test cases for verifying the conformance of products based on the Futurebus+ family of standards and a means of selecting the exact set of test cases necessary for testing any given product are provided. The test suite is abstract in that it was not created with a specific test environment in mind. It can be effectively used in conjunction with logic simulators during product development as well as with physical testers after prototype or production units are available. This standard is intended for use by Futurebus+ product suppliers and users and by independent test labs. Consistent use of this standard should promote comparability and, therefore, mutual recognition of test results produced by different testers.

Keywords: application environment profile (AEP), BICS pro forma, BIXIT pro forma, block copy, implementation under test (IUT), interactive BICS

Institute of Electrical and Electronics Engineers, Inc.

345 East 47th Street, New York, NY 10017-2394, USA

Copyright 1994 by the Institute of Electrical and Electronics Engineers, Inc.

All right reserved. Published 1994. Printed in the United States of America.

ISBN 1-55937-374-1

No part of this publication may be reproduced in any form, in an electronic retrieval system or otherwise, without the prior written permission of the publisher, with the exception of the BICS pro forma, which may be freely reproduced by users of this standard so that they may be used for their intended purpose.

Permission is further granted for the completed BICS to be published.

IEEE Standards documents are developed within the Technical Committees of the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Board. Members of the committees serve voluntarily and without compensation. They are not necessarily members of the Institute. The standards developed within IEEE represent a consensus of the broad expertise on the subject within the Institute as well as those activities outside of IEEE that have expressed an interest in participating in the development of the standard.

Use of an IEEE Standard is wholly voluntary. The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least every five years for revision or reaffirmation. When a document is more than five years old and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

Comments for revision of IEEE Standards are welcome from any interested party, regardless of membership affiliation with IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of all concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason IEEE and the members of its technical committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

Comments on standards and requests for interpretations should be addressed to:

Secretary, IEEE Standards Board
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331
USA

IEEE standards documents may involve the use of patented technology. Their approval by the Institute of Electrical and Electronics Engineers does not mean that using such technology for the purpose of conforming to such standards is authorized by the patent owner. It is the obligation of the user of such technology to obtain all necessary permissions.

Introduction

(This introduction is not a part of IEEE Std 896.4-1993, IEEE Standard for Conformance Test Requirements for Futurebus+.)

The IEEE 896.4 Working Group would like to gratefully acknowledge the following persons for their roles in developing this standard, and to thank them for their contributions.

Joel Witt, Chair
Steve Cecil, Chair
Michael Roby, Vice Chair
Kim Burris, Technical Editor

Daniel Doades
Samuel Duncan
Karl Franklin
Joe Jachim
Edgar Jacques

Jeffrey Lear
Karl McClure
Thanos Mentzelopoulos
Thai Nguyen
Joseph Schneider

Tim Scott
Patricia Smith
Joanne Spiller
Scott Thomas
Karl Wieber

The following persons were on the balloting committee:

Amir Abouelnaga
Edward W. Aichinger
Scott Akers
Ray S. Alderman
John Allen
Richard P. Ames
Keith D. Anthony
Chris Bezirtoglou
John Black
Paul Borrill
Charles Brill
Kim Burris
Clyde Camp
Steve Cecil
Andy Cheese
Mark Chestnutwood
Steven Cobb
David Cohen
Steven R. Corbesero
Dante Del Corso
Ian Dobson

Jean-Jacques Dumont
Samuel Duncan
Wilhelm P. Evertz
Wayne Fischer
Gordon Force
Karl Franklin
Paul Fulton
Julio Gonzalez-Sanz
Michael c. Hayward
Joe Jachim
Edgar Jacques
Ernst H. Kristiansen
Lak Ming Lam
Michael Lambrou
Karl McClure
Thanos Mentzelopoulos
Brian D. Morrison
Klaus Dieter Mueller
Thai Nguyen
Steve Quinton
William Ramsay

Michael Roby
Frederick E. Sauer
Donald Senzig
Patricia Smith
Joanne Spiller
Scott Thomas
Michael G. Thompson
Joseph P. Trainor
Robert Tripi
Yoshiaki Wakimura
Eike Waltz
Dave Wickliff
Robert Widlicka
Karl Wieber
Joel Witt
Mark Woodbury
David L. Wright
Yoshio Yamaguchi
Oren Yuen
Janusz Zalewski

When the IEEE Standards Board approved this standard on September 15, 1993, it had the following membership:

Wallace S. Read, Chair
Donald C. Loughry, Vice Chair
Andrew G. Salem, Secretary

Gilles A. Baril
José A. Berrios de la Paz
Clyde R. Camp
Donald C. Fleckenstein
Jay Forster*
David F. Franklin

Ramiro Garcia
Donald N. Heirman
Jim Isaak
Ben C. Johnson
Walter J. Karplus
Lorraine C. Kevra

E.G. "Al" Kiener
Ivor N. Knight
Joseph L. Koepfinger*
D. N. "Jim" Logothetis
Don T Michael*
Marco W. Migliaro

L. John Rankine
Arthur K. Reilly

Ronald H. Reimer
Gary S. Robinson

Leonard L. Tripp
Donald W. Zipse

*Member Emeritus

Also included are the following nonvoting IEEE Standards Board liaisons:

Satish K. Aggarwal
James Beall

Richard B. Engelman
David E. Soffrin

Stanley I. Warshaw

Rachel A. Meisel
IEEE Standards Project Editor

CLAUSE	PAGE
1. Overview.....	1
1.1 Scope.....	1
1.2 Background.....	2
1.3 Terminology.....	5
2. References.....	6
3. Definitions.....	6
3.90 3.1 Definitions for bus lines and signals.....	10
Figure 2—3.2 Futurebus+ logo.....	11
4. Considerations for implementing a conformance test program	12
4.1 Physical tester	12
4.2 Executable test suite.....	14
4.3 Application of conformance tests	16
4.4 Limitations of the conformance test process.....	17
4.5 Test case format	18
4.6 Test case numbering scheme.....	19
5. Test cases	23
5.1 Parallel protocol.....	23
5.2 Distributed arbitration	58
5.3 Central arbitration	67
5.4 Message passing.....	68
5.5 CSR, bus/system management	71
6. Monitors	77
6.1 Parallel protocol	77
6.2 Distributed arbitration	93
6.3 Central arbitration	96
6.4 Bus system management monitors.....	110
7. General definitions for header file	110
8. Test case functions and prototypes	111
8.1 Function prototypes.....	111
8.2 Test case prototypes	111
8.3 C functions	111
9. BICS header file.....	111
10. Electrical test cases	111
10.1 Backplane signal skew	111
10.2 Module signal skew	115

CLAUSE	PAGE
Annex A (Normative) BICS pro forma entry-to-test case mapping information	120
Annex B (Normative) BICS pro forma.....	128
Annex C (Normative) Arbitration waveforms	157
Annex D (Normative) Parallel protocol waveforms	223
Annex E (Normative) Bus/system management and CSR waveforms	263

IEEE Standard for Conformance Test Requirements for Futurebus+

1. Overview

1.1 Scope

This standard defines tester-independent conformance test procedures for Futurebus+. Specifically, this standard includes test procedures for verifying a module's conformance to the requirements of ISO/IEC 10857 : 1994¹ and sections 3 and 4 of IEEE Std 896.2-1991. This standard does not include test procedures for physical layer or performance-related requirements defined by the profile specification sections of IEEE Std 896.2-1991 or of any other Futurebus+ standards.

Because the requirements contained in the non-profile sections of ISO/IEC 10857 : 1994 and IEEE Std 896.2-1991 are for the logical layer, this is primarily a logical layer test standard. Section 9 contains tests for those electrical requirements that are: a) part of ISO/IEC 10857 : 1994 and b) not profile specific. Since the majority of the requirements for the bus signaling environment are contained in profiles, the tests in section 9 of this standard are not, nor are they intended to be, a complete set of electrical tests for any Futurebus+ profile.

Similarly, section 4 of IEEE Std 896.2-1991 contains mainly description and points to profile for the requirements governing the implementation of live insertion. Since this is the case, no live insertion tests are included in this standard.

Readers interested in tests for these requirements should look to ongoing standardization efforts for Futurebus+ electrical and live insertion requirements and for Futurebus+ profiles.

This standard is intended for use in verifying the conformance of Futurebus+ modules to the Futurebus+ family of standards. Some of the information may be useful in performing production tests and tests that measure a module's level of performance, but these types of testing are not within the scope of this standard.

This standard is intended for the use of Futurebus+ manufacturers, users, and third-party test agencies. Use of a single-test standard should allow different testers to arrive at similar test results.

¹Information on references can be found in clause 2.

1.2 Background

This standard consists mainly of ANSI C program segments, including test case functions, data structure definitions, and function prototypes for the tester and module dependent functions a test facility must develop. The primary motivation in writing the test procedures in ANSI C was to provide a concise and unambiguous test definition. The C code included in this standard was written to be easily understandable. It has not been optimized for speed or for efficient use of memory. Those with even the slightest exposure to C should find the test procedures easily understandable.

While C was used to document the test procedures, test facilities are not required to use C when writing their actual executable test suites. Even a test facility that does choose to use C when writing its executable test suites may choose to modify the test cases to increase execution efficiencies or to provide a more seamless interface to its test environment. The procedures contained in this standard must not be modified, but the code may be changed as necessary to meet a test facility's needs.

The test procedures are very well defined, but at the same time, tester independent. This was accomplished by relegating tester-specific interactions to low-level functions. These functions as well as several high-level functions shall be written by the test facility. This standard includes detailed descriptions and function prototypes for most of these functions.

Figure 1 is an overview of the code segments contained or defined in this standard. The shaded blocks represent program segments each test facility must create.

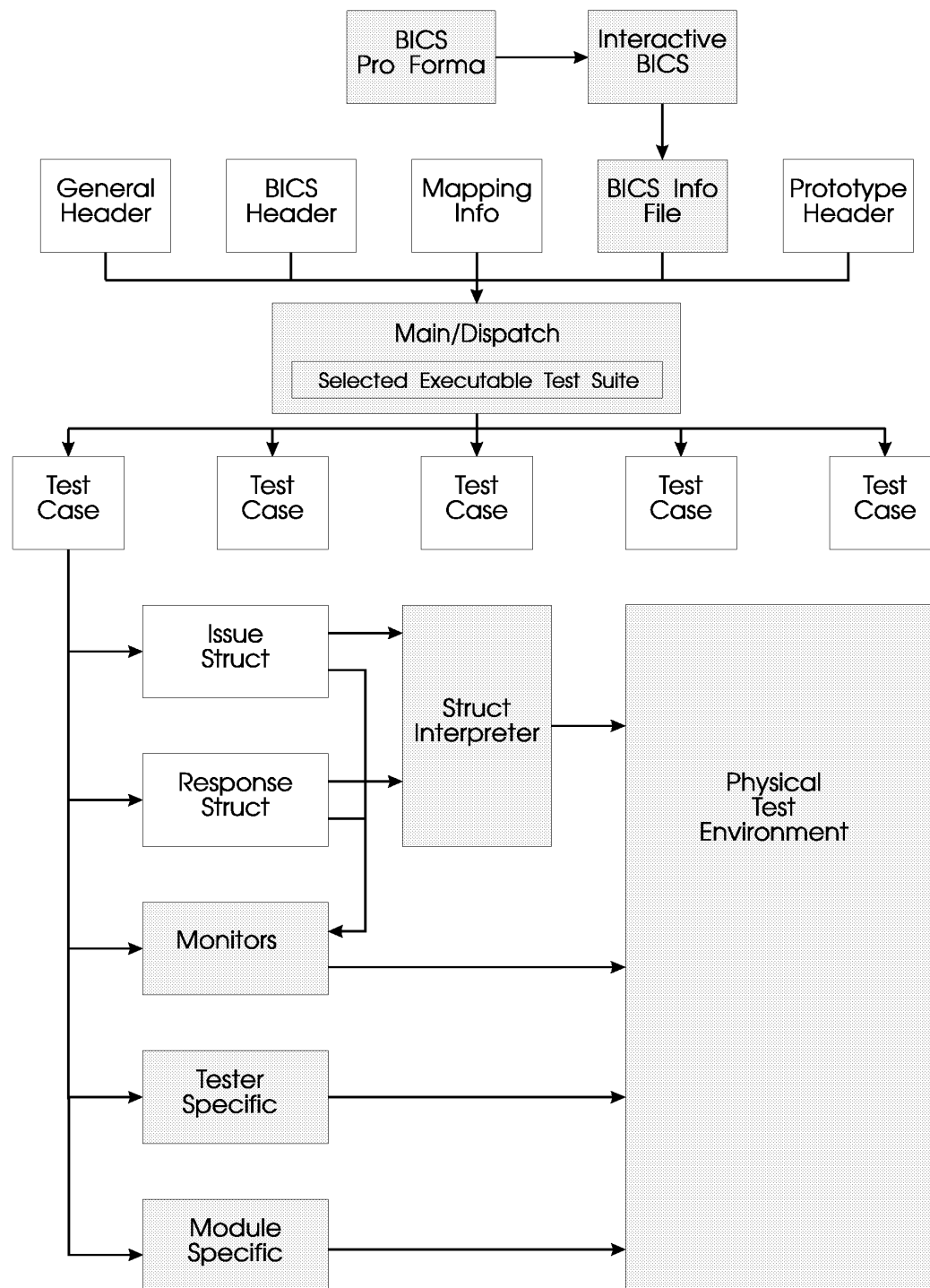


Figure 1—Test suite overview

Each of the components depicted in figure 1 are described as follows:

