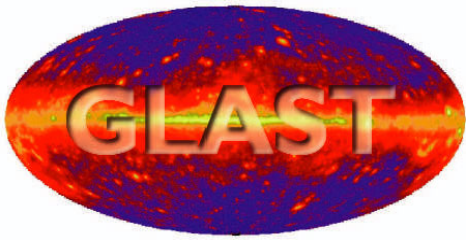

LAT Flight Software

FMX Manual

Type:	User Manual
Version:	V3-2-0
Author:	A.P.Waite
Created:	8 January 2007
Updated:	18 January 2007
Printed:	18 January 2007

Manual for the FMX filebase management system

Contents

Figures

Tables

0 Preface

FMX is a system to track activity on the embedded processor file systems on the various instruments used in the GLAST/LAT project. The LAT instrument itself is the most obvious example, but FMX is also designed to track file activity on a collection of ground based test stands as well.

0.0 History

This manual will describe the "second incarnation" of FMX. The first incarnation was a FSW tool very much designed around FSW needs. During integration activities at SLAC, it became apparent that there was a more widespread need for a file tracking tool (I&T and ISOC), with more extensive capabilities:

- To track file activity on `/ram` (in memory) disks
- To successfully "go on the road", when the LAT moved to NRL.
- To integrate with higher level file based configuration management tools (MOOT/MOOD).

That was a considerable leap to make from the existing tool, so the strategy was to rewrite FMX, basing the design on more extensive criteria. The result was "second incarnation" FMX.

However, it's worth bearing FMX's FSW heritage in mind while reading this manual. This version of FMX still implements many of the FSW specific capabilities of the original. As far as possible, those "added goodies" will be annotated as such, and there will be additional sections at the end of this manual which will only appeal to FSW developers who can take advantage of them.

0.1 Is It a Tool? Is It a Database?

It's both.

This is one place where I wish I'd emulated Joanne. She has precisely the same distinction to make in her configuration file tracking system, and had the foresight to give them different names: MOOT is the *tool*, while MOOD is the *database*.

The term FMX, unfortunately, encompasses both. FMX consists of a random access database, implemented using MySQL, and a command-line driven script written in Perl that manipulates it. Of these, the database is by far the more "fundamental". It is quite possible to write additional tools in a variety of languages to access the database, thus by-passing the FMX script altogether.

To date, that hasn't happened, so both the tool and the database will be described in this manual.

0.2 Structure Of This Manual

This manual is written in several sections, but the sections can be grouped into just four major topics.

- Orientation:
  - A section describing the (simplified) life-cycle of a file in FMX. This is to set up a general context for the material that follows.
- FMX database:
  - User account management.
  - Hardware topology tables.
  - Logical file tables.
  - Physical file tables.
  - Special database topics.
- FMX script:
  - Enumeration commands.
  - User account commands.
  - Logical file commands.
  - Physical file commands.
  - Special commands.
- Additional topics:
  - Secondary boot scripts.
  - File-Of-Files.
  - External software requirements.
  - FMX future.

1 Life Cycle Of A File In FMX

FMX was conceived as a long-lived repository, recording all files and all file transactions to embedded system file storage for the project's collection of instruments. It is designed to provide an accurate picture of a file system's contents (and supply a copy of any given file) for any particular date during the lifetime of the project. To achieve this, the database is divided into "logical" and "physical" sections. The (simplified) life cycle of a file looks like this:

A file is "added" to the FMX database. This can be a code module, a configuration file, or any file that needs to be "uploaded" to an instrument. The source of the file is unspecified. In response, FMX:

- Assigns the file a unique number, called the *logical file key*.
- Captures the original file into the database.
- Constructs an "uploadable" version of the file:
    - Prepends a file header (which contains the logical file key).
    - (Optionally) compresses it.
- Captures the uploadable version of the file into the database.
- Places read-only copies of both files in the regular host file system.
- Keeps records of these files, indexed by the logical file key.

That's a fair amount of work, but note that this process has not placed the file on any embedded file system. Adding a file to FMX simply makes the file *available* for uploading to an embedded file system. The file has been added to the *logical* side of the FMX database. No more.

The next step in the life-cycle is the physical upload. This function is not actually performed by FMX. It is currently performed by a LICOS script:

- LICOS interrogates FMX for the location of the file in the host file system, and for the "file name" the file should be given on the target. This operation is completely read-only and does not alter the state of the FMX database.
- LICOS does the necessary file chunking/CCSDS heading to upload the file to the target instrument.
- On successful completion, LICOS tells FMX that the file has been committed. This causes entries to be made in the physical side of the FMX database.

