High level design of a file operations log

By Nikita Danilov <nikita_danilov@xyratex.com>
Date: 2010/09/06
Revision: 1.0

[Text in square brackets and with a light-green background is a commentary explaining the structure of a design document. The rest is a fictional design document, used as a running example.]

This document presents a high level design (HLD) of a file operations log (a *fol*) of Mero M0 core. The main purposes of this document are: (i) to be inspected by M0 architects and peer designers to ascertain that high level design is aligned with M0 architecture and other designs, and contains no defects, (ii) to be a source of material for Active Reviews of Intermediate Design (ARID) and detailed level design (DLD) of the same component, (iii) to serve as a design reference document.

The intended audience of this document consists of M0 customers, architects, designers and developers.

0. Introduction

[This section succinctly introduces the subject matter of the design. 1--2 paragraphs.]

A fol is a central M0 data-structure, maintained by every node where M0 core is deployed and serving multiple goals:

- it is used by a node data-base component to implement local transactions through WAL logging;
- it is used by DTM to implement distributed transactions. DTM uses fol for multiple purposes internally:
  - on a node where a transaction originates: for replay;
  - on a node where a transaction update is executed: for undo, redo and for replay (sending redo requests to recovering nodes);
  - to determine when a transaction becomes stable;
- it is used by a cache pressure handler to determine what cached updates have to be re-integrated into upward caches;
- it is used by FDML to feed file system updates to fol consumers asynchronously;
- more generally, a fol is used by various components (snapshots, addb, *etc.*) to consistently reconstruct the system state as at a certain moment in the (logical) past.

Roughly speaking, a fol is a partially ordered collection of fol records, each corresponding to (part of) a consistent modification of file system state. A fol record contains information determining durability of the modification (how many volatile and persistent copies it has and where, *etc.*) and dependencies between modifications, among other things.

When a client node has to modify a file system state to serve a system call from a user, it places a record it its (possibly volatile) fol. The record keeps track of operation state: has it been re-integrated to servers, has it been committed on the servers, *etc.* A server, on receiving a request to execute an update on a client behalf, inserts a record, describing the request into its fol. Eventually, fol is purged to reclaim storage, culling some of the records.

1. Definitions

[Definitions of terms and concepts used by the design go here. The definitions must be as precise as possible. References to the M0 Glossary are permitted and encouraged. Agreed upon terminology should be incorporated in the glossary.]

- a (file system) *operation* is a modification of a file system state preserving file system consistency (*i.e.*, when applied to a file system in a consistent state it produces consistent state). There is a limited repertoire of operation types: mkdir, link, create, write, truncate, *etc.* M0 core maintains *serializability* of operation execution;

- an *update* (of an operation) is a sub-modification of a file system state that modifies state on a single node only. For example, a typical write operation against a RAID-6 striped file includes updates which modify data blocks on a server A and updates which modify parity blocks on a server B;

- an operation or update *undo* is a reversal of state modification, restoring original state. An operation can be undone only when the parts of state it modifies are compatible with the operation having been executed. Similarly, an operation or update *redo* is modifying state in the "forward" direction, possibly after undo;

- the *recovery* is a distributed process of restoring file system consistency after a failure. M0 core implements recovery in terms of undoing and redoing individual updates in a coherent way;

- an update (or more generally, a piece of a file system state) is *persistent* when it is recorded on a persistent storage, where persistent storage is defined as one whose contents survives a reset (a particular type of failure, to be defined in the Glossary);

- an operation (or more generally, a piece of a file system state) is *durable* when it has enough persistent copies to survive any failure (as defined in the Glossary). Note that even as a record of a durable operation exists after a failure, the recovery might decide to undo the operation to preserve overall system consistency. Also note that the notion of failure is configuration dependent;

- an operation is *stable* when it is guaranteed that the system state would be consistent with this operation having been executed. A stable operation is always durable. Additionally, M0 core guarantees that the recovery would never undo the operation. The system can renege stability guarantee only in the face of a catastrophe or a system failure (as defined in the Glossary);

- updates U and V are *conflicting* or *non-commutative* if the final system state after U and W are executed depends on the relative order of their execution (note that system state includes information and result codes returned to the user applications); otherwise the updates are *commutative*. The later of two non-commutative updates *depends* on the earlier one. The earlier one is a *pre-requisite* of a later one (this ordering is well-defined due to serializability);

