LAT Flight Software

FMX Manual

Type:	User Manual
Version:	V1-1-0
Author:	S.Maldonado
Created:	4 October 2004
Updated:	10 February 2005
Printed:	10 February 2005

Manual for FMX, the LAT FSW file database management tool

Contents

Figures

Tables

0 Problem Overview

The file system on the SIBs (wherever located) will not be a particularly user friendly environment. Management of what files are where is going to be a headache. The tool proposed here is a first attempt to get a handle on the problem. Note that this section is not intended to cover all aspects of the parameter space the tool will have to handle, its intent is rather to give the flavour of the file system environment.

0.0 Why Is This So Hard?

While it is true that the SIBs will carry a fully-fledged DOS file system, FSW and the underlying hardware have placed constraints on layout and operation.

0.0.0 Hardware Constraints

The EEPROM memory used in the SIB does not have a great reputation, and in the high radiation environment of space, we must be prepared for file corruption. The FSW response has been to require that every file be prefixed with a (fixed length) header containing (among other things) a checksum across the header and another checksum across the file. Obviously, the header has to be attached before the file can be uploaded.

0.0.1 Software Constraints

To avoid arbitrary strings in telecommands/telemetry, all commands refer to files using a simple 32-bit integer number. The integer is divided into fields for:

- Device (in this context, lower or upper EEPROM bank)
- Directory
- File

FSW converts this integer algorithmically into a string when it has to talk to the file system. Thus all files in the SIB file system have names like:

```
/ee0/d012/f00000023
```

This parses to “lower EEPROM bank, directory twelve, file twenty-three”.

Note that this restricts the file system to one level of directories.

It was also a design goal that a given file name never be re-used for the duration of the mission. There's enough room in the file name for millions of files so that is not an issue. Reality sets in when ground operations have to assign new file numbers (picture operations staff running around asking "what *was* the highest file number assigned in that directory?")

A related problem is the fact that one "logical" file (e.g. the FSW binary constituent that handles inter-task communication) will be loaded onto multiple SIB boards (one for each CPU). Not only must the "logical" file (version, build date, etc.) be tracked, but the individual instances ("what is the physical file name of this logical file on this board") must be tracked.

1 Solution Overview

The solution proposed here employs the following technologies/techniques:

- A random access database (probably MySQL)
- A well-defined directory layout in which to maintain the files
- A command-line driven script written in Python (`fmx`)

1.0 Relational Database Overview

The design contains two core tables:

- The logical file table.
- The physical file table.

The logical file table records all files to be put into a SIB, but a logical file is not associated with any particular SIB at this level. The primary key for this table is an auto-incrementing number so that all files added to this table get a unique number associated with them. Note that records are never physically deleted from this table (though they can be marked "inactive"). Later sections will specify all the fields in this table in gory detail. In an overview section I will simply note that each file has a file type associated with it and that the primary key and the file type from this record are inserted into the mandatory file header of files being uploaded to a SIB.

Records in the physical file table associate a logical file with a physical file on a SIB board. This is the traditional database "associative record", listing a key from the logical table against the SIB, device, directory and file number from when the file is uploaded to a SIB.

Supporting the core tables are some peripheral tables:

- The SIB tables
- The file type table

The SIB tables are used to maintain a list of hardware targets. The most obvious examples are the real SIBs, but I'd like to see the system support "virtual SIBs" as well. These would be host-based file systems with the lowest level of directory structure emulating the device/directory structure of a real SIB. The idea is to provide a SIB-like target without going through all the pain of the file upload to a real SIB. Why call this the SIB table? It might have been more recognizable if I'd called it the "unit" table. I went with SIB because that's really what gets burned (and we have a bad habit of moving SIBs between units).

The file type table stores information about file transforms. The products of flight software build procedures are not usually directly useable from a SIB. A code module, for instance, must be stripped, compressed and have a header added before it's uploaded. This database table keeps a record of how to perform these transforms for all file types.

1.1 Directory Structure Overview

The directory structure descends from one fixed root. The root is defined in an environment variable (to allow this system to be portable across sites). The environment variable is called $LDB_C_LDB. The next level of directory structure represents major functional divisions:

Root	Name	Use
$LDB_C_LDB	/frdb	"File random access database". The actual MySQL database.
	/lfmx	"Logical file database". Corresponds to the logical file table in the random access database.
	/pfmx	"Physical file database". Corresponds to the physical file table in the random access database.

Table 1 Top level of $LDB_C_LDB directory

The directory structure then varies by major functional division. Note that only SLAC has the /frdb directory.