At this point, the original file now has records (still indexed against its logical file key) in both the logical and physical side of the FMX database. It is rare however, that a file is uploaded to just a single embedded file system. More often the step just described is repeated many times, loading

the same file to many different targets (different CPUs or different devices on those CPUs). Each time a file is successfully committed, LICOS informs the FMX database, so that at the end of the day, a single *logical* file entry can have multiple *physical* file entries.

But many files have finite lifetimes. Many will need to be deleted from the embedded file systems to make way for other files. This process is simply the reverse of the *commit* procedure:

- LICOS interrogates FMX for "file name" the file has on the target. This operation is completely read-only and does not alter the state of the FMX database.
- LICOS constructs and sends the command instructing the embedded system to delete the file.
- On successful completion, LICOS tells FMX that the file has been deleted. This causes entries to be made in the physical side of the FMX database.

The only surprise in that sequence may be that deleting a file causes entries to be *made* in the physical side of FMX. This is a consequence of the desire to be able to "roll back the clock" and ask FMX for the state of the file system at a given date in the past. When a file is "deleted" from FMX, FMX adds a time-stamped entry noting the deletion of the file in a transaction log.

One last note on a file's life-cycle. Nowhere in this description will you find mention of a file, logical or physical, original or prepared-for-flight, being deleted. Once a file is added to FMX, it is never deleted. In this way, not only can FMX work out what files were uploaded where, for any date in the past, it can also provide a copy of those files.

A more graphical representation of the preceding material looks like:

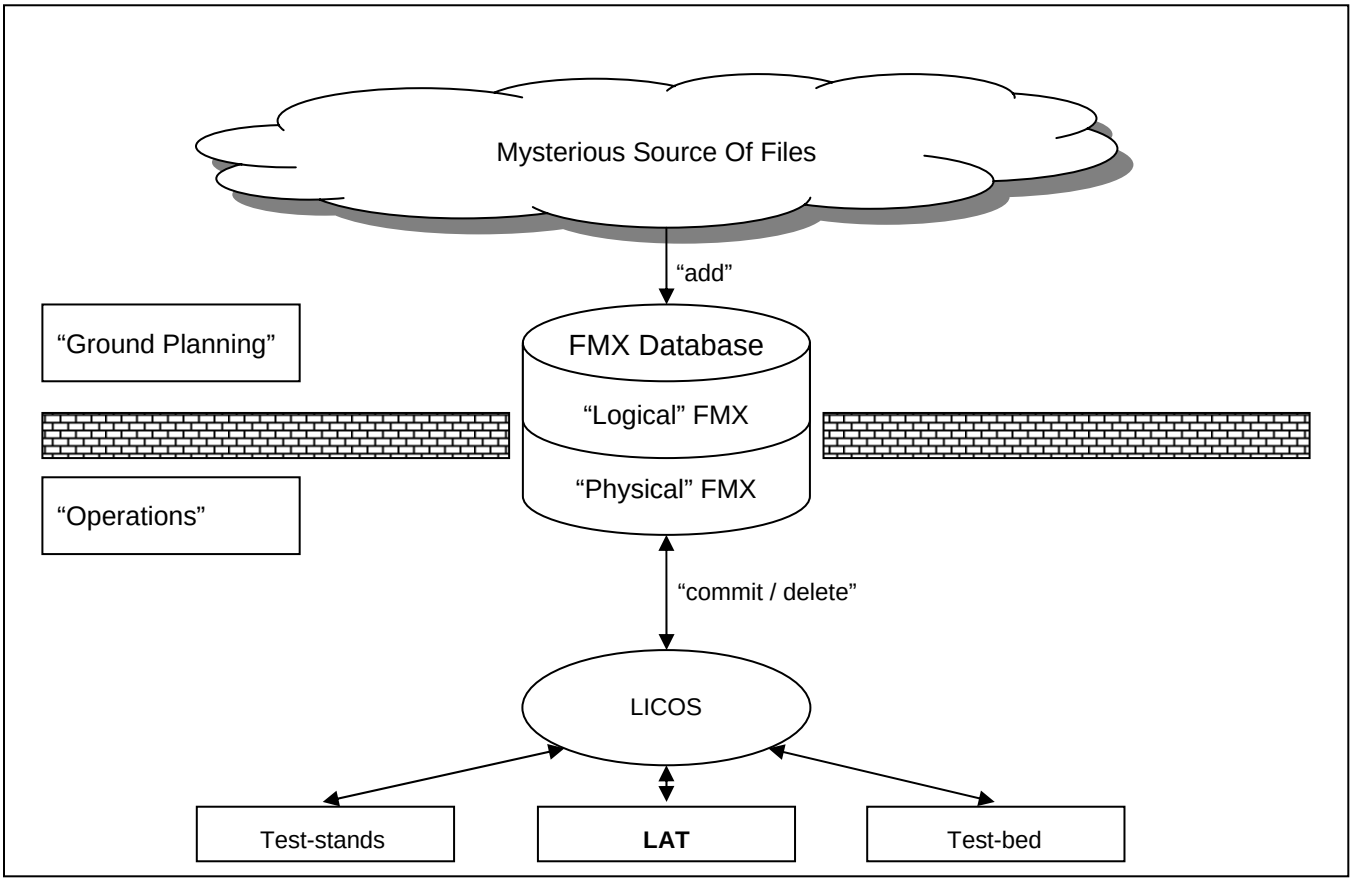

Figure 1 Life-cycle of a file in FMX

I've also taken the liberty of annotating "policy" in terms of who is responsible for what. This is not meant to dictate policy, merely to demonstrate that FMX lends itself to a separation of functions. People on the "ground side" should never have to deal with minutiae of how files get uploaded, whereas the operations people should not have to worry about a file's provenance.

Taking an even bigger liberty, the following figure elaborates on the previous one by detailing more of the flow details of the current implementation. This figure should be used with extreme caution because the details are not guaranteed to stay the same. Nevertheless it provides a useful context for later discussions.