- to maintain operation serializability, all conflicting updates of a given file system object must be serialized. An *object version* is a number associated with a storage object with a property that for any two conflicting updates U (a pre-requisite) and V (a dependent update) modifying the object, the object version at the moment of U execution is less than at the moment of V execution;

- a *FOL* of a node is a sequence of records, each describing an update carried out on the node, together with information identifying operations these updates are parts of, file system objects that were updated and their versions, and containing enough data to undo or redo the updates

and to determine operation dependencies;

- a record in a node fol is uniquely identified by a *log sequence number* (lsn). Log sequence numbers have two crucial properties:
    - a fol record can be found efficiently (*i.e.*, without fol scanning) given its lsn, and
    - for any pair of conflicting updates recorded in the fol, the lsn of the pre-requisite is less than that of the dependent update (*Note*: clearly this property implies that lsn data-type has infinite range and, hence, is unimplementable in practice. What is in fact required is that this property holds for any two conflicting updates *sufficiently close in logical time*, where precise closeness condition is defined by the fol pruning algorithm. The same applies to object versions.);

Note: it would be nice to refine the terminology to distinguish between operation description (*i.e.*, intent to carry it out) and its actual execution. This would make description of dependencies and recovery less obscure, at the expense of some additional complexity.

2. Requirements

[This section enumerates requirements collected and reviewed at the Requirements Analysis (RA) and Requirements Inspection (RI) phases of development. References to the appropriate RA and RI documents should go here. In addition this section lists architecture level requirements for the component from the Summary requirements table and appropriate architecture documentation.]

- [r.fol.every-node]: every node where M0 core is deployed maintains fol;
- [r.fol.local-txn]: a node fol is used to implement local transactional containers.

The following requirements from the Summary requirements table are relevant to fol:

- [R.FOL]: A File Operations Log is maintained by M0;
- [R.FOL.VARIABILITY]: FOL supports various system configurations. FOL is maintained by every M0 back-end. FOL stores enough information to efficiently find modifications to the file system state that has to be propagated through the caching graph, and to construct network-optimal messages carrying these updates. A FOL can be maintained in volatile or persistent transactional storage;
- [R.FOL.LSN]: A FOL record is identified by an LSN. There is a compact identifier (LSN, Log Sequence Number) with which a log record can be identified and efficiently located;
- [R.FOL.CONSISTENCY]: A FOL record describes a storage operation. A FOL record describes a complete storage operation, that is, a change to a storage system state that preserves state consistency;
- [R.FOL.IDEMPOTENCY]: A FOL record application is idempotent. A FOL record contains enough information to detect that operation is already applied to the state, guaranteeing EOS (Exactly Once Semantics);
- [R.FOL.ORDERING]: A FOL records are applied in order. A FOL record contains enough information to detect when all necessary pre-requisite state changes have been applied;
- [R.FOL.DEPENDENCIES]: Operation dependencies can be discovered through FOL. FOL contains enough information to determine dependencies between operations;

- [R.FOL.DIX]: FOL supports DIX;
- [R.FOL.SNS]: FOL supports SNS;
- [R.FOL.REINT]: FOL can be used for cache reintegration. FOL contains enough information to find out what has to be re-integrated;
- [R.FOL.PRUNE]: FOL can be pruned. A mechanism exists to determine what portions of FOL can be re-claimed;
- [R.FOL.REPLAY]: FOL records can be replayed;
- [R.FOL.REDO]: FOL can be used for redo-only recovery;
- [R.FOL.UNDO]: FOL can be used for undo-redo recovery;
- [R.FOL.EPOCHS]: FOL records for a given epoch can be found efficiently;
- [R.FOL.CONSUME.SYNC]: storage applications can process FOL records synchronously;
- [R.FOL.CONSUME.ASYNC]: storage applications can process FOL records asynchronously;
- [R.FOL.CONSUME.RESUME]: a storage application can be resumed after a failure;
- [R.FOL.ADDB]: FOL is integrated with ADDB. ADDB records matching a given FOL record can be found efficiently;
- [R.FOL.FILE]: FOL records pertaining to a given file(-set) can be found efficiently.