1.1.0 Directory Structure For Logical Files

The next directory down from /lfmx describes the type of logical file. This corresponds directly to the type specified in the header of an on-board file. Very few types have been assigned so far. I have given two of the types "aliases" (they're just easier to read), but I've also included one other "undefined" type to show how it fits in:

Function	Type	Use
/lfmx	/abs	Absolute code module. Basically, the VxWorks operating system.
	/rel	Relocateable code module. Almost any of our constituents.
	/t0023	Typical use: a configuration file for a specific code module.

Table 2 The /lfmx directory

1.1.0.0 Directory Structure For Logical Code Module Files

Continuing down the tree for types /abs and /rel:

Type	Package	Version	Tag	Constituent	Files
/abs	VXW	V6-2-1	mv2304	vxw_flight	vxw_flight.o vxw_flight.z
				vxw_symbol	vxw_symbol.o vxw_symbol.z
			mcp750	vxw_flight	vxw_flight.o vxw_flight.z
				vxw_symbol	vxw_symbol.o vxw_symbol.z

			rad750	vxw_flight	vxw_flight.o vxw_flight.z
				vxw_symbol	vxw_symbol.o vxw_symbol.z
/rel	MSG	V2-1-2	mv2304	msg_mt	msg_mt.o msg_mt.z
			mcp750	msg_mt	msg_mt.o msg_mt.z

Table 3 The directory structure for code modules

Note that the relocateable modules have had their “lib” prefix removed. I’m hoping that by throwing away trash filename decorations, we can use the constituent name directly as the name that gets embedded in the SIB file header.

1.1.0.1 Directory Structure For Other Type Files

The directory structure for other type files (in my mind, package configuration files read in at task start), is similar to that for code modules. I’m assuming that even configuration files will need versioning through CMX methods in case their format changes. Otherwise, the file database structure is a little looser. The example I’m giving is one particular way a configuration file could be maintained (start with an XML file and convert to some compressed binary format).

Type	Package	Version	Directory	Files
/t0004	LHK	V3-2-1	lhk_config	lhk_config.xml lhk_config.dat
			lhk_schedule	lhk_schedule.xml lhk_schedule.dat

Table 4 Directory structure for non-code-modules

1.1.1 Directory Structure For Physical Files

For real SIBs, the files are on the SIB and not on the host file system. The physical directory is intended to support *virtual* SIBs.

Function	SIB name	Device	Directory	Filename
/pfmx	vSIB001	ee0	d010	f0000000123
/pfmx	VSIB001	boot	n/a	f0000000004
/pfmx	VSIB001	ram	d011	f0000000007

Table 5 Directory structure for physical files

1.2 Command Overview

1.2.0 Adding Files To The Logical File Database

`fmx add [absolute|relocateable|type=nnnn] <pkg> <ver> <variant-args>`

If type is `absolute` or `relocateable` then `<variant-args>`:

`<constituent> --tag=<cmx-tag>`

If type is not `absolute` and not `relocateable` then `<variant-args>`:

`<file-name> --tag=<cmx-tag>`

1.2.1 Adding Files To The Physical File Database

To transfer a file from the logical filebase to the physical filebase should be accompanied in real life by the file upload and commit operations. The command syntax reflects that:

`fmx upload [absolute|relocateable|type=nnnn] <pkg> <ver> <variant-args> <target-args>`

`<variant-args>` follow the same logic as for `fmx add`. `<target-args>` specifies where the upload goes to.

`<target-args> : --sib=<sib-name> --dev=<device> --directory=<nn>`

`<sib-name>` must be a valid SIB name (from the RDB table), `<device>` must be either `ee0` or `ee1` and `<nn>` must be a number between 1 (?) and 127(?).

`fmx commit [absolute|relocateable|type=nnnn] …`

Commit is a separate operation in this model to map to the real life situation where a file is “queued for upload” (and thus has reserved a file name), but has not actually been loaded (and cannot yet be used).

`fmx delete [absolute|relocateable|type=nnnn] …`

Delete is a separate operation in this model to map to the real life situation where a file is deleted from a sib file system. Note that the file is never deleted from the fmx database, but is tagged as “deleted” for the named sib.