- a) *BICS Pro Forma*. The Bus Information Conformance Statement (BICS) Pro Forma is a questionnaire that allows a board manufacturer to very precisely describe the board to be tested in terms of the Futurebus+ features and options it includes. Each item in the BICS Pro Forma has associated with it a specific set of requirements that shall be tested in order to verify a module's conformance to the Futurebus+ standards. When used in conjunction with the BICS Pro Forma, this mapping information can be used to automatically custom tailor a test plan to any Futurebus+ product. A sample BICS Pro Forma is included in annex B. A test facility may devise some other means of determining the exact set of tests that shall be performed for a given module.
- b) *Interactive BICS*. This is an electronic, interactive form of the BICS Pro Forma. The Main/Dispatch function must have access to the information contained in the completed BICS Pro Forma if it is to automatically select the appropriate set of tests for a given module. The optional Interactive BICS places this information into the BICS Info File. A test facility may devise some other means of allowing the Main/Dispatch function to perform the appropriate set of tests for a given module. The result of the Interactive BICS program should be the assignment of values to the variables in the BICS structure, similar to the result of calling the `set_bics` function contained in the file 'example/set_bics.c' from the main/dispatch program.
- c) *BICS Info File*. This file contains the information from the completed BICS Pro Forma. As depicted, the file is created by the Interactive BICS, but some other means may be devised. The information in this file is used to initialize the variables contained in the BICS Header file.
- d) *General Header File*. This file contains various definitions used by the test cases. It is completely defined in ANSI C. This file can be found on one of the accompanying diskettes as 'headers/gendefs.h'.
- e) *BICS Header File*. This file contains the BICS struct, which is used by the test cases to determine which subtests shall be performed to test a given module. The struct may also be used by the Main/Dispatch function to determine which tests shall be performed to test a given module. The variables within the struct shall be initialized before any test case functions are called. As depicted, the Main/Dispatch function (or some other test-facility-defined function) would initialize the variables based on the contents of the BICS Info File. Some other means may be devised to initialize the struct. This file is completely defined in ANSI C. This file can be found on one of the accompanying diskettes as 'headers/bics.h'. The test lab shall initialize the structure contained in this file for each implementation being tested prior to selecting and running the executable tests.
- f) *Mapping Info File*. This file contains BICS Pro Forma entry-to-test case mapping information. The Main/Dispatch function uses this information to determine which tests shall be run for each given implemented feature. The Mapping Info File is used to generate the Selected Executable Test Suite File, a part of the Main/Dispatch function. Test labs should use annex A as a guide to developing the Mapping Info File.
- g) *Selected Executable Test Suite File*. This file contains the suite of tests to be run against an implementation and can be created by using the BICS Info File of the implementation and the BICS Reference to Test Case Mapping in annex A to select the test cases to run on the implementation under test. In the example on the diskettes there is no selected executable test suite file, because the selected executable test suite is embedded in the main/dispatch function. For this example the test suite was selected by hand.
- h) *Prototype Header File*. This file contains the function prototypes for the test-facility-defined functions. Along with each prototype is a description of the necessary actions the function shall take. There will be prototypes for functions defined by this standard, such as the test case prototypes defined in the file 'headers/tc_proto.h' and this standard's function prototypes defined in the file 'headers/funcprot.h'. There will also be some test-lab-defined prototype header files, such as 'examples/evalprot.h'. This file is completely defined in ANSI C.
- i) *Main/Dispatch Function*. This function is responsible for controlling the overall test flow. Each test case is a function of type null. The Main/Dispatch Function shall call each of the appropriate test functions in some test-facility-defined sequence. The function will typically determine which test functions to call based on the content of variables in the BICS Header File, but some other means may be used. Most of the test case functions use variables defined in the BICS Header File, therefore the Main/Dispatch Function shall initialize them before it calls any test functions. The `set_bics.c` does the initialization of the BICS structure. In the example main program given with the test suite, the test cases are chosen by hand, but a test lab could (and probably should) write a program to automatically select tests based upon the BICS Info File and the BICS Reference to Test Case Mapping in annex A. A Main/Dispatch function, called "example/testcase.c" is

included on the diskettes accompanying this standard, but this is only an example main function, and test labs are encouraged to develop their own.

- j) *Test Case*. Each test case includes a high-level description of the test procedure and the requirements to be verified, the actual test procedure (written in ANSI C), and a definition of the behavior necessary for the implementation to conform to the requirement. The test case functions interface to the test environment through the Issue and Response structs and through several tester and module specific functions. There are numerous test case files. For specific file names and an indication of test coverage of a specific file, refer to section 4. Each of the files under the 'testcase' directory on each diskette is a test case file.
- k) *Issue and Response Structs*. These structs are defined in the General Header file and act as the primary interface between the Test Case Functions and the Physical Test Environment. These structs contain tester transaction definitions and expected module response definitions. They are initialized by each test case function before the actual test transactions are initiated.
- l) *Struct Interpreter*. This function reads the Issue and Response Structs and configures the tester to perform a transaction using the desired characteristics. This function has no definition other than the fact that it shall be capable of reading the Issue and Response Structs.
- m) *Monitors*. The monitors perform most of the actual low-level timing and protocol verification. They may be thought of as rule-based logic analyzers that observe bus activity and compare the observed behavior with expected behavior. The observed behavior is obtained by watching the bus signals. The expected behavior is obtained from the Issue and Response Structs. This standard includes the definition of two types of monitors: timing and protocol. The timing monitor verifies characteristics such as set up and hold times of information lines relative to synchronization lines. The protocol monitor verifies characteristics such as the sequencing of synchronization lines. This standard does not include actual C code for the monitors. Instead, it defines the set of rules on which executable monitors should be based.
- n) *Tester Specific Functions*. Since this standard was written to be tester independent, each test facility shall write its own functions to interface between the C code contained in this standard and its specific test environment. Function prototypes are provided in the Prototype Header File for those Tester Specific Functions that are called by the Test Case Functions.
- o) *Module Specific Functions*. To test a module, it shall be caused to generate and respond to the bus transactions it supports. The Module Specific Functions are used to elicit such behaviors. Function prototypes are provided in the Prototype Header File for those Module Specific Functions that are called by the Test Case Functions.
- p) *Physical Test Environment*. The physical test environment is the combination of hardware and software used to test the implementation. In the example given on the diskettes, the file 'examples\eval.c' contains some of the software that would be a part of a legitimate physical test environment. The functions found in 'examples\eval.c' print messages to a file opened by the Main/Dispatch Function. These messages indicate the progress of each test case called by the main function, and were used by working group members to ensure that this standard's test suite compiles properly and executes as intended. These routines are examples only and should in no way be considered a complete test environment for a Futurebus+ implementation.

1.3 Terminology

The word *may* indicates flexibility of choice with no implied preference.

The word *shall* indicates a mandatory requirement. Designers shall implement all such mandatory requirements to ensure interoperability with other ISO/IEC 10857 : 1994 conformant products and claim conformance with this standard.

The word *should* indicates flexibility of choice with a strongly preferred implementation. The phrase *it is recommended* is used interchangeably with the word *should*.

2. References

The following publications should, when applicable, be used in conjunction with this standard:

IEEE Std 896.2-1991, IEEE Standard for Futurebus+—Physical Layer and Profile Specification.²

IEEE P896.3/D4.2, February 1993, IEEE Recommended Practices for Futurebus+.³

IEEE Std 1212-1991, IEEE Standard Control and Status Register (CSR) Architecture for Microcomputer Buses (ANSI).

ISO/IEC 10857:1994 [ANSI/IEEE Std 896.1, 1994 Edition], Information technology—Microprocessor systems—Futurebus+—Logical protocol specification.⁴

Kernighan, Brian W. and Dennis M. Ritchie, *The C Programming Language*, 2nd Ed. Englewood Cliffs, NJ: Prentice Hall, 1988.

3. Definitions

3.1 address only transaction: A bus transaction that does not include a data phase. The only information transferred is contained within the connection phase and, in some cases, the disconnection phase.

3.2 aligned address: This is an integer multiple of the data block size. The maximum data block size that can be transferred by an implementation under test (IUT) Master is the product of data width and data length.

3.3 application environment profile (aep): A document that describes functional requirements and points to existing standards, selecting and binding options within those standards. An implementer who then designs a specific module and/or system should be reasonably assured that another designer's (manufacturer's or supplier's) modules will properly function within the same system. This includes all aspects of definition: mechanical, electrical, protocol, environmental, and system considerations.

3.4 arbitration: The process of selecting the next bus master.

3.5 arbiter: The module that is performing the arbitration.

3.6 arbitrated message: An event number broadcast on the arbitrated message bus lines to all modules on the bus.

3.7 backplane: An electronic circuit board and connectors used to interconnect modules together electrically. The backplane connects selected pins of the connectors, thus providing the medium for the transfer of signals needed for the operation of the bus.

3.8 backplane bus: A means to connect circuit modules using common signal traces on a backplane and a standard set of rules.

3.9 beat: An event that begins with the transition on a synchronization line by the master followed by the release of an acknowledge line by one or more slaves. Command and data information may be transferred from the master to one or more slaves in the first half of the beat. During the second half of the beat the slaves may transfer capability, status, and data information back to the master.

3.10 block copy: This operation is characterized by a long series of read or write transactions to sequential memory locations.

²IEEE publications are available from the Institute of Electrical and Electronics Engineers, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, US.

³At the time this document went to press, IEEE Std 896.3-1993 was not yet published. It is, however, available in manuscript form from the IEEE.

⁴This standard is available from the IEC, Case Postale 131, 1 rue de Varembe, CH-1211, Genève 20, Switzerland/Suisse and from the IEEE.

3.11 bus bridge: This is an interconnect between two or more buses that provides signal and protocol translation from one bus to another. The buses may adhere to different bus standards for mechanical, electrical, and logical operation (such as a bus bridge from Futurebus+ to VMEbus or to Multibus II).

3.12 bus-dependent: This term is used to describe technology-dependent parameters. Although the CSR Architecture may specify the size and address of these parameters, their format and definition is provided by the appropriate bus standards.

3.13 bus implementation conformance statement (BICS): This is a completed BICS pro forma questionnaire.

3.14 bus implementation conformance statement pro forma (BICS pro forma): A questionnaire that lists implementation capabilities.

3.15 bus implementation extra information for testing (BIXIT): A completed BIXIT pro forma.

3.16 bus implementation extra information for testing pro forma (BIXIT pro forma): This questionnaire provides extra information about the module that might be necessary to configure and perform the tests. The BIXIT for this standard is located at the end of the BICS pro forma in annex B.

3.17 bus line: This is the medium for the transmission of signals. Since Futurebus+ requires drivers with wire-OR capability, a bus line may be driven by several modules simultaneously. Therefore, the signal carried by the bus line is the combination of signals applied to that line from each module.

3.18 bus tenure: The duration of a master's control of the bus; i.e., the time during which a module has the right to initiate and execute bus transactions.

3.19 bus transaction: An event initiated with a connection phase and terminated with a disconnection phase. Data may or may not be transferred during a bus transaction.

3.20 busy: If a slave is unable to accept a bus transaction from a master, it may issue a Busy status to the master of the transaction. The master must relinquish the bus and may reacquire the bus and retry the transaction after a suitable time interval.

3.21 byte: A set of eight adjacent binary digits.

3.22 byte lane: A data path formed by eight data lines and one parity line used to carry a single byte among the system modules.

3.23 cache agent: A module that uses split transactions to assume all the rights and responsibilities of some number of remote cache modules.

3.24 cache coherence: A system of caches is said to be coherent with respect to a cache line if each cache and main memory in the coherence domain Observes all modifications of that same cache line. A modification is said to be observed by a cache when any subsequent read would return the newly written value.

3.25 cache memory: A buffer memory inserted between one or more processors and the bus, used to hold currently active copies of blocks from main memory. Cache memories exploit spatial locality by what is brought into a cache. Temporal locality is exploited by the strategy employed for determining what is removed from the cache.

3.26 central arbiter requester: A module that requests access to the bus on a system that uses central arbitration.

3.27 coherence domain: A region in a coherence domain is a region in a multiple-cache system, inside of which cache consistency measures are enforced. In a system that contains bus bridges, a coherence domain may or may not be extensible beyond the local bus through a bus bridge to remote buses.

3.28 coherence line: A data block for which cache consistency attributes are maintained.

3.29 compelled data transfer protocol: A technology-independent transfer mechanism in which the slave is compelled to provide a response before the master proceeds to the next transfer.

3.30 competitor: A module actively participating in the current control acquisition cycle of the arbitration process.

3.31 connected transaction: A transaction in which both the request and response are performed within the same bus transaction.

3.32 connection phase: A beat that begins with the assertion of the address synchronization line followed by the release of an address acknowledge line. It is used to broadcast the address and command information. Modules determine whether they wish to take part in the transaction based on this information.