3. Design highlights

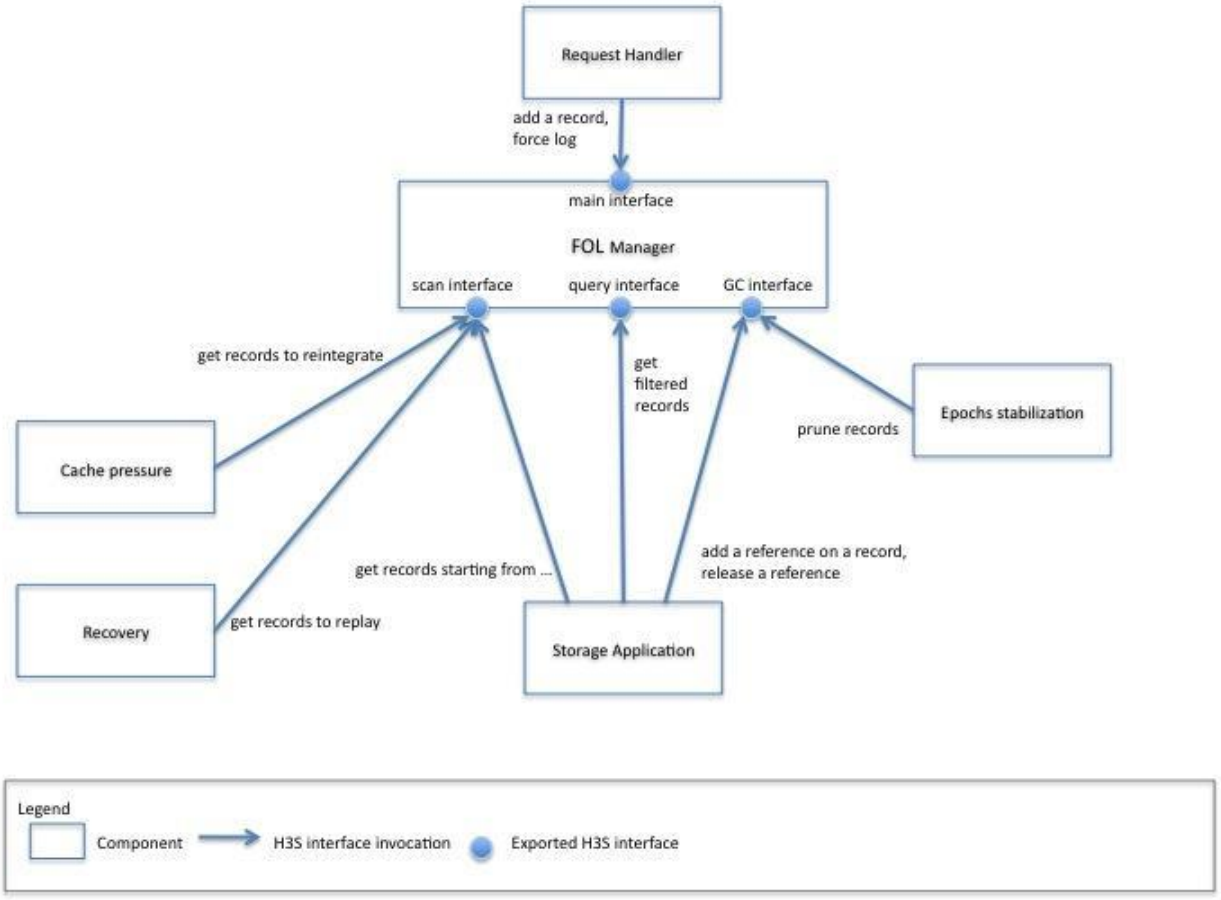
[This section briefly summarises key design decisions that are important for understanding of the functional and logical specifications and enumerates topics that a reader is advised to pay special attention to.]

A fol record is identified by its LSN. LSN are defined and selected as to be able to encode various partial orders imposed on fol records by the requirements.

4. Functional specification

[This section defines a functional structure of the designed component: the decomposition showing *what* the component does to address the requirements.]

A fol context diagram from the FOL architecture view packet:



C&C Shared Data View Packet 0 File Operations Log: Context Diagram, 2009.08.07, Nikita, version 1.

