

SEMI G91-0513

STANDARD TEST DATA FORMAT (STDF) MEMORY FAIL DATALOG

This Standard was technically approved by the Automated Test Equipment Global Technical Committee. This edition was approved for publication by the global Audits and Reviews Subcommittee on December 24, 2011. Available at www.semiviews.org and www.semi.org in May 2013.

NOTICE: This Standard or Safety Guideline has an Inactive Status because the conditions to maintain Current Status have not been met. Inactive Standards or Safety Guidelines are available from SEMI and continue to be valid for use.

1 Purpose

1.1 The purpose of this Standard is to provide a common format for memory fail datalog specification along with necessary synchronization information enabling an efficient dataflow for volume diagnostics applications for memories.

1.2 The purpose of this Standard is to provide a standard format to log electrical failure information during test for embedded as well as stand-alone memories. The Standard provides the definition of records and their use for storing failure information.

2 Scope

2.1 The scope of the proposed Standard is the memory failure information collected during electrical test for embedded as well as stand-alone volatile memories as shown Figure 1 below:

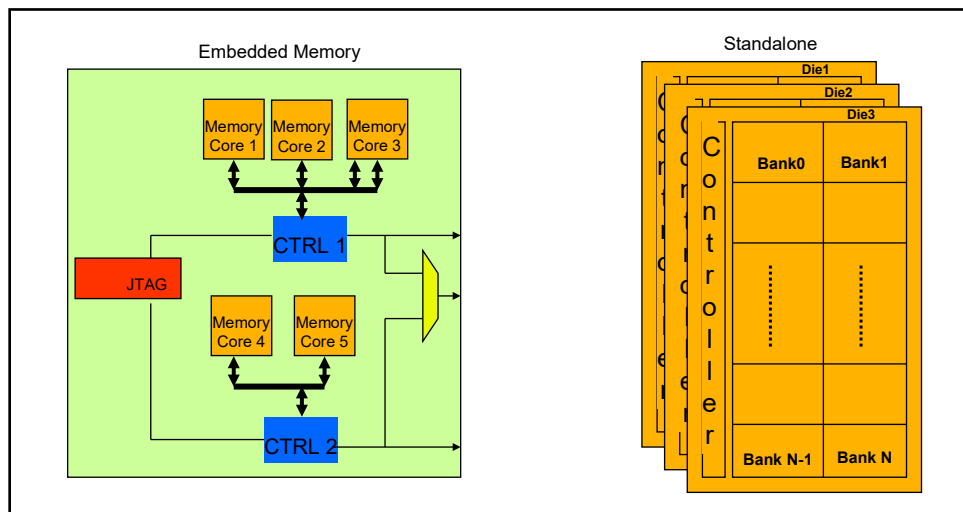


Figure 1
Supported Memory Types

2.2 The scope of this Standard is as follows:

2.2.1 Targeted Memories

- Embedded
 - Direct Access
 - BISTed
- Stand-Alone
 - Single Die

- Stacked Dice
- Datalog Format
 - Definition of records and their use for storing failure information
 - Data format for storing memory design information in the datalog
 - Definition of format for enabling data integrity and consistency checking
 - Reduction in data volume for storing a given amount of failure information

3 Limitations

3.1 While the proposed format can be used to store the datalog for the nonvolatile memories, the Standard did not explicitly target them. So, it may fall short on the requirements for nonvolatile memories.

3.2 The Standard is limited to store only logic design information in the format. The links to physical information is provided through the external file references.

4 Referenced Standards and Documents

4.1 *IEEE Standard*¹

IEEE 1149.1 — 1990 IEEE Standard Test Access Port and Boundary-Scan Architecture

4.2 *Other Documents*

STDF V4 — Standard Test Data Format Version 4.0

STDF V4-2007 — Standard Test Data Format Version 2007 – Scan Datalog Extension

NOTICE: Unless otherwise indicated, all documents cited shall be the latest published versions.

5 Terminology

5.1 *Abbreviations and Acronyms*

5.1.1 *ATE* — automated test equipment

5.1.2 *BIST* — built-in self test

5.1.3 *STDF* — standard test data format

5.2 *Definitions*

5.2.1 *built-in self test (BIST)* — a test methodology in which the stimulus generator and/or response analyzers is built into the same integrated circuit as the block that is tested.

5.2.2 *built-in self test (BIST) controller* — a logic block that coordinates the test of the memory block that it tests.

5.2.3 *column* — a series of bits that can be enabled for access using a column address.

5.2.4 *datalog* — the collection of information that is collected during the application of test. It consists of information collected from the device, the environment and the equipment that is used to apply the test.