3.33 copyback cache: A cache memory scheme with the attribute that data written from the processor is normally written to the cache rather than the main memory. Modified data in the cache is written to the main memory to avoid loss of the data when a cache line flush or replacement occurs.

3.34 CSR: Control and Status Register. A quadlet register that is accessed through read4 or write4 transactions and is used to observe a node's state or to control its operation.

3.35 CSRA: Control and Status Register Architecture (see IEEE Std 1212-1991).

3.36 control acquisition: The total of all bus activity associated with acquiring exclusive control of the bus.

3.37 data phase: A period within a transaction used to transfer data.

3.38 deadlock: This occurs when modules are awaiting actions that can only be performed by those waiting, and those waiting cannot perform actions.

3.39 direct memory access (DMA): This refers to the ability of I/O controller modules to independently access memory. An I/O controller with DMA capabilities can access commands, fetch data, and report status by accessing memory directly.

3.40 disconnection phase: A period within a transaction used to return the bus signals to their quiescent state. In addition, this phase might be used to transfer additional information required to perform or abort the requested operation.

3.41 doublet: A set of two adjacent bytes.

3.42 emperor processor: The monarch processor selected to direct the configuration and initialization of an entire system with multiple interconnected logical buses.

3.43 entrant: A live inserted module in the process of aligning itself with the arbitration protocol.

3.44 exclusive modified (EM): An attribute assigned to a cache line if there is an up-to-date copy of the line in the module's cache and the module has the only valid copy in the system.

3.45 exclusive unmodified (EU): An attribute assigned to a cache line if there is an up-to-date copy of the line in the module's cache and the module is to assume that no other copies of the line are valid in any other cache in the system.

3.46 forward progress: This occurs when a module is not blocked from performing the tasks necessary to achieve its goal. Forward progress is guaranteed only in the absence of deadlock or starvation.

3.47 geographical address: A unique identifier assigned to each physical module slot on the bus and assumed by any module connected to that slot.

3.48 glitch filter: As explained in ISO/IEC 10857 : 1994, this filters out a fundamental transmission line effect found in bussed backplane implementations. The effect is commonly called the *wire-OR glitch*.

3.49 global identification: A unique identifier assigned to each physical module slot in a system. This identifier would typically include both a bus identifier and a slot identifier. IEEE Std 1212-1991 specifies the format for such a global identifier.

3.50 implementation under test: The Futurebus+ module that is being tested for conformance.

3.51 intervening slave: The participating slave that, although not the repository of last resort of the requested data, finds it necessary to prevent the repository of last resort from providing the requested data. Having done so, the intervening slave provides the data instead.

3.52 invalid (I): An attribute assigned to a cache line if there is not an up-to-date copy in the module's cache.

3.53 live insertion: The process of inserting boards into a backplane when power is on.

3.54 livelock: A metastable situation in which some modules acquire and release resources in a way that none of them make forward progress.

3.55 locking: A facility whereby a module is requested to guarantee exclusive access to addressed data, blocking other modules from accessing that data. This allows indivisible operations to be performed on addressed resources.

3.56 master: A module that has acquired control of the bus through the control acquisition procedure.

3.57 master elect: A module that has won the most recent arbitration competition.

3.58 memory agent: A module that uses split transactions to assume all the rights and responsibilities of some number of remote memory modules.

3.59 memory boundary: The last address of an aligned data block. The maximum data block size that can be transferred by an IUT Master is the product of data width and data length.

3.60 module: A collection of circuitry designed to perform specific functions that includes an interface to Futurebus+.

3.61 monarch processor: The monarch processor, or simply the monarch, is the processor selected to manage the configuration and initialization of all modules on one logical bus.

3.62 node: A set of control and status register addresses (including identification-ROM and reset command registers) that are initially defined in a 4 kb (minimum) initial node address space. Each node can be reset independently (a reset of one node has no effect on other nodes).

3.63 octlet: A set of eight adjacent bytes.

3.64 packet data transfer protocol: A very fast but technology-dependent noncompelled transfer mechanism that uses a compelled protocol over the entire packet to provide flow control.

3.65 parallel contention arbitration: A process whereby modules assert their unique arbitration number on a parallel bus and release signals according to an algorithm such that after a period of time the winner's number appears on the bus.

3.66 participating slave: A slave involved in a transaction as a selected slave, an intervening slave, a broadcast slave, or a slave involved in multiple packet mode.

3.67 potential master: A module capable of acquiring control of the bus through the control acquisition process.

3.68 preemption: The release of the bus by the current bus master due to the request of another module. Note that in some systems, preemption occurs when the current bus master relinquishes the bus because another module has requested it. In some systems, any module may cause preemption; in some systems, only a module with a higher priority request may cause preemption.

3.69 quadlet: A set of four adjacent bytes.

3.70 request: A command, generated by a requester, to initiate an action on a responder. For a processor-to-memory read transaction, for example, the request transfers the memory address and command from the processor to memory. In the case of a split transaction, the request would be a separate bus transaction. In the case of a connected transaction, the request would be the connection phase of a bus transaction.

3.71 requester: A module that initiates a transaction by sending a request (containing address, command, and sometimes data) is referred to as the requester.

3.72 responder: A module that completes a transaction by sending a response (containing the completion status and sometimes data) is referred to as the responder.

3.73 response: A reply generated by a responder, to complete a transaction initiated by a requester. For a processor-to-memory read transaction, for example, the response returns the data and status from the memory to the processor. In the case of a split transaction, the response would be a separate bus transaction. In the case of a connected transaction, the response would be the data and disconnection phases of a bus transaction.

3.74 round robin: A bus allocation rule where, after a module acquires and uses the bus, it will not be granted use of the bus again until all other modules currently requesting the bus at the same priority level have had control of the bus.

3.75 selected slave: A slave that is selected during the connection phase by the master when it recognizes its address on the bus lines.

3.76 shared memory: The address space in the system accessible to all caching modules.

3.77 shared unmodified (SU): An attribute assigned to a cache line if there is an up-to-date copy of the line in the module's cache and the module is to assume that a copy also exists in another module's cache.

3.78 skew: The difference between the propagation delays of two or more signals on any bus lines.

3.79 slave: A module that can be addressed and is capable of participating in bus transactions.

3.80 snarf: A module is said to snarf a transaction if it takes a copy of data passing by on the bus even though it did not request it.

3.81 snoop: A module is said to snoop a transaction if it is not the master that originated the transaction or the repository of last resort for the data, but it still monitors the transaction. Cache memories snoop transactions to maintain coherence.

3.82 split acceptor: A master module that splits the current transaction upon detecting SR* asserted.

3.83 split initiator: A slave module that asserts sr* to require that the current transaction be split.

3.84 split transaction: An operation in which the request is transmitted in one bus transaction and the response is transmitted in a separate subsequent bus transaction.

3.85 transaction bystander: A module that is not participating in the current transaction. A transaction bystander monitors, asserts, and releases the synchronization signals, even though it is not the initiator of a transaction or the target of it.

3.86 transfer line size: The size of the block of data transferred to or from main memory in a caching environment.

3.87 unaligned address: An unaligned address is a noninteger multiple of the data block size. The maximum data block size that can be transferred by an IUT Master is the product of data width and data length.

3.88 unit: A unit is a logical component of a node that is accessed by I/O driver software. After the node is initialized and configured, the units normally operate independently. Note that one node could have multiple units (for example, processor, memory, and SCSI controller).

3.89 unselected slave: A slave that does not recognize its address on the bus lines during the connection phase of a bus transaction.

3.90 vendor-defined: This is used to describe parameters that identify and particularize a specific vendor implementation. Although the CSR Architecture specified the size, location, and a few values within these fields, their remaining values are provided by the module vendor. *Syn:vendor-dependent.*

3.1 Definitions for bus lines and signals

3.1.1 activate: The action of applying a set of signals to a group of bus lines. Similarly, the term **activated** is used to describe the state of a group of bus lines when they carry signals.

3.1.2 *: When appended to signal's name, the suffix “*” (asterisk) indicates that the logic one state of the signal is such that it will override the logic zero state applied by any other module on that line.

3.1.3 assert: The action of applying a logic one signal to a bus line. Similarly, the term **asserted** is used to describe the state of a bus line when the signal it carries represents a logic one.

3.1.4 bus line: The medium for the transmission of signals. Since Futurebus+ requires drivers with wire-OR capability, a bus line may be driven by several modules simultaneously. Therefore, the signal carried by the bus line is the combination of signals applied to that line from each module.

3.1.5 release: The action of applying a logic zero signal to a bus line. Similarly, the term **released** is used to describe the state of a bus line when the signal it carries represents a logic zero.

3.1.6 signal names: Where a group of bus lines are represented by the same letters, the lines within the group are numbered, e.g., **AD0***, **AD1***, **AD2*** etc. In order to represent a group of lines or signals in more convenient form, the notation **AD[63..0]*** is used. Also, the notation **AD[]*** is used to refer to all of the lines within the group.

Figure 2 shows an example of the signal conventions used in ISO/IEC 10857 : 1994. The example is shown with open collector technology (although this specification is independent of this logic family). The signal that a specific module applies to the input of its driver is designated by a lowercase label; i.e., **ai**. The signal that a specific module applies to a bus line is designated by a lowercase label with an asterisk; i.e., **ai***. The signal that appears on a bus line as the result of the combined signals applied to it by all modules is designated by an uppercase label with an asterisk; e.g., **AI***.

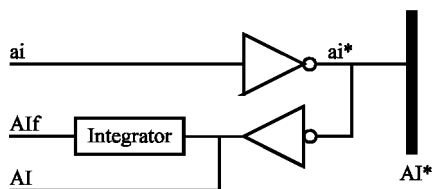


Figure 2—Signal naming convention

When appended to a signal name, the suffix “f” (filtered) refers to the bus line signal after it has passed through a receiver and a wire-OR glitch filter (integrator). For example, **AIf** refers to the signal on the **AI*** line, after it has passed through an inverting receiver to become **AI** and a wire-OR glitch filter to become **AIf**.

In a table with an appended asterisk; i.e., **WR***, a “0” would mean the signal was released, a “1” would mean an asserted signal.

3.2 Futurebus+ logo

The IEEE 896 logo shown in figure 3 is copyrighted to the IEEE Computer Society, and may be used only on products that fully conform to all mandatory requirements of an IEEE-approved 896 profile document, or adhere to either ISO/IEC 10857 : 1994 or IEEE Std 896.1-1987.



Figure 3—IEEE 896 logo

4. Considerations for implementing a conformance test program

4.1 Physical tester

This standard is nontester-specific in that the tests it contains can be carded out on many different testers. The tester depicted in figure 4 includes the logical functions necessary to implement the tests. The tester model was developed to aid in defining the test procedures and is included to help the reader to understand them, but the logical tester is not intended to dictate how a physical tester must be implemented.

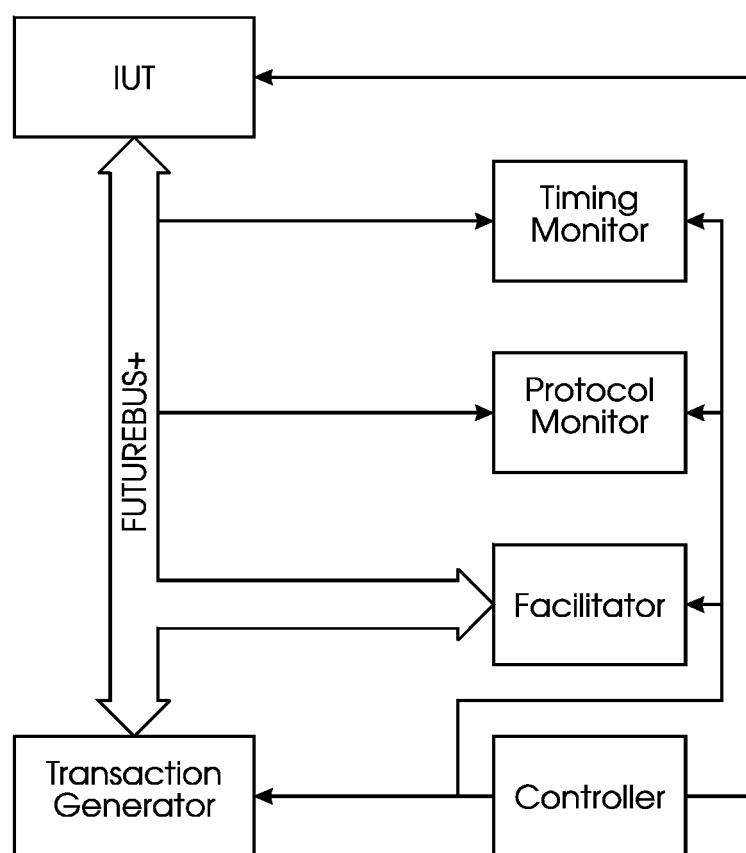


Figure 4—Logical tester

4.1.1 Logical tester model

The logical tester depicted in figure 4 consists of the following subsystems:

- a) *Transaction generator.* The transaction generator initiates and responds to bus transactions. During the testing process, the transaction generator initiates transactions targeting the IUT and responds to transactions initiated by it. During tests that require the IUT to be a nonparticipating bystander, the transaction generator initiates and responds to its own transactions. For such tests, multiple transaction generators may be used, one to initiate the transaction and one or more to respond to it. All test results are derived from the observed interaction between the transaction generator(s) and the IUT or from the effects of it.

