Background Statement for SEMI Draft Document 4782C
NEW STANDARD: STANDARD TEST DATA FORMAT (STDF) MEMORY FAIL DATALOG

Note: This background statement is not part of the balloted item. It is provided solely to assist the recipient in reaching an informed decision based on the rationale of the activity that preceded the creation of this document.

Note: Recipients of this document are invited to submit, with their comments, notification of any relevant patented technology or copyrighted items of which they are aware and to provide supporting documentation. In this context, "patented technology" is defined as technology for which a patent has issued or has been applied for. In the latter case, only publicly available information on the contents of the patent application is to be provided.

Background

Yield learning in modern technologies requires fail data logging from the scan and memory structural tests to gain insight into the failing location inside a chip. Currently there is no standard format to store the fail data in an efficient way. A group of more than 20 companies from ATE, EDA, Semiconductor and Yield Management companies has been working to enhance the Standard Fail Datalog Format (STDF) V4 to enable efficient fail datalog for scan and memory fails. This ballot describes the proposed memory fail datalog format.

Ballot History

Draft Document 4782C is the latest attempt by the STDF Task Force toward an approved SEMI Standard in this area. The first ballot, 4782, failed to receive sufficient return of votes during the Cycle 6, 2010 voting period, while ballot 4782A also failed to receive sufficient return of votes during the Cycle 1, 2011 voting period. Ballot 4782B received sufficient return of votes, but was failed at SEMICON West 2011 (July 13) due to persuasive negatives received. The task force has reviewed and considered the voting responses received and has submitted ballot 4782C for voting.

Review and Adjudication Information

	Task Force Review	**Committee Review & Adjudication**
Group:	STDF Task Force	North America Automated Test Equipment Committee
Date:	Thursday, October 6, 2011	Wednesday, October 26, 2011
Time & Timezone:	9:00 AM to 10:00 AM, Pacific Time	4:30 PM to 6:00 PM, Pacific Time
Location:	Via teleconference/ web meeting *Please contact Paul Trio for meeting bridge information.*	SEMI Headquarters 3081 Zanker Road
City, State/Country:	Via teleconference / web meeting	San Jose, California / U.S.A.
Leader(s):	Ajay Khoche (Khoche Consulting Services)	Ajay Khoche (Khoche Consulting Services)
Standards Staff:	Paul Trio (SEMI NA) 408.943.7041 ptrio@semi.org	Paul Trio (SEMI NA) 408.943.7041 ptrio@semi.org

This meeting's details are subject to change, and additional review sessions may be scheduled if necessary. Contact SEMI Standards staff for confirmation.

Telephone and web information will be distributed to interested parties as the meeting date approaches. If you will not be able to attend these meetings in person but would like to participate by telephone/web, please contact Standards staff.

SEMI Draft Document 4782C
NEW STANDARD: STANDARD TEST DATA FORMAT (STDF) MEMORY FAIL DATALOG

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 standalone 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 standalone volatile memories as shown Figure 1 below:

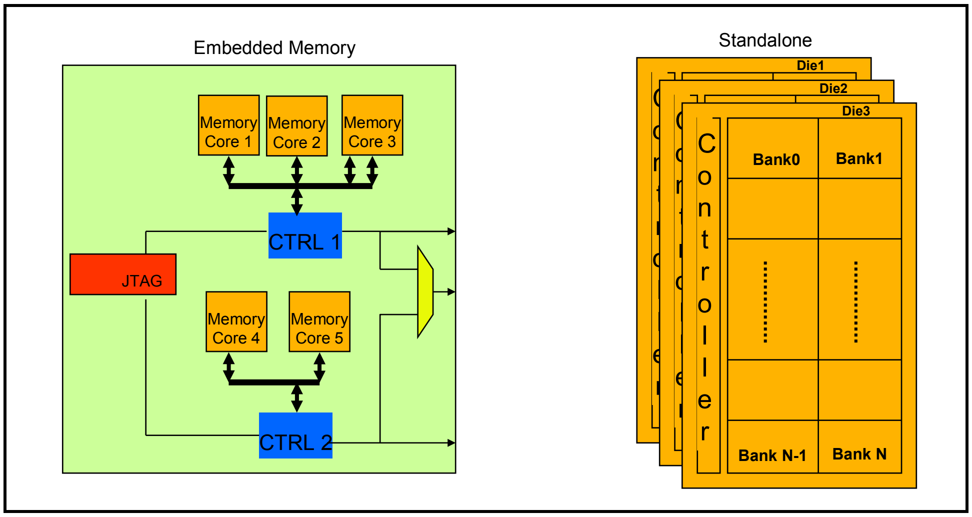


Figure 1
Supported Memory Types

The scope of this standard is as follows:

- Targeted Memories
  - Embedded
    - Direct Access
    - BISTed
  - Standalone
    - 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 non-volatile memories, the standard did not explicitly target them. So, it may fall short on the requirements for non-volatile 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 Documents

4.1 *SEMI Standards*

None

4.2 *IEEE Standards*[1]

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

4.3 *Other References*

Standard Test Data Format Version 4.0
Standard Test Data Format Version 2007 (STDF V4-2007) – Scan datalog extension.

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 *BIST Controller* — a logic block that coordinates the test of the memory block that it tests.

5.2.2 *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.

5.2.3 *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.

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 *non-volatile memory* — memory that can retain it 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.

[1] Institute of Electrical and Electronics Engineers
IEEE Operations Center, 445 Hoes Lane, P.O. Box 1331, Piscataway, New Jersey 08855-1331, USA. Telephone: 732.981.0060; Fax: 732.981.1721
Website: www.ieee.org

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

5.2.19 *standalone memory* — a memory integrated circuit that contains only memory blocks.

5.2.20 *volatile memory* — a semiconductor memory which looses it data when power is turned off.

6 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

Field	*Field Description*
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 upto 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

REC_TYP	*Code Meaning and STDF REC_SUB Codes*
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

Code	*Description*	*C Type Specifier*
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 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

Data Type	*Missing/Invalid Data Flag*
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 is 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 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 File Attributes Record (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 Version Update Record (VUR) which is a required record for STDF V4 – 2007 must appear after the last ATR and before MIR

6.8.1.5 The Master Information Record (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 though 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.

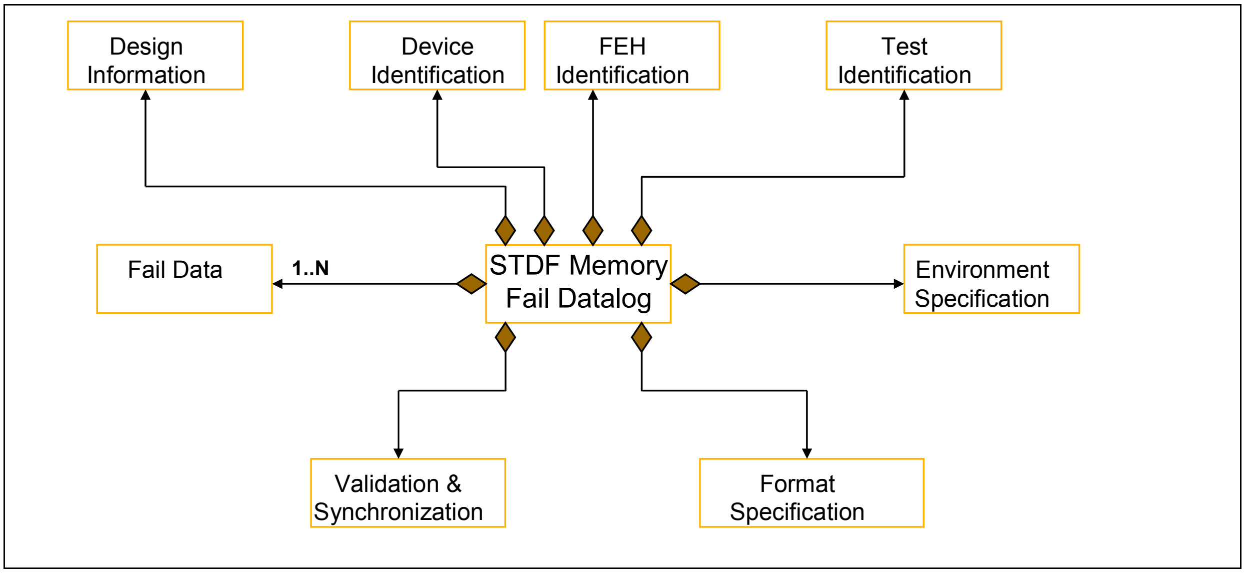


Figure 2
Object Model

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	*Sub Type*	*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 standalone 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 Record*

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

NOTE: 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 Record

Field Name	*Data Type*	*Field Description*	*Missing/Invalid Data Flag*
REC_LEN	U*2	Bytes of data following header	
REC_TYPE	U*1	Record Type (0)	
REC_SUB	U*1	Record sub-type (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 File Atrribute Record (FAR) and Master Information Record (MIR). In case the STDF contains an Audit Trail Record, it follows the ATR record(s) otherwise, it appears right after the FAR record.

8.4 *Memory Structure Record*

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 Memory Structure Record (MSR)

Field Name	*Type*	*Field Description*	*Missing/ Invalid*
REC_LEN	U*2	Bytes of data following header	
REC_TYP	U*1	Record Type(1)	
REC_SUB	U*1	Record Sub-type(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 *CTLR_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 IO pins and not controlled by an embedded controller.

8.4.2.6 *INST_LST* — INST_LST contains an array of 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 define din 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 anytime after Retest Data Record (RDR) if no Site Description Record(s) are present, otherwise after all the SDRs. This record must occur before Memory Controller Records (MCRs) and Instance Description Records (IDRs)

8.5 *Memory Controller Record (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 Memory Structure Record (MSR) through its CTRL_LST field.

Table 8 Memory Controller Record (MCR)

FieldName	*Type*	*Field Description*	*Missing/Invalid*
REC_LEN	U*2	Bytes of Data following header	
REC_TYP	U*1	Record Type(1)	
REC_SUB	U*1	Record Sub-type(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 want to add a reference to an external file that describes the memory controller. There is not format specified for that file by this standard. It is up to the environment to provided 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 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 Memory Structure Records(MSRs) and before Instance Description Records (IDRs)

8.6 *Instance Description Record*

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 Instance Description Record (IDR)

FieldName	*Type*	*Field Description*	*Missing/Invalid*
REC_LEN	U*2	Bytes of Data following header	
REC_TYP	U*1	Record Type(1)	
REC_SUB	U*1	Record Sub-type(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)

LETTER (YELLOW) BALLOT

FieldName	*Type*	*Field Description*	*Missing/Invalid*
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 record. MDL_REF contains an index value identifying the associated MMR record containing the memory model information. The 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 Memory Controller Records(MCRs) and before Memory Model Records (MMRs)

8.7 *Memory Model Record*

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 Memory Model Record (MMR)

FieldName	*Type*	*Field Description*	
REC_LEN	U*2	Bytes of Data following header	
REC_TYP	U*1	Record Type(1)	
REC_SUB	U*1	Record Sub-type(102)	
MDL_IDX	U*2	Unique Index of this Memory Model Record	

FieldName	*Type*	*Field Description*	
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 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 the 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
MDl_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 e.g. the MDL_REF of the Instance Description Record (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 rientation 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]

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 Instance Description Records(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*

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

Table 11 Algorithm Specification Record (ASR) Record

FieldName	*Type*	*Field Description*	*Missing/Invalid*
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 Name 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_INDX* — 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 he 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 Memory Model Records(MMRs) and before any Test specific records e.g. Parametric Test Record (PTR), Functional Test Record (FTRs), Scan Test Record (STR) and Memory Test Record (MTRs).

8.9 *Frame Specification Record*

8.9.1 Function: Frame Specification Record (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 controllers to ATE in a serial frame format. Each vendor has its own frame format. So to deal with different frame format from various vendors the FSR allows encapsulating one or more specific frame definitions used within the STDF file.

Table 12 Frame Specification Record (FSR)

FieldName	*Type*	*Field Description*	*Missing/Invalid*
REC_LEN	U*2	Bytes of Data following header	
REC_TYP	U*1	Record Type(1)	
REC_SUB	U*1	Record Sub-type(96)	
FRM_IDX	U*2	Unique ID for this frame specification record	
FLD_CNT	U*2	Count (k) of fields in the frame	
FLD_TYP	kxU*2	Field type; From a pre-defined list	
FLD_NAM	kxC*n	Field Name	
FLD_SIZE	kxU*2	field width in number of bits	

8.9.1.1 *FRM_IDX* — This field stores the unique identifier for frame definition that is used by other records to reference this frame definition.

8.9.1.2 *FLD_CNT* — The number of fields in the frame definition. This field also indicates the size of arrays in FLD_TYP, FLD_NAM, FLD_SIZE and DEF_VAL fields.

8.9.1.3 The following three fields i.e. FLD_TYP, FLD_NAM and FLD_SIZE are synchronized with each other i.e. for given index the FLD_TYP contains the type of the field, the entry at that index in FLD_NAM contains the name of the corresponding field and the entry in FLD_SIZE array in that position contains the size description of the field.

8.9.1.4 *FLD_TYP* — The array of types for each field in the frame. The possible values of the types are as follows:

0 — Controller
1 — Memory
2 — Port
3 — Algorithm
4 — March element
5 — March Step
6 — Logical Address
7 — Pin No
8 — Cycle No
9 — Write Port
10 — Write Port logic address
11 — Expected data
12 — Measured data

13 — Failed data
14 — Physical Column Address
15 — Physical Row Address
16 — Physical Bank Address
17 — Physical Bit Position
18 — Optional data

8.9.1.5 FLD_NAM: The array of symbolic names of the fields

8.9.1.6 FLD_SIZE: The array of the size of fields in number of bits.

8.9.2 Frequency: Once per memory Algorithm.

8.9.3 Location: It can occur after all the Memory Model Records(MMRs) and before any Test specific records e.g. Parametric Test Record (PTR), Functional Test Record (FTRs), Scan Test Record (STR) and Memory Test Record (MTRs).

8.10 *Bitstream Specification Record (BSR)*

8.10.1 Function: This record is used to enable string bit stream data from the memories. This record defines the format of the bit stream in which the data can be recorded in Memory Test Record (MTR). The bit streams are stored as stream of clusters for compaction. i.e. only the data words that have meaningful information are stored in the stream. Each cluster is defined as the starting address where the meaningful information starts followed by the count of words with meaningful information followed by the words themselves.

Table 13 BitStream Specification Record (BSR)

Field Name	*Type*	*Field Description*	*Missing/Invalid*
REC_LEN	U*2	Bytes of Data following header	
REC_TYP	U*1	Record Type(1)	
REC_SUB	U*1	Record Sub-type(97)	
BSR_IDX	U*2	Unique ID this Bit stream specification	
BIT_TYP	U*1	Meaning of bits in the stream	
ADDR_SIZ	U*1	Address field size (U1, U2, U4 or U8), values other than 1,2,4 & 8 are illegal(unused bits in each word are at lower bit locations)	
WC_SIZ	U*1	Word Count Field Size (U1, U2, U4, or U8) values other than 1,2,4 & 8 are illegal	
WRD_SIZ	U*2	Number of bits used in the word field (unused bits in each word are at lower bit locations)	

8.10.1.1 *BSR_IDX* — This field stores the unique identifier for frame definition that is used by other records to reference this bit stream definition.

8.10.1.2 *BIT_TYP* — This field indicates the meanings of the bits in the bit stream where bits are actual data, exp ecteddata, fails type. The allowed values are as follows:

0 — Actual/Measured data:
0 bit in log → data 0
1 bit in log → data 1
1 — Fail data:
0 bit in log → no fail;
1 bit in log → bit failed
2 — Expected data:
0 bit in log → data 0
1 bit in log → data 1

8.10.1.3 *ADDR_SIZ* — This is the size of address field in the bit stream number of bytes. The allowed values are 1,2,4, and 8 depending on the size of the memory. The user should use a value that is large enough to hold the address bits of the memory. e.g. 2 bytes for memories of size 64k or lower.

8.10.1.4 *WC_SIZ* — This is the size of word count field in the frame. The word count field can be U1, U2, U4 or U8 type depending on how many consecutive words may be stored in the stream, worst case it will be of the size of the address.

8.10.1.5 *WRD_SIZ* — Size of each data word in bits. This value is used to extract the word values from the bit stream.

8.10.2 Frequency: Once per memory Algorithm.

8.10.3 Location: It can occur after all the Memory Model Records(MMRs) and before any Test specific records e.g. Parametric Test Record (PTR), Functional Test Record (FTRs), Scan Test Record (STR) and Memory Test Record (MTRs).

NOTE 1: Please see the appendix for an example of the BSR field usage.

8.11 *Memory Test Record*

8.11.1 Function: This is the record is used to store fail data along with capture test conditions and references to test test descriptions. It allows the fail data to be stored in various formats describe below using the field highlighting. The highlighting convention used is as follows:

> Yellow — These fields are used for count only logging and allows logging of total fail counts for each dimension in a memory.
> Cyan — These fields are used to (Pin, Cycle number based logging).
> Light green — The fields are used to (row,col) based logging.
> Gold — These fields are used for frame based logging.
> Purple — These fields are used for Bitstream based logging.
> Pink — These fields are used for storing compressed bit maps.