5.2.5 *diagnosis* — a process of identifying the root cause of an observed incorrect response.

5.2.6 *direct access memory* — an embedded memory which can be accessed directly from the interface of the integrated circuit.

5.2.7 *embedded memory* — is a semiconductor memory that is embedded inside an integrated circuit along with other logic blocks.

5.2.8 *handler* — an equipment that is used to move the packaged part to and from the test bed during the test.

¹ Institute of Electrical and Electronics Engineers, 3 Park Avenue, 17th Floor, New York, NY 10016-5997, USA; Telephone: +1.212.419.7900, Fax: +1.212.752.4929, <http://www.ieee.org>

5.2.9 *march element* — sequence of read and write into the memory. It is used to excite a potential fault and observe corresponding response.

5.2.10 *memory bank* — a memory can be organized as set of sub-blocks. Each such block is called a bank.

5.2.11 *memory core* — a memory block that can be used in integrated circuits.

5.2.12 *memory instance* — a memory instance in an integrated circuit is an instance of memory block.

5.2.13 *memory test algorithm* — a sequence of march elements.

5.2.14 *nonvolatile memory* — memory that can retain its storage even when the power is turned off.

5.2.15 *probe card* — an interface to connect device pads on the wafer to channels of an ATE.

5.2.16 *prober* — an equipment that is used to move the wafer to the test bed for wafer testing.

5.2.17 *row* — a series of bits in a memory/memory bank that can be enabled for access using a row address.

5.2.18 *semiconductor memory* — a device that can store data.

5.2.19 *stand-alone memory* — a memory integrated circuit that contains only memory blocks.

5.2.20 *volatile memory* — a semiconductor memory which loses its data when power is turned off.

6 Standard Test Data Format (STDF) Record Structure

6.1 This section describes the basic STDF record structure. It describes the following general topics, which are applicable to all the record types:

- STDF record header
- Record types and subtypes
- Data type codes and representation
- Optional fields and missing/invalid data

6.2 STDF Record Header

6.2.1 Each STDF record begins with a record header consisting of the following three fields:

Table 1 STDF Record Header

<i>Field</i>	<i>Field Description</i>
REC_LEN	The number of bytes of data following the record header.
REC_TYP	An integer identifying a group of related STDF record types.
REC_SUB	An integer identifying a specific STDF record type within each REC_TYP group.

6.2.2 Record Types and Subtypes

6.2.2.1 The header of each STDF record contains a pair of fields called REC_TYP and REC_SUB. Each REC_TYP value identifies a group of related STDF record types. Each REC_SUB value identifies a single STDF record type within a REC_TYP group. The combination of REC_TYP and REC_SUB values uniquely identifies each record type. This design allows groups of related records to be easily identified by data analysis programs, while providing unique identification for each type of record in the file.

6.3 STDF Record Structure

6.3.1 The following table lists the meaning of the REC_TYP codes in STDF up to version V4-2007. The new record types added for memory fail data log will be described later in the document. The description for rest of these records can be found in STDF V4 and STDF V4-2007 specification documents [1,2].

Table 2 STDF V4 and V4-2007 Record Types

<i>REC_TYP</i>	<i>Code Meaning and STDF REC_SUB Codes</i>
0	Information about the STDF file 10 File Attributes Record (FAR) 20 Audit Trail Record (ATR) 30 Version Update Record (VUR)
1	Data collected on a per lot basis 10 Master Information Record (MIR) 20 Master Results Record (MRR) 30 Part Count Record (PCR) 40 Hardware Bin Record (HBR) 50 Software Bin Record (SBR) 60 Pin Map Record (PMR) 62 Pin Group Record (PGR) 63 Pin List Record (PLR) 70 Retest Data Record (RDR) 80 Site Description Record (SDR) 90 Pattern Sequence Record (PSR) 91 Name Map Record (NMR) 92 Cell Name Record (CNR) 93 Scan Structure Record (SSR) 94 Scan Chain Record (SCR)
2	Data collected per wafer 10 Wafer Information Record (WIR) 20 Wafer Results Record (WRR) 30 Wafer Configuration Record (WCR)
5	Data collected on a per part basis 10 Part Information Record (PIR) 20 Part Results Record (PRR)
10	Data collected per test in the test program 30 Test Synopsis Record (TSR)
15	Data collected per test execution 10 Parametric Test Record (PTR) 15 Multiple-Result Parametric Record (MPR) 20 Functional Test Record (FTR) 30 Scan Test Record (STR)
20	Data collected per program segment 10 Begin Program Section Record (BPS) 20 End Program Section Record (EPS)
50	Generic Data 10 Generic Data Record (GDR) 30 Datalog Text Record (DTR)
180	Reserved for use by Image software
181	Reserved for use by IG900 software

6.4 Data Type Support

6.4.1 The STDF specification uses a set of data type codes that are concise and easily recognizable. For example, R*4 indicates a REAL (float) value stored in four bytes. A byte consists of eight bits of data. For purposes of this Document,

the low order bit of each byte is designated as bit 0 and the high order bit as bit 7. The following table gives the complete list of STDF data type codes, as well as the equivalent C language type specifier.