The transaction generator is capable of initiating and responding with all possible variations in command, capability, and status lines. It can be programmed to cause various error conditions to occur and to signal that it has detected error conditions when none have occurred. The transaction generator can also be programmed to vary the time relationships between signals to test interlocked handshake relationships and timing requirements.

- b) *Timing monitor.* The timing monitor is a passive device, incapable of initiating, responding to, or participating in bus transactions. It measures the time relationships between the Futurebus+ synchronization lines and associated information lines and then determines whether or not the measured values fall within the allowed ranges.
- c) *Protocol monitor.* Like the timing monitor, the protocol monitor is totally passive. It monitors the sequence in which the synchronization lines are asserted and released relative to each other and determines whether or not the observed sequence matches the sequence specified for the specific set of circumstances (e.g., error detection, split request, intervention, etc.).
- d) *Facilitator.* The facilitator is used to initialize the IUT's memory and CSRs before test transactions begin. It is also used to read the IUT's memory and CSRs after the test transactions have been completed to determine whether or not events that cannot be observed from the bus have occurred. It is used extensively for verifying proper operation of the error-related CSRs and for verifying that the IUT captures the data written to it. Since it must interact with the IUT, the facilitator fully supports the Futurebus+ protocols as a master. In some physical tester implementations, the Facilitator functions may be carried out by the Transaction Generator.
- e) *Controller.* The controller uses the other modules in the tester to perform tests using the test procedures defined in this standard. The controller has no direct connection to the tester Futurebus+ backplane, so all communication with the IUT is done indirectly through the facilitator or through another access port on the IUT.

4.1.2 Issues in implementing a physical tester

Defining the implementation of a physical tester is counter to the abstract nature of this standard. However, several issues that shall be considered when implementing a physical tester have been identified.

4.1.2.1 Real-time vs. posttransaction observation

Characteristics like signal timing can only be observed in real time while a test transaction is in progress, because after the transaction has been completed, no artifacts from which timing information can be deduced exist. Other characteristics, like how a module records error-symptoms, can only be observed after a transaction has been completed, by reading its CSRs. Some requirements can be verified using either real-time or posttransaction observation. For example, to verify that a module drives the appropriate data lines with valid data during write cycles, a monitor could capture the state of each of the data lines as the write cycle progresses. Alternatively, a facilitator could read the contents of the targeted memory board. If the correct data pattern exists in the memory, it can be deduced that the module drove valid data on the correct set of data lines.

Each approach has its advantages and disadvantages. Real-time observation requires more complicated hardware but it results in a true measure of the subject characteristic. Posttransaction observation requires less complicated hardware but the resulting data may be corrupted by other failures.

Continuing with the previous example of valid write data during write cycles, suppose the module being tested violated an interlocked handshake requirement, causing the targeted slave to fail to capture the data properly. The module would pass the test if real-time observation was used but it would fail the test if posttransaction observation was used. Use of posttransaction observation may result in reported failures where no failure exists. If the goal of the physical tester is to provide a pass or fail result, the use of posttransaction observation can simplify the tester. However, if the goal of the physical tester is to provide an accurate report of the specific requirements violated, real-time observation shall be used. Either method may be used with the test procedures in this standard. It is recommended where a choice exists, the real-time method of observation be used whenever practical, due to the potential for error in the posttransaction method.

4.1.2.2 Data capture using passive monitors

The Futurebus+ protocols allow a module to slow down each transaction by delaying the assertion or release of synchronization lines. Since the timing and protocol monitors described in are totally passive, they must be capable of capturing the information on the bus signals without stretching the valid signal time. This means that to be guaranteed that a monitor will function properly under all conditions, it must be designed to capture data at greater than the theoretical maximum Futurebus+ signal rate. Creating such a monitor is impractical, but as advances in technology allow Futurebus+ modules to signal at higher rates, monitors used to test them must improve in data capture rates.

4.2 Executable test suite

The test procedures contained in this standard consist of a combination of ANSI C code segments and descriptions of physical tester and module specific functions. Before conformance testing can be performed, each test facility must create the overall framework for an executable test suite and incorporate into it the code segments provided in this standard as well as the custom code segments described by this standard. The test case description and test code should be alike. Where the description and test code do not match the code shall take precedence. The following are issues, that must be considered when utilizing the code segments defined in this standard as the basis of an executable test suite.

4.2.1 Exhaustive vs. representative tests

Because of unavoidable limitations in any practical test program, the tests defined in this standard cannot ensure that all incorrect behaviors can be detected for all modules. It is totally impractical to test a Futurebus+ module under every possible combination of variables. Because of the number of transaction types and the number of variations allowed during each, the number of legal combinations is an unfathomable number. Developing an exhaustive set of tests would be prohibitively expensive in terms of time and money, and running the tests after they were developed would likely take from months to years to test a single module.

The more practical approach to testing a Futurebus+ module is to identify a much more modest set of representative tests that can demonstrate to of certainty that the module performs as required. This is the approach taken in most effective method in reducing an exhaustive set of tests down to a representative set of tests is to identify the noninteractive aspects of the Futurebus+ interface and then write tests to test them independently of one another.

For example, the levels on the CM[*] lines probably have no effect on the way a slave responds to a parity error on the AD[*] lines during the connection phase. To test the slave's response exhaustively would require the test to be repeated 336 times (21 transaction types $\times$ 16 combinations of AW*, DW1*, DW0*, and WR*). Representative testing would require the test to be performed only once, using a typical encoding on the CM[*] lines.

Identifying a representative set of tests is not a simple matter. It requires insight into how designers are likely to implement the various features and options of Futurebus+. If the test writer guesses correctly, representative testing is equally as beneficial as exhaustive testing because all possible behaviors are exercised. If he guesses incorrectly, the representative tests will not exercise all possible behaviors, and consequently, incorrect behaviors will likely go undetected.

Exhaustive testing would detect errant behaviors that representative testing could miss, but if adequate care is taken during test development, representative testing can be an acceptable alternative.

4.2.2 Sequence-driven vs. requirement-driven

When creating a suite of conformance tests for a given standard, two distinct approaches may be used to identify test sequences: the sequence-driven approach and the requirement-driven approach.

When using the sequence-driven approach, identify all of the possible sequences such as, 'read unlocked transaction with command parity error in the connection phase' and 'read unlocked transaction with address parity error in the connection phase.' Once the sequences have been identified, one identifies the requirements that may be verified during each sequence. The primary advantage to this approach is that it minimizes the number of actual bus transaction that must be generated and monitored, which in turn, minimizes the actual test time. It also reduces the volume of information needed to define a complete test suite since each sequence is documented only once. This approach has two main disadvantages. The first is that it is difficult to precisely identify how a given requirement is verified. Since a requirement may be partially verified during each of many sequences, one cannot find a complete description of the verification process in a one place. Another disadvantage of this approach is that since many requirements may be verified or partially verified during each sequence, a very slight, seemingly negligible modification to a given test sequence may have the effect of causing some requirements to not be totally verified, or in some cases, to not be verified at all.

When using the requirement-driven approach, one first identifies all of the individual requirements and then identifies one or more sequences that can demonstrate a module's compliance with each requirement. The primary advantage to using this approach is that a complete description of how a requirement is to be verified accompanies the statement of the requirement. The disadvantage to this approach is that the same general test sequence must be run many times, having a profound impact on test time. Since each of the sequences must also be documented many times, the volume of test documentation increases dramatically.

The abstract test suite defined in this Standard uses the sequence-driven approach.

4.2.3 Monitor concept

The test cases in this standard utilize monitors to verify most of the timing and protocol related requirements. As discussed in , monitors are totally passive in that they do not participate in any of the transactions. The monitors 'run in the background' during all interactions between the IUT and the test environment. They include sets of rules that described how Futurebus+ transactions are to be carried out. As the monitors watch the bus signals, they compare the observed behavior to the expected behavior defined by their rule bases and report any apparent violations.

Monitors implicitly verify most of the timing and protocol related requirements during every transaction, therefore, this standard includes very few explicit timing and protocol test cases.

4.2.4 Module initialization

The test case functions included in this standard assume that the module begins from a known, nominal state. For example, the functions assume that no error symptoms bits are set when the function is invoked. If a test case requires a set of conditions to exist before the actual test begins, the test case generates the necessary activity to create that set of conditions. In most cases the test case functions leave the module in a nominal state, but this is not guaranteed. The main/dispatch function written by the test facility must ensure that module is in a nominal state before each test case function is called.

Each test case includes a function call (test_init()) before the actual test procedure begins. This function call is not required to perform any specific tasks. Instead, it is intended as a mechanism by which the test facility may initialize the IUT and test environment.

Most of the test case functions include nested loops that result in a bus transaction being generated several times. If unexpected events occur, such as a bus parity error when none was intentionally created, the module may transition into a state that will cause subsequent loop iterations to fail. A possible solution to this problem is to reset the module between each loop iteration. However, because of the relatively long reset time and the number of tests that must be performed on a given module, doing so would result in unacceptably long test times.

The test facility must examine each reported failure to be certain that it was legitimate and not the result of correct but unexpected events.

4.2.5 Modification of code segments

The code segments contained in this standard were written to be instructive, not to be execution time or code efficient. Consequently, a test facility may want to rewrite the code to make it more efficient or to add special hooks into the test case functions. The test facility may want to translate the test suite from ANSI C code to some other programming language. Regardless of the reason, most test facilities will make changes to the C code. Changing the code is not prohibited as long as the sequence of events and the substantive portions of the procedure are not changed. Since the code segments define conformance, unintentional changes could allow a module to pass a given test at one test facility and fail it at another. This divergence undermines all of the advantages of utilizing standardized conformance test suites.

4.3 Application of conformance tests

A physical tester combined with an executable test suite is nothing more than a tool. To achieve a thorough conformance test, the tool must be applied properly. Following are issues to consider when performing conformance tests.

4.3.1 Testing CSRs

Most Futurebus+ modules include read/write CSRs. Testing control and status registers in general is a challenging problem, but because the CSRs present on Futurebus+ modules are well defined, testing them is even more challenging. The functionality of each of the bits or bit fields in most of the CSRs are verified through tests written specifically for the CSRs or as part of other tests.

What may be overlooked is that the CSRs are also slaves and must, therefore, be tested as slaves. To complicate matters, some of the CSRs are likely to be implemented in ROM, others in RAM, and others by chains of logic embedded within ASICs. CSRs implemented using different technologies will almost certainly exhibit different timing characteristics. It is possible that one CSR may pass all of the slave tests while another does not, even if they occupy consecutive addresses. It is also possible that one CSR bit may pass all of the slave tests while another does not even if they are different bits within the same CSR.

To completely test the CSRs as slaves, the test facility must perform timing-related tests on CSRs and CSR bits representative of the technologies used (e.g., some RAM-based bits should be tested as well as ROM-based bits, etc.).

4.3.2 Testing modules with multiple subsystems

Futurebus+ modules often include more than one type of master or slave. For conformance test purposes, it must be understood that a module's architecture and logic partitioning before an adequate test plan can be developed. For example, a module that contains a DMA master and a CPU master will typically require two separate test plans, one for each type of master. Because the DMA and CPU timing characteristics are likely different, each should be tested individually. Further, CPU masters typically have different capabilities from DMA masters. Bus interfaces for CPU masters are usually capable of accessing single bytes through short bursts of data. DMA masters, on the other hand, are often optimized for high-speed data transfers through the use of wide data paths and large bursts.

On the other hand, if the CPU and DMA both share the same bus interface logic and if that logic hides the timing characteristics of the CPU and DMA controllers, only one test plan may be necessary.

4.3.3 Data verification

Test cases shall perform data verification when possible. The test lab shall make the decision when it is possible to perform data verification.

4.4 Limitations of the conformance test process

Regardless of the steps taken to ensure that the physical tester and the executable test suite limitations are minimized, the conformance test concept has its own inherent limitations.

4.4.1 Module testability

In many cases, incorrect behaviors go undetected because of a module's lack of testability.

4.4.1.1 Nonmemory slaves

The slave tests in this standard were written with read/write memory in mind. The primary motivation behind this was to simplify the definition of the tests. Real Futurebus+ modules, however, often contain non-memory slaves such as registers and FIFOs. Testing these types of slaves can be significantly more complex than testing memory slaves. The main problem in testing nonmemory slaves is that the test cases must be adapted to accommodate the characteristics of the specific nonmemory slave being tested.

The following describe various types of nonmemory slaves and identify the issues that must be addressed in adapting the memory slave test cases to test them.