The fol manager exports two interfaces:

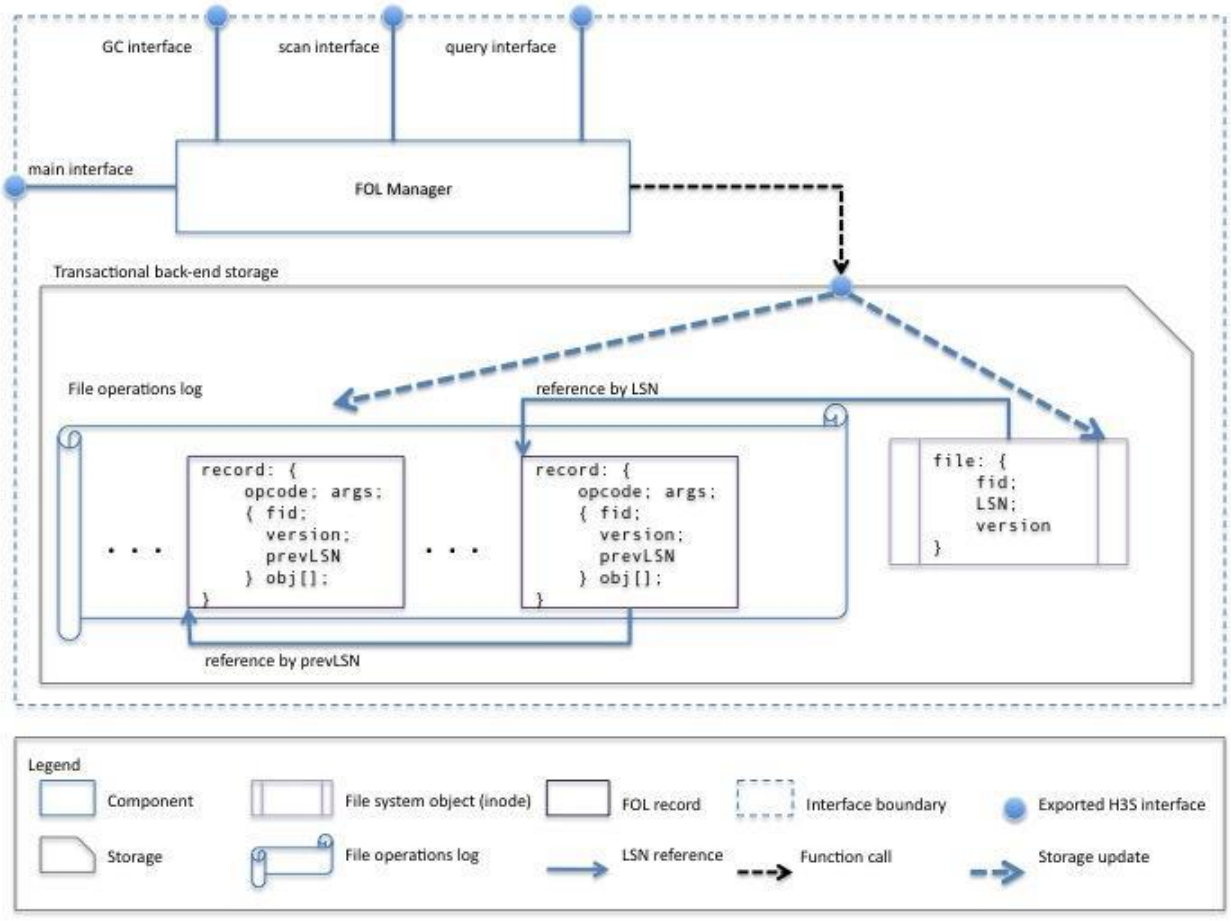
- *main interface* used by the request handler. Through this interface fol records can be added to the fol and the fol can be *forced* (i.e., made persistent up to a certain record);
- *auxiliary interfaces*, used for fol pruning and querying.

5. Logical specification

[This section defines a logical structure of the designed component: the decomposition showing *how* the functional specification is met. Subcomponents and diagrams of their interrelations should go in this section.]

5.a. Overview

Fol is stored in a transactional container[1] populated with records indexed[2] by lsn. An lsn is used to refer to a point in fol from other meta-data tables (epochs table, object index, sessions table, *etc.*). To make such references more flexible, a fol, in addition to genuine records corresponding to updates, might contain pseudo-records marking points on interest in the fol to which other file system tables might want to refer to (for example, an epoch boundary, a snapshot origin, a new server secret key, *etc.*). By abuse of terminology, such pseudo-records will be called fol records too. Similarly, as part of redo-recovery implementation, DTM might populate a node fol with records describing updates to be performed on *other* nodes.



C&C Shared Data View Packet 0 File Operations Log: Primary Presentation, 2009.08.07, Nikita, version 1.