Table 3 Data Types

<i>Code</i>	<i>Description</i>	<i>C Type Specifier</i>
C*12	Fixed length character string: If a fixed length character string does not fill the entire field, it must be left-justified and padded with spaces.	char[12]
C*n	Variable length character string: first byte = unsigned count of bytes to follow (maximum of 255 bytes)	char[]
S*n	Variable length character string: first two bytes = unsigned count of bytes to follow (maximum of 65535 bytes)	char[]
C*f	Variable length character string: string length is stored in another field	char[]
U*f	An unsigned integer whose size in bytes (f) is defined in separate field within the containing record or a separate but related record	
U*1	One byte unsigned integer	unsigned char
U*2	Two byte unsigned integer	unsigned short
U*4	Four byte unsigned integer	unsigned long
U*8	Eight byte unsigned integer	unsigned long
I*1	One byte signed integer	char
I*2	Two byte signed integer	short
I*4	Four byte signed integer	long
R*4	Four byte floating point	float
R*8	Eight byte floating point number	long float (double)
B*6	Fixed length bit-encoded data	char[6]
V*n	Variable data type field: The data type is specified by a code in the first byte, and the data follows (maximum of 255 bytes)	
B*n	Variable length bit-encoded field: First byte = unsigned count of bytes to follow (maximum of 255 bytes). First data item in least significant bit of the second byte of the array (first byte is count.)	char[]
D*n	Variable length bit-encoded field: First two bytes = unsigned count of bits to follow (maximum of 65,535 bits). First data item in least significant bit of the third byte of the array (first two bytes are count). Unused bits at the high order end of the last byte must be zero.	char[]
N*1	Unsigned integer data stored in a nibble. (Nibble = 4 bits of a byte). First item in low 4 bits, second item in high 4 bits. If an odd number of nibbles is indicated, the high nibble of the byte will be zero. Only whole bytes can be written to the STDF file.	char
kxTYPE	Array of data of the type specified. The value of 'k' (the number of elements in the array) is defined in an earlier field in the record. For example, an array of short unsigned integers is defined as kxU*2.	TYPE[]

6.5 Optional Fields and Missing/Invalid Data

6.5.1 Certain fields in STDF records are defined as optional. An optional field must be present in the record, but there are ways to indicate that its value is not meaningful, that is, that its data should be considered missing or invalid. STDF V4-2007 builds on mechanisms of STDF V4 for missing or invalid fields. It uses the same mechanism as the STDF V4 for indicating missing/invalid data as described below:

- Some optional fields have a predefined value that means that the data for the field is missing. For example, if the optional field is a variable-length character string, a length byte of 0 means that the data is missing. If the field is numeric, a value of -1 may be defined as meaning that the data is missing.
- For other optional fields, all possible stored values, including -1, are legal. In this case, the STDF specification for the record defines an Optional Data bit field. Each bit is used to designate whether an optional field in the record contains valid or invalid data. Usually, if the bit for an optional field is set, any data in the field is invalid and should be ignored.

Table 4 Missing/Invalid Field Specification

<i>Data Type</i>	<i>Missing/Invalid Data Flag</i>
Variable-length string	Set the length byte to 0
Fixed-length character string	Fill the field with spaces.
Fixed-length binary string	Set a flag bit in an Optional Data byte.
Time and date fields	Use a binary 0.
Signed and unsigned integers and floating point values	Use the indicated reserved value or set a flag bit in an Optional Data byte. ^{#1}

#1 However for the omission of the optional fields the new standard uses explicit control flags/bits to indicate the absence of optional fields and the first missing optional field does not mean that the rest of the optional fields in the records are missing as well.

6.6 Continuation of Records

6.6.1 The amount of data required for some of the new record types will occasionally exceed what can be accommodated in a single 65k record. To facilitate this expanded data volume the concept of 'continuation records' is introduced. Any number of continuation records may be added after an initial record type.