- a) *Read-only static slaves.* The most common read-only static slave is ROM. The data may be easily read from it, but cannot be deliberately modified, and the data pattern read will never change. The main problem in testing this type of slave is providing the test cases with a means of determining whether or not the data read is correct. In testing memory slaves, a known pattern is generally written to the target locations before the test begins. In this case, verifying that the data is correct is as simple as comparing the data read during the test with the data written before the test. Since a known data pattern cannot be written to ROM, the test cases must use some other means of data verification.
- b) *Read-only dynamic slaves.* The most common read-only dynamic slave is a status register. The data may be easily read from it, but cannot be deliberately modified, but the data pattern read may be different each time it is read. As with the static read-only slave, the main problem is modifying the test cases so that they can determine whether or not the read data is correct, which in some cases is virtually impossible to do. The best approach in these cases is to log the data read and examine it manually to see if it seems reasonable.
- c) *Virtually read-only dynamic slaves.* The most common form of this category of slaves is the output of a FIFO. This category of slaves is very similar to read-only dynamic slaves, the difference being that the pattern read may be deliberately modifiable through indirect means. By writing the desired pattern into the input of the FIFO, it may be read from the output of the FIFO. The trick here is writing the pattern to the FIFO. If the FIFO input is not accessible from the bus, writing data to the FIFO may be quite difficult. Another problem is that the FIFO data is volatile in that it may be read only once. If a test case is to read the data more than once, the data must be initialized prior to each read.
- d) *Write-only slaves.* The most common write-only slave is a hardware control register. The bits in such registers control various aspects of a module's functionality. In many cases there is no means by which the tester can verify that the data written to the register was written properly.
- e) *Virtually write-only slaves.* The most common virtually write-only slave is a hardware control register or a FIFO. This class of slaves is similar to the write-only slaves, the difference being that the data written to the slave may be verified indirectly. The effects of setting a bit in a hardware control register may be indirectly observable by reading a related status register. The data written to a FIFO's input may be verified by looking at the data coming out of the FIFO output. In some cases, verification through such indirect means may be possible but impractical.
- f) *Read/write slaves with pattern sensitivity.* The most common form of this class of slave is a hardware control register. In many cases the patterns written to control registers may change the character of the module to the degree that test results are meaningless. For example, setting a given bit in a register may cause the module to undergo an internal reset sequence, which in turn modifies the value that would be read back from the register. The problem in testing this type of slave is to choose the data patterns written carefully to ensure that no such events occur.

4.4.1.2 Prescribed action on command

Before a module can be thoroughly tested, it must be caused to exhibit all of its various behaviors. Causing a module to generate and respond to transactions using all of the variations it supports may be relatively easy or it may be virtually impossible, depending on its architecture.

To cause a module to exhibit behavior that takes place under a specific set of conditions, the test environment must be caused to provide that set of conditions. For example, Futurebus+ slaves often have the ability to respond to a transaction by signaling a Busy or Wait status. Suppose the slave being tested functions as a FIFO, where write transactions on the bus fill the FIFO and a communication channel empties it. Such a slave would likely signal a Busy status only if the FIFO is full at the time a write transaction targeting the FIFO occurs. To cause the slave to signal a Busy status, the tester would have to cause the FIFO to overflow, which could be quite difficult if the FIFO is very large or if the data sent to the FIFO had to meet special protocol requirements. If the communication channel is very high speed, the tester may not be fast enough to cause the FIFO to overflow.

Steps may be taken during the design phase to help minimize this type of difficulty. For example, the Diagnostic Extensions for Conformance Test section in IEEE P896.3/D4 .2, February 1993, provides a means of eliciting certain behaviors from a module, simply by asking it to exhibit the behaviors. The use of such extensions is highly encouraged.

4.4.2 Conformance test vs. design verification

Conformance tests allow a test facility to determine the extent to which a module meets the appropriate set of requirements. Generally, conformance tests are an excellent subset of design verification tests, but are subject to the following limitations. During conformance tests, it is generally assumed that the module being tested is stable and consistent. For example, each time a memory module responds to a bus transaction, it is expected to reliably retrieve read data or store write data. Further, the module is expected to exhibit similar timing characteristics each time it is accessed. Consequently, most conformance test sequences are performed only once.

To do thorough design verification, the same test sequence is typically repeated many times. Common practice is to run the tests in an endless loop for hours or days at a time. This repetition helps to identify errant behaviors that occur only when a certain set of circumstances occurs. For example, the dual port memory on a CPU module may only fail when the local master, the bus master, and the refresh controller access requests occur within the same 10 ns window. Unless tests are performed an adequate number of times to ensure that all such combinations of conditions are exercised, the module's design will not have been thoroughly proven.

4.5 Test case format

The majority of this standard consists of test cases. The typical Futurebus+ product will not require that all of these tests be run. However, depending upon the product's feature set and its implementation details, some of the tests may have to be run more than once. The test cases are written in ANSI C and may be used as the main core for Futurebus+ module test programs.

Each test case shall include the following:

- a) TEST_ID—Reference number for use with BICS entry to test case mapping information.
- b) TEST_NAME—Descriptive name.
- c) TEST DESCRIPTION—Description of what the test case is verifying and how the test case is performed.
- d) BICS REFERENCE—It identifies the BICS entry or entries that determine(s), based upon the features of the IUT, if the test case is applicable.
- e) TEST PROCEDURE—It describes the sequence of events necessary to verify that the IUT meets the requirement identified in the test description.

4.6 Test case numbering scheme

The following numbering scheme is used to organize the tests and to give each test a unique identification number. The largest identification number that can be used is 6 digits long.

Example: #1.#2.#3.#4.#5.#6

#1—The most significant digit identifies what main function of Futurebus+ is being tested. These are defined below.

- 1) Parallel protocol
- 2) Distributed arbitration
- 3) Central arbitration
- 4) Message passing
- 5) Bus/system management and CSR

Each of these main functions have a different subnumbering scheme. Each subnumbering scheme will be defined separately.

4.6.1 Parallel protocol numbering scheme

4.6.1.1 Transaction type

#2—This digit identifies the type of transaction that is to be performed. These are defined below.

- 1) Read unlocked
- 2) Write unlocked
- 3) Address only unlocked
- 4) Read locked
- 5) Write locked
- 6) Address only locked
- 7) Read partial
- 8) Write partial
- 9) Read partial locked
- 10) Write partial locked
- 11) Read response Write response
- 12) Write no acknowledge
- 13) Read invalid
- 14) Write invalid
- 15) Read shared
- 16) Copyback
- 17) Read modified
- 18) Invalidate
- 19) Shared response
- 20) Modified response
- 21) Read unlocked packet
- 22) Write unlocked packet
- 23) Read response packet
- 24) Write no acknowledge packet

4.6.1.2 Master/slave

#3—This digit identifies if the IUT is being tested as a master or slave. These are defined below.

- 1) The IUT is tested as a slave.

- 2) The IUT is tested as a master.

4.6.1.3 Condition

#4—This digit identifies what condition is being tested. These are defined below.

- 1) Normal status
- 2) Transaction error (TE)
- 3) Beat error (BE)
- 4) Wait (WT)
- 5) Busy (BS)
- 6) End of data (ED)
- 7) Broadcast (BC)
- 8) Transaction flag (TF)
- 9) Intervention (IV)
- 10) Split response (SR)
- 11) Unselected
- 12) Self-detected condition
- 13) Miscellaneous

4.6.1.4 Error conditions or user-specific conditions

#5—This digit is used for Read and Write Unlocked transactions to identify in which phase errors occur. These are defined below. In other transactions this digit can be user specific and could be used to identify other subdivisions of the test case.

- 1) Connection phase error
- 2) Data phase error
- 3) Disconnection phase error

4.6.1.5 Test specific

#6—This digit is test specific and could be used to identify other subdivisions of the test case.

4.6.2 Distributed arbitration numbering scheme

A value of 2 for digit #1 signifies distributed arbitration test cases. Distributed arbitration test case numbering goes to four digits.

4.6.2.1 Test group type

#2—This digit identifies the type of test group being performed.

- 1) Basic functional tests
- 2) Extended functional tests

4.6.2.2 Basic functional tests

If the test group basic functional tests is selected in digit #2, then digit #3 identifies the competition type being performed.

- 1) One pass
- 2) Two pass
- 3) Distributed message

- 4) Central arbitrated message
- 5) General arbitrated message

4.6.2.3 Extended functional tests

If the test group extended functional tests is selected in digit #2, then digit #3 identifies the distributed arbitration bus feature being tested.

- 1) Reset
- 2) Deposition
- 3) Arbitration owner
- 4) Round robin
- 5) Phase 2 time-out error
- 6) Parity error
- 7) Tester asserts error
- 8) Phase 4 time-out
- 9) Competition settling time
- 10) $\sim AS^*$ to $\sim AP^*$ handshake
- 11) $\sim AK^*$ (or $\sim AD^*$, CM^*) to AR^* handshake
- 12) Transaction preemption
- 13) Master transaction begin cancel

4.6.2.4 Subdivision

#4—This digit is test specific and could be used to identify other subdivisions of the test case.

4.6.3 Central arbitration number scheme

A value of 3 for digit #1 signifies central arbitration test cases. Central arbitration numbering goes to 3 digits.

4.6.3.1 Test group

#2—This digit is used to indicate whether the test group covers central arbitration requesters or central arbiter modules.

- 1) Central arbitration requester
- 2) Central arbiter module

4.6.3.2 Subdivision

#3—This digit is test specific and could be used to identify other subdivisions of the test case.

4.6.4 Message passing numbering scheme

4.6.4.1 Message protocol

#2—This digit identifies the type of message protocol used.

- 1) Frame level protocol
- 2) Message level protocol

4.6.4.2 General test category

If the frame level protocol is selected in digit #2, then digit #3 identifies the general test category as defined below.

- 1) Frame size ≤ 64 bytes, data width varied
- 2) Address only unlocked transaction
- 3) Protocol field check
- 4) Queue reordering/preemption
- 5) Frame size, bus signals
- 6) Message passing broadcast, frame level
- 7) Message destination busy

If the message level protocol is selected in digit #2, then digit #3 identifies the general test category as defined below.

- 1) Message source time-out
- 2) Message ID and frame size field checks
- 3) Unacknowledged event frame
- 4) Acknowledged event frame
- 5) Ordered multiple frame message
- 6) Sequenced multiple frame message
- 7) Exceptions
- 8) Several multiple frame messages outstanding

4.6.4.3 Subdivision

#4 and #5—These digits are test specific and are used to further subdivide the test case.

4.6.5 Bus/system management and CSR numbering scheme

4.6.5.1 CSR group name

#2—This digit identifies the Bus/system management or CSR group. These are defined below.

- 1) Bus system management
- 2) Core CSRs
- 3) Futurebus+ specific CSRs
- 4) ROM CSRs
- 5) Unimplemented register

4.6.5.2 Third digit subdivision

The third digit has different meanings for each of the CSR groups listed above.

4.6.5.2.1 Bus system management subdivision

#3—This digit is test specific and could be used to identify other subdivisions of the test case.

4.6.5.2.2 Core and Futurebus++-specific CSR subdivision

#3—This digit identifies the specific CSR being tested. CSRs are numbered in the order that they are listed in IEEE Std 896.2-1991 .

4.6.5.2.3 ROM CSRs

#3—This digit specifies the ROM group being tested.

- 1) ROM entry CSRs
- 2) Other ROM CSRs

4.6.5.2.4 Unimplemented registers

#3—This digit is not used.

4.6.5.3 #4 and #5 digit

#4—This digit is test specific and could be used to identify other subdivisions of the test case.

#5—This digit is test specific and could be used to identify other subdivisions of the test case.

5. Test cases

The test cases that comprise this section can be found on the diskettes that accompany this standard. The documentation in this section of the standard lists the file name and the test identification (TEST ID) and test name (TEST NAME) for each test that can be found under that file name.

5.1 Parallel protocol

5.1.1 Read unlocked transaction

5.1.1.1 IUT slave

The tests for this subclause can be found in file “5_1_1_1.c” on the accompanying diskette. Tests that are included in that file are as follows:

5.1.1.1.1 Normal status

```
TEST_ID = "1.1.1.1.1.1";
TEST_NAME = "Read Unlocked, IUT Slave, Aligned Address.";
TEST_ID = "1.1.1.1.1.2";
TEST_NAME = "Read Unlocked, IUT Slave, Unaligned Address.";
TEST_ID = "1.1.1.1.1.3";
TEST_NAME = "Read Unlocked, IUT Slave, Unrestricted Data Length.";
```

5.1.1.1.2 Transaction error (TE)

```
TEST_ID = "1.1.1.2.3.1";
TEST_NAME = "Read Unlocked, IUT Slave, Connection Command Parity \
Error";
TEST_ID = "1.1.1.2.3.2";
TEST_NAME = "Read Unlocked, IUT Slave, Address Parity Error in the \
Connection Phase.";
TEST_ID = "1.1.1.2.3.3";
TEST_NAME = "Read Unlocked, IUT Slave, Unsupported Data Width.";
TEST_ID = "1.1.1.2.3.4";
TEST_NAME = "Read Unlocked, IUT Slave, Unsupported Address Width.";
TEST_ID = "1.1.1.2.3.5";
TEST_NAME = "Read Unlocked IUT Slave, Unsupported Connection Command \
Error";
"; TEST_ID = "1.1.1.2.3.6";
TEST_NAME = "Read Unlocked IUT Slave, Invalid Connection Command \
```

```

Error";
TEST_ID = "1.1.1.2.3.7";
TEST_NAME = "Read Unlocked IUT Slave, TE* Held Until Next Connection \
Phase";
TEST_ID = "1.1.1.2.3.8";
TEST_NAME = "Read Unlocked IUT Slave, Connection Tag Parity Error";
TEST_ID = "1.1.1.2.3.11";
TEST_NAME = "Read Unlocked, IUT Slave, Data Phase Command Parity Error";
TEST_ID = "1.1.1.2.3.17";
TEST_NAME = "Read Unlocked IUT Slave, Disconnection Phase Command \
Parity Error";
TEST_ID = "1.1.1.2.3.19";
TEST_NAME = "Read Unlocked, IUT Slave, Disconnection Phase Tag Parity \
Error";

```

5.1.1.1.3 Beat error (BE)

There are no tests for this category.