2 Specifics

2.0 Database Tables

2.0.0 Logical File Table

```
CREATE TABLE logical_files
(
 logical_id        INT UNSIGNED AUTO_INCREMENT PRIMARY KEY   NOT NULL,
 type_key          INT UNSIGNED                              NOT NULL,
 ext_key           VARCHAR(8)                                NOT NULL,
 package           VARCHAR(32)                               NOT NULL,
 version           VARCHAR(32)                               NOT NULL,
 con_or_name       VARCHAR(32)                               NOT NULL,
 tag_or_ext        VARCHAR(32)                               NOT NULL,
 create_site       VARCHAR(32)                               NOT NULL,
 create_user       VARCHAR(32)                               NOT NULL,
 create_time       TIMESTAMP                                 NOT NULL,
 state             VARCHAR(32)                               NOT NULL,
 hdr_ver           VARCHAR(32)                               NOT NULL,
 fmx_ver           VARCHAR(32)                               NOT NULL,
 input_file        LONGBLOB                                          ,
 output_file       LONGBLOB                                          ,
 UNIQUE( package,version,con_or_name,tag_or_ext ),
 INDEX(type_key),
 FOREIGN KEY(type_key) REFERENCES file_types (type_key)
 ON DELETE CASCADE
 ON UPDATE CASCADE,
 INDEX(ext_key),
 FOREIGN KEY(ext_key) REFERENCES file_exts (ext_key)
 ON DELETE CASCADE
 ON UPDATE CASCADE
) TYPE=InnoDB;
```

Figure 1 Layout of the `logical_files` database table

Column	Meaning
`logical_id`	Unique logical file number for every file added to the file database.
`type_key`	File type. Dan's definition.
`ext_key`	File extension key from file type table

package	Package name
version	Package version
con_or_name	Constituent name if abs or rel. Otherwise a “trivial” file-stem name.
tag_or_ext	CMX tag if abs or rel. Otherwise a file extension
create_site	CMX site name.
create_user	Unix ID of person adding the file.
create_time	Creation timestamp. Listed as Unix time, but database probably has better datestamp facilities. Datestamp should be in UTC, UT, GMT (you get the idea).
state	Logical file state - “created”…
hdr_ver	Version number of command that adds the header to a file.
fmx_ver	Version number of FMX package used to add logical file
input_file	The real, physical input file.
output_file	The real physical output file.

Table 6 Field descriptions for the logical_files database table

2.0.1 Physical File Table

```
CREATE TABLE physical_files
(
  physical_id       INT UNSIGNED AUTO_INCREMENT PRIMARY KEY   NOT NULL,
  logical_id        INT UNSIGNED                              NOT NULL,
  sib_name          VARCHAR(32)                               NOT NULL,
  device            VARCHAR(32)                               NOT NULL,
  directory         SMALLINT                                  NOT NULL,
  file              INT                                       NOT NULL,
  file_id           INT                                       NOT NULL,
  upload_site       VARCHAR(32)                               NOT NULL,
  upload_user       VARCHAR(32)                               NOT NULL,
  upload_time       TIMESTAMP                                 NOT NULL,
  commit_site       VARCHAR(32)                               NOT NULL,
  commit_user       VARCHAR(32)                               NOT NULL,
  commit_time       TIMESTAMP                                 NOT NULL,
  delete_site       VARCHAR(32)                               NOT NULL,
  delete_user       VARCHAR(32)                               NOT NULL,
  delete_time       TIMESTAMP                                 NOT NULL,
  state             VARCHAR(32)                               NOT NULL,
  fmx_ver           VARCHAR(32)                               NOT NULL,
  UNIQUE( physical_id,logical_id, sib_name, device, directory ),
  INDEX(logical_id),
  FOREIGN KEY(logical_id) REFERENCES logical_files (logical_id)
  ON DELETE CASCADE
  ON UPDATE CASCADE,
  INDEX(sib_name),
  FOREIGN KEY(sib_name) REFERENCES sib_names (sib_name)
  ON DELETE CASCADE
  ON UPDATE CASCADE
) TYPE=InnoDB;
```

Figure 2 Layout of the physical_files database table

Column	Meaning

physical_id	Unique physical file number for every file “uploaded”.
logical_id	Key into logical_files table..
sib_name	Key into sib_localize table.
device	Device within SIB (still pondering keying this or just /ee0 /ee1
directory	Directory number (1-127 I think)
file	File number (0-2**24)
file_id	File ID as specified in FILE document
upload_site	CMX site name.
upload_user	Unix ID of person uploading the file.
upload_time	Upload timestamp. Listed as Unix time, but database probably has better datestamp facilities. Datestamp should be in UTC, UT, GMT (you get the idea).
commit_site	CMX site name.
commit_user	Unix ID of person uploading the file.
commit_time	Upload timestamp. Listed as Unix time, but database probably has better datestamp facilities. Datestamp should be in UTC, UT, GMT (you get the idea).
delete_site	CMX site name.
delete_user	Unix ID of person deleting the file.
delete_time	Delete timestamp. Listed as Unix time, but database probably has better datestamp facilities. Datestamp should be in UTC, UT, GMT (you get the idea).
state	“uploading”, “commited”, “deleted”.
fmx_ver	FMX version used to add physical file