6.6.2 The continuation mechanism is implemented by adding a CONT_FLG (B*1) immediately after the Record SubType field. This flag is set to 1 if there are follow on continuation records after the current one. The last record in this sequence of records will have the flag set to 0. In addition for each field that extends over multiple records, a local count is added for that field to indicate the entries that are present in the current record.

6.7 STDF Record Types

6.7.1 This section contains the definitions for the STDF record types. The following information is provided for each record type:

6.7.1.1 A statement of function: how the record type is used in the STDF file.

6.7.1.2 A table defining the data fields: first the standard STDF header, then the fields specific to this record type. The information includes the field name, the data type (see the previous section for the data type codes), a brief description of the field, and the flag to indicate missing or invalid data (see the previous section for a discussion of optional fields).

6.7.1.3 Any additional notes on specific fields.

6.7.1.4 Possible uses for this record type in data analysis reports. Note that this entry states only where the record type can be used. It is not a statement that the reports listed always use this record type, even if Teradyne has written those reports. For definitive information on how any data analysis software uses the STDF file, see the documentation for the data analysis software.

6.7.1.5 Frequency with which the record type appears in the STDF file: for example, once per lot, once per wafer, one per test, and so forth.

6.7.1.6 The location of the record type in the STDF file. See the note on 'initial sequence'.

6.8 Initial Record Sequences

6.8.1 For several record types, the 'Location' says that the record must appear 'after the initial sequence.' The phrase 'initial sequence' refers to the records that must appear at the beginning of the STDF file. The requirements for the initial sequence are as follows:

6.8.1.1 Every file must contain one file attributes record (FAR), one version update record (VUR), one master information record (MIR), one or more part count records (PCR), and one master results record (MRR). All other records are optional.

6.8.1.2 The first record in the STDF file must be the FAR.

6.8.1.3 If one or more audit trail records (ATRs) are used, they must appear immediately after the FAR.

6.8.1.4 VUR which is a required record for STDF V4-2007 must appear after the last ATR and before the MIR.

6.8.1.5 The MIR must appear in every STDF file. Its location must be after the FAR, ATRs (if ATRs are used) and VUR.

6.8.1.6 If the retest data record (RDR) is used, it must appear immediately after the MIR.

6.8.1.7 If one or more site description records (SDRs) are used, they must appear immediately after the MIR and RDR (if the RDR is used).

6.8.2 Given these requirements, every STDF V4-2007 must contain one of these initial sequences:

- FAR – VUR – MIR
- FAR – ATRs – VUR – MIR
- FAR – VUR – MIR – RDR
- FAR – ATRs – VUR – MIR – RDR
- FAR – VUR – MIR – SDRs
- FAR – ATRs – VUR – MIR – SDRs
- FAR – VUR – MIR – RDR – SDRs
- FAR – ATRs – VUR – MIR – RDR – SDRs

6.8.2.1 All other STDF record types appear after the initial sequence.

7 Fail Datalog Requirements

7.1 Figure 2 shows the information objects/classes that need to be supported in the Standard to enable an efficient volume diagnosis flow. This object model has been created through the analysis of the design to test to design flow for volume diagnosis. It allows the design information to be carried forward into test environment and the test information to be carried back to analysis tools. The proposed Standard maps these object onto various STDF records.