5.1.1.1.4 Wait (WT)

```

TEST_ID = "1.1.1.4.1.1";
TEST_NAME = "Read Unlocked, IUT Slave, Wait";

```

5.1.1.1.5 Busy (BS)

```

TEST_ID = "1.1.1.5.1.1";
TEST_NAME = "Read Unlocked, IUT Slave, Busy";

```

5.1.1.1.6 End of date (ED)

```

TEST_ID = "1.1.1.6.2.1";
TEST_NAME = "Read Unlocked, IUT Slave, ED Resulting from Crossing the \
Value in MEMORY_BOUND.";
TEST_ID = "1.1.1.6.2.2";
TEST_NAME = "Read Unlocked, IUT Slave, ED Resulting from Exceeding \
Buffer Size.";

```

5.1.1.1.7 Broadcast (BC)

```

TEST_ID = "1.1.1.7.1.1";
TEST_NAME = "Read Unlocked, IUT Slave, Broadcast.";

```

5.1.1.1.8 Transaction flag (TF)

There are no tests for this category.

5.1.1.1.9 Intervention (IV)

There are no tests for this category.

5.1.1.1.10 Split response (SR)

There are no tests for this category.

5.1.1.1.11 Unselected

TEST_ID = "1.1.1.11.1.1";
TEST_NAME = "Read Unlocked, IUT Slave, Address Unselected";

5.1.1.1.12 Self-detected condition

There are no tests for this category.

5.1.1.1.13 Miscellaneous

BeTEST_ID = "1.1.1.13.1.1";
TEST_NAME = "Read Unlocked, IUT Slave, Nonbenign";
TEST_ID = "1.1.1.13.1.2";
TEST_NAME = "Read Unlocked, IUT Slave, DELAY DS* ASSERT ODD";
TEST_ID = "1.1.1.13.1.3";
TEST_NAME = "Read Unlocked, IUT Slave, DELAY DS* RELEASE EVEN";
TEST_ID = "1.1.1.13.1.4";
TEST_NAME = "Read Unlocked, IUT Slave, DELAY AS* RELEASE DISCON";

5.1.1.2 IUT master

The tests for this subclause can be found in file "5_1_1_2.c" on the accompanying diskette. Tests that are included in that file are as follows:

5.1.1.2.1 Normal status

TEST_ID = "1.1.2.1.1.1";
TEST_NAME = "Read Unlocked, IUT Master, Aligned Address";
TEST_ID = "1.1.2.1.1.2";
TEST_NAME = "Read Unlocked, IUT Master Unaligned Address.";
TEST_ID = "1.1.2.1.1.3";
TEST_NAME = "Read Unlocked, IUT Master Unrestricted Data Length.";

5.1.1.2.2 Transaction error (TE)

TEST_ID = "1.1.2.2.3.1";
TEST_NAME = "Read Unlocked, IUT Master Address/Data Parity Error in \ the Data Phase";
TEST_ID = "1.1.2.2.3.2";
TEST_NAME = "Read Unlocked, IUT Master Tag Parity Error in the \ Data Phase";
TEST_ID = "1.1.2.2.3.14";
TEST_NAME = "Read Unlocked, IUT Master Tester Asserts IV* (Illegal \ Operation)";

5.1.1.2.3 Beat error (BE)

There are no tests for this category.

5.1.1.2.4 Wait (WT)

TEST_ID = "1.1.2.4.1.1";
TEST_NAME = "Read Unlocked, IUT Master, Wait";

5.1.1.2.5 Busy (BS)

TEST_ID = "1.1.2.5.1.1";
TEST_NAME = "Read Unlocked, IUT Master, Busy";
TEST_ID = "1.1.2.5.1.2";
TEST_NAME = "Read Unlocked, IUT Master, Busy and Beat Error Asserted \ in the Connection Phase";
TEST_ID = "1.1.2.5.1.3";
TEST_NAME = "Read Unlocked, IUT Master, Busy and SR in the \ Connection Phase";

5.1.1.2.6 End of data (ED)

TEST_ID = "1.1.2.6.2.1";
TEST_NAME = "Read Unlocked, IUT Master, ED* is Asserted by the Tester.";

5.1.1.2.7 Broadcast (BC)

TEST_ID = "1.1.2.7.1.1";
TEST_NAME = "Read Unlocked, IUT Master, Broadcast";

5.1.1.2.8 Transaction flag (TF)

There are no tests for this category.

5.1.1.2.9 Intervention (IV)

There are no tests for this category.

5.1.1.2.10 Split response (SR)

There are no tests for this category.

5.1.1.2.11 Unselected

TEST_ID = "1.1.2.11.1.1";
TEST_NAME = "Read Unlocked, IUT Master, Address Unselected";

5.1.1.2.12 Self-detected condition

There are no tests for this category.

5.1.1.2.13 Miscellaneous

TEST_ID = "1.1.2.13.1.1";
TEST_NAME = "Read Unlocked, IUT Master, Nonbenign";
TEST_ID = "1.1.2.13.1.2";
TEST_NAME = "Read Unlocked, IUT Master, DELAY AI* RELEASE CONN";
TEST_ID = "1.1.2.13.1.3";
TEST_NAME = "Read Unlocked, IUT Master, DELAY DI* ASSERT CONN";
TEST_ID = "1.1.2.13.1.4";
TEST_NAME = "Read Unlocked, IUT Master, DELAY DI RELEASE ODD";
TEST_ID = "1.1.2.13.1.5";
TEST_NAME = "Read Unlocked, IUT Master, DELAY DK* RELEASE EVEN";

TEST_ID = "1.1.2.13.1.6";
TEST_NAME = "Read Unlocked, IUT Master, DELAY AK* RELEASE DISCON";

5.1.2 Write unlocked transaction

5.1.2.1 IUT slave

The tests for this subclause can be found in file "5_1_2_1.c" on the accompanying diskette. Tests that are included in that file are as follows:

5.1.2.1.1 Normal status

TEST_ID = "1.2.1.1.1.1";
TEST_NAME = "Write Unlocked, IUT Slave, Aligned Address";
TEST_ID = "1.2.1.1.1.2";
TEST_NAME = "Write Unlocked, IUT Slave, Unaligned Address.";
TEST_ID = "1.2.1.1.1.3";
TEST_NAME = "Write Unlocked, IUT Slave, Unrestricted Data Length.";

5.1.2.1.2 Transaction error (TE)

TEST_ID = "1.2.1.2.3.1";
TEST_NAME = "Write Unlocked, IUT Slave, Address/Data Parity Error in \ the Data Phase";
TEST_ID = "1.2.1.2.3.2";
TEST_NAME = "Write Unlocked, IUT Slave, Tag Parity Error in the Data \ Phase";
TEST_ID = "1.2.1.2.3.18";
TEST_NAME = "Write Unlocked, IUT Slave, Address/Data Parity Error in \ Disconnection Phase.";

5.1.2.1.3 Beat error (BE)

There are no tests for this category.

5.1.2.1.4 Wait (WT)

There are no tests for this category.

5.1.2.1.5 Busy (BS)

There are no tests for this category.

5.1.2.1.6 End of data (ED)

TEST_ID = "1.2.1.6.2.1";
TEST_NAME = "Write Unlocked, IUT Slave, ED Resulting from Crossing the \ Value in MEMORY_BOUND.";

5.1.2.1.7 Broadcast (BC)

TEST_ID = "1.2.1.7.1.1";
TEST_NAME = "Write Unlocked, IUT Slave, Broadcast";
TEST_NAME = "Read Unlocked, IUT Master, Broadcast";

5.1.2.1.8 Transaction flag (TF)

There are no tests for this category.

5.1.2.1.9 Intervention (IV)

There are no tests for this category.

5.1.2.1.10 Split response (SR)

There are no tests for this category.

5.1.2.1.11 Unselected

There are no tests for this category.

5.1.2.1.12 Self-detected condition

There are no tests for this category.

5.1.2.1.13 Miscellaneous

```
TEST_ID = "1.2.1.13.1.1";
TEST_NAME = "Write Unlocked, IUT Slave, Nonbenign";
TEST_ID = "1.2.1.13.1.2";
TEST_NAME = "Write Unlocked, IUT Slave, DELAY DS* ASSERT ODD";
TEST_ID = "1.2.1.13.1.3";
TEST_NAME = "Write Unlocked, IUT Slave, DELAY DS* RELEASE EVEN";
TEST_ID = "1.2.1.13.1.4";
TEST_NAME = "Write Unlocked, IUT Slave, DELAY AS* RELEASE DISCON";
```

5.1.2.2 IUT master

The tests for this subclause can be found in file "5_1_2_2.c" on the accompanying diskette. Tests that are included in that file are as follows:

5.1.2.2.1 Normal status

```
TEST_ID = "1.2.2.1.1.1";
TEST_NAME = "Write Unlocked, IUT Master, Aligned Address";
TEST_ID = "1.2.2.1.1.2";
TEST_NAME = "Write Unlocked, IUT Master, Unaligned Address.";
TEST_ID = "1.2.2.1.1.3";
TEST_NAME = "Write Unlocked, IUT Master, Unrestricted Data Length.";
```

5.1.2.2.2 Beat error (BE)

There are no tests for this category.

5.1.2.2.3 Beat error (BE)

There are no tests for this category.

5.1.2.2.4 Wait (WT)

There are no tests for this category.

5.1.2.2.5 Busy (BS)

There are no tests for this category.

5.1.2.2.6 End of data (ED)

TEST_ID = "1.2.2.6.2.1";
TEST_NAME = "Write Unlocked, IUT Master, ED* is Asserted by the \ Tester.";

5.1.2.2.7 Broadcast (BC)

TEST_ID = "1.2.2.7.1.1";
TEST_NAME = "Write Unlocked, IUT Master, Broadcast";

5.1.2.2.8 Transaction flag (TF)

There are no tests for this category.

5.1.2.2.9 Intervention (IV)

There are no tests for this category.

5.1.2.2.10 Split response (SR)

There are no tests for this category.

5.1.2.2.11 Unselected

There are no tests for this category.

5.1.2.2.12 Self-detected condition

There are no tests for this category.

5.1.2.2.13 Miscellaneous

TEST_ID = "1.2.2.13.1.1";
TEST_NAME = "Write Unlocked IUT Master, Nonbenign";
TEST_ID = "1.2.2.13.1.2";
TEST_NAME = "Write Unlocked IUT Master, DELAY AI* RELEASE CONN";
TEST_ID = "1.2.2.13.1.3";
TEST_NAME = "Write Unlocked IUT Master, DELAY DI* ASSERT CONN";
TEST_ID = "1.2.2.13.1.4";
TEST_NAME = "Write Unlocked IUT Master, DELAY DI* RELEASE ODD";
TEST_ID = "1.2.2.13.1.5";
TEST_NAME = "Write Unlocked IUT Master, DELAY DK* RELEASE EVEN";
TEST_ID = "1.2.2.13.1.6";
TEST_NAME = "Write Unlocked IUT Master DELAY AK* RELEASE DISCON";

5.1.3 Address only unlocked transaction

5.1.3.1 IUT slave

The test for this subclause can be found in file “5_1_3_1.c” on the accompanying diskette. The test is the following:

5.1.3.1.1 Normal status

```
TEST_ID = "1.3.1.1.1.1";  
TEST_NAME = "Address Only Unlocked, IUT Slave, Normal.";
```

5.1.3.2 IUT master

The test for this subclause can be found in file “5_1_3_2.c” on the accompanying diskette. The test is the following:

5.1.3.2.1 Normal status

```
TEST_ID = "1.3.2.1.1.1";  
TEST_NAME = "Address Only Unlocked, IUT Master, Normal.";
```

5.1.4 Read locked transaction

5.1.4.1 IUT slave

The tests for this subclause can be found in file “5_1_4_1.c” on the accompanying diskette. Tests that are included in that file are as follows:

5.1.4.1.1 Normal status

```
TEST_ID = "1.4.1.1.1.1";  
TEST_NAME = "Read Locked, IUT Slave, Normal, LK Released.";  
TEST_ID = "1.4.1.1.1.2";  
TEST_NAME = "Read Locked, IUT Slave, Normal, Unlocked Transaction.";  
TEST_ID = "1.4.1.1.1.3";  
TEST_NAME = "Read Locked, IUT Slave, Normal, Release Master Tenure.";
```

5.1.4.1.2 Transaction error (TE)

```
TEST_ID = "1.4.1.2.3.1";  
TEST_NAME = "Read Locked, IUT Slave, Illegal Operation - TE.";
```

5.1.4.2 IUT master

The test for this subclause can be found in “5_1_4_2.c” on the accompanying diskette. The test included in that file is the following:

5.1.4.2.1 Normal status

```
TEST_ID = "1.4.2.1.1.1";  
TEST_NAME = "Read Locked, IUT Master, Normal.";
```

5.1.5 Write locked transaction

5.1.5.1 IUT slave

The tests for this subclause can be found in file “5_1_5_1.c” on the accompanying diskette. Tests that are included in that file are as follows:

5.1.5.1.1 Normal status

```
TEST_ID = "1.5.1.1.1.1";
TEST_NAME = "Write Locked, IUT Slave, Normal, LK Released.";
TEST_ID = "1.5.1.1.1.2";
TEST_NAME = "Write Locked, IUT Slave, Normal, Unlocked Transaction.";
TEST_ID = "1.5.1.1.1.3";
TEST_NAME = "Write Locked, IUT Slave, Normal, Release Master Tenure.";
TEST_ID = "1.5.1.1.2.1";
TEST_NAME = "Mask & Swap Lock Command, IUT Slave, Normal.";
TEST_ID = "1.5.1.1.6.1";
TEST_NAME = "Compare & Swap Lock Command, IUT Slave, Normal, Operands \
Match.";
TEST_ID = "1.5.1.1.6.2";
TEST_NAME = "Compare & Swap Lock Command, IUT Slave, Normal, Operands \
Don't Match.";
TEST_ID = "1.5.1.1.7.1";
TEST_NAME = "Fetch & Add Big Endian Lock Command, IUT Slave, Normal.";
TEST_ID = "1.5.1.1.8.1";
TEST_NAME = "Fetch & Add Little Endian Lock Command, IUT Slave, \ Normal.";
```

5.1.5.1.2 Transaction error

```
TEST_ID = "1.5.1.2.3.1";
TEST_NAME = "Write Locked, IUT Slave, Illegal Operations - TE.";
TEST_ID = "1.5.1.2.3.2";
TEST_NAME = "Mask & Swap Lock Command, IUT Slave, Normal, No Swap \
Operand - Errors: TE, Possible BE.";
TEST_ID = "1.5.1.2.3.3";
TEST_NAME = "Compare & Swap Lock Command, IUT Slave, No Swap Operand \
Errors: TE, Possible BE.";
```

5.1.5.2 IUT master

The tests for this subclause can be found in file “5_1_5_2.c” on the accompanying diskette. Tests that are included in that file are as follows:

5.1.5.2.1 Normal status

```
TEST_ID = "1.5.2.1.1.1";
TEST_NAME = "Write Locked, IUT Master, Normal.";
TEST_ID = "1.5.2.1.2.1";
TEST_NAME = "Mask & Swap Lock Command, IUT Master, Normal.";
TEST_ID = "1.5.2.1.3.1";
TEST_NAME = "Compare & Swap Lock Command, IUT Master, Normal.";
TEST_ID = "1.5.2.1.4.1";
TEST_NAME = "Fetch & Add Big Endian Lock Command, IUT Master, Normal.";
```

```
TEST_ID = "1.5.2.1.5.1";  
TEST_NAME = "Fetch & Add Little Endian Lock Command, IUT Master, \  
Normal.";
```

5.1.6 Address only locked transaction

5.1.6.1 IUT slave

The tests for this subclause can be found in file "5_1_6_1.c" on the accompanying diskette. Tests that are included in that file are as follows:

5.1.6.1.1 Normal status

```
TEST_ID = "1.6.1.1.1.1";  
TEST_NAME = "Address Only Locked, IUT Slave, Normal - LK Released.";  
TEST_ID = "1.6.1.1.1.2";  
TEST_NAME = "Address Only Locked, IUT Slave, Normal - Unlocked \  
Transaction.";  
TEST_ID = "1.6.1.1.1.3";  
TEST_NAME = "Address Only Locked, IUT Slave, Normal - Release Master \  
Tenure.";
```

5.1.6.2 IUT master

The test for this subclause can be found in file "5_1_6_2.c" on the accompanying diskette. The test included in that file is as follows:

5.1.6.2.1 Normal status

```
TEST_ID = "1.6.2.1.1.1";  
TEST_NAME = "Address Only Locked, IUT Master, Normal.";
```

5.1.7 Read partial transaction

5.1.7.1 IUT slave

The test for this subclause can be found in file "5_1_7_1.c" on the accompanying diskette. The test included in that file is as follows:

5.1.7.1.1 Normal status

```
TEST_ID = "1.7.1.1.1.1";  
TEST_NAME = "Read Partial, IUT Slave, Normal";
```

5.1.7.2 IUT master

The tests for this subclause can be found in file "5_1_7_2.c" on the accompanying diskette. Tests that are included in that file are as follows:

5.1.7.2.1 Normal status

```
TEST_ID = "1.7.2.1.1.1";  
TEST_NAME = "Read Partial, IUT Master, Normal";  
TEST_ID = "1.7.2.1.1.2";
```

```

TEST_NAME = "Read Partial, IUT Master, Deselected Byte Lane Parity \
Error";
TEST_ID = "1.7.2.2.3.1";
TEST_NAME = "Read Partial, IUT Master, Selected Byte Lane Parity Error";

```

5.1.8 Write partial transaction

5.1.8.1 IUT slave

The tests for this subclause can be found in file "5_1_8_1.c" on the accompanying diskette. Tests that are included in that file are as follows:

5.1.8.1.1 Normal status

```

TEST_ID = "1.8.1.1.1.1";
TEST_NAME = "Write Partial, IUT Slave, Normal";
TEST_ID = "1.8.1.1.1.2";
TEST_NAME = "Write Partial, IUT Slave, Deselected Byte Lane Data Phase \
Parity Error";
TEST_ID = "1.8.1.2.3.1";
TEST_NAME = "Write Partial, IUT Slave, Lane Deselect Parity Error";
TEST_ID = "1.8.1.2.3.2";
TEST_NAME = "Write Partial, IUT Slave, Selected Byte Lane Data Phase \
Parity Error";

```

5.1.8.2 IUT master

The test for this subclause can be found in file "5_1_8_2.c" on the accompanying diskette. The test included in that file is as follows:

5.1.8.2.1 Normal status

```

TEST_ID = "1.8.2.1.1.1";
TEST_NAME = "Write Partial, IUT Master, Normal";

```

5.1.9 Read partial locked transaction

5.1.9.1 IUT slave

The tests for this subclause can be found in file "5_1_9_1.c" on the accompanying diskette. Tests that are included in that file are as follows:

5.1.9.1.1 Normal status

```

TEST_ID = "1.9.1.1.1.1";
TEST_NAME = "Read Partial Locked, IUT Slave, Normal.";

```

5.1.9.1.2 Transaction error

```

TEST_ID = "1.9.1.2.3.1";
TEST_NAME = "Read Partial Locked Transaction with Illegal Operation, \
IUT Slave, No Split, Compelled Data Transaction, Errors: TE, Possible \
BE.";

```

5.1.9.2 IUT master

The test for this subclause can be found in file “5_1_9_2.c” on the accompanying diskette. The test included in that file is as follows:

5.1.9.2.1 Normal status

```
TEST_ID = "1.9.2.1.1.1";
TEST_NAME = "Read Partial Locked, IUT Master, Normal.";
```

5.1.10 Write partial locked transaction

5.1.10.1 IUT slave

The tests for this subclause can be found in file “5_1_10_1.c” on the accompanying diskette. Tests that are included in that file are as follows:

5.1.10.1.1 Normal status

```
TEST_ID = "1.10.1.1.1.1";
TEST_NAME = "Write Partial Locked Transaction, IUT Slave, Normal.";
TEST_ID = "1.10.1.1.2.1";
TEST_NAME = "Write Partial Locked Transaction with Mask & Swap Lock \
Command, IUT Slave, Normal.";
TEST_ID = "1.10.1.1.3.1";
TEST_NAME = "Write Partial Locked Transaction with Compare & Swap Lock \
Command, IUT Slave, Normal - Match.";
TEST_ID = "1.10.1.1.3.2";
TEST_NAME = "Write Partial Locked Transaction with Compare & Swap Lock \
Command, IUT Slave, Normal - Operands Don't Match.";
TEST_ID = "1.10.1.1.4.1";
TEST_NAME = "Write Partial Locked Transaction with Fetch and Add Big \
Endian Lock Command, IUT Slave, Normal.";
TEST_ID = "1.10.1.1.5.1";
TEST_NAME = "Write Partial Locked Transaction with Fetch and Add \
Little Endian Lock Command, IUT Slave, Normal.";
```

5.1.10.1.2 Transaction error

```
TEST_ID = "1.10.1.2.3.1";
TEST_NAME = "Write Partial Locked Transaction with Illegal Operation, \
IUT Slave, Errors: TE, Possible BE.";
```

5.1.10.2 IUT master

The tests for this subclause can be found in file “5_1_10_2.c” on the accompanying diskette. Tests that are included in that file are as follows:

5.1.10.2.1 Normal status

```
TEST_ID = "1.10.2.1.1.1";
TEST_NAME = "Write Partial Locked, IUT Master, Normal.";
TEST_ID = "1.10.2.1.2.1";
TEST_NAME = "Write Partial Locked with Mask & Swap Lock Command, IUT \
```

Master, Normal.”;
TEST_ID = “1.10.2.1.3.1”;
TEST_NAME = “Write Partial Locked with Compare & Swap Lock Command, \ IUT Master, Normal - Match.”;
TEST_ID = “1.10.2.1.3.2”;
TEST_NAME = “Write Partial Locked with Compare & Swap Lock Command, \ IUT Master, Normal - No Match.”;
TEST_ID = “1.10.2.1.4.1”;
TEST_NAME = “Write Partial Locked with Fetch & Add Big Endian Lock \ Command, IUT Master, Normal.”;
TEST_ID = “1.10.2.1.5.1”;
TEST_NAME = “Write Partial Locked with Fetch & Add Little Endian Lock \ Command, IUT Master, Normal.”;

5.1.11 Read response transaction

5.1.11.1 IUT slave

The tests for this subclause can be found in file “5_1_11_1.c” on the accompanying diskette. Tests that are included in that file are as follows:

5.1.11.1.1 Normal status

TEST_ID = “1.11.1.1.1.1”;
TEST_NAME = “Read Response, IUT Slave to Read Response, Split Read \ with IUT as Split Acceptor, Normal, Aligned Address.”;
TEST_ID = “1.11.1.1.2.1”;
TEST_NAME = “Read Response, IUT Slave to Read Response, Split \ Read Locked with IUT as Split Acceptor, Normal, Aligned Address.”;
TEST_ID = “1.11.1.1.1.2”;
TEST_NAME = “Read Response, IUT Slave to Read Response, Split Read \ with IUT as Split Acceptor, Tester Responds with One Less Data Beat \ Than Requested by the IUT”;
TEST_ID = “1.11.1.2.1.1”;
TEST_NAME = “Read Response, IUT Slave to Read Response, Split Read \ with Tester Asserting BE* during Read Unlocked”;
TEST_ID = “1.11.1.2.1.2”;
TEST_NAME = “Read Response, IUT Slave to Read Response, Split Read \ with Tester Asserting BS* during Read Unlocked”;
TEST_ID = “1.11.1.2.1.3”;
TEST_NAME = “Read Response, IUT Slave to Read Response, Split Read \ with Tester Asserting the AW Signal on the Command Lines during the \ Connection Phase of the Read Response Transaction”;
TEST_ID = “1.11.1.2.1.4”;
TEST_NAME = “Read Response, IUT Slave to Read Response, Split Read \ with Tester Asserting a Different Data Width on the Command Lines \ during the Connection Phase of the Read Response Transaction”;
TEST_ID = “1.11.1.2.1.5”;
TEST_NAME = “Read Response, IUT Slave to Read Response, Split Read \ with Tester not Asserting the AD31 Signal on the Address/Data Lines \ during the Connection Phase of the Read Response Transaction”;

5.1.11.2 IUT master

The test for this subclause can be found in file “5_1_11_2.c” on the accompanying diskette. The test included in that file is as follows:

5.1.11.2.1 Normal status

```
TEST_ID = "1.11.2.1.1.1";
TEST_NAME = "Read Response, IUT Master to Read Response, Split Read \
Unlocked with IUT as Split Requestor, Normal, Aligned Address";
```

5.1.12 Write response transaction

5.1.12.1 IUT slave

The tests for this subclause can be found in file “5_1_12_1.c” on the accompanying diskette. Tests that are included in that file are as follows:

5.1.12.1.1 Normal status

```
TEST_ID = "1.12.1.1.1.1";
TEST_NAME = "Write Response, IUT Slave to Write Response, Split Write \
with IUT as Split Acceptor, Normal, Aligned Address.";
TEST_ID = "1.12.1.1.2.1";
TEST_NAME = "Write Response, IUT Slave to Write Response, Split Write \
Locked with IUT as Split Acceptor, Normal, Aligned Address.";
```

5.1.12.2 IUT master

The test for this subclause can be found in file “5_1_12_2.c” on the accompanying diskette. The test included in that file is as follows:

5.1.12.2.1 Normal status

```
TEST_ID = "1.12.2.1.1.1";
TEST_NAME = "Write Response, IUT Master to Write Response, Split Write \
Unlocked with IUT as Split Requestor, Normal, Aligned Address";
```

5.1.13 Write no acknowledge transaction

5.1.13.1 IUT slave

The tests for this subclause can be found in file “5_1_13_1.c” on the accompanying diskette. Tests that are included in that file are as follows:

5.1.13.1.1 Normal status

```
TEST_ID = "1.13.1.1.1.1";
TEST_NAME = "Write No Acknowledge, IUT Slave, Aligned Address";
TEST_ID = "1.13.1.1.1.3";
TEST_NAME = "Write No Acknowledge, IUT Slave, Unrestricted Data \
Length.";
```

5.1.13.1.2 Transaction error (TE)

There are no tests for this category.