Table 7 Field descriptions for the physical_files database table

2.0.2 SIB Tables

```
CREATE TABLE sib_localize
(
  sib_key           INT UNSIGNED AUTO_INCREMENT PRIMARY KEY   NOT NULL,
  localize          VARCHAR(32)                               NOT NULL
) TYPE=InnoDB;
```

Figure 3 Layout of the sib_localize database table

Column	Meaning
sib_key	Unique SIB number used to associate multiple SIB names with a single resolution.
localize	Blank for real SIBs. Directory ($LDB_C_LDB/pfmx relative) for virtual SIBs.

Table 8 Field descriptions for the sib_localize database table

```
CREATE TABLE sib_names
(
  sib_name          VARCHAR(32) PRIMARY KEY                    NOT NULL,
  sib_key           INT UNSIGNED                               NOT NULL,
  site              VARCHAR(32)                                NOT NULL,
  descr             VARCHAR(32),
  INDEX(sib_key),
  FOREIGN KEY(sib_key) REFERENCES sib_localize (sib_key)
  ON DELETE CASCADE
  ON UPDATE CASCADE
) TYPE=InnoDB;
```

Figure 4 Layout of the `sib_names` database table

Column	Meaning
`sib_name`	Trivial SIB name (e.g. GLAT0840 or SEI 789 or vSIB0001)
`sib_key`	Key into `sib_localize` table.
`site`	CMX site where SIB resides
`descr`	SIB describption

Table 9 Field descriptions for the `sib_names` database table

2.0.3 File Type Table

```
CREATE TABLE file_types
(
  type_key          INT UNSIGNED PRIMARY KEY                   NOT NULL,
  command           VARCHAR(32),
  descr             VARCHAR(64)
) TYPE=InnoDB;
```

Figure 5 Layout of the `file_types` database table

Column	Meaning
`type_key`	Unique type identifier.
`command`	Command to transform input file into output file.
`descr`	File type description

Table 10 Field descriptions for the `file_types` database table

2.0.4 File Extension Table

```
CREATE TABLE file_exts
(
 ext_key           VARCHAR(8)                                    NOT NULL,
 type_key          INT UNSIGNED                                  NOT NULL,
 io_type           VARCHAR(8)                                    NOT NULL,
 intermed_flag     VARCHAR(8)                                    NOT NULL,
 comp_flag         VARCHAR(8)                                    NOT NULL,
 descr             VARCHAR(64),
 PRIMARY KEY( ext_key, type_key ),
 INDEX(type_key),
 FOREIGN KEY(type_key) REFERENCES file_types (type_key)
 ON DELETE CASCADE
 ON UPDATE CASCADE
) TYPE=InnoDB;
```

Figure 6 Layout of the `file_exts` database table

Column	Meaning
`ext_key`	Unique extension type identifier
`type_key`	Key into file type table.
`io_type`	Input/output designation {input|output]
`intermed_flag`	Intermediate file designation [yes|no]
`comp_flag`	Compression flag [yes|no]
`descr`	File extension description

Table 11 Field descriptions for the `file_exts` database table

2.1 Command Processing

2.1.0 Adding Logical Files

Lots of validation of course, and decent clean-up in case of error, but when everything goes right:

- Create temporary directory in the right place in the `$LDB_C_LDB` tree.
- Use parameters to find the input file. Copy/rename to new directory location.
- Use transform table to convert input file to output file.
- Rename temporary directory to correct name.
- Create new record in the `logical_files` table and fill it in (including grabbing the full input and output files).

Whoops. I have a logical contradiction here. The database record must be created to supply the unique file number inserted into the SIB file header, but the sequence described above has the database record being constructed *after* the file transform. I know why I did it. I didn't want dead records in the `logical-files` table if the transform failed. Perhaps the right approach is to do the transform twice. The first time (with a file ID of zero) just ensures that the command will work, the second time does it for real (with a good, i.e. database derived, file ID number).