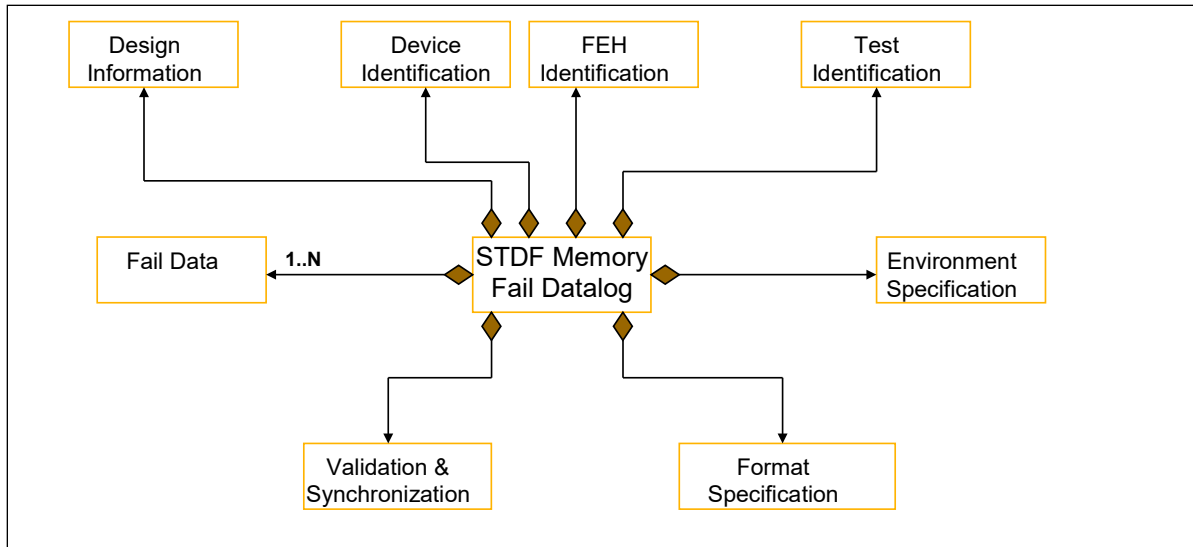


Figure 2
Object Model

8 STDF Memory Fail Datalog Proposal

8.1 The STDF memory fail datalog proposal is an extension to the current STDF V4 and V4-2007 specifications. Records currently defined in these specifications would continue to be used to datalog information about equipment setup, device information, and parametric, functional and scan test results. This Standard defines the additional records shown in Table 5 to provide an improved means of datalogging memory-specific electrical fail data. The new records are intended to co-exist with existing records in STDF output files.

Table 5 New Memory Fail Data Records

Type	Subtype	Record Acronym	Description
0			Information about STDF file
	30	VUR	Version update record
1			Data collection on a per lot basis
	95	ASR	Algorithm specification record
	96	FSR	Frame specification record
	97	BSR	Bit stream specification record
	99	MSR	Memory structure record
	100	MCR	Memory controller record
	101	IDR	Instance description record
	102	MMR	Memory model record
15			Data collected per test execution
	40	MTR	Memory test record

8.2 Design Information

8.2.1 The design information objects contain the information about the design that needs to be passed around the design-to-test-design loop to enable proper diagnosis and data translations. In particular, for the embedded memory architecture example shown in Figure 1, the design information object includes information about the controllers, the instances accessed by each controller, the logical memory (e.g., rows/columns, banks), the ports, and orientation of the memory. It also contains a link to the layout file to enable logical to physical mapping for diagnosis and localization

of failures. For stand-alone memories the design information only contains information about the memory model and its orientation. This design information object in the object model is implemented using four STDF records: MSR, MCR, IDR, and MMR (see Table 4). A single MSR record contains the top level information about the chip/part. A set of memory controller records (MCR) contains the information about the controllers in the designs. Each MCR represents a single controller and contains references to the memory instances it controls. Each memory instance is described using an IDR record which can be linked to their model information. Several examples how these records can be used to store design information are provided in the Appendix.

8.3 VUR

8.3.1 *Function* — This record is used to indicate the version of STDF update that is being used for the STDF file.

NOTE 1: This record was introduced as part of the STDF V4-2007 Scan Fail Datalog standard and is described here just for reference and because it is required to use this extension.

Table 6 VUR

<i>Field Name</i>	<i>Data Type</i>	<i>Field Description</i>	<i>Missing/Invalid Data Flag</i>
REC_LEN	U*2	Bytes of data following header	
REC_TYPE	U*1	Record type (0)	
REC_SUB	U*1	Record subtype (30)	
UPD_CNT	U*1	Count (k) of entries in the UPD_NAME field	
UPD_NAME	kxC*n	Update version name	k=0

8.3.2 Field Description

8.3.2.1 *UPD_NAME* — This field will contain an array of the version update names. For example the entry for the new memory standard name will be stored as ‘Memory:2010.1’ string in the UPD_NAME field.

8.3.3 *Frequency* — Once per STDF file.

8.3.4 *Location* — This record occurs between FAR and MIR. In case the STDF contains an ATR, it follows the ATR record(s) otherwise, it appears right after the FAR record.

8.4 MSR

8.4.1 *Function* — This record is the top level record for storing memory design information. It supports both the direct access memories as well as the embedded memories controlled by embedded controllers. For embedded memories it contains the references to the controllers and for direct access memories it contains the references to the memory instances.