5.b. Record structure

A fol record, added via the main fol interface, contains the following:

[1][R.BACK-END.TRANSACTIONAL] ST

[2][R.BACK-END.INDEXING] ST

- an operation opcode, identifying the type of file system operation;
- lsn;
- information sufficient to undo and redo the update, described by the record, including:
    - for each file system object affected by the update, its identity (a fid) and its object version identifying the state of the object in which the update can be applied;
    - any additional operation type dependent information (file names, attributes, *etc.*) necessary to execute or roll-back the update;
- information sufficient to identify other updates of the same operation (if any) and their state. For the purposes of the present design specification it's enough to posit that this can be done by means of some opaque identifier;
- for each object modified by the update, a reference (in the form of lsn) to the record of the previous update to this object (null is the update is object creation). This reference is called *prev-lsn* reference;
- distributed transaction management data, including an epoch this update and operation are parts of;
- liveness state: a number of outstanding references to this record.

5.c. Liveness and pruning

A node fol must be prunable if only to function correctly on a node without persistent storage. At the same time, a variety of sub-systems both from M0 core and outside of it, might want to refer to fol records. To make pruning possible and flexible, each fol record is augmented with a reference counter, counting all outstanding references to the record. A record can be pruned iff its reference counter drops to 0 together with reference counters of all earlier (in lsn sense) unpruned records in the fol.

5.1. Conformance

[For every requirement in the Requirements section, this sub-section explicitly describes how the requirement is discharged by the design. This section is part of a requirements tracking mechanism, so it should be formatted in some way suitable for (semi-)automatic processing.]

- [r.fol.every-node]: on nodes with persistent storage, M0 core runs in the user space and the fol is stored in a data-base table. On a node without persistent storage, M0 core runs in the kernel space and the fol is stored in memory-only index. Data-base and memory-only index provide the same external interface, making fol code portable;

- [r.fol.local-txn]: request handler inserts a record into FOL table in the context of the same transaction where update is executed. This guarantees WAL property of fol;

- [R.FOL]: vacuous;

- [R.FOL.VARIABILITY]: fol records contain enough information to determine where to forward updates to;

- [R.FOL.LSN]: explicitly by design;
- [R.FOL.CONSISTENCY]: explicitly by design;
- [R.FOL.IDEMPOTENCY]: object versions stored in every fol record are used to implement EOS;
- [R.FOL.ORDERING]: object versions and lsn are used to implement ordering;
- [R.FOL.DEPENDENCIES]: object versions and epoch numbers are used to track operation dependencies;
- [R.FOL.DIX]: distinction between operation and update makes multi-server operations possible;
- [R.FOL.SNS]: same as for r.fol.DIX;
- [R.FOL.REINT]: cache pressure manager on a node keeps a reference to the last re-integrated record using auxiliary fol interface;
- [R.FOL.PRUNE]: explicitly by design;
- [R.FOL.REPLAY]: the same as r.fol.reint: a client keeps a reference to the earliest fol record that might require replay. Liveness rules guarantee that all later records are present in the fol;
- [R.FOL.REDO]: by design fol record contains enough information for update redo. See DTM documentation for details;
- [R.FOL.UNDO]: by design fol record contains enough information for update undo. See DTM documentation for details;
- [R.FOL.EPOCHS]: an epoch table contains references (lsn) of fol (pseudo-)records marking epoch boundaries;
- [R.FOL.CONSUME.SYNC]: request handler feed a fol record to registered synchronous consumers in the same local transaction context where the record is inserted and where the operation is executed;
- [R.FOL.CONSUME.ASYNC]: asynchronous fol consumers receive batches of fol records from multiple nodes and consume them in the context of distributed transactions on which these records are parts of;
- [R.FOL.CONSUME.RESUME]: the same mechanism is used for resumption of fol consumption as for re-integration and replay: a record to the last consumed fol records is updated transactionally with consumption;
- [R.FOL.ADDB]: see ADDB documentation for details;

- [R.FOL.FILE]: an object index table, enumerating all files and file-sets for the node contains references to the latest fol record for the file (or file-set). By following previous operation lsn references the history of modifications of a given file can be recovered.

5.2. Dependencies

[This sub-section enumerates other system and external components the component depends on. For every dependency a type of the dependency (uses, generalizes, *etc.*) must be specified together with the particular properties (requirements, invariants) the design depends upon. This section is part of a requirements tracking mechanism.]