5.1.13.1.3 Beat error (BE)

There are no tests for this category.

5.1.13.1.4 Wait (WT)

There are no tests for this category.

5.1.13.1.5 Busy (BS)

There are no tests for this category.

5.1.13.1.6 End of data (ED)

TEST_ID = "1.13.1.6.2.1";

TEST_NAME = "Write No Acknowledge, IUT Slave, ED Resulting from \ Crossing the Value in MEMORY_BOUND.";

5.1.13.2 IUT master

The tests for this subclause can be found in file "5_1_13_2.c" on the accompanying diskette. Tests that are included in that file are as follows:

5.1.13.2.1 Normal status

TEST_ID = "1.13.2.1.1.1";

TEST_NAME = "Write No Acknowledge, IUT Master Aligned Address";

TEST_ID = "1.13.2.1.1.3";

TEST_NAME = "Write No Acknowledge, IUT Master Unrestricted Data \ Length.";

TEST_ID = "1.13.2.1.1.4";

TEST_NAME = "Write No Acknowledge, IUT Master Illegal Operation, \ Tester Attempts to Split";

5.1.13.2.2 Transaction error (TE)

There are no tests for this category.

5.1.13.2.3 Beat error (BE)

There are no tests for this category.

5.1.13.2.4 Wait (WT)

There are no tests for this category.

5.1.13.2.5 Busy (BS)

There are no tests for this category.

5.1.13.2.6 End of data (ED)

TEST_ID = "1.13.2.6.2.1";
TEST_NAME = "Write No Acknowledge, IUT Master, ED* is Asserted by the \ Tester. ";

5.1.14 Read invalid transaction**5.1.14.1 IUT slave**

The tests for this subclause can be found in file "5_1_14_1.c" on the accompanying diskette. Tests that are included in that file are as follows:

5.1.14.1.1 Normal status

TEST_ID = "1.14.1.1.1";
TEST_NAME = "Read Invalid, IUT Memory Slave, Aligned Address";

5.1.14.1.2 Transaction error (TE)

TEST_ID = "1.14.1.2.1";
TEST_NAME = "Read Invalid, IUT Memory Slave, Unsupported Data Width";

5.1.14.1.3 Beat error (BE)

There are no tests for this category.

5.1.14.1.4 Wait (WT)

TEST_ID = "1.14.1.4.1";
TEST_NAME = "Read Invalid, IUT Cache Slave, Wait";

5.1.14.1.5 Busy (BS)

TEST_ID = "1.14.1.5.1";
TEST_NAME = "Read Invalid, IUT Memory Slave, Busy";

5.1.14.1.6 End of data (ED)

There are no tests for this category.

5.1.14.1.7 Broadcast (BC)

There are no tests for this category.

5.1.14.1.8 Transaction flag (TF)

TEST_ID = "1.14.1.8.1";
TEST_NAME = " Read Invalid, IUT Cache Slave, Transaction Flag ";
TEST_ID = "1.14.1.8.2";
TEST_NAME = " Read Invalid, IUT Memory Slave, Transaction flag";
TEST_ID = "1.14.1.8.3";
TEST_NAME = "Read Invalid, IUT Cache Agent Snooping, Aligned Address";

5.1.14.1.9 Intervention (IV)

```

TEST_ID = "1.14.1.9.1";
TEST_NAME = " Read Invalid, IUT Cache Slave, Intervention.";
TEST_ID = "1.14.1.9.2";
TEST_NAME = " Read Invalid, IUT Memory Slave, Intervention";
TEST_ID = "1.14.1.9.3";
TEST_NAME = " Read Invalid, IUT Cache Slave, Intervention, Some SU";

```

5.1.14.1.10 Split response (SR)

```

TEST_ID = "1.14.1.10.1";
TEST_NAME = "Read Invalid, IUT Cache Slave, Split Response with \
Intervention";
TEST_ID = "1.14.1.10.2";
TEST_NAME = "Read Invalid, IUT Memory Slave, Split Response with \
intervention";
TEST_ID = "1.14.1.10.3";
TEST_NAME = "Read Invalid, IUT Cache Slave, Split Response, IV* and \
TF*, Some SU";
TEST_ID = "1.14.1.10.4";
TEST_NAME = " Read Invalid, IUT Memory Slave, Split Response";

```

5.1.14.2 IUT master

The tests for this subclause can be found in file "5_1_14_2.c" on the accompanying diskette. Tests that are included in that file are as follows:

5.1.14.2.1 Normal status

```

TEST_ID = "1.14.2.1.1";
TEST_NAME = "Read Invalid, IUT Master, Aligned Address";

```

5.1.14.2.2 Transaction error (TE)

There are no tests for this category.

5.1.14.2.3 Beat error (BE)

There are no tests for this category.

5.1.14.2.4 Wait (WT)

```

TEST_ID = "1.14.2.4.1";
TEST_NAME = "Read Invalid, IUT Master, Wait";

```

5.1.14.2.5 Busy (BS)

```

TEST_ID = "1.14.2.5.1";
TEST_NAME = "Read Invalid, IUT Master, Busy ";

```

5.1.14.2.6 End of data(ED)

There are no tests for this category.

5.1.14.2.7 Broadcast (BC)

There are no tests for this category.

5.1.14.2.8 Transaction flag (TF)

```
TEST_ID = "1.14.2.8.1";
TEST_NAME = "Read Invalid, IUT Master, Broadcast";
```

5.1.14.2.9 Intervention (IV)

```
TEST_ID = "1.14.2.9.1";
TEST_NAME = " Read Invalid, IUT Master, Intervention ";
TEST_ID = "1.14.2.9.2";
TEST_NAME = " Read Invalid, IUT Master, Intervention with Broadcast ";
```

5.1.14.2.10 Split response (SR)

```
TEST_ID = "1.14.2.10.1";
TEST_NAME = " Read Invalid, IUT Master, Split Response ";
TEST_ID = "1.14.2.10.2";
TEST_NAME = "Read Invalid, IUT Master, Split Response, TF * ";
TEST_ID = "1.14.2.10.3";
TEST_NAME = " Read Invalid, IUT Master, Split Response with \
Intervention ";
```

5.1.15 Write invalid transaction**5.1.15.1 IUT slave**

The tests for this subclause can be found in file "5_1_15_1.c" on the accompanying diskette. Tests that are included in that file are as follows:

5.1.15.1.1 Normal status

```
TEST_ID = "1.15.1.1.1";
TEST_NAME = "Write Invalid, IUT Memory Slave, Aligned Address";
TEST_ID = "1.15.1.1.2";
TEST_NAME = "Write Invalid, IUT Cache Slave, Aligned Address, EU to I";
TEST_ID = "1.15.1.1.3";
TEST_NAME = "Write Invalid, IUT Cache Slave, Aligned Address, SU to I";
TEST_ID = "1.15.1.1.4";
TEST_NAME = "Write Invalid, IUT Cache Slave, Aligned Address, EM to I";
TEST_ID = "1.15.1.1.5";
TEST_NAME = "Write Invalid, IUT Cache Slave, Invalidate to Read \ Modified";
TEST_ID = "1.15.1.1.6";
TEST_NAME = "Write Invalid, IUT Cache Slave, Copyback to Read Shared";
```

5.1.15.1.2 Transaction error (TE)

```
TEST_ID = "1.15.1.2.1";
TEST_NAME = "Write Invalid, IUT Slave, Illegal Transaction Flag";
TEST_ID = "1.15.1.2.2";
TEST_NAME = "Write Invalid, IUT Slave, Illegal Transaction Flag w/ SR* ";
```

```
TEST_ID = "1.15.1.2.3";  
TEST_NAME = "Write Invalid, IUT Slave, Illegal Intervention";  
TEST_ID = "1.15.1.2.4";  
TEST_NAME = "Write Invalid, IUT Slave, Illegal Intervention w/ TF*";  
TEST_ID = "1.15.1.2.5";  
TEST_NAME = "Write Invalid, IUT Slave, Illegal Intervention w/ SR* ";
```

5.1.15.1.3 Beat error (BE)

There are no tests for this category.

5.1.15.1.4 Wait (WT)

```
TEST_ID = "1.15.1.4.1";  
TEST_NAME = "Write Invalid, IUT Slave, Wait";
```

5.1.15.1.5 Busy (BS)

```
TEST_ID = "1.15.1.5.1";  
TEST_NAME = "Write Invalid, IUT Slave, Busy";
```

5.1.15.1.6 End of data (ED)

There are no tests for this category.

5.1.15.1.7 Broadcast (BC)

There are no tests for this category.

5.1.15.1.8 Transaction flag (TF)

There are no tests for this category.

5.1.15.1.9 Intervention (IV)

There are no tests for this category.

5.1.15.1.10 Split response (SR)

```
TEST_ID = "1.15.1.10.1";  
TEST_NAME = "Write Invalid, IUT Slave, Split Request";
```

5.1.15.2 IUT master

The tests for this subclause can be found in file "5_1_15_2.c" on the accompanying diskette. Tests that are included in that file are as follows:

5.1.15.2.1 Normal status

```
TEST_ID = "1.15.2.1.1";  
TEST_NAME = "Write Invalid, IUT Master, Aligned Address.";
```

5.1.15.2.2 Transaction error (TE)

```
TEST_ID = "1.15.2.2.1";
TEST_NAME = "Write Invalid, IUT Master, Illegal Intervention.";
TEST_ID = "1.15.2.2.2";
TEST_NAME = "Write Invalid, IUT Master, Illegal Intervention w/ TF* ";
TEST_ID = "1.15.2.2.3";
TEST_NAME = "Write Invalid, IUT Master, Illegal Intervention w/ SR* ";
TEST_ID = "1.15.2.2.4";
TEST_NAME = "Write Invalid, IUT Master, Illegal Intervention with SR* \
and TF* ";
TEST_ID = "1.15.2.2.5";
TEST_NAME = "Write Invalid, IUT Master, Illegal Transaction Flag";
TEST_ID = "1.15.2.2.6";
TEST_NAME = "Write Invalid, IUT Master, Illegal Transaction Flag \
with Split Response";
```

5.1.15.2.3 Beat error (BE)

There are no tests for this category.

5.1.15.2.4 Wait (WT)

```
TEST_ID = "1.15.2.4.1";
TEST_NAME = "Write Invalid, IUT Master, Wait";
```

5.1.15.2.5 Busy (BS)

```
TEST_ID = "1.15.2.5.1";
TEST_NAME = "Write Invalid, IUT Master, Busy.";
```

5.1.15.2.6 End of data (ED)

There are no tests for this category.

5.1.15.2.7 Broadcast (BC)

There are no tests for this category.

5.1.15.2.8 Transaction flag (TF)

There are no tests for this category.

5.1.15.2.9 Intervention (IV)

There are no tests for this category.

5.1.15.2.10 Split response (SR)

```
TEST_ID = "1.15.2.10.1";
TEST_NAME = "Write Invalid, IUT Master, Split Request.";
```

5.1.16 Copyback transaction

5.1.16.1 IUT slave

The tests for this subclause can be found in file “5_1_16_1.c” on the accompanying diskette. Tests that are included in that file are as follows:

5.1.16.1.1 Normal status

```
TEST_ID = "1.16.1.1.1";
TEST_NAME = "Copyback, IUT Memory Slave, Aligned Address";
```

5.1.16.1.2 Transaction error (TE)

```
TEST_ID = "1.16.1.2.1";
TEST_NAME = "Copyback, IUT Cache Slave, Illegal Intervention";
TEST_ID = "1.16.1.2.2";
TEST_NAME = "Copyback, IUT Slave, Illegal Intervention with TF* ";
TEST_ID = "1.16.1.2.3";
TEST_NAME = "Copyback, IUT Cache Slave, Illegal Intervention with SR* ";
TEST_ID = "1.16.1.2.4";
TEST_NAME = "Copyback, IUT Slave, Illegal Intervention with SR* and \
TF* ";
TEST_ID = "1.16.1.2.5";
TEST_NAME = "Copyback, IUT Slave, Illegal Split Response";
TEST_ID = "1.16.1.2.6";
TEST_NAME = "Copyback, IUT Slave, Illegal Split Response with TF*";
```

5.1.16.1.3 Beat error (BE)

There are no tests for this category.

5.1.16.1.4 Wait (WT)

```
TEST_ID = "1.16.1.4.1 ";
TEST_NAME = "Copyback, IUT Slave, Wait";
```

5.1.16.1.5 Busy (BS)

```
TEST_ID = "1.16.1.5.1";
TEST_NAME = "Copyback, IUT Slave, Busy";
```

5.1.16.1.6 End of data (ED)

There are no tests for this category.

5.1.16.1.7 Broadcast (BC)

There are no tests for this category.

5.1.16.1.8 Transaction flag (TF)

```
TEST_ID = "1.16.1.8.1";
TEST_NAME = "Copyback, IUT Cache Slave, Snarf";
```