2.1.0.0 Input To Output File Transforms

The transform engines are assumed to be findable from the command line and be sourced from some CMX package. All transform engines must be written to a standard interface:

```
<transform-engine> <input-file> <output-file> <logical-file-number> <file-type>
```

Mnemonic	Meaning/use
`transform-engine`	A command that transforms a file into a form suitable for uploading to a SIB.
`input-file`	File name always constructable by using the `con_or_name` field of the `logical_files` table (or whatever will go into this field) with the input-file extension identified for this type in the `file_types` table.
`output-file`	File name always constructable by using the `con_or_name` field of the `logical_files` table (or whatever will go into this field) with the output-file extension identified for this type in the `file_types` table.
`logical-file-number`	From the `logical_files` table. Passed in so that the transform engine can attach the unique logical file number.
`file-type`	Similar idea for the file type.

Table 12 Standardized file transform

2.1.1 Adding Physical Files

Mostly a book-keeping chore. Yes it's two-step ("upload" which creates the record in the `physical_files` table, and "commit" which marks the state column in the record as "commited"). The only clever bit is allocating a physical file number (not to be confused with the logical file number!). This can be done by using the arguments to identify the target <sib>/<device>/<directory>, finding (from the database) the highest file number currently allocated in that directory and incrementing by one. Clever bit on top of clever bit. Ensure that the operation is interlocked somehow so that two simultaneous accesses can't cross each other up.

3 Extensions

3.0 Extending the fmx Command Set

3.0.0 Site Update from Database

The actual files are kept in the database. This makes it possible to write a command like:

```
fmx synchronize [logical|physical]
```

This scans all the database records, compares to the exposed directories on the given site, and updates where necessary. Choose to update the logical or physical file base.

3.0.1 Database Queries

The database provides for an extensive query facility.

```
        fmx query <logical|physical|advanced=[query_string]>

                <logical>=Query the logical table
                            [--logical_id=<logical_file_id>]
                            [--type=<file_type_code>]
                            [--package=<cmx_package>]
                            [--version=<package_version>]
                            [--constituent=<cmx_constituent>]
                            [--tag=<cmx_build_tag>]
                            [--site=<cmx_site>]
                            [--user=<username>]
                            [--state=<logical_file_state>]

                <physical>=Query the physical table
                            [--logical_id=<logical_file_id>]
                            [--physical_id=<physical_file_id>]
                            [--device=<device>]
                            [--directory=<directory>]
                            [--file_num=<file_number>]
                            [--file_id=<file_id>]
                            [--site=<cmx_site>]
                            [--user=<username>]
                            [--state=<physical_file_state>]

                <advanced>=Submit user defined query string
                            query_string="SQL query string"
                            ex>fmx query advanced="SELECT sib_key from sib_names"
```

Figure 7 Database query command

3.1 FMX Scripting

This combines techniques from the old `cmx tornado` command and the functioning of LTX. Its value lies in the fact that it makes it possible to write a “logical script” for loading and initializing an embedded system.

The script syntax is:

```
    load cmx:<package>/<ver>/<constituent>
    load fmx:<package>/<ver>/<constituent>
    load sib:<sib_name>/<device>/<dir>/<file>

    call foo_init( 0x0, 0x0, cmx:<package>/<ver>/<constituent> )
    call bar_init( 0x0, 0x0, fmx:<pkg>/<ver>/<constituent> )
    call baz_init( 0x0, 0x0, sib<sib_name>/<device>/<dir>/<file> )
```

Figure 8 A logical script for loading and initialization

This follows the usual design of loading a bunch of modules, then making a bunch of initialization calls. The trick lies in how files are localized. Three different resolution modes are identified:

`cmx:` Resolve the file reference the CMX way. (i.e. the old `cmx tornado` command).

`fmx:` Resolve the file reference in the file database.

`sib:` Resolve the file reference from a (real or virtual) SIB.

3.1.0 Commands

```
fmx tornado <script_file>
          --tag=<cmx_tag>
          --target=<target_ip>
          [--script] - Generate shell script without execution

fmx xyplex <script_file>
          --tag=<cmx_tag>
          --target=<target_ip>
          [--script] - Generate shell script without execution

fmx serial <script_file>
          --tag=<cmx_tag>
          --port=<serial_port>
          [--script] - Generate shell script without execution

fmx sbc <script_file>
```

Figure 9 Script execution commands

This makes is possible to mix and match file sourcing during development. It would be possible to run a RAD750 from both its internal SIB board and from a host file system (a good way to check out files before committing them to the SIB!)