- back-end:
  - [R.BACK-END.TRANSACTIONAL] ST: back-end supports local transactions so that fol could be populated atomically with other tables;
  - [R.BACK-END.INDEXING] ST: back-end supports containers with records indexed by a key.

5.3. Security model

[The security model, if any, is described here.]

FOL manager by itself does not deal with security issues. It trusts its callers (request handler, DTM, *etc.*) to carry out necessary authentication and authorization checks before manipulating fol records. The fol stores some security information as part of its records.

5.4. Refinement

[This sub-section enumerates design level requirements introduces by the design. These requirements are used as input requirements for the detailed level design of the component. This sub-section is part of a requirements tracking mechanism.]

The fol is organized as a single indexed table containing records with lsn as a primary key. The structure of an individual record is outlined above.

Detailed main fol interface is straightforward. Fol navigation and querying in the auxiliary interface are based on a fol cursor.

6. State

[This section describes the additions or modifications to the system state (persistent, volatile) introduced by the component. As much of component behavior from the logical specification should be described as state machines as possible. The following sub-sections are repeated for every state machine.]

Fol introduces no extra state.

7. Use cases

[This section describes how the component interacts with rest of the system and with the outside world.]

7.1. Scenarios

[This sub-section enumerates important use cases (to be later used as seed scenarios for ARID) and describes them in terms of logical specification.]

FOL QAS list is included here by reference.

[UML use case diagram can be used to describe a use case.]

7.2. Failures

[This sub-section defines relevant failures and reaction to them. Invariants maintained across the failures must be clearly stated. Reaction to Byzantine failures (*i.e.*, failures where a compromised component acts to invalidate system integrity) is described here.]

Failure of the underlying storage container in which fol is stored is treated as any storage failure. All other fol related failures are handled by DTM.

8. Analysis

8.1. Scalability

[This sub-section describes how the component reacts to the variation in input and configuration parameters: number of nodes, threads, requests, locks, utilization of resources (processor cycles, network and storage bandwidth, caches), *etc.* Configuration and work-load parameters affecting component behavior must be specified here.]

8.2. Other

[As applicable, this sub-section analyses other aspects of the design, *e.g.*, recoverability of a distributed state consistency, concurrency control issues.]

At alternative design is to store fol in a special data-structure, instead of a standard indexed container. For example, fol can be stored in an append-only flat file with starting offset of a record serving as its lsn. Perceived advantage of this solution is avoiding an overhead of a full-fledged indexing (b-tree). Indeed, general purpose indexing is not needed, because records with lsn less than the maximal one used in the past are never inserted into the fol (aren't they?).

Yet another possible design is to use db4 extensible logging to store fol records directly in a db4 transactional log. The advantage of this is that forcing fol up to a specific record becomes possible (and easy to implement), and the overhead of indexing is again avoided. On the other hand, it is not clear how to deal with pruning.

8.2. Rationale

[This sub-section describes why particular design was selected; what alternatives (alternative designs and variations of the design) were considered and rejected.]

Simplest solution first.

9. Deployment

9.1. Compatibility

[Backward and forward compatibility issues are discussed here. Changes in system invariants (event ordering, failure modes, *etc.*)]

9.1.1. Network

9.1.2. Persistent storage

9.1.3. Core

[Interface changes. Changes to shared in-core data structures.]

9.2. Installation

[How the component is delivered and installed.]

10. References

[References to all external documents (specifications, architecture and requirements documents, *etc.*) are placed here. The rest of the document cites references from this section. Use Google Docs bookmarks to link to the references from the main text.]

[0] FOL QAS
[1] FOL architecture view packet
[2] FOL overview
[3] WAL
[4] Summary requirements table
[5] M0 glossary
[6] HLD of request handler

11. Inspection process data

[The tables below are filled in by design inspectors and reviewers. Measurements and defects are transferred to the appropriate project data-bases as necessary.]

11.1. Logt

	Task	Phase	Part	Date	Planned time (min.)	Actual time (min.)	Comments
Anatoliy	fol	HLDINSP	1		200		
Huang Hua	fol	HLDINSP	1		200		

11.2. Logd

No.	Task	Summary	Reported by	Date reported	Comments
1	fol				
2	fol				
3	fol				
4	fol				
5	fol				
6	fol				
7	fol				
8	fol				
9	fol				