Table 14 Memory Test Record (MTR)

Field Name	*Type*	*Field Description*	*Missing/Invalid*
REC_LEN	U*2	Bytes of Data following header	
REC_TYP	U*1	Record Type(15)	
REC_SUB	U*1	Record Sub-type(40)	
CONT_FLG	B*1	Continuation flag	
TEST_NUM	U*4	Test number	
HEAD_NUM	U*1	Test head number	
SITE_NUM	U*1	Test site number	
ASR_REF	U*2	ASR Index (Algorithm Specification Record)	
TEST_FLG	B*1	Test flags (fail, alarm, etc.)	
LOG_TYP	C*n	User defined description of datalog	length byte = 0
TEST_TXT	C*n	Descriptive text or label	length byte = 0
ALARM_ID	C*n	Name of alarm	length byte = 0
PROG_TXT	C*n	Additional Programmed information	length byte = 0
RSLT_TXT	C*n	Additional result information	length byte = 0
COND_CNT	U*2	Count (k) of conditions	
COND_LST	kxC*n	Conditions specified as <condition name> = <Value> arrays;	COND_CNT=0
CYC_CNT	U*8	Total cycles executed during the test	
TOTF_CNT	U*8	Total fails during the test	
TOTL_CNT	U*8	Total fails logged during in complete MTR data set for a complete Memory Test	
OVFL_FLG	B*1	Flag to indicate failures after logging stopped	

Field Name	*Type*	*Field Description*	*Missing/Invalid*
FILE_INC	B*1	File incomplete	
LOG_TYPE	B*1	Type of datalog (Frame, BIT Stream, pin/cycle, compressed bit stream etc)	
FDIM_CNT	U*1	Count (m) of FDIM_FNAM and FDIM_FCNT array	
FDIM_NAM	mxC*n	Array of logged Dimension names for fail count logging	
FDIM_FCNT	mxU*8	Array of count of failures for dimensions	
CYC_BASE	U*8	Cycle offset to apply for the values in the CYC_OFST array	
CYC_SIZE	U*1	Size (f) of data (1,2,4, or 8 byes) in CYC_OFST field	
PMR_SIZE	U*1	Size (f) of data (1 or 2 bytes) in PMR_ARR field	
ROW_SIZE	U*1	Size (f) of data (1,2,4 or 8 bytes) in ROW_ARR field	
COL_SIZE	U*1	Size (f) of data (1,2,4 or 8 bytes) in COL_ARR field	
DLOG_MSK	U*1	mask to indicate presence or absence of the field for fail based logging	
PMR_CNT	U*4	Count (n) of pins in PIN_ARR	
PMR_ARR	nxU*f	Array of PMR indexes for pins	PMR_CNT=0
CYCO_CNT	U*4	Count (n) of failed cycles in CYC_OFST ARR	
CYC_OFST	nxU*f	array of cycle indexes for each fail	CYC_CNT=0
ROW_CNT	U*4	Count (d) of number of rows in the current record	
ROW_ARR	dxU*f	Array of row addresses for each fail	ROW_CNT=0
COL_CNT	U*4	Count (d) of number of cols in the current record	
COL_ARR	dxU*f	Array of column addresses for each fail	COL_CNT=0
STEP_CNT	U*4	Count (d)of march steps stored in this record	
STEP_ARR	dxU*1	Array of march steps for each fail	STEP_CNT=0
DIM_CNT	U*1	Number (k) of dimensions	
DIM_NAMS	k*C*n	Names of the dimensions	DIM_CNT=0
DIM_DCNT	U*4	Number (n) of data values for a dimension. Repeated per dimension as shown below.	DIM_CNT=0
DIM_DSIZ	U*1	Size (f) of data values for a dimension (1,2,4 or 8 bytes). Repeated per dimension as shown below.	DIM_CNT=0
DIM_VALS	n*U*f	Array of data values for a dimension. Repeated per dimension as shown in th description that follows the table .	DIM_CNT=0
TFRM_CNT	U*8	Total frames in frame based logging	
TFSG_CNT	U*8	Total segments across all the continuation records for this test	
LFSG_CNT	U*2	Local number of frame segments. where each frame segment is defined by FRM_IDX, FRM_MASK, FRM_CNT, LFBT_CNT, and FRAMES	TFRM_CNT=0
FRM_IDX	U*2	Index of the frame record with the frame structure definition	TFRM_CNT=0
FRM_MASK	D*n	mask for the frame field to indicate presence of absence of a field	TFRM_CNT=0
FRM_CNT	U*4	Count (q) of frame with current frame definition and the current mask	TFRM_CNT=0
LFBT_CNT	U*4	Count(q) of bits out of p*f bits that are stored in this record (it is used to handle a single frame > 65K)	

Field Name	*Type*	*Field Description*	*Missing/Invalid*
FRAMES	D*n	Bit encoded data one or more (q) frames of unmasked frame fields where the fields and their sizes (in bits) are defined by the currently referenced FSR.	TFRM_CNT=0
TBSG_CNT	U*8	Total number of logged bit stream segments	
LBSG_CNT	U*2	No of bit stream segments out of all the segments that are stored in this record	TBSG_CNT=0
BSR_IDX	U*2	Index of the bit stream record with the definition of bit stream fields	TBSG_CNT=0
STRT_ADR	U *f	Starting row address in the current segment; f defined by ADDR_SIZ field in BSR	TBSG_CNT=0
WORD_CNT	U*f	Count (r) of words in the current stream segment; f defined by WC_SIZ field in in BSR	TBSG_CNT=0
WORDS	D*n	Bit encoded data for one or more (r) words where the number of bits in each word is defined by the referenced BSR WRD_SIZ field.	
TBMP_SIZE	U*8	count (k) of compressed bit map in bytes	
LBMP_SIZE	U*2	Number of bytes from the map in the current record	BMP_SIZE =0
CBIT_MAP	kxU*1	Compressed bit map	BMP_SIZE=0

8.11.1.1 CONT_FLG – It is flag to indicate if another MTR follows as continuation of current log for a test . If this flag is set to 1 then a continuation MTR record follows and a value 0 indicates that the current MTR is last MTR record of a datalog for that test.

8.11.1.2 *TEST_NUM* — It is the identifier for the test for which data is collected. It should be populated with the Test Number.

8.11.1.3 *HEAD_NUM, SITE_NUM* — If a test system does not support parallel testing, and does not have a standard way of identifying its single test site or head, these fields should be set to 1. When parallel testing, these fields are used to associate individual datalogged results with a PIR/PRR pair. An FTR belongs to the PIR/PRR pair having the same values for HEAD_NUM and SITE_NUM.

8.11.1.4 *TEST_FLG* — Contains the following fields:

bit 0 — 0 = No alarm
1 = Alarm detected
bit 1 — Reserved for future
bit 2 — 0 = Test result is reliable
1 = Test result is unreliable
bit 3 — 0 = No timeout
1 = Timeout occurred
bit 4 — 0 = Test was executed
1 = Test not executed
bit 5 — 0 = No abort
1 = Test aborted
bit 6 — 0 = Pass/fail flag
1 = Test completed
bit 7 — 0 = Test passed
1 = Test failed

8.11.1.5 *LOG_TYP* — This is a user defined description of the datalog type to indicate what type of information is stored in the current record.

8.11.1.6 LOG_TYPE: This a B*1 field where a log type is indicated by setting the corresponding bit to 1. The bit assignment is as follows:

bit 0 — Fail Count log
0 = No Count only log
1 = Fail Count log present

bit 1 — Pin/Cycle type log
0 = No Pin/Cycle type log
1 = Pin/Cycle type log present
bit 2 — Row, Column & Step based log
0 = No Row, Column & Step based log
1 = Row, Column & Step based log present
bit 3 — frame based log
0 = No frame based log
1 = frame based log present
bit 4 — bit stream based log
0 = No bit stream based log
1 = bit stream based log present
bit 5 - Compressed Bit Stream
0 - No Compressed Bit stream based data datalog present
1 - Compressed Bit stream based datalog present

8.11.1.7 *TEST_TXT* — This is a user given description name of the test

8.11.1.8 *ALARM_ID* — If the alarm flag (bit 0 of TEST_FLG) is set, this field can optionally contain the name or ID of the alarm or alarms that were triggered. The names of these alarms are tester-dependent.

8.11.1.9 *PROG_TXT* — This is also a user provided information. Any additional information regarding programming can be provided.

8.11.1.10 *RSLT_TXT* — This is also a user provided information. Any additional information about the results can be provided

8.11.1.11 *ASR_REF* — This is the reference to ASR(s) that describes the algorithms that were applied during the test and for which the data log is stored in the current MTR.

8.11.2 *Test Condition Specification*

8.11.2.1 Test conditions are specified using an array of strings in the COND_LST field array. Each condition is represented as “condition=value” in ASCII without the quotes.

8.11.2.2 Each condition specification is stored in a separate entry in the COND_LST array. The size of this array (or the number of conditions specified) is specified by the COND_CNT field.

8.11.2.3 A user can add other environment specific conditions using this field by adding their custom condition names and values to this array e.g. CORE_ID=<core-id-name>.

8.11.2.4 There is one exception to the test condition specification and that is Temperature specification. Since, the temperature is more of a global test condition and a field exists in the Master Information Record (MIR) to represent that, the temperature does not need to be present in the MTR. The standard however does not preclude one from putting another temperature parameter in the test conditions array mentioned above

8.11.3 *Validation and Synchronization Fields*

8.11.3.1 *CYC_CNT* — The total number of cycles that were executed in this test. If 0 then it should be inferred that this value was not determinable

8.11.3.2 *CYC_BASE* — This field is of type U*8 and contains the offset for the cycle numbers in the fail log. This mechanism allows only U*4 values for the cycle numbers in the fail log (vs. U*8) and thus reduces the data volume. The actual cycle number for the fail log entries would be CYC_BASE+ <fail Cycle No in the log>.

8.11.3.3 *TOTF_CNT* — The total number of failures that were detected in this test, logged or not. A failure is defined as a pin and cycle, such as in three pins failed in the same cycle, that would be counted as 3 failures. If MAXINT then it should be inferred that this value was not determinable

8.11.3.4 *TOTL_CNT* — The total number of failures that were detected and logged in this test. A failure is defined as a pin and cycle, such as in three pins failed in the same cycle that would be counted as 3 failures. This count includes the total failures from the test execution and includes any following continuation STR records.

8.11.3.5 *OVFL_FLG* – This flag determines if there were failures after logging stopped.

8.11.3.6 *FILE_INC* — This flag determine if the fail data is contained in one file or the current file is incomplete. The assignment of the values is as follows:

0 — File is complete

>0 — File is incomplete

This field is expected to be used to indicate that the data that need to be logged exceed the operating systems limit for a file size. This field is only valid when the CONT_FLG flag is 0 i.e. it the last record in the series of continuation records.

8.11.4 *Fail Count only Logging*

8.11.4.1 It is implemented using FDIM_CNT, FDIM_NAM and FDIM_CNT fields

8.11.4.1.1 *FDIM_CNT* — It stores the number of dimensions for which a fail count is stored. Is is also the size of the FDIM_NAM and FDIM _FCNT arrays.

8.11.4.1.2 *FDIM_NAM* — This array stores the names of the dimension for which fail count is stored.

8.11.4.1.3 *FDIM_FCNT* — This array stores the actual fail count for the dimensions in FDIM_NAM array.

8.11.5 *Fail Datalog*

8.11.5.1 CYC_SIZE - Size (f) of data (1,2,4 or 8 bytes) in CYC_OFST field. It allows user to have CYC_OFST array being a 1, 2, 4, or 8 byte array.

8.11.5.2 PMR_SIZE - Size (f) of data (1 or 2 bytes) in PMR_ARR field. It allows user to have PMR_ARR array being a 1 or 2byte array.

8.11.5.3 ROW_SIZE - Size (f) of data (1,2,4 or 8 bytes) in ROW_ARR field. It allows user to have CYC_OFST array being a 1, 2, 4, or 8 byte array.

8.11.5.4 COL_SIZE: Size (f) of data (1,2,4 or 8 bytes) in COL_ARR field. It allows user to have CYC_OFST array being a 1, 2, 4, or 8 byte array.

8.11.5.5 *DLOG_MSK* — This flag indicates which type of data is stored in this record. The record as such allows multiple types of fail log e.g. count only, pin-cycle, frame based bit stream based log but the field for the logs are optional and can be omitted from MTR. DLOG_MASK field indicated which fields are not present in the current MTR record. Table Below shows the bit position assignment for log types. Setting a bit to 1 indicates that particular log type is not present in the record.

Table 15 Log Types

Log type	*Bit position*
Count only	Bit 0
(Pin, Cycle)	Bit 1
(Row, Col)	Bit 2
Fame Based	Bit 3
Bit Stream	Bit 4
Compressed Bit Stream	Bit 5

8.11.5.6 *PMR_CNT* — The number of entries in the PIN_ARR for Pin-Cycle based fail datalogging.

8.11.5.7 *PMR_ARR* — The array for storing failed pins in the test. This array is synchronized with the CYC_OFST array such that the pins in PMR_ARR together with cycle number in CYC_OFST form a pin-cycle number pair to capture the fail

8.11.5.8 *CYCO_CNT* — The number of entries in the CYC_OFST or Pin-Cycle based fail datalogging.

8.11.5.9 *CYC_OFST* — The array of cycle OFST from the CYC_BASE where a failure was observed and captured. The actual cycle number where the failure was recorded is obtained by adding CYC_BASE to CYC_OFST. This implementation is done to reduce the impact on the data volume due to large cycle number field. Some ATE support 64 bit cycle numbers which otherwise would lead to 8 byte per cycle number recorded.

8.11.5.10 *ROW_CNT* — The number of entries in the ROW_ARR for Row-Column based fail datalogging.

8.11.5.11 *ROW_ARR* — The array to store the failing row number. This array is synchronous with the COL_ARR such that the corresponding entries form a failed row-column pair.

8.11.5.12 *COL_CNT* — The number of entries in the ROW_ARR for Row-Column based fail datalogging.

8.11.5.13 *COL_ARR* — The array to store the failing row number. This array is synchronous with the ROW_ARR such that the corresponding entries form a failed row-column pair.

8.11.5.14 *STEP_CNT* — The number of entries in the STEP_ARR field for Row-Column based fail datalogging. Storing the failing algorithm step is optional.

8.11.5.15 *STEP_ARR* — The array to record the MARCH steps where the failure was seen. This array can be synchronized and used with either pin-cycle based logging or row-col based logging. But only one type of logging at a time.

8.11.6 Generic Dimension Based Logging

8.11.6.1 *DIM_CNT* — The number of dimensions represented in the DIM_VALS array

8.11.6.2 *DIM_NAMS* — An array of string names for each dimension represented in the DIM_VALS array

8.11.6.3 *DIM_DSIZ*— The data size, in bytes, of the values in the DIM_VALS data to follow. This can be 1, 2, 4, or 8. This field is part of a series of fields that are repeated as needed for each dimension in DIM_CNT.

8.11.6.4 *DIM_DCNT* — The number of values in the DIM_VALS data to follow. This field is part of a series of fields that are repeated as needed for each dimension in DIM_CNT

8.11.6.5 *DIM_VALS* — An array of DIM_DCNT data values or size DIM_DSIZ bytes for a given dimension defined in DIM_CNT and DIM_NAMS. This field is part of a series of fields that are repeated as needed for each dimension in DIM_CNT.

8.11.6.5.1 The following is an example of how DIM_VALS data is organized and how fields are repeated for each dimension. An example of bit layout is shown in figure R9-1, R9-2 and R9-3

DIM_CNT k	
DIM_NAMS[0]	1st dimension name
DIM_NAMS[1]	2nd dimension name
DIM_NAMS[…]	…
DIM_NAMS[k-1]	kth dimension name
DIM_DSIZ	1st dimension data size (f_1)
DIM_DCNT	Number (n_1) of values for 1st dimension
DIM_VALS[0]	1st value of 1st dimension, data size defined by f_1
DIM_VALS[1]	2nd value of 1st dimension, data size defined by f_1
DIM_VALS[…]	…
DIM_VALS[n_1-1]	last value of 1st dimension, data size defined by f_1
DIM_DSIZ	2nd dimension data size (f_2)
DIM_DCNT	Number (n_2) of values for 2nd dimension
DIM_VALS[0]	1st value of 2nd dimension, data size defined by f_2
DIM_VALS[1]	2nd value of 2nd dimension, data size defined by f_2
DIM_VALS[…]	…
DIM_VALS[n_2-1]	last value of 2nd dimension, data size defined by f_2
…	
…	
DIM_DSIZ	kth dimension data size (f_k)
DIM_DCNT	Number (n_k) of values for kth dimension
DIM_VALS[0]	1st value of kth dimension, data size defined by f_k
DIM_VALS[1]	2nd value of kth dimension, data size defined by f_k
DIM_VALS[…]	…
DIM_VALS[n_k-1]	last value of kth dimension, data size defined by f_k

8.11.7 *Frame Based Logging*

8.11.7.1 Following Fields are used to implement frame based logging with the MTR.

8.11.7.1.1 *TFRM_CNT* — This field stores the total number of frames in the frame based logging that are stored in this record.

8.11.7.1.2 *TFSG_CNT* — This field stores the total number of frame segments for this test stores across all the MTR records for this test. Each segment is defined as a series of frames which comply with a frame mask.

8.11.7.1.3 *LFSG_CNT* — This field stores the number of frame segments that are stored in the current record. The rest are stored in the continuation records that follow.

8.11.7.1.4 *FRM_IDX* — This is the reference to the frame Specification record which contain the definition for the frames in this record.

8.11.7.1.5 FRM_MASK — The mask field allows certain field from the frame definition to be eliminated for frames in the log until a new mask is defined. Setting a bit to 1 means that field is masked. The least significant bit corresponds to first field in FSR. A *FRM_MASK* of a segment remains valid across continuation record for a segment.

8.11.7.1.6 *FRM_CNT* — The number of frames that comply with the current mask setting in FRM_MASK

8.11.7.1.7 *LFBT_CNT* — This field stores the number of bits in the current frame segment that are stored in the current MTR record the rest are stored in continuation records.

8.11.7.1.8 *FRAMES* — The fail datalog bit encoded frame data representing the unmasked fields contained in one or more (FRM_CNT) frames. Each field of the frame is defined by the currently referenced FSR (FRM_IDX). Masked fields are defined by the currently referenced FRM_MASK and data from masked fields is NOT included in this bit encoded data. The length of the bit encoded D*n data in bits is the (bits in unmasked fields) * (FRM_CNT). See Figures R8-1, R8-2, R8-3 and R8-4 for an examples of bit layout .In addition, the first LSB bit in the first byte corresponds to bit 0 of the first bit of the frame data as shown in Figure R8-2.

8.11.8 *BitStream Based Logging*

8.11.8.1 Following field enable bit stream based logging in a MTR

8.11.8.1.1 *TBSG_CNT* — Total number of bit stream clusters possibly spanning across multiple MTRs from the current test

8.11.8.1.2 *LBSG_CNT* — The number of Bit stream segments stored in the current record

8.11.8.1.3 *BSR_IDX* — The reference to bit stream specification record that contains the format definition for the bit stream.

8.11.8.1.4 *STRT_ADR* — This field stores the start address of the Segment.

8.11.8.1.5 *WORD_CNT* — The number of words starting with the STRT_ADR that have some fails and are logged

8.11.8.1.6 *WORDS* — Bit encoded fail data for one or more (WORD_CNT) words defined by the currently referenced BSR (BSR_INDX). The length of the bit encoded D*n data in bits is defined by the BSR's WRD_SIZ * WRD_CNT. See Figure R7-1 and R7-2 for an example.

8.11.9 Compressed Bit map

8.11.9.1 TBMP_SIZE: This field contains the count of total bytes in the compressed bit map.

8.11.9.2 LBMP_SIZE: This field stores the count of bytes of the compressed bitmap that are stored in the current record. The rest of the bytes will be stored in the successive record by using continue flag

8.11.9.3 CBIT_MAP: This is the array of bytes that stores the compressed bit map in the current record.

8.11.10 Frequency: Number of memory tests times records required to log the fails for the test (counting continuation record)

8.11.11 Location: It can occur after all the memory design specific records i.e. any Memory Structure Record (MSR),any Memory Controller Records (MCRs), any Memory Instance Records (IDRs), any Memory Model Records(MMRs) , any Algorithms Specification Records (ASRs), any Frame Specification Records (FSRs) and any Bitstream Specificaion Records (BSRs)

NOTICE: SEMI makes no warranties or representations as to the suitability of the standard(s) set forth herein for any particular application. The determination of the suitability of the standard(s) is solely the responsibility of the user. Users are cautioned to refer to manufacturer's instructions, product labels, product data sheets, and other relevant literature respecting any materials or equipment mentioned herein. These standards are subject to change without notice.

By publication of this standard, Semiconductor Equipment and Materials International (SEMI) takes no position respecting the validity of any patent rights or copyrights asserted in connection with any item mentioned in this standard. Users of this standard are expressly advised that determination of any such patent rights or copyrights, and the risk of infringement of such rights are entirely their own responsibility.

RELATED INFORMATION 1
EXAMPLES OF USING FAIL DATALOG

R1-1 Storing Embedded Memory Design Information

R1-1.1 Following example show a case of a design with embedded memory shown in figure below. The design has two embedded memory blocks which are controlled by a single memory controller. The address buses are 4 bit wide and the data is two bit wide for each of the memory.

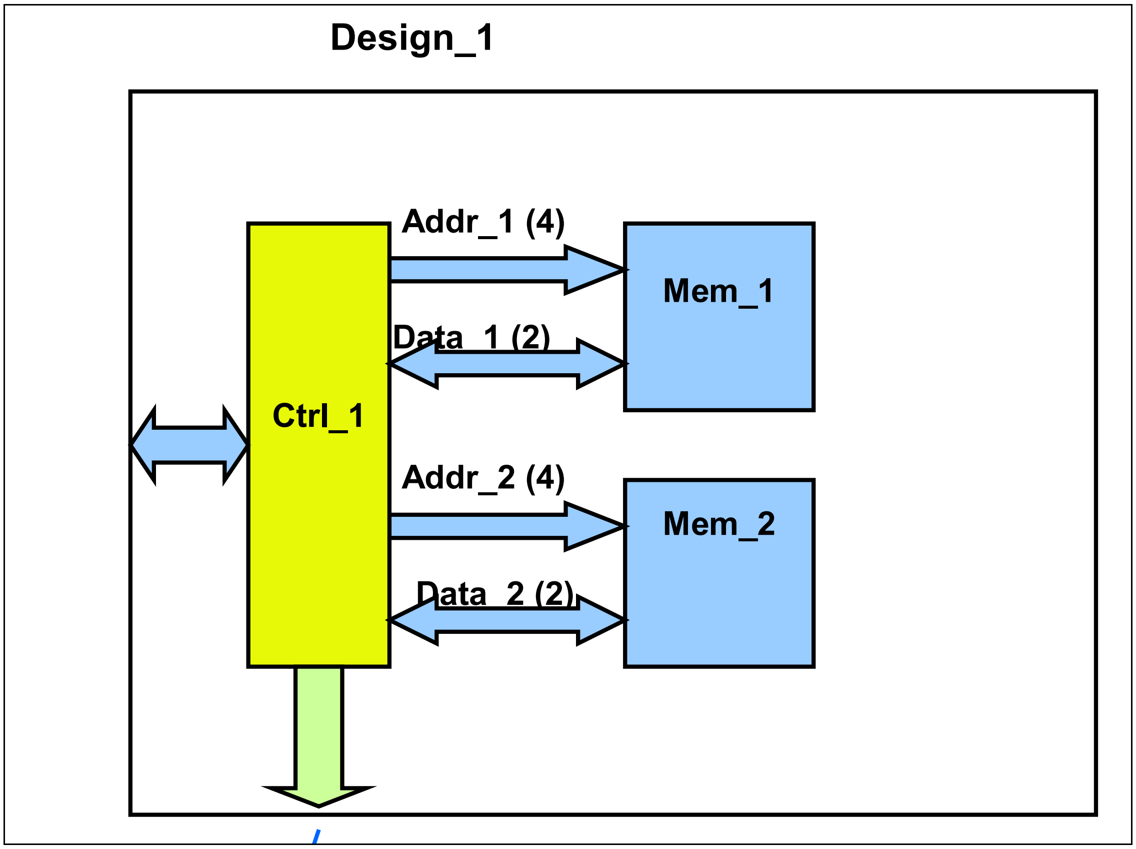
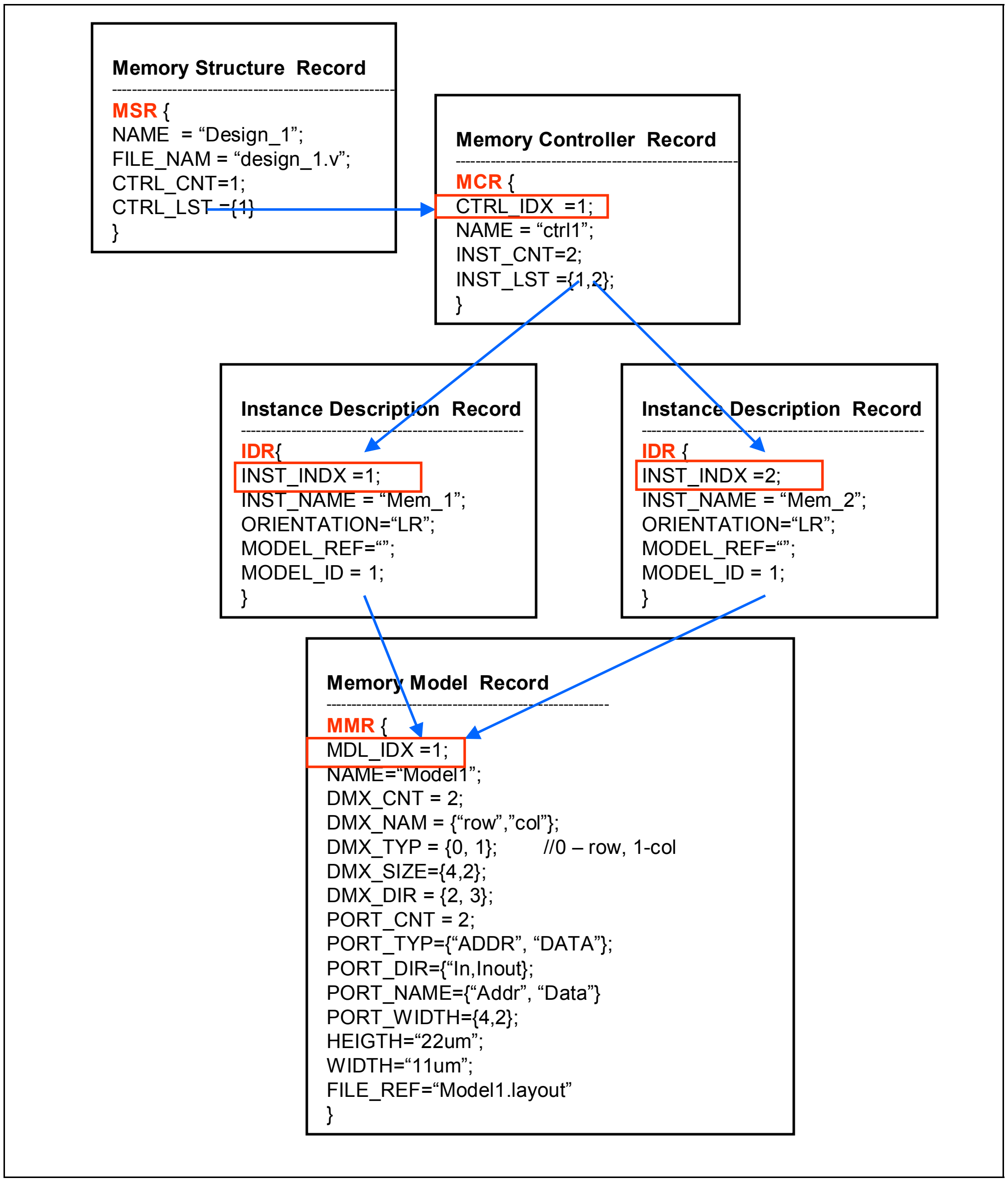


Figure R1-1
Embedded Memory Example

R1-1.2 Figure R1-2 shows the record structure in a graphical form. The Memory structure record contains the pointer to the single memory controller in this design. The controller record has a unique ID 1 assigned to it in the memory controller name space. The Memory controller record contains pointers to Instance Description Records for two memory instances in the design. Both the instance description records point to the same Memory model record as instances are of same memory type. The memory Model Description Record (MDR) describes the details of the memory's logical structures.



Memory Structure Record

```
MSR {
NAME  = “Design_1”;
FILE_NAM = “design_1.v”;
CTRL_CNT=1;
CTRL_LST ={1}
}
```

Memory Controller Record

```
MCR {
CTRL_IDX  =1;
NAME = “ctrl1”;
INST_CNT=2;
INST_LST ={1,2};
}
```

Instance Description Record

```
IDR{
INST_INDX =1;
INST_NAME = “Mem_1”;
ORIENTATION=“LR”;
MODEL_REF=“”;
MODEL_ID = 1;
}
```

Instance Description Record

```
IDR {
INST_INDX =2;
INST_NAME = “Mem_2”;
ORIENTATION=“LR”;
MODEL_REF=“”;
MODEL_ID = 1;
}
```

Memory Model Record

```
MMR {
MDL_IDX =1;
NAME=“Model1”;
DMX_CNT = 2;
DMX_NAM = {“row”,”col”};
DMX_TYP = {0, 1};       //0 – row, 1-col
DMX_SIZE={4,2};
DMX_DIR = {2, 3};
PORT_CNT = 2;
PORT_TYP={“ADDR”, “DATA”};
PORT_DIR={“In,Inout};
PORT_NAME={“Addr”, “Data”}
PORT_WIDTH={4,2};
HEIGTH=“22um”;
WIDTH=“11um”;
FILE_REF=“Model1.layout”
}
```

Figure R1-2

Records for Embedded Memory Design Description

R1-2 Storing Direct Access/Standalone Memory Design Information

R1-2.1 This example shows a case of direct access memory where the memory ports are directly accessible from the chip IO. The same case also applies for a standalone memory. The address ports are 4 bit wide and the data port is 2 bit wide. The control port is one bit wide.

LETTER (YELLOW) BALLOT

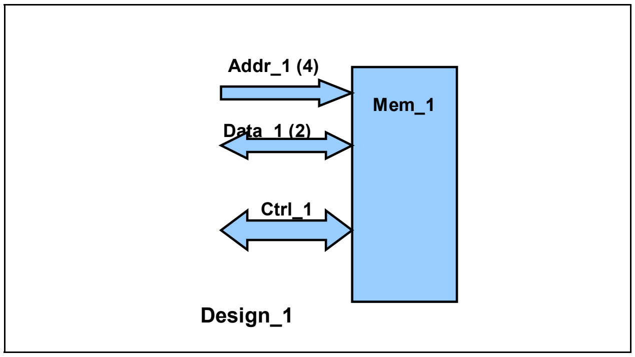
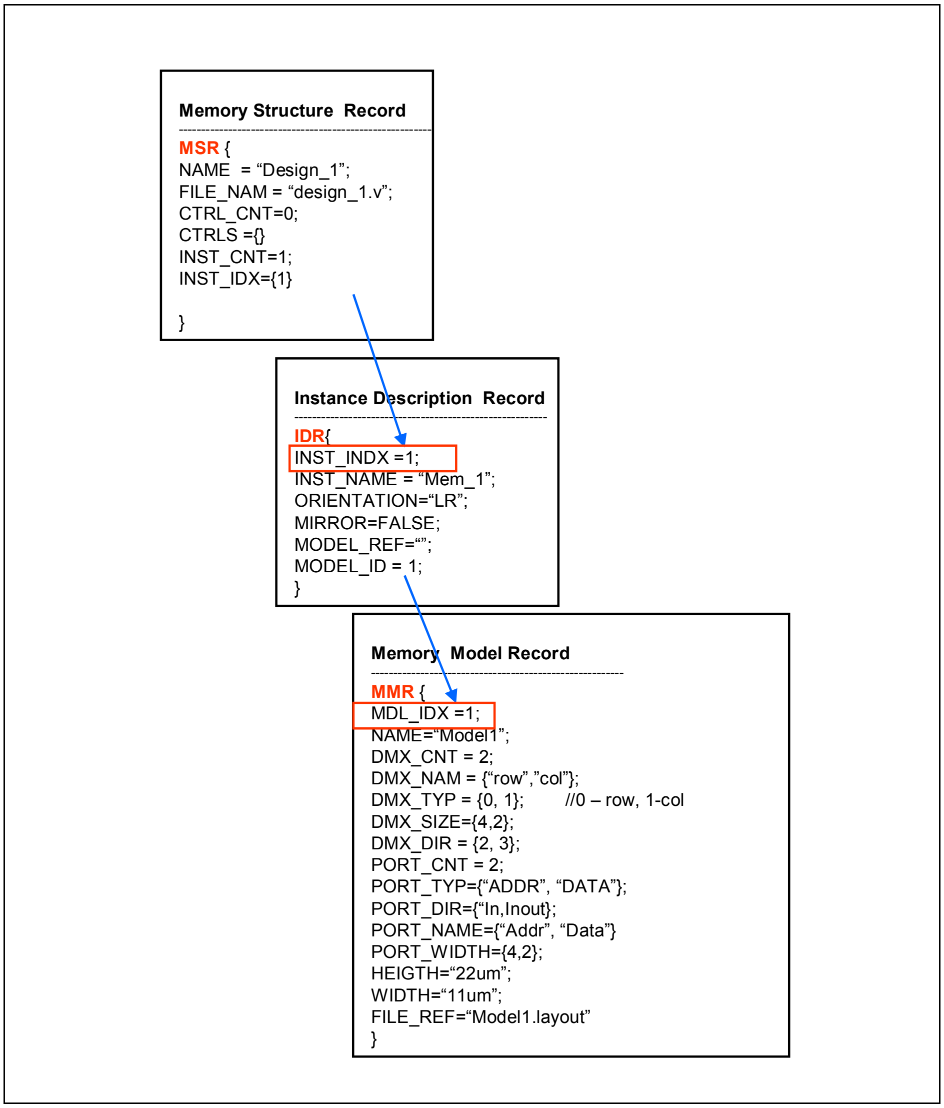


Figure R1-3
Direct Memory Example



Memory Structure Record

```
MSR {
NAME  = “Design_1”;
FILE_NAM = “design_1.v”;
CTRL_CNT=0;
CTRLS ={}
INST_CNT=1;
INST_IDX={1}

}
```

Instance Description Record

```
IDR{
INST_INDX =1;
INST_NAME = “Mem_1”;
ORIENTATION=“LR”;
MIRROR=FALSE;
MODEL_REF=“”;
MODEL_ID = 1;
}
```

Memory Model Record

```
MMR {
MDL_IDX =1;
NAME=“Model1”;
DMX_CNT = 2;
DMX_NAM = {“row”,”col”};
DMX_TYP = {0, 1};        //0 – row, 1-col
DMX_SIZE={4,2};
DMX_DIR = {2, 3};
PORT_CNT = 2;
PORT_TYP={“ADDR”, “DATA”};
PORT_DIR={“In,Inout};
PORT_NAME={“Addr”, “Data”}
PORT_WIDTH={4,2};
HEIGTH=“22um”;
WIDTH=“11um”;
FILE_REF=“Model1.layout”
}
```

Figure R1-4
Design With Direct Memory Access

R1-2.2 In this case, since there is no embedded memory controller, the controller fields are empty in the Memory Structure Record (MSR) and by the same reason there is no Memory Controller Record (MCR). The MSR records instead points directly to the memory instance which in turn points to its Model Description Record.

R1-3 Storing Design Information For Designs with Embedded and Direct Access Memory

R1-3.1 This example shows a case of a design which has one embedded memory and a direct access memory. Both of the memory instances in the example are shows to be of same type for simplicity however, there is no such requirement of them to be the same type.

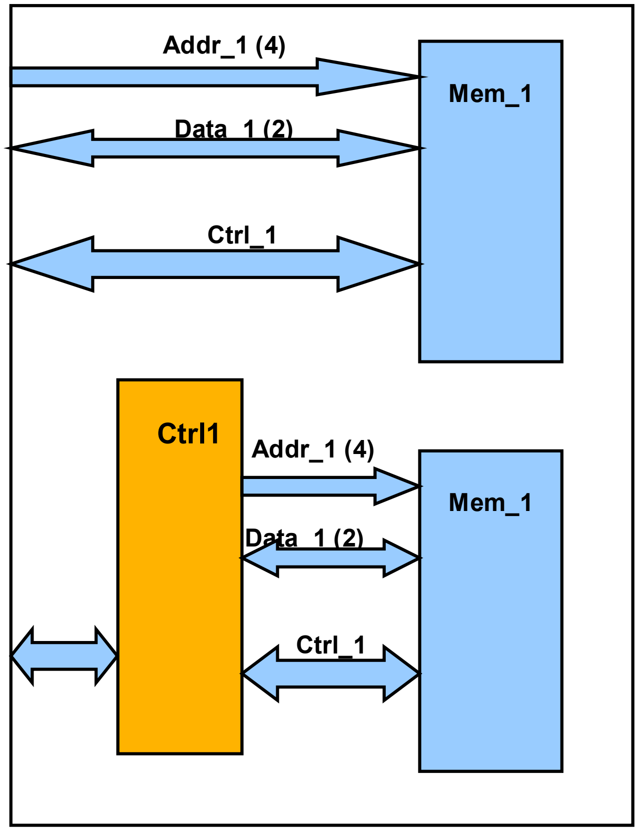
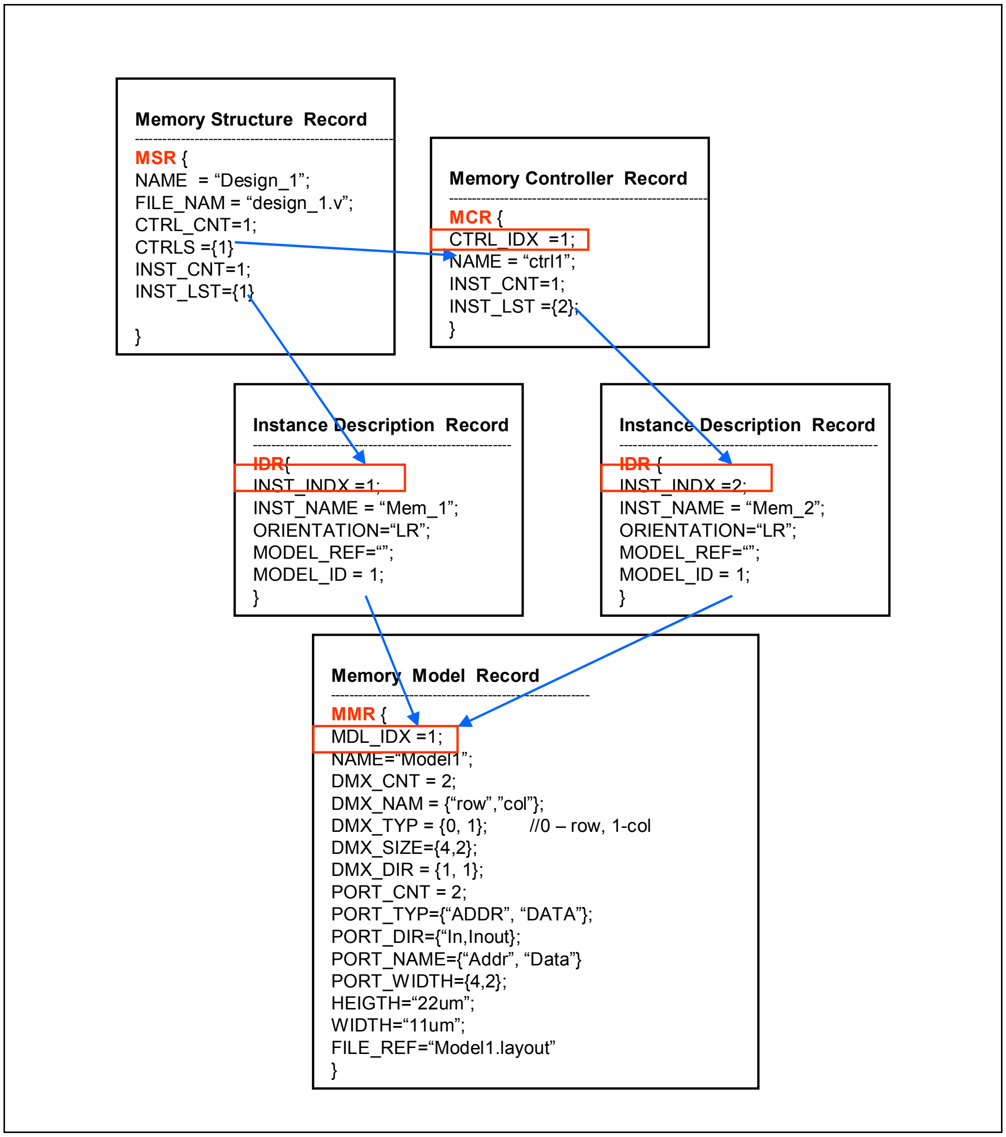


Figure R1-5
Design with Embedded and Direct Access Memory

R1-3.2 The main point highlight in this example is that the Memory Specification Record (MSR) in this contains a point to the memory controller for embedded memory as well as a pointer the instance description record of the direct access memory. Each of these pointer fields are arrays hence multiple direct access memories as well as multiple embedded memories can be described using this record structure.

LETTER (YELLOW) BALLOT



Memory Structure Record

```
MSR {
NAME  = “Design_1”;
FILE_NAM = “design_1.v”;
CTRL_CNT=1;
CTRLS ={1}
INST_CNT=1;
INST_LST={1}

}
```

Memory Controller Record

```
MCR {
CTRL_IDX  =1;
NAME = “ctrl1”;
INST_CNT=1;
INST_LST ={2};
}
```

Instance Description Record

```
IDR{
INST_INDX =1;
INST_NAME = “Mem_1”;
ORIENTATION=“LR”;
MODEL_REF=“”;
MODEL_ID = 1;
}
```

Instance Description Record

```
IDR {
INST_INDX =2;
INST_NAME = “Mem_2”;
ORIENTATION=“LR”;
MODEL_REF=“”;
MODEL_ID = 1;
}
```

Memory Model Record

```
MMR {
MDL_IDX =1;
NAME=“Model1”;
DMX_CNT = 2;
DMX_NAM = {“row”,”col”};
DMX_TYP = {0, 1};        //0 – row, 1-col
DMX_SIZE={4,2};
DMX_DIR = {1, 1};
PORT_CNT = 2;
PORT_TYP={“ADDR”, “DATA”};
PORT_DIR={“In,Inout};
PORT_NAME={“Addr”, “Data”}
PORT_WIDTH={4,2};
HEIGTH=“22um”;
WIDTH=“11um”;
FILE_REF=“Model1.layout”
}
```

Figure R1-6

Record Structure for Design With Embedded and Direct Access Memory

R1-4 Access to Physical Information

R1-4.1 As mentioned earlier in the document, the standard provides records to store the logical information only and only supports storing links to files with physical information. The links to physical information can be added inside the Instance Description Record in which case there is no logic or physical information is stored in STDF. The other place where links can be added is inside Memory Model Record (MMR). This leads to three possible scenarios as shows in Figure R1-7 below.

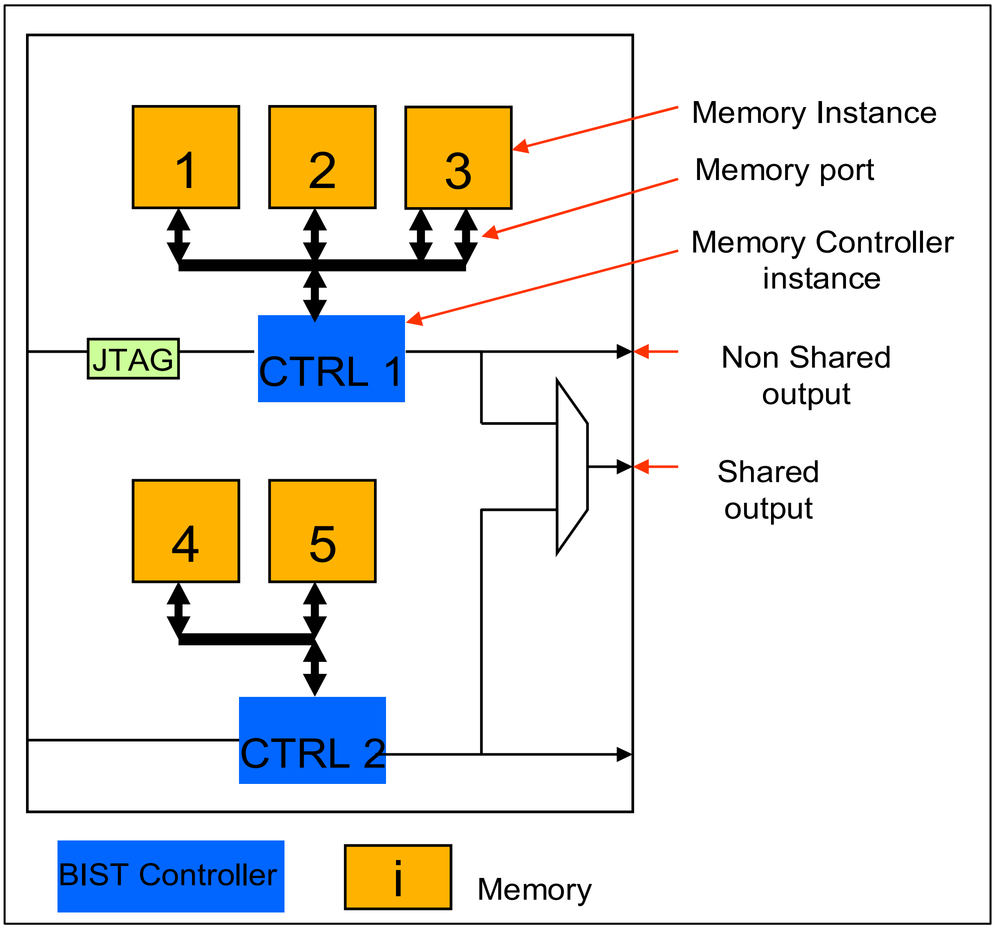


Figure R1-7(a)
Storing Links to Physical information

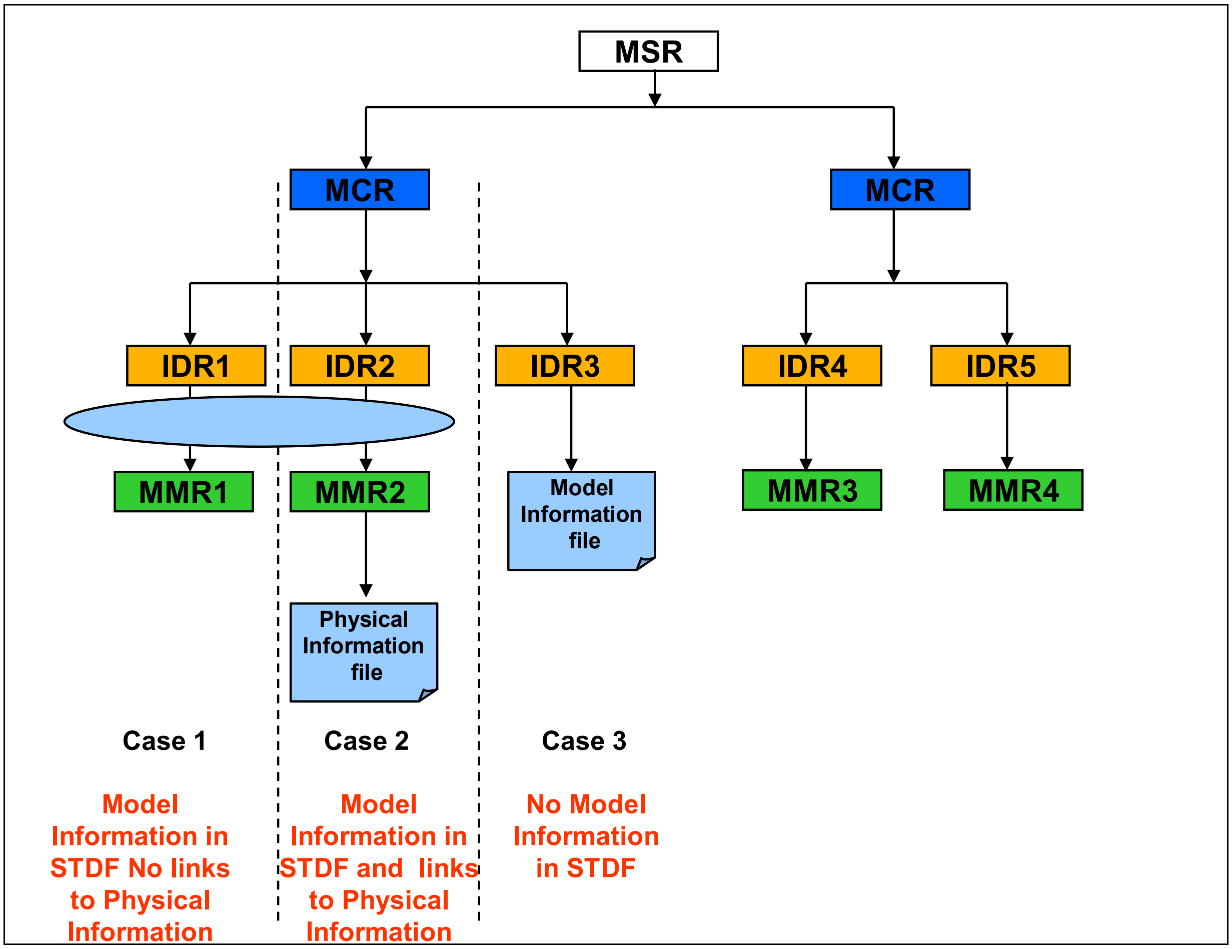

MSR
MCR
MCR
IDR1
IDR2
IDR3
IDR4
IDR5
MMR1
MMR2
Model Information file
MMR3
MMR4
Physical Information file
Case 1
Case 2
Case 3
Model Information in STDF No links to Physical Information
Model Information in STDF and links to Physical Information
No Model Information in STDF

Figure R1-8(b)
Storing Links to Physical information

RELATED INFORMATION 2
EXAMPLES OF STORING ALGORITHM INFORMATION

R2-1 Storing Memory Test Algorithm Information in Memory Fail Datalog

R2-1.1 The standard allows storing information about the algorithms that are used in a test. Algorithm specification record is used to describe the algorithms for a test in the STDF file. Figure R2-1 shows an example of an Algorithm Specification Record (ASR) where the test applies one algorithm. The ASR stores the name of the algorithms, its complexity and library name where the description of algorithms is stored. It also allows storing the cycle begin and end information for the algorithm which can be used to translate cycles numbers from tester to location in the algorithm.

```
Algorithm Specification Record
------------------------------------------------------------------------------------------
ASR {
ASR_IDX=1;
STRT_IDX=0;  //Cycle numbers start from 0
ALGO_CNT=1;
ALGO_NAME=”MARCH-C”;  //
ALGO_LEN=”13N”;  // Algorithm complexity is 13N
FILE_ID=”Mem_Algo_Lib”;  //File containing the algorithm descriptions.
CYC_BGN=0
CYC_END=13312
}
```

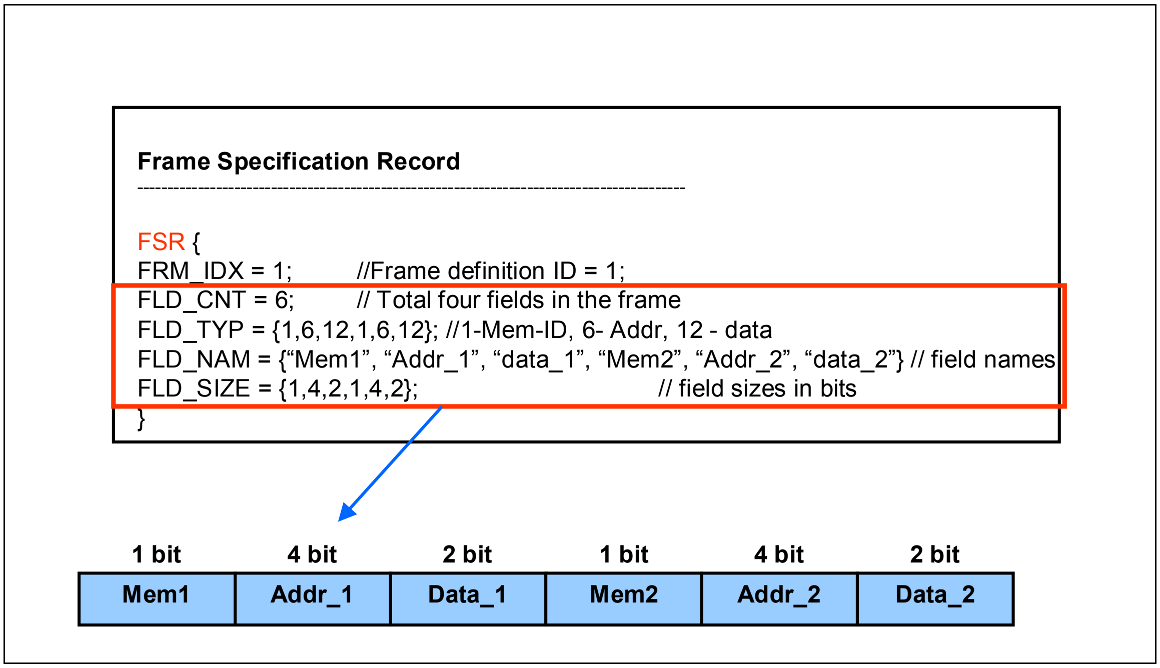
Figure R2-1
Algorithm Specification Record Example

LETTER (YELLOW) BALLOT

RELATED INFORMATION 3
EXAMPLE OF USING FRAME SPECIFICATION RECORD

R3-1 Using Frame Specification Record Example

R3-1.1 The Frame Specification Record allows specification of the structure of the datalog frames where the data is communicated in stored in the form of serial frames. The example below shows how the frame shown in Figure R3-1 is specified using a FSR record. In this example the frame contains 6 fields shows in figure. These fields are described using the record fields in the red box. The FLD_TYPE entries indicate the type of fields using a standard described encoding described earlier in the document. The FLD_NAM field is an array and stores the user defined names of the fields. The FLD_SIZE information stores the size in number of bits for each of theses fields. These three fields are position synchronized i.e. the first entry in FLD_TYPE corresponds to first entries in FLD_NAM and FLD_SIZE fields.



Frame Specification Record

```
FSR {
FRM_IDX = 1;        //Frame definition ID = 1;
FLD_CNT = 6;        // Total four fields in the frame
FLD_TYP = {1,6,12,1,6,12}; //1-Mem-ID, 6- Addr, 12 - data
FLD_NAM = {“Mem1”, “Addr_1”, “data_1”, “Mem2”, “Addr_2”, “data_2”} // field names
FLD_SIZE = {1,4,2,1,4,2};                        // field sizes in bits
}
```

1 bit	4 bit	2 bit	1 bit	4 bit	2 bit
Mem1	Addr_1	Data_1	Mem2	Addr_2	Data_2

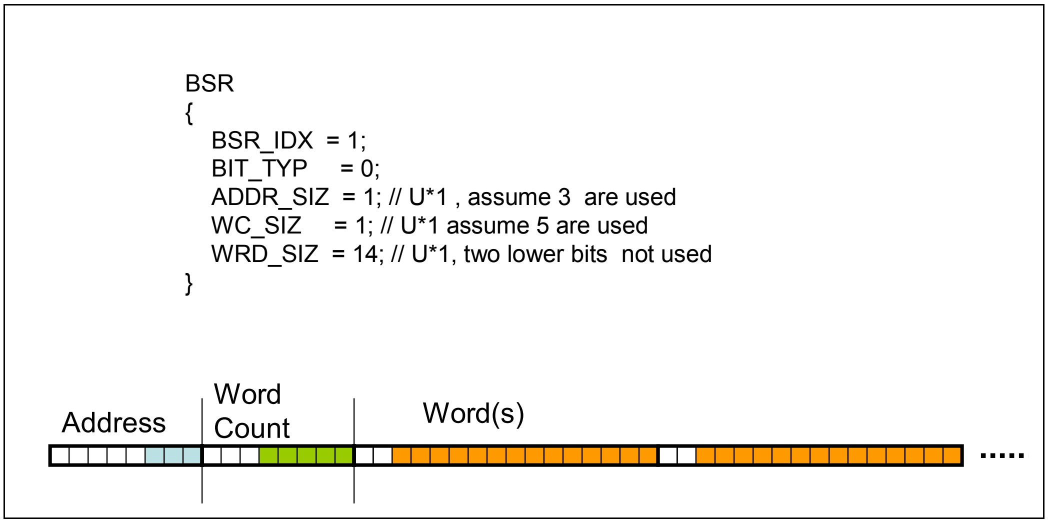
Figure R3-1
Frame Specification Record Example

LETTER (YELLOW) BALLOT

RELATED INFORMATION 4
EXAMPLES OF USING BIT STEAM SPECIFICATION RECORD

R4-1 Bit Stream Specification Record Example

R4-1.1 The Bit Stream Specification Record allows storing the structure information of a datalog stored in a bit stream format. The Bit stream is stored a sequence of Word segments. Each segment is a series of words in the memory for which information in stored in the bit stream. This type of bit stream structure allows one to eliminate the words with non-relevant words be removed from the log and thereby reducing the data volume of the log. Figure R4-1 shows an example of a specification of word cluster where the ADDR_SIZ field stores the number of bits allocated to store the starting address of the cluster. The WC_SIZ field stores the number of bit allocated to store the count of the words from the start cluster starting address that are stored in the log. The WRD_SIZ field contains the size of the memory words in bits. The used bits are shown with the uncolored boxes in each of the fields.



```
BSR
{
   BSR_IDX  = 1;
   BIT_TYP     = 0;
   ADDR_SIZ  = 1; // U*1 , assume 3  are used
   WC_SIZ     = 1; // U*1 assume 5 are used
   WRD_SIZ  = 14; // U*1, two lower bits  not used
}
```

Figure R4-1
Bit Stream Specification Record Example

RELATED INFORMATION 5
EXAMPLES OF USING SUPPORTED DATALOG TYPES

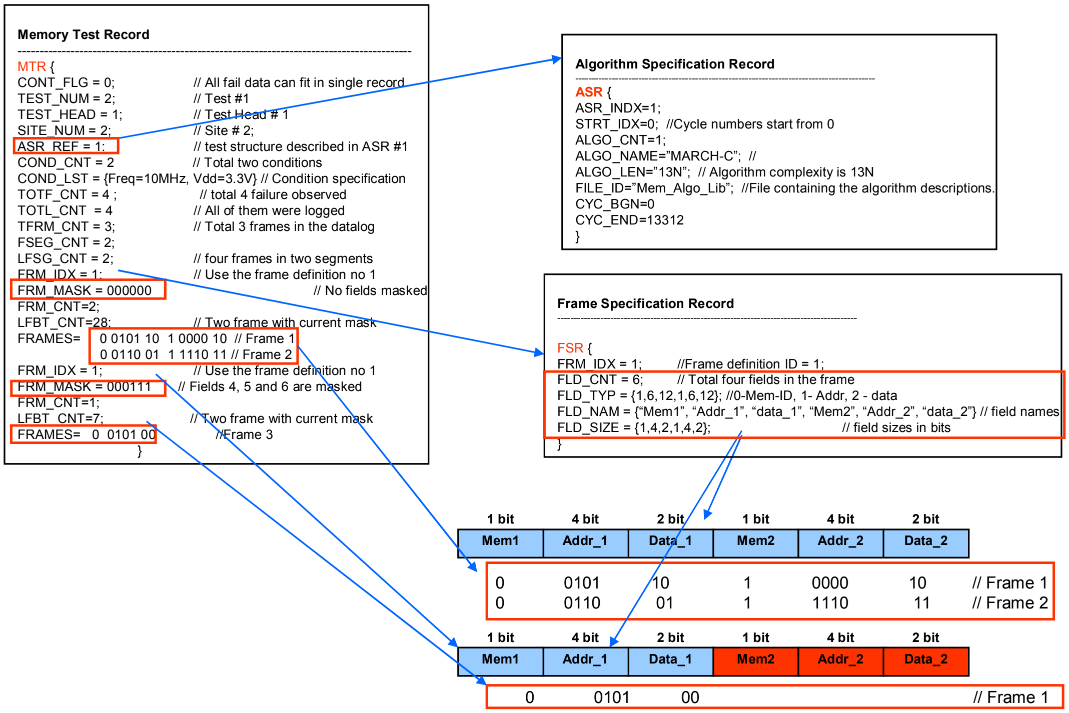
R5-1 Count only Log

R5-1.1 This type of log stores only the fail count in the Memory Test Record. Figure R5-1 shows an example of MTR record. The fail counts are stored in the DIM_CNT, DIM_NAME, DIM_FCNT. This example also shows the setting of test conditions using COND_LST and COND_CNT as well as the header fields which are similar to Header fields for Functional Test Records (FTRs).

```
CONT_FLAG = 0; // No Continuation record exists
TEST_NUM = 1;    // Test Id in the flow = 1
HEAD_NUM = 1   // the device is tested on Head number 1
SITE_NUM = 2;    // The device is site number 2 on header number 1
ASR_REF = 2;    // the memory algorithm  used in the test is described in ASR # 2
TEST_FLG = 0;  // No test flags set
LOG_TYP = “Memory test log”; //User give description of the log
TEST_TXT = “”; // No give description
ALARM_ID= “” ; //No Alarm ID set;
PROG_TXT= “”  // No Additional program information
RSLT_TXT = “” // No additional test result information.
COND_CNT = 2; // Total two conditions set logged for the test
COND_LST = {“Frequency=100MHz”, “Voltage=3.3V”}; // The description of condition as <ConditionName= Value>
TOTF_CNT= 1000; // total 1000 fails detected;
TOTL_CNT=1000; // All 1000 fails were logged
FILE_INC = 0; // The file is complete;
LOGTYPE=11111110;  // Log type is Fail count only, other log types are masked
DIM_CNT=2; // Fail count on two dimensions logged;
DIM_NAME = {“ROW,”COL”};
DIM_FCNT = {10,5}; // 10 rows  and 5 columns had failures
```

R5-2 Frame Based Logging

R5-2.1 This example shows the Frame based logging. Figure R5-1 shows how the MTR, FSR and ASR records are used to log failures in the frame format. The frame contains six fields described in the FSR record. In this example two segments of frame sequences stores in the log. In the first segment, two frames where all 6 fields are present are stored. The second segment contains only one frame but has only 3 out of 6 field filed from the frame. The other 3 are masked. The masking feature is allowed to reduce the data volume due to repeating information or from place for the missing information.



Memory Test Record

```
----------------------------------------------------------------------------------------
MTR {
CONT_FLG = 0;                 // All fail data can fit in single record
TEST_NUM = 2;                 // Test #1
TEST_HEAD = 1;                // Test Head # 1
SITE_NUM = 2;                 // Site # 2;
ASR_REF = 1:                  // test structure described in ASR #1
COND_CNT = 2                  // Total two conditions
COND_LST = {Freq=10MHz, Vdd=3.3V} // Condition specification
TOTF_CNT = 4 ;                 // total 4 failure observed
TOTL_CNT  = 4                 // All of them were logged
TFRM_CNT = 3;                 // Total 3 frames in the datalog
FSEG_CNT = 2;
LFSG_CNT = 2;                 // four frames in two segments
FRM_IDX = 1;                  // Use the frame definition no 1
FRM_MASK = 000000                                  // No fields masked
FRM_CNT=2;
LFBT_CNT=28;                  // Two frame with current mask
FRAMES=   0 0101 10  1 0000 10  // Frame 1
          0 0110 01  1 1110 11 // Frame 2
FRM_IDX = 1;                  // Use the frame definition no 1
FRM_MASK = 000111   // Fields 4, 5 and 6 are masked
FRM_CNT=1;
LFBT_CNT=7;                  // Two frame with current mask
FRAMES=  0  0101 00              //Frame 3
                   }
```

Algorithm Specification Record

```
-------------------------------------------------------------------------------
ASR {
ASR_INDX=1;
STRT_IDX=0;  //Cycle numbers start from 0
ALGO_CNT=1;
ALGO_NAME=”MARCH-C”;  //
ALGO_LEN=”13N”;  // Algorithm complexity is 13N
FILE_ID=”Mem_Algo_Lib”;  //File containing the algorithm descriptions.
CYC_BGN=0
CYC_END=13312
}
```

Frame Specification Record

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

FSR {
FRM_IDX = 1;         //Frame definition ID = 1;
FLD_CNT = 6;         // Total four fields in the frame
FLD_TYP = {1,6,12,1,6,12}; //0-Mem-ID, 1- Addr, 2 - data
FLD_NAM = {“Mem1”, “Addr_1”, “data_1”, “Mem2”, “Addr_2”, “data_2”} // field names
FLD_SIZE = {1,4,2,1,4,2};                        // field sizes in bits
}
```

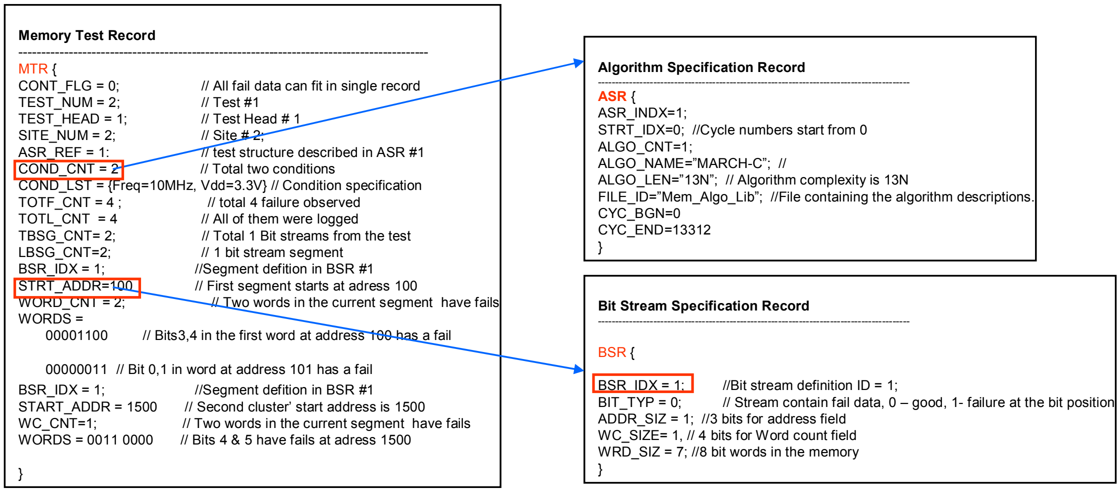
1 bit	4 bit	2 bit	1 bit	4 bit	2 bit	
Mem1	**Addr_1**	**Data_1**	**Mem2**	**Addr_2**	**Data_2**	
0	0101	10	1	0000	10	// Frame 1
0	0110	01	1	1110	11	// Frame 2

1 bit	4 bit	2 bit	1 bit	4 bit	2 bit	
Mem1	**Addr_1**	**Data_1**	**Mem2**	**Addr_2**	**Data_2**	
0	0101	00				// Frame 1

Figure R5-1
Frame based Logging Example

R5-3 Bit Stream Based Logging

R5-3.1 This example shows how to store bit stream based log. There are total 4 failures in the test. Two fails are clustered in two consecutive words starting at address 100 and the other two fails are in a word at address 1500. These clustered are stored in bit stream segments. The segment structure is defined in BSR record number 1.



Memory Test Record

```
------------------------------------------------------------------------------------------------
MTR {
CONT_FLG = 0;                    // All fail data can fit in single record
TEST_NUM = 2;                    // Test #1
TEST_HEAD = 1;                   // Test Head # 1
SITE_NUM = 2;                    // Site # 2;
ASR_REF = 1:                     // test structure described in ASR #1
COND_CNT = 2                     // Total two conditions
COND_LST = {Freq=10MHz, Vdd=3.3V} // Condition specification
TOTF_CNT = 4 ;                    // total 4 failure observed
TOTL_CNT  = 4                    // All of them were logged
TBSG_CNT= 2;                     // Total 1 Bit streams from the test
LBSG_CNT=2;                      // 1 bit stream segment
BSR_IDX = 1;                    //Segment defition in BSR #1
STRT_ADDR=100                   // First segment starts at adress 100
WORD_CNT = 2;                      // Two words in the current segment  have fails
WORDS =
     00001100       // Bits3,4 in the first word at address 100 has a fail

     00000011  // Bit 0,1 in word at address 101 has a fail
BSR_IDX = 1;                    //Segment defition in BSR #1
START_ADDR = 1500      // Second cluster' start address is 1500
WC_CNT=1;                      // Two words in the current segment  have fails
WORDS = 0011 0000      // Bits 4 & 5 have fails at adress 1500

}
```

Algorithm Specification Record

```
--------------------------------------------------------------------------------------
ASR {
ASR_INDX=1;
STRT_IDX=0;  //Cycle numbers start from 0
ALGO_CNT=1;
ALGO_NAME="MARCH-C";  //
ALGO_LEN="13N";  // Algorithm complexity is 13N
FILE_ID="Mem_Algo_Lib";  //File containing the algorithm descriptions.
CYC_BGN=0
CYC_END=13312
}
```

Bit Stream Specification Record

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

BSR {

BSR_IDX = 1;         //Bit stream definition ID = 1;
BIT_TYP = 0;         // Stream contain fail data, 0 – good, 1- failure at the bit position
ADDR_SIZ = 1;  //3 bits for address field
WC_SIZE= 1, // 4 bits for Word count field
WRD_SIZ = 7; //8 bit words in the memory
}
```

Figure R5-2
BitStream Based Datalog Example

RELATED INFORMATION 6
EXAMPLES OF USING ORIENTATION CODE

R6-1 Orientation Code Examples

R6-1.1 As mentioned earlier, each Instance Description Record allows user to store the orientation information. This information consists of a two byte code. The 1st character represents the direction of increasing column direction and the 2nd character represents the direction of increasing row direction

R6-1.2 4 characters are used to describe the direction

U → Up

D → Down

L → Left

R → Right

R6-1.3 The naming of these directional indicators is based on notch down placement for wafer and bottom left corner placement of package reference designator for the packages. These codes will need to be and rotate with other notch/package referencedesignator placement (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 left setting).

R6-1.4 The two byte code with the 4 character encoding allows description of 8 possible memory orientations as show in the figure below

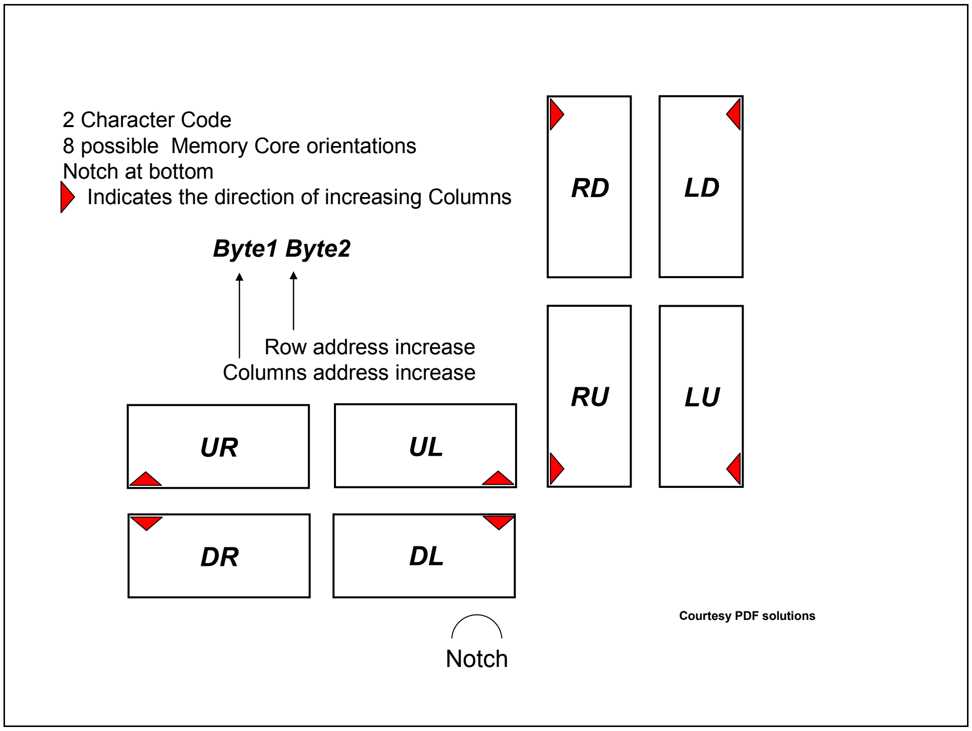


Figure R6-1(a)
Memory Orientations for Wafer

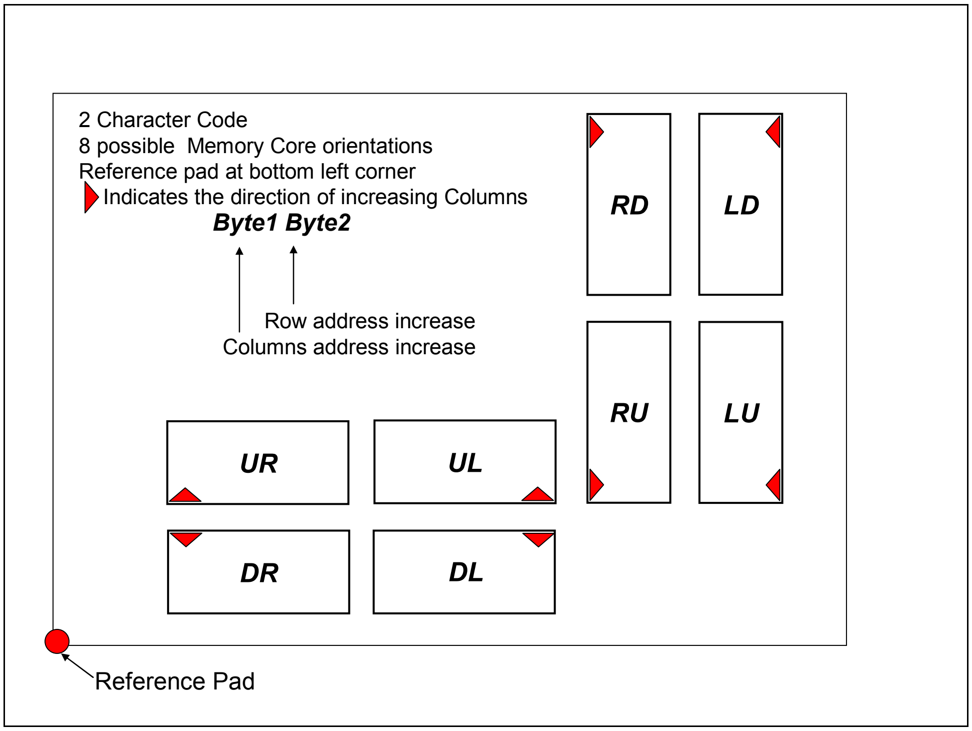

2 Character Code
8 possible Memory Core orientations
Reference pad at bottom left corner
Indicates the direction of increasing Columns
Byte1 Byte2
Row address increase
Columns address increase
RD
LD
RU
LU
UR
UL
DR
DL
Reference Pad

Figure R6-1 (b)
Memory Orientation for Package

R6-1.5 Figure R6-2 shows convention used in assigning codes to rows and columns. Figure R6-3 shows an example of a memory with its code.

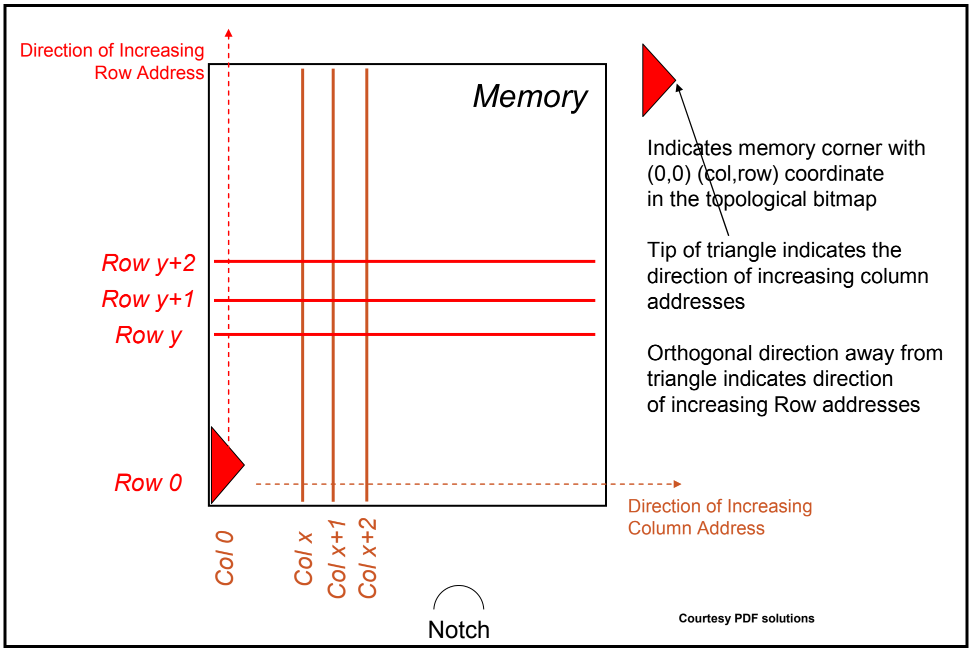
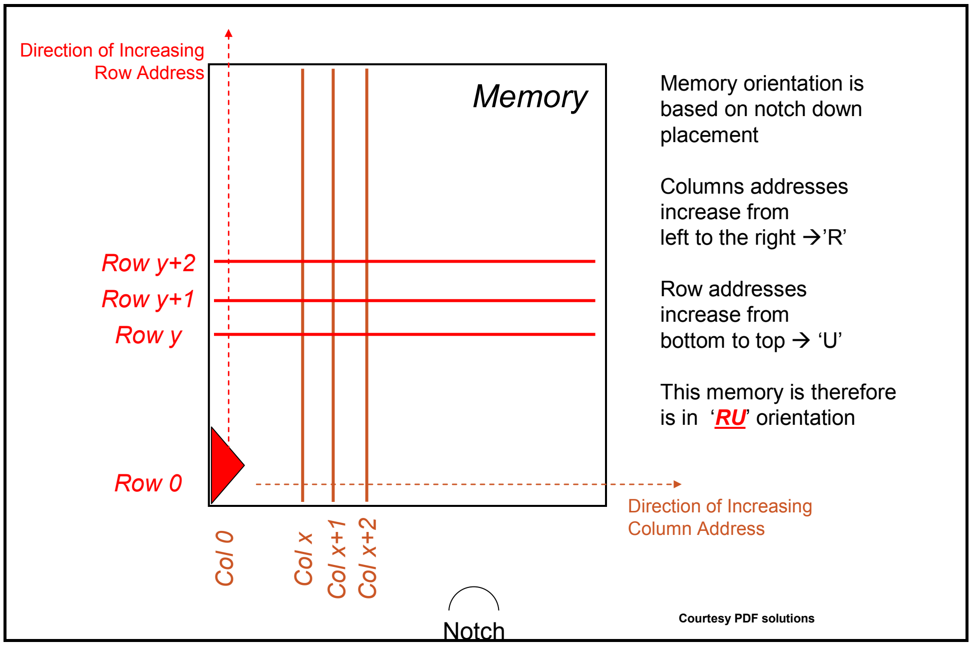

Direction of Increasing
Row Address
Memory
Indicates memory corner with
(0,0) (col,row) coordinate
in the topological bitmap
Tip of triangle indicates the
direction of increasing column
addresses
Orthogonal direction away from
triangle indicates direction
of increasing Row addresses
Row y+2
Row y+1
Row y
Row 0
Direction of Increasing
Column Address
Col 0
Col x
Col x+1
Col x+2
Notch
Courtesy PDF solutions

Figure R6-2
Memory Orientation Conventions



Memory orientation is based on notch down placement

Columns addresses increase from left to the right →'R'

Row addresses increase from bottom to top → 'U'

This memory is therefore is in '***RU***' orientation

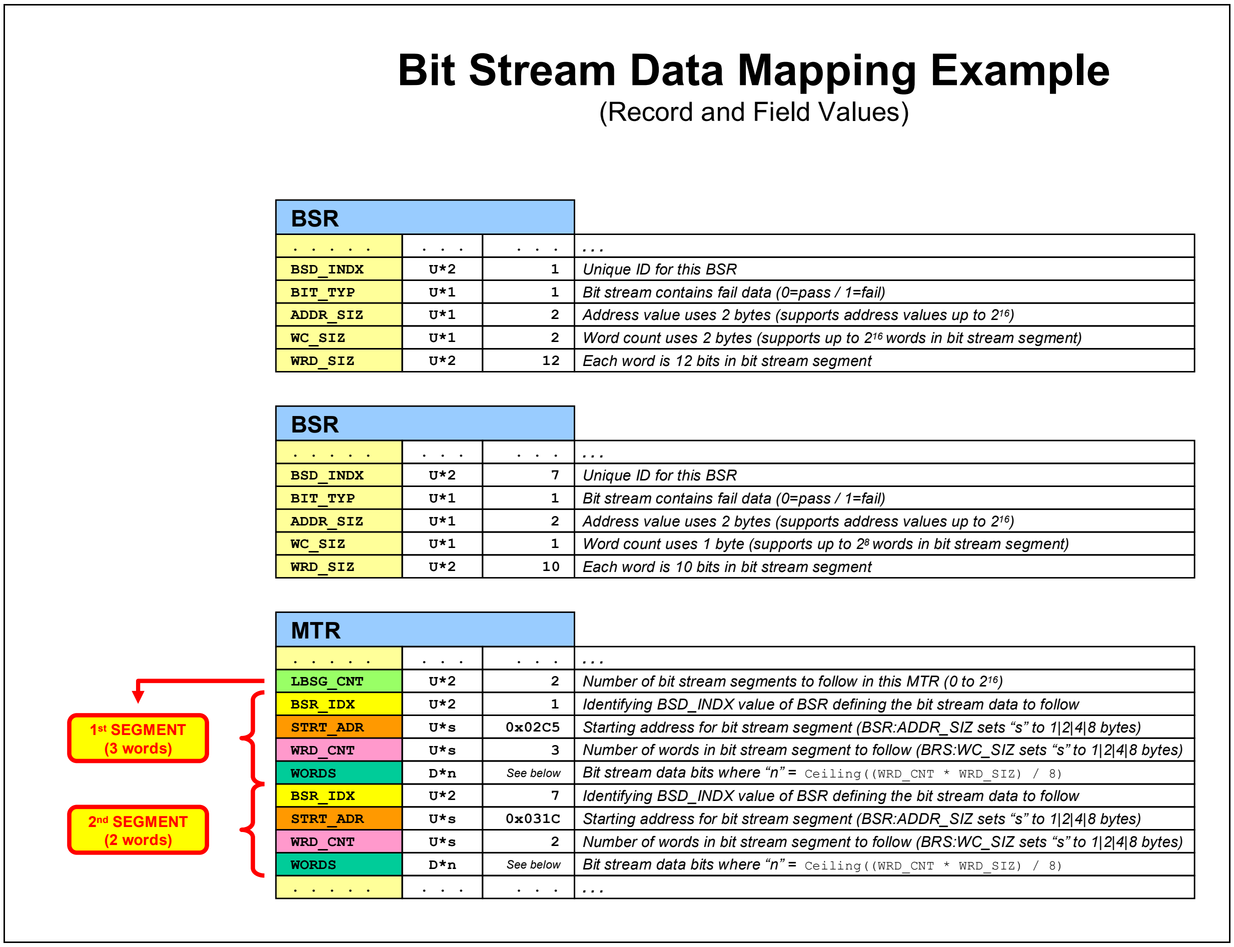
Courtesy PDF solutions

Figure R6-3
Example Coding

RELATED INFORMATION 7
EXAMPLES OF BIT STREAM DATA LAYOUT

R7-1 Bit Stream Data Mapping

R7-1.1 The Figure R7-1 shows the layout of the bits in a bit stream based logging for an example.



Bit Stream Data Mapping Example

(Record and Field Values)

BSR			
. . . . .	. . .	. . .	. . .
BSD_INDX	U*2	1	Unique ID for this BSR
BIT_TYP	U*1	1	Bit stream contains fail data (0=pass / 1=fail)
ADDR_SIZ	U*1	2	Address value uses 2 bytes (supports address values up to 2^{16})
WC_SIZ	U*1	2	Word count uses 2 bytes (supports up to 2^{16} words in bit stream segment)
WRD_SIZ	U*2	12	Each word is 12 bits in bit stream segment

BSR			
. . . . .	. . .	. . .	. . .
BSD_INDX	U*2	7	Unique ID for this BSR
BIT_TYP	U*1	1	Bit stream contains fail data (0=pass / 1=fail)
ADDR_SIZ	U*1	2	Address value uses 2 bytes (supports address values up to 2^{16})
WC_SIZ	U*1	1	Word count uses 1 byte (supports up to 2^8 words in bit stream segment)
WRD_SIZ	U*2	10	Each word is 10 bits in bit stream segment

MTR			
. . . . .	. . .	. . .	. . .
LBSG_CNT	U*2	2	Number of bit stream segments to follow in this MTR (0 to 2^{16})
BSR_IDX	U*2	1	Identifying BSD_INDX value of BSR defining the bit stream data to follow
STRT_ADR	U*s	0x02C5	Starting address for bit stream segment (BSR:ADDR_SIZ sets "s" to 1|2|4|8 bytes)
WRD_CNT	U*s	3	Number of words in bit stream segment to follow (BRS:WC_SIZ sets "s" to 1|2|4|8 bytes)
WORDS	D*n	See below	Bit stream data bits where "n" = Ceiling((WRD_CNT * WRD_SIZ) / 8)
BSR_IDX	U*2	7	Identifying BSD_INDX value of BSR defining the bit stream data to follow
STRT_ADR	U*s	0x031C	Starting address for bit stream segment (BSR:ADDR_SIZ sets "s" to 1|2|4|8 bytes)
WRD_CNT	U*s	2	Number of words in bit stream segment to follow (BRS:WC_SIZ sets "s" to 1|2|4|8 bytes)
WORDS	D*n	See below	Bit stream data bits where "n" = Ceiling((WRD_CNT * WRD_SIZ) / 8)
. . . . .	. . .	. . .	. . .

Figure R7-1

Example of layout of data bits in a Bit Stream based Log (1)

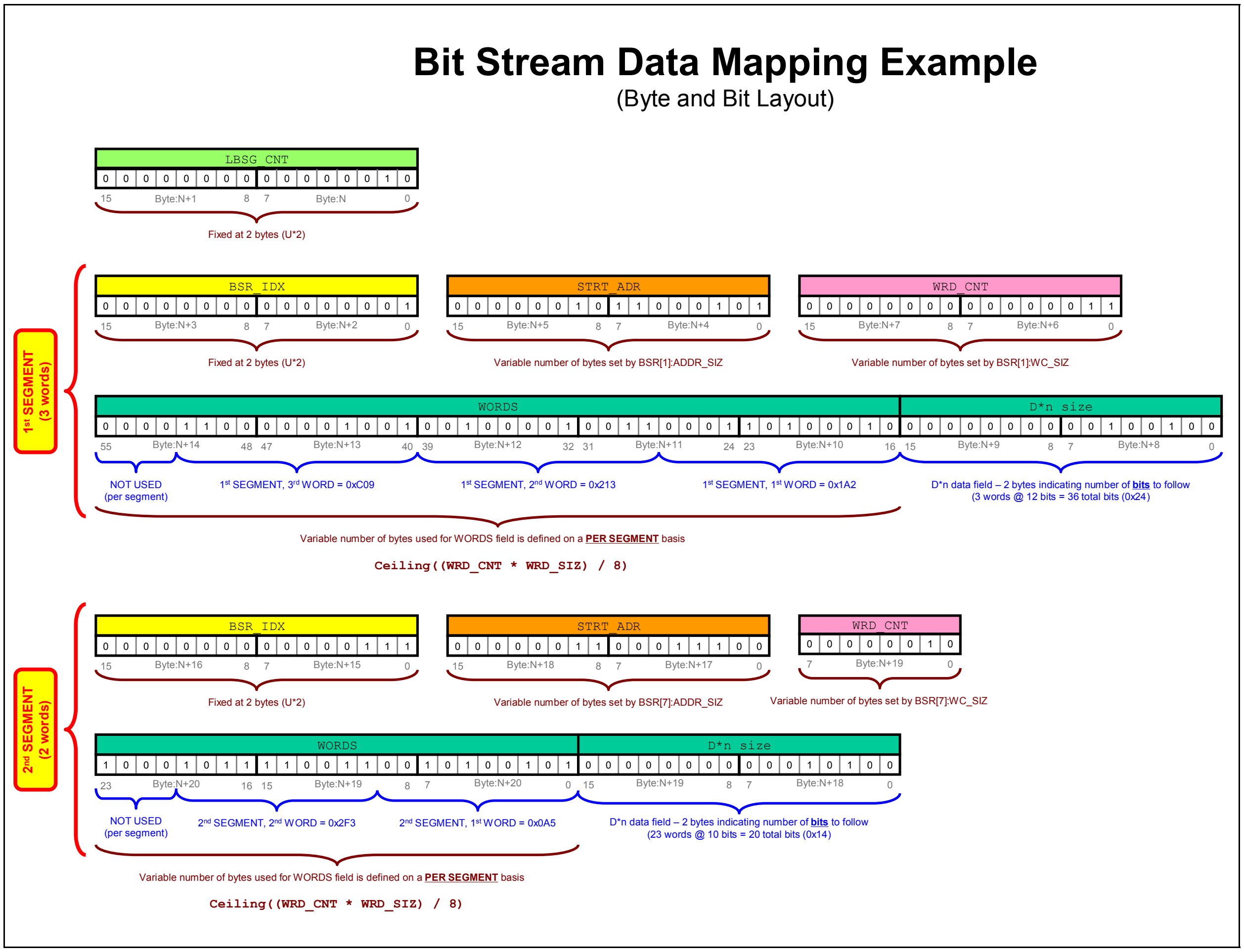

Bit Stream Data Mapping Example
(Byte and Bit Layout)
LBSG_CNT
0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 1 0
15 Byte:N+1 8 7 Byte:N 0
Fixed at 2 bytes (U*2)
1st SEGMENT (3 words)
BSR_IDX
0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 1
15 Byte:N+3 8 7 Byte:N+2 0
Fixed at 2 bytes (U*2)
STRT_ADR
0 0 0 0 0 0 1 0 | 1 1 0 0 0 1 0 1
15 Byte:N+5 8 7 Byte:N+4 0
Variable number of bytes set by BSR[1]:ADDR_SIZ
WRD_CNT
0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 1 1
15 Byte:N+7 8 7 Byte:N+6 0
Variable number of bytes set by BSR[1]:WC_SIZ
WORDS
0 0 0 0 1 1 0 0 | 0 0 0 0 1 0 0 1 | 0 0 1 0 0 0 0 1 | 0 0 1 1 0 0 0 1 | 1 0 1 0 0 0 1 0
55 Byte:N+14 48 47 Byte:N+13 40 39 Byte:N+12 32 31 Byte:N+11 24 23 Byte:N+10 16
D*n size
0 0 0 0 0 0 0 0 | 0 0 1 0 0 1 0 0
15 Byte:N+9 8 7 Byte:N+8 0
NOT USED (per segment)
1st SEGMENT, 3rd WORD = 0xC09
1st SEGMENT, 2nd WORD = 0x213
1st SEGMENT, 1st WORD = 0x1A2
D*n data field – 2 bytes indicating number of bits to follow (3 words @ 12 bits = 36 total bits (0x24)
Variable number of bytes used for WORDS field is defined on a PER SEGMENT basis
Ceiling((WRD_CNT * WRD_SIZ) / 8)
2nd SEGMENT (2 words)
BSR_IDX
0 0 0 0 0 0 0 0 | 0 0 0 0 0 1 1 1
15 Byte:N+16 8 7 Byte:N+15 0
Fixed at 2 bytes (U*2)
STRT_ADR
0 0 0 0 0 0 1 1 | 0 0 0 1 1 1 0 0
15 Byte:N+18 8 7 Byte:N+17 0
Variable number of bytes set by BSR[7]:ADDR_SIZ
WRD_CNT
0 0 0 0 0 0 1 0
7 Byte:N+19 0
Variable number of bytes set by BSR[7]:WC_SIZ
WORDS
1 0 0 0 1 0 1 1 | 1 1 0 0 1 1 0 0 | 1 0 1 0 0 1 0 1
23 Byte:N+20 16 15 Byte:N+19 8 7 Byte:N+20 0
D*n size
0 0 0 0 0 0 0 0 | 0 0 0 1 0 1 0 0
15 Byte:N+19 8 7 Byte:N+18 0
NOT USED (per segment)
2nd SEGMENT, 2nd WORD = 0x2F3
2nd SEGMENT, 1st WORD = 0x0A5
D*n data field – 2 bytes indicating number of bits to follow (23 words @ 10 bits = 20 total bits (0x14)
Variable number of bytes used for WORDS field is defined on a PER SEGMENT basis
Ceiling((WRD_CNT * WRD_SIZ) / 8)

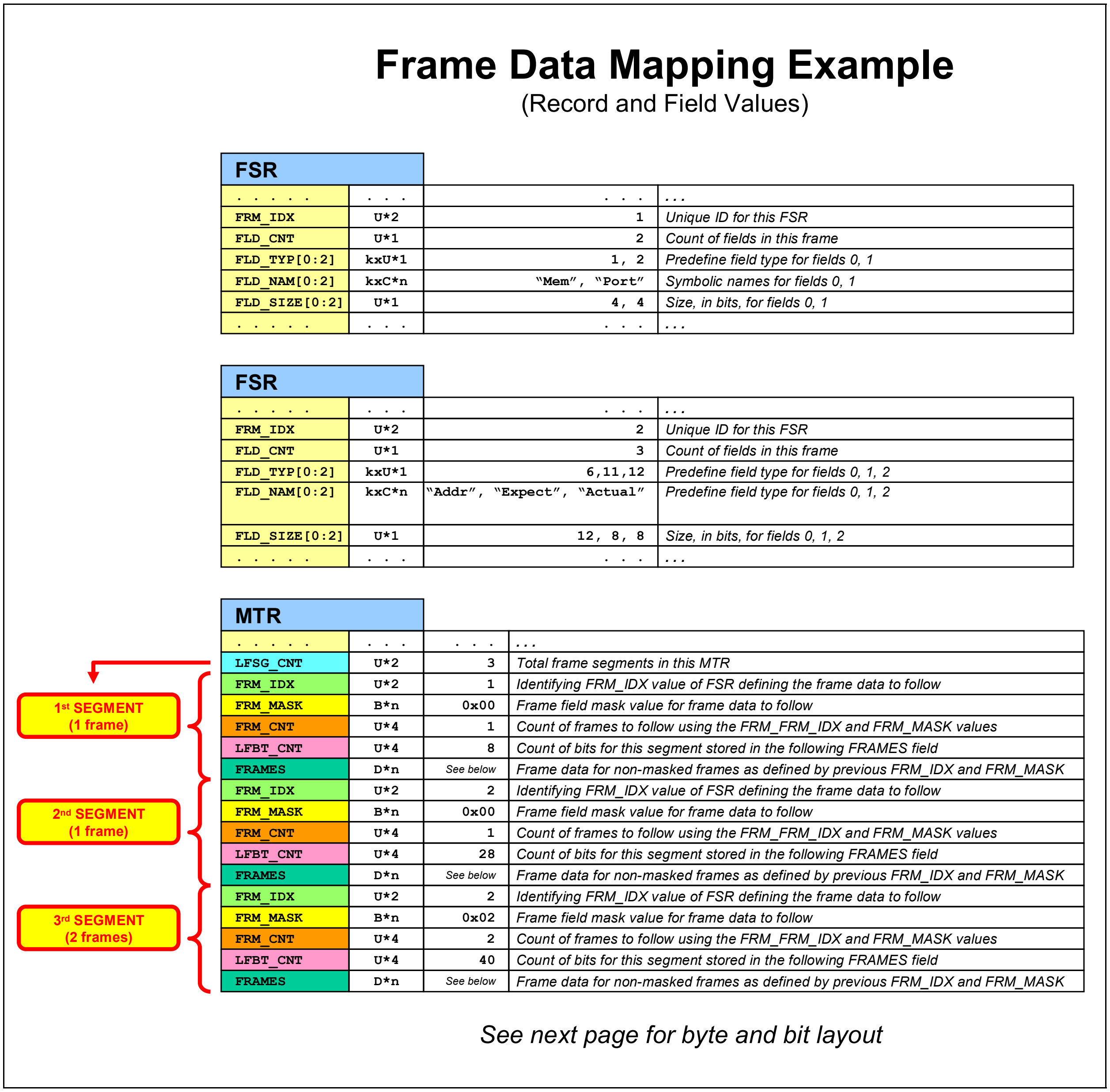
Figure R7-2

Example of layout of data bits in a Bit Stream based Log (2)

RELATED INFORMATION 8
EXAMPLE OF BITS LAYOUT IN FRAME BASED LOGGING

R8-1 Bits Layout in Frame Based Logging

R8-1.1 Figure R8-1 and Figure R8-2 show an example of data bit layout in frame based datalog.



Frame Data Mapping Example

(Record and Field Values)

FSR

. . . . . .	. . .	. . .	. . .
FRM_IDX	U*2	1	*Unique ID for this FSR*
FLD_CNT	U*1	2	*Count of fields in this frame*
FLD_TYP[0:2]	kxU*1	1, 2	*Predefine field type for fields 0, 1*
FLD_NAM[0:2]	kxC*n	"Mem", "Port"	*Symbolic names for fields 0, 1*
FLD_SIZE[0:2]	U*1	4, 4	*Size, in bits, for fields 0, 1*
. . . . . .	. . .	. . .	. . .

FSR

. . . . . .	. . .	. . .	. . .
FRM_IDX	U*2	2	*Unique ID for this FSR*
FLD_CNT	U*1	3	*Count of fields in this frame*
FLD_TYP[0:2]	kxU*1	6,11,12	*Predefine field type for fields 0, 1, 2*
FLD_NAM[0:2]	kxC*n	"Addr", "Expect", "Actual"	*Predefine field type for fields 0, 1, 2*
FLD_SIZE[0:2]	U*1	12, 8, 8	*Size, in bits, for fields 0, 1, 2*
. . . . . .	. . .	. . .	. . .

MTR

. . . . . .	. . .	. . .	. . .
LFSG_CNT	U*2	3	*Total frame segments in this MTR*
FRM_IDX	U*2	1	*Identifying FRM_IDX value of FSR defining the frame data to follow*
FRM_MASK	B*n	0x00	*Frame field mask value for frame data to follow*
FRM_CNT	U*4	1	*Count of frames to follow using the FRM_FRM_IDX and FRM_MASK values*
LFBT_CNT	U*4	8	*Count of bits for this segment stored in the following FRAMES field*
FRAMES	D*n	See below	*Frame data for non-masked frames as defined by previous FRM_IDX and FRM_MASK*
FRM_IDX	U*2	2	*Identifying FRM_IDX value of FSR defining the frame data to follow*
FRM_MASK	B*n	0x00	*Frame field mask value for frame data to follow*
FRM_CNT	U*4	1	*Count of frames to follow using the FRM_FRM_IDX and FRM_MASK values*
LFBT_CNT	U*4	28	*Count of bits for this segment stored in the following FRAMES field*
FRAMES	D*n	See below	*Frame data for non-masked frames as defined by previous FRM_IDX and FRM_MASK*
FRM_IDX	U*2	2	*Identifying FRM_IDX value of FSR defining the frame data to follow*
FRM_MASK	B*n	0x02	*Frame field mask value for frame data to follow*
FRM_CNT	U*4	2	*Count of frames to follow using the FRM_FRM_IDX and FRM_MASK values*
LFBT_CNT	U*4	40	*Count of bits for this segment stored in the following FRAMES field*
FRAMES	D*n	See below	*Frame data for non-masked frames as defined by previous FRM_IDX and FRM_MASK*

See next page for byte and bit layout

Figure R8-1
Data bit layout in a Frame based datalog

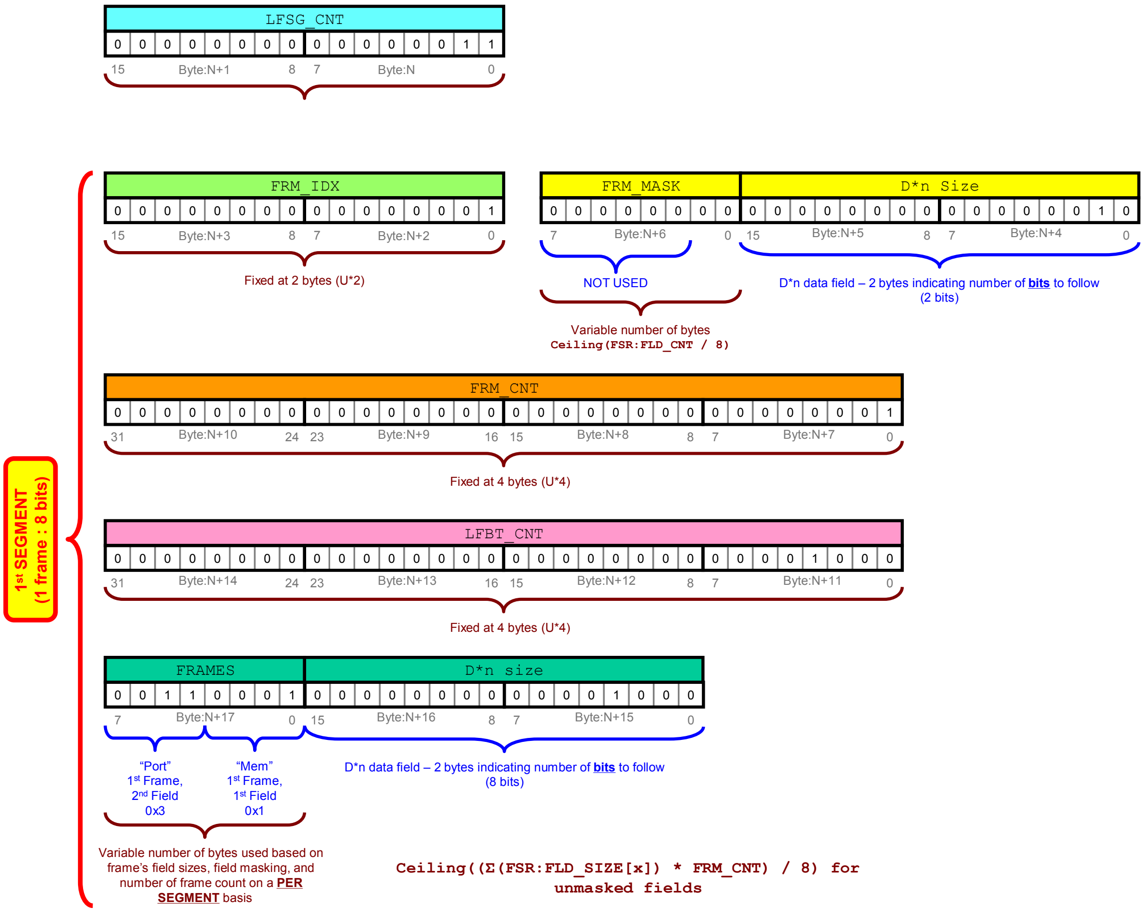


Figure R8-2
Bit Data bit layout in a Frame based datalog (Contd)

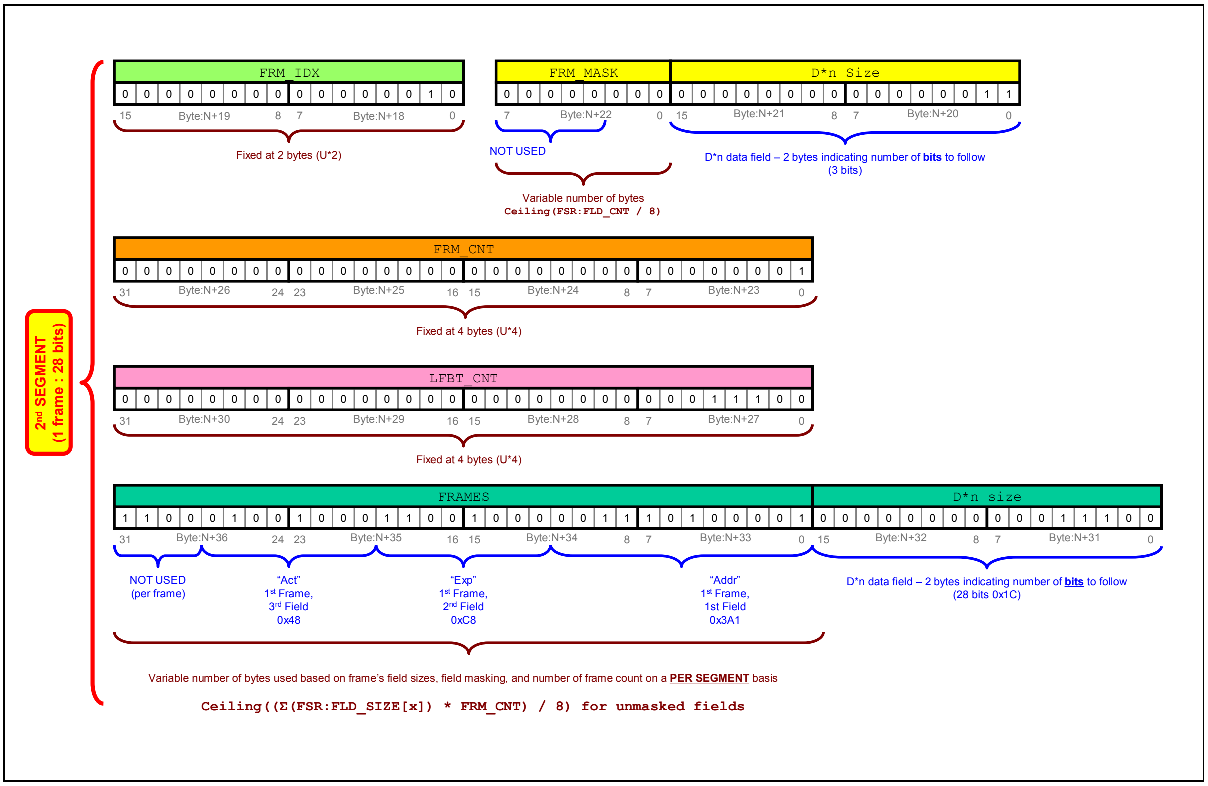


Figure R8-3
Bit Data bit layout in a Frame based datalog (Contd)

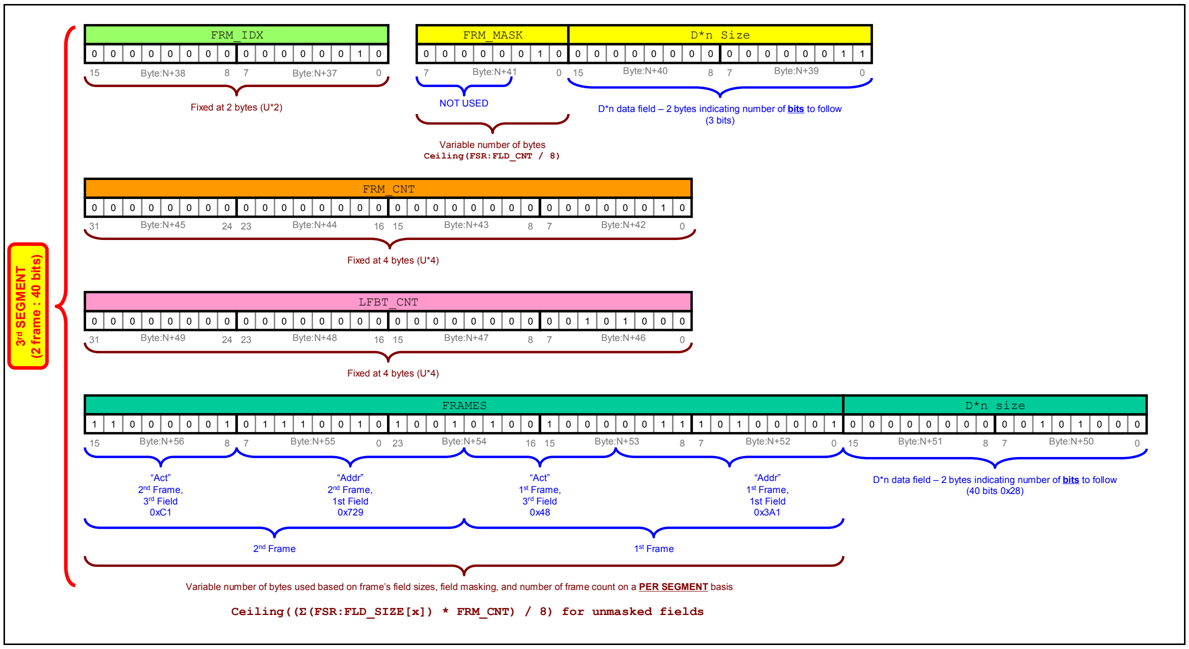
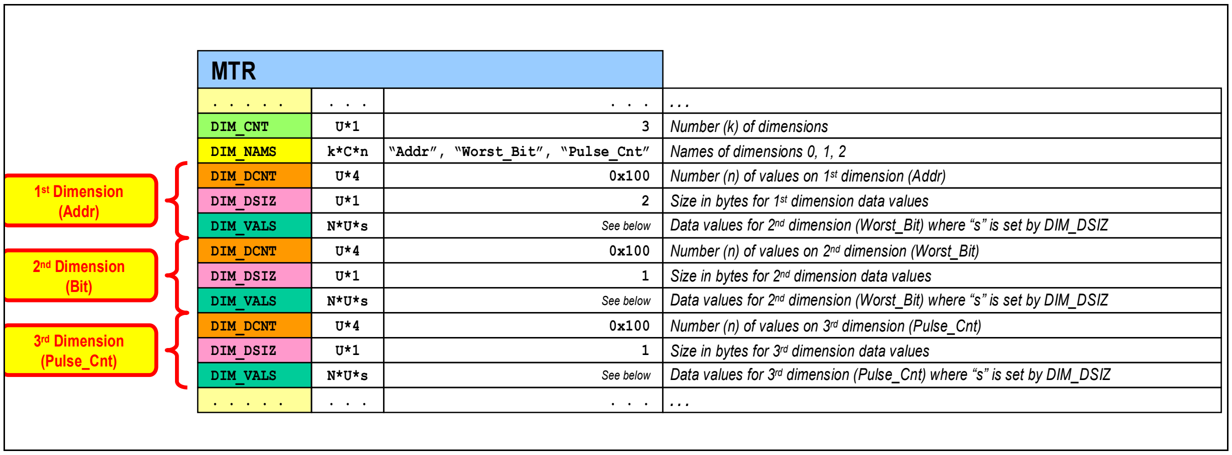


Figure R8-4
Bit Data bit layout in a Frame based datalog (Contd)

RELATED INFORMATION 9
EXAMPLE OF GENERIC MULTI-DIMENSIONAL LOGGING

R9-1 Example of Data Layout in Generic Multi-Dimensional Logging

R9-1.1 Figure **R9-1**, **R9-2** and **R9-3** show and example of data layout in generic multi-dimension log



. . . . .	. . .	. . .	...
DIM_CNT	U*1	3	Number (k) of dimensions
DIM_NAMS	k*C*n	"Addr", "Worst_Bit", "Pulse_Cnt"	Names of dimensions 0, 1, 2
DIM_DCNT	U*4	0x100	Number (n) of values on 1st dimension (Addr)
DIM_DSIZ	U*1	2	Size in bytes for 1st dimension data values
DIM_VALS	N*U*s	See below	Data values for 2nd dimension (Worst_Bit) where "s" is set by DIM_DSIZ
DIM_DCNT	U*4	0x100	Number (n) of values on 2nd dimension (Worst_Bit)
DIM_DSIZ	U*1	1	Size in bytes for 2nd dimension data values
DIM_VALS	N*U*s	See below	Data values for 2nd dimension (Worst_Bit) where "s" is set by DIM_DSIZ
DIM_DCNT	U*4	0x100	Number (n) of values on 3rd dimension (Pulse_Cnt)
DIM_DSIZ	U*1	1	Size in bytes for 3rd dimension data values
DIM_VALS	N*U*s	See below	Data values for 3rd dimension (Pulse_Cnt) where "s" is set by DIM_DSIZ
. . . . .	. . .	. . .	...

Figure R9-1
Example of Generic Multi-dimensional Data logging

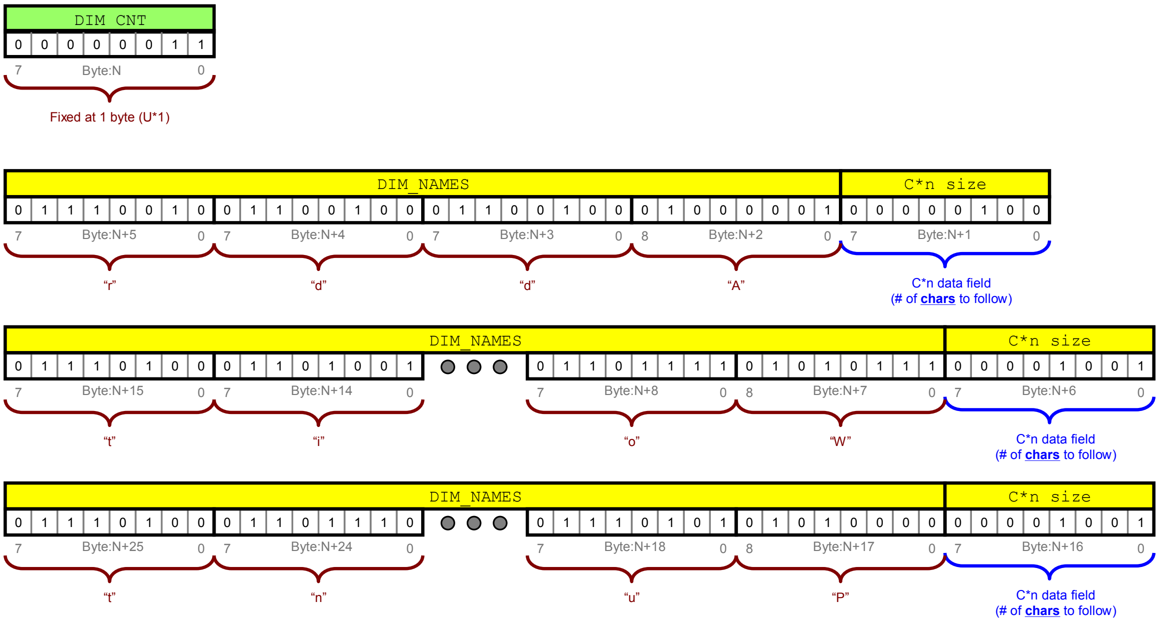


Figure R9-2
Example of Generic Multi-dimensional Datalogging (Contd)

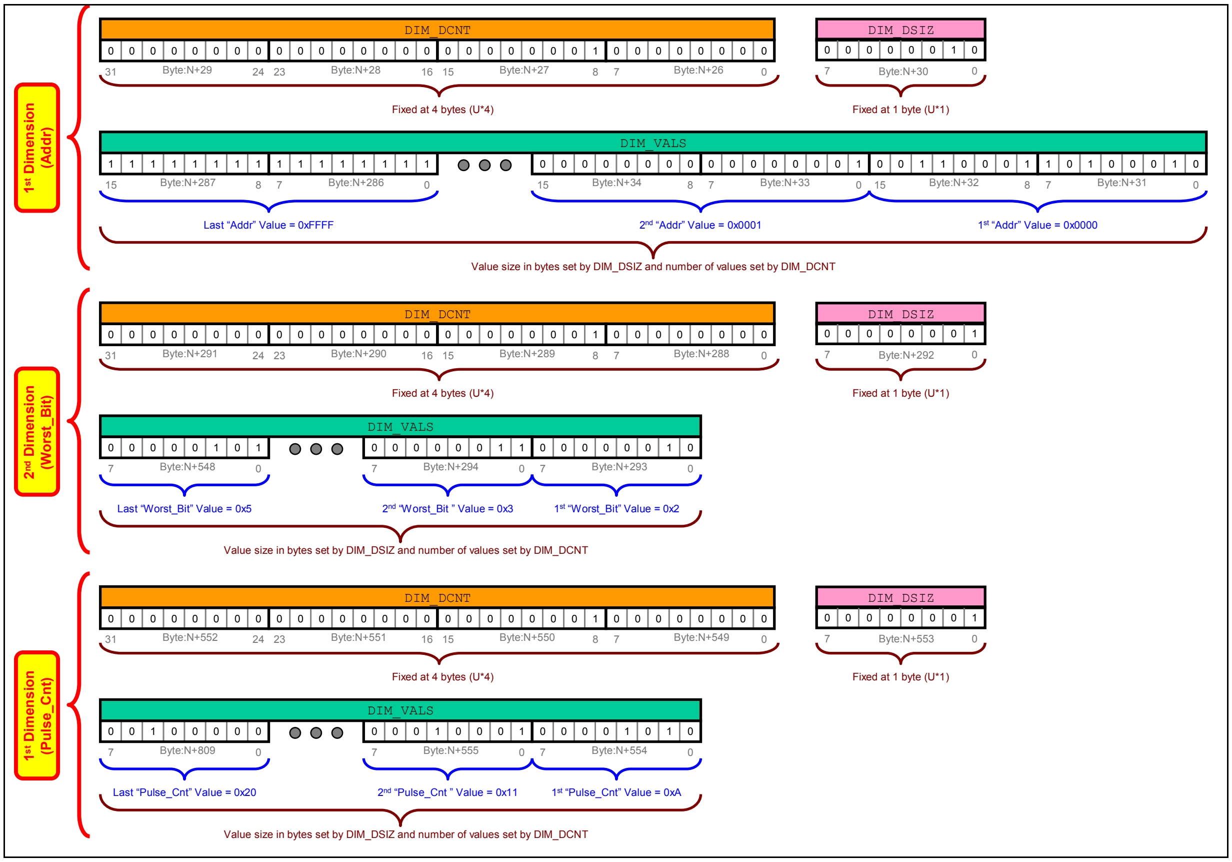

1st Dimension (Addr)
DIM_DCNT
0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 1 | 0 0 0 0 0 0 0 0
31 Byte:N+29 24 23 Byte:N+28 16 15 Byte:N+27 8 7 Byte:N+26 0
Fixed at 4 bytes (U*4)
DIM_DSIZ
0 0 0 0 0 0 1 0
7 Byte:N+30 0
Fixed at 1 byte (U*1)
DIM_VALS
1 1 1 1 1 1 1 1 | 1 1 1 1 1 1 1 1 ● ● ● 0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 1 | 0 0 1 1 0 0 0 1 | 1 0 1 0 0 0 1 0
15 Byte:N+287 8 7 Byte:N+286 0
15 Byte:N+34 8 7 Byte:N+33 0 15 Byte:N+32 8 7 Byte:N+31 0
Last "Addr" Value = 0xFFFF
2nd "Addr" Value = 0x0001
1st "Addr" Value = 0x0000
Value size in bytes set by DIM_DSIZ and number of values set by DIM_DCNT
2nd Dimension (Worst_Bit)
DIM_DCNT
0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 1 | 0 0 0 0 0 0 0 0
31 Byte:N+291 24 23 Byte:N+290 16 15 Byte:N+289 8 7 Byte:N+288 0
Fixed at 4 bytes (U*4)
DIM_DSIZ
0 0 0 0 0 0 0 1
7 Byte:N+292 0
Fixed at 1 byte (U*1)
DIM_VALS
0 0 0 0 0 1 0 1 ● ● ● 0 0 0 0 0 0 1 1 | 0 0 0 0 0 0 1 0
7 Byte:N+548 0
7 Byte:N+294 0 7 Byte:N+293 0
Last "Worst_Bit" Value = 0x5
2nd "Worst_Bit " Value = 0x3
1st "Worst_Bit" Value = 0x2
Value size in bytes set by DIM_DSIZ and number of values set by DIM_DCNT
1st Dimension (Pulse_Cnt)
DIM_DCNT
0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 1 | 0 0 0 0 0 0 0 0
31 Byte:N+552 24 23 Byte:N+551 16 15 Byte:N+550 8 7 Byte:N+549 0
Fixed at 4 bytes (U*4)
DIM_DSIZ
0 0 0 0 0 0 0 1
7 Byte:N+553 0
Fixed at 1 byte (U*1)
DIM_VALS
0 0 1 0 0 0 0 0 ● ● ● 0 0 0 1 0 0 0 1 | 0 0 0 0 1 0 1 0
7 Byte:N+809 0
7 Byte:N+555 0 7 Byte:N+554 0
Last "Pulse_Cnt" Value = 0x20
2nd "Pulse_Cnt " Value = 0x11
1st "Pulse_Cnt" Value = 0xA
Value size in bytes set by DIM_DSIZ and number of values set by DIM_DCNT

Figure R9-3
Example of Multi-dimensional Datalogging (Contd)

NOTICE: SEMI makes no warranties or representations as to the suitability of the standard(s) set forth herein for any particular application. The determination of the suitability of the standard(s) is solely the responsibility of the user. Users are cautioned to refer to manufacturer's instructions, product labels, product data sheets, and other relevant literature respecting any materials or equipment mentioned herein. These standards are subject to change without notice.

By publication of this standard, Semiconductor Equipment and Materials International (SEMI) takes no position respecting the validity of any patent rights or copyrights asserted in connection with any item mentioned in this standard. Users of this standard are expressly advised that determination of any such patent rights or copyrights, and the risk of infringement of such rights are entirely their own responsibility.