Table 7 MSR

<i>Field Name</i>	<i>Type</i>	<i>Field Description</i>	<i>Missing/Invalid</i>
REC_LEN	U*2	Bytes of data following header	
REC_TYP	U*1	Record type(1)	
REC_SUB	U*1	Record subtype(99)	
NAME	C*n	Name of the design under test	Length Byte = 0
FILE_NAM	C*n	name of the file containing design information	Length Byte = 0
CTRL_CNT	U*2	Count (k) of controllers in the design	
CTRL_LST	kxU*2	Array of controller record indexes	CTRL_CNT = 0;
INST_CNT	U*2	Count(m) of top level memory instances	
INST_LST	mxU*2	Array of instance description record indexes	INST_CNT=0

8.4.2 Field Description

8.4.2.1 *NAME* — This is a place holder for the design name. It is optional but is recommended.

8.4.2.2 *FILE_NAM* — Name of the file that has information about the design. This field allows one to link STDF to information in the files outside of STDF. User can decide how much design information is stored in the STDF file itself and how much is stored in the external file that is linked.

8.4.2.3 *CTRL_CNT* — This field is used to record the number of controllers in an embedded memory design. It also indicates the number of entries in the *CTRL_LST* field.

8.4.2.4 *CTRL_LST* — *CTRL_LST* contains an array of MCR indexes. The size of the array is defined by *CTRL_CNT*. The MCR index stored in this array are the indexes defined in the MCR's *CTRL_IDX* field. The order of the entries in the array is not significant.

8.4.2.5 *INST_CNT* — This field is used to record the number of top level memory instances that are directly accessible from device I/O pins and not controlled by an embedded controller.

8.4.2.6 *INST_LST* — *INST_LST* contains an array of instance description record (IDR) record indexes. The size of the array is defined by the *INST_CNT* field above. The IDR indexes stored in this array are the indexes defined in the IDR's *INST_IDX* field.

8.4.3 *Frequency* — One for each STDF file for a design.

8.4.4 *Location* — It can occur any time after RDR if no SDR(s) are present, otherwise after all the SDRs. This record must occur before MCRs and IDRs

8.5 MCR

8.5.1 *Function* — This record is used to store information about an embedded memory controller in a design. There is one MCR record in an STDF file for each controller in a design. These records are referenced by the top level MSR through its *CTRL_LST* field.

Table 8 MCR

<i>FieldName</i>	<i>Type</i>	<i>Field Description</i>	<i>Missing/Invalid</i>
REC_LEN	U*2	Bytes of data following header	
REC_TYP	U*1	Record type(1)	
REC_SUB	U*1	Record subtype(100)	
CTRL_IDX	U*2	Index of this memory controller record	
CTRL_NAM	C*n	Name of the controller	Length Byte = 0 (Not recommend to be missing though)
MDL_FILE	C*n	Pointer to the file describing model (e.g., CTL)	Length Byte = 0
INST_CNT	U*2	Count (k) of INST_IDX array	
INST_LST	Kx U*2	Array of memory instance indexes	

8.5.1.1 *CTRL_IDX* — This is a unique numeric identifier for this specific memory controller. This is used by other records types to reference this specific memory controller.

8.5.1.2 *CTRL_NAM* — The name of the controller. This is a symbolic name that is recommended to match with controller name in the design. The name could be full hierarchical.

8.5.1.3 The name of the file that contains the model information for this memory controller. This is an optional field if user wants to add a reference to an external file that describes the memory controller. There is no format specified for that file by this Standard. It is up to the environment to provide the readers to read such file in the target application. The file name could be local or hierarchical.

8.5.1.4 *INST_CNT* — The number of memory instances that are controlled by this controller. It also indicates the size of the *INST_IDX* array.

8.5.1.5 *INST_LST* — *INST_LST* contains an array of IDR indexes whose size is defined by *INST_CNT*. The IDR index is defined in the IDR's *INST_IDX* field. The order of the entries in the array is not significant.

8.5.2 *Frequency* — Once per controller in the design.

8.5.3 *Location* — It can occur after all the MSRs and before IDRs.

8.6 IDR

8.6.1 *Function* — This record is used to store the information for a memory instance within a design. It contains a reference to the model records which define the design information for this specific memory instance.

Table 9 IDR

<i>FieldName</i>	<i>Type</i>	<i>Field Description</i>	<i>Missing/Invalid</i>
REC_LEN	U*2	Bytes of data following header	
REC_TYP	U*1	Record type(1)	
REC_SUB	U*1	Record subtype(101)	
INST_IDX	U*2	Unique index of this instance definition record	
INST_NAM	C*n	Name of the instance	Length Byte = 0 (Not recommend to be missing though)
REF_COD	U*1	0-Wafer (Notch based), 1-Pkg (Package reference designator)	
ORNT_COD	C*2	Orientation of the instance (two characters)	
MDL_FILE	C*n	Pointer to the file describing model (e.g., CTL)	Length Byte = 0
MDL_REF	U*2	Reference to the model record for this instance	0; Model record 0 is undefined

8.6.1.1 *INST_IDX* — This is a unique numeric identified for this specific memory instance definition. This is used by other records types to reference this specific memory instance definition.