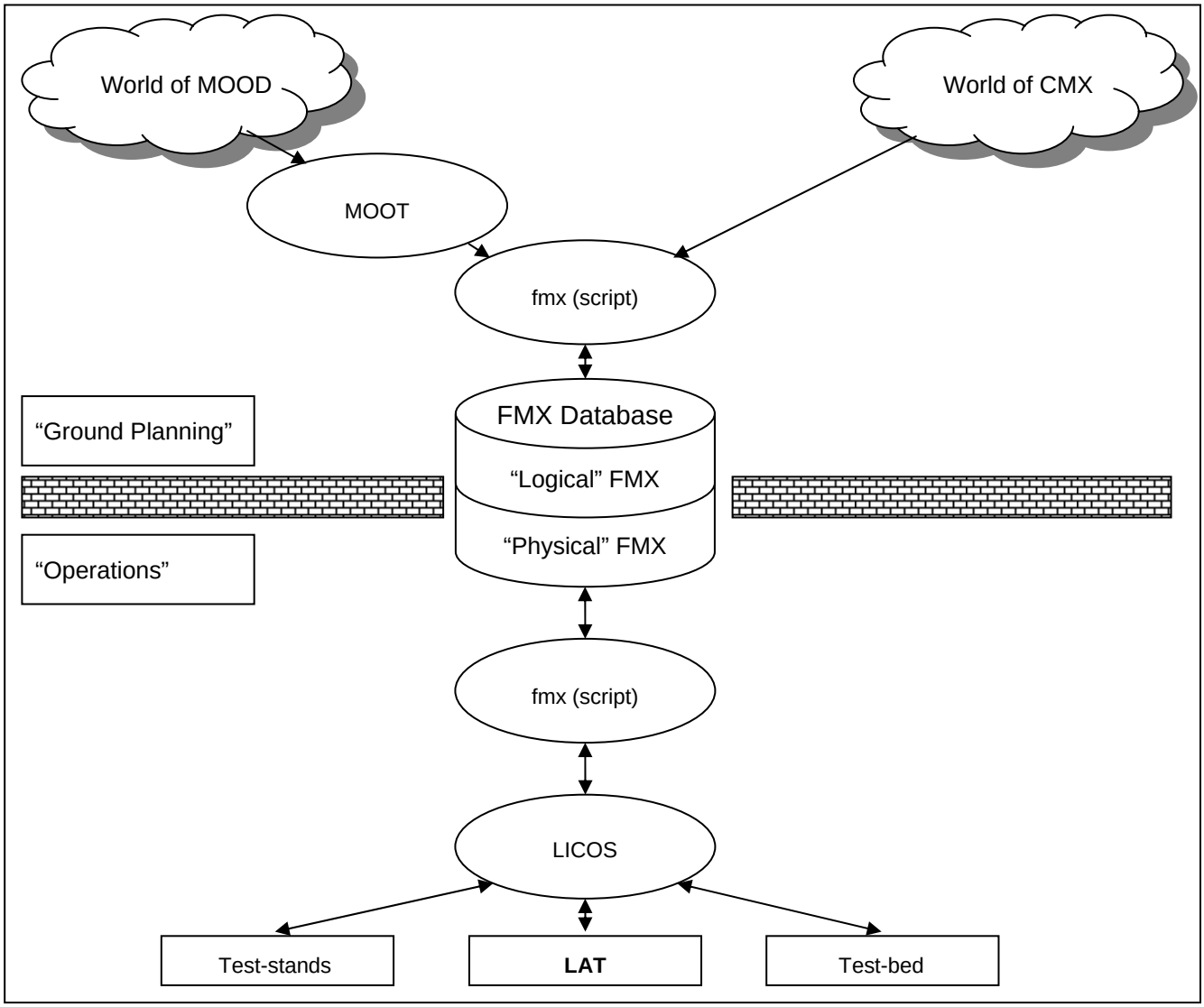

Figure 2 Life-cycle of file showing more details of the current implementation

2 User Account Management

One of the first places where this version of FMX was used was during LAT instrument integration at SLAC. This immediately pushed FMX into the world of Virtual Private Networks (VPNs) and "secure Linux". In keeping with that environment, FMX tries to provide a high level of security, a feature that finds expression through it's handling of user accounts.

Note that FMX user accounts have nothing to do with Unix (or Windows or …) login accounts. MySQL databases (like most SQL databases) have a completely independent security infrastructure. This in turn allows for very fine-grained, database specific, security rules. For instance, it is quite possible to set up a user account so that a user can *connect* to the database, but then fail to give that user access to any tables at all! MySQL provides some very specialized access controls, but for the average user account, there is little need for control over more than the common "select", "insert", "update", and "delete" privileges. Those privileges can be tailored (by user) down to individual tables, or even individual columns in a table.

So much for theory. What does FMX actually do?

2.0 Account Types

FMX classifies user accounts into two types, role based accounts and user accounts.

2.0.0 Role Based Accounts

Role based accounts are used to fulfill very specialized functions. FMX has defined four such accounts:

Account Name	**Account Purpose**
`magician`	This is the FMX database's equivalent of the Unix "super-user". This account has full privileges to do anything it pleases (and the account password is a well guarded secret … I hope).
`barbican`	The exact opposite of the `magician` account. This is the *least* privileged account in FMX. This account couldn't tie its own shoelaces without help. Its sole purpose is to know just enough to register a new user account request. Think about it. By definition a new user doesn't have an account, so it needs an existing account to register the account request. Such a role based account must be accessible to everyone, so it must have very few privileges to avoid exposing the database to all and sundry!

`replication`	This is an *extremely* specialized (and quite powerful) account. It only has one function and that is to manage replication between databases. To avoid someone trying to hijack the replication account, it also has well guarded password.
`mortician`	Used to perform nightly backups of the database. Another well guarded password to avoid account hijacking.

Table 1 Role based account in FMX

2.0.1 User Accounts

All other accounts in FMX are individual user accounts. Each such account is granted a combination of privileges from the following list, depending on the access required by the user:

Privilege Group	**Purpose**
`optician`	`optician` privilege gives the user the ability to read any FMX database table (in database parlance, that's "select" privilege).
`logician`	`logician` privilege gives the user the ability to add or edit records in the FMX tables that record logical files (in database parlance, that's "insert" and "update" privileges … notably absent is the "delete" privilege!).
`physician`	`physician` privilege gives the user the ability to add, edit or delete records in the FMX tables that record physical files (in database parlance, that's "insert", "update" and "delete" privileges). Actually that's a slight overstatement. `physicians` can delete records in table `physical`, but they cannot delete records in table `physlog`.
`technician`	`technician` privilege gives the user the ability to add, edit or delete records in the FMX tables that record instrument hardware topologies.

Table 2 User account privilege groups

2.1 User Account Philosophy

This is obviously quite a lot of mechanism to handle user accounts. Why so elaborate?

If you're serious about account security, individual user accounts (with users that jealously guard their account password) are the way to go. Consider an alternative approach. Suppose I set up another role based account called `operator` and shared it between all the operations crew. The first thing lost is the ability to annotate who changed what in the database. It would just read "one of the operators did it". Worse still the list of "who is an operator" changes over time as staff come and go. To attach a database change to even a group of people, it would be necessary to record a timestamp with every such entry. It gets worse. Assuming that operations want to invalidate an operator's access after he or she leaves, the only alternative is to change the password of the `operator` account and then pass that information to all the remaining operators. Dealing with that "simple" role based account suddenly starts getting very messy!

Which is not to say that a database cannot be run this way. It all depends on the level of control and security the project demands. As stated in the first paragraph, the setup of the mobile rack certainly suggested a desire for a high level of security, so FMX's account management was designed accordingly.

One last note on FMX's user privilege structure. The privilege level `optician` probably looks a bit anomalous. What would be the use of a user account that *doesn't* have optician privileges? The thinking behind making this a specific privilege level goes back to the idea that staff will occasionally leave and when they do, their user accounts should be invalidated. One way to do