8.6.1.2 *INST_NAM* — The name of the instance in the design hierarchy.

8.6.1.3 *ORNT_COD* — This field contains the information about the orientation of the memory instance. It is a two character code as described below:

- Each memory orientation is represented by a 2 character code
- The 1st character represents the direction of increasing column direction
- The 2nd character represents the direction of increasing row direction
- 4 characters are used to describe the direction
- U → Up
- D → Down
- L → Left
- R → Right

8.6.1.3.1 The naming of these directional indicators is based on notch down placement for wafers and rotates with other notch placement. For packaged devices it is with reference to the Pad 0 location of the Die/Core (e.g., 'U' direction for columns means that the columns addresses are increasing from left to right if the wafer is looked at in notch right setting).

8.6.1.4 *MDL_FILE* — The name of the file that contains the model information for this memory instance. This is an optional field if the model information is not contained in STDF file itself in a model record. The file name could be local or hierarchical.

8.6.1.5 *MDL_REF* — This is an optional field only used if the memory model information for this memory instance is contained within the STDF file in an MMR. MDL_REF contains an index value identifying the associated MMR record containing the memory model information. The memory model record (MMR) index is defined in the MMR's MDL_IDX field.

8.6.2 *Frequency* — Once per memory instance.

8.6.3 *Location* — It can occur after all the MCRs and before MMRs.

8.7 MMR

8.7.1 *Function* — This record is used to store the memory model information in STDF. One record per model is used. The record allows storing the logic level information of the model. It does not have any fields to store the physical information except height and width. The physical information can be optionally linked to the record through a reference to the file.

Table 10 MMR

<i>FieldName</i>	<i>Type</i>	<i>Field Description</i>	
REC_LEN	U*2	Bytes of data following header	
REC_TYP	U*1	Record type(1)	
REC_SUB	U*1	Record subtype(102)	
MDL_IDX	U*2	Unique index of this MMR	
MDL_NAM	C*n	Name of the model	Length Byte = 0 (Not recommend to be missing though)
DIMS_CNT	U*2	Count (k) of dimensions	
DIMS_NAM	kxC*n	Array of names of the dimension axis	
DIMS_TYP	kxU*1	Array of types (e.g., BANK, COL, COL_MUX, etc.)	
DIMS_SIZ	kxU*8	Array of the size of each dimension	
DIMS_DIR	kxU*1	Array of direction of each dimension	
DIMS_OCD	kxC*2	Array of orientation code for banks	Length Byte = 0
PORT_CNT	U*2	Count (m) of ports in the memory	
PORT_TYP	mxC*n	Enumeration of type (e.g., ADDR/DATA/CTRL)	
PORT_DIR	mxC*n	In/Out/InOut	
PORT_NAM	mxC*n	Type and name string of the port	
PORT_SIZ	mxU*1	The size of port in number of bits	
MEM_HT	C*n	Physical height of the memory (with units)	Length Byte = 0
MEM_WID	C*n	Physical width of the memory (with units)	Length Byte = 0
MDI_FIL	C*n	Name of the file with the physical information for this memory	Length Byte = 0

8.7.1.1 *MDL_IDX* — This is the unique identifier for this model record that is used as a reference by other records for example, the MDL_REF of the IDR to link to a model record. Note: MDL_IDX must be >0. ID 0 is undefined.

8.7.1.2 *MDL_NAM* — This is the name of this memory model.

8.7.1.3 *DIMS_CNT* — The is the count of the dimensions (e.g., row, column, etc.) that a memory mode has.

8.7.1.4 *DIMS_NAM* — This is an array of the names for memory model dimensions. Instead of providing a fixed dimension names this allows the user the flexibility in terms of number and names of the dimensions.

8.7.1.5 *DIMS_TYP* — This is an array of type of the dimension for the dimensions in the model. The values of the supported types are as follows:

- 0 = Row
- 1 = Column
- 3 = Bank
- 4 = Column MUX
- 5–127 = Reserved for future extensions
- 128–255 = User defined

8.7.1.6 *DIMS_SIZ* — This is an array of the size specifications for each dimension.

8.7.1.7 *DIMS_DIR* — This is an array of the direction for each dimension w.r.t. origin of the core as defined by the design. The possible values of the directions are:

- 0 = Not present
- 1 = Up
- 2 = Down
- 3 = Left
- 4 = Right
- 5 = CenterOut

8.7.1.8 *DIMS_OCD* — This an array of orientation codes of the memory bank within the memory. Only supposed to be used in case of *DIMS_TYP* being Bank.

8.7.1.9 *PORT_CNT* — The number of port in the memory model. It also indicates the size of the array for *PORT_TYP*, *PORT_DIR*, *PORT_NAM* and *PORT_SIZ* fields.

8.7.1.10 *PORT_TYP* — This is an array of types of the ports in the model (e.g., DATA, ADDRESS, CONTROL, etc.). The type is indicated by a string values. The predefined types are as follows, the other can be added as needed by the environment:

- ADDRESS: Indicates an address type port
- DATA: Indicates a DATA type port
- CONTROL: Indicates a control type port
- CLOCK: Indicates a clock type port.

(User can add optional additional port types if needed)

8.7.1.10.1 *PORT_DIR* — This is an array of direction for each port. The possible values are:

- In: Input only
- Out: Output only
- InOut: Bi-Directional

8.7.1.11 *PORT_NAM* — The array of names of the ports.

8.7.1.12 *PORT_SIZ* — The array of port size in number of bits.

8.7.1.13 *MEM_HT* — The character string representing the physical height of the memory. The value is specified with it units in the character string.

8.7.1.14 *MEM_WID* — The character string representing the physical height of the memory. The value is specified with it units in the character string.

8.7.1.15 *MDL_FILE* — This field is used to add a link to external file containing model information. This is an optional field and can co-exist with the rest of the data in the record. Please note that this field just allows one to link STDF to external file. STDF specification neither dictates the format of this file nor prescribes the tools that can read this file. The file name can be relative or absolute hierarchical file name.

8.7.2 *Frequency* — Once per memory model.

8.7.3 *Location* — It can occur after all the IDRs and before any frame specification records (FSRs), bit stream specification records (BSRs) and any test specific records (e.g., parametric test record (PTR), functional test record (FTRs), scan test record (STR), and memory test record (MTRs).

8.8 Algorithm Specification Record (ASR)

8.8.1 *Function* — This record is used to store the algorithms that are applied during a memory test.

Table 11 ASR

<i>FieldName</i>	<i>Type</i>	<i>Field Description</i>	<i>Missing/Invalid</i>
REC_LEN	U*2	Bytes of data following header	
REC_TYP	U*1	Record type(1)	
REC_SUB	U*1	Record sub-type(95)	
ASR_IDX	U*2	Unique identifier for this ASR record	
STRT_IDX	U*1	Cycle start index flag	
ALGO_CNT	U*1	count (k) of algorithms descriptions	
ALGO_NAM	kxC*n	Array of names of the algorithm	
ALGO_LEN	kxC*n	Array of complexity of algorithm (e.g., 13N)	
FILE_ID	kxC*n	Array of name of the file with algorithm description	
CYC_BGN	kxU*8	Array of starting cycle number for the algorithms	
CYC_END	kxU*8	Array of end cycle number for the algorithm	

8.8.1.1 *ASR_IDX* — This field stores the unique identifier for this record that is used to reference it in other records.

8.8.1.2 *STRT_IDX* — This is a flag that indicates if the cycle numbers for this algorithm starts with 0 or 1. If this field contains a 0 then it starts from 0 else it start with 1.

8.8.1.3 *ALGO_CNT* — The number of algorithms stored in this record. This field also indicates the size of the arrays for ALGO_NAM, ALGO_LEN, FILE_ID, CYC_BGN and CYC_END fields.

8.8.1.4 *ALGO_NAM* — The array of names of the algorithms.

8.8.1.5 *ALGO_LEN* — The array of algorithms complexities.

8.8.1.6 *FILE_ID* — The array of file names that contain the description of the algorithm. The name could be of a library file that contains multiple algorithms. In that case the name of the algorithm in the ALGO_NAM field is used to identify the specific algorithm in that library.

8.8.1.7 *CYC_BGN* — The array of starting cycle of the algorithm in the test.

8.8.1.8 *CYC_END* — The array of ending cycle of the algorithms.

8.8.2 *Frequency* — Once per unique memory test specification.

8.8.3 *Location* — It can occur after all the MMRs and before any test specific records (e.g., PTR, FTRs, STR and MTRs).

8.9 Frame Specification Record (FSR)

8.9.1 *Function* — FSR is used to define a frame structure that can be used to store the fail data in a frame format. In most of the embedded memory test architecture available in the industry, the data is communicated from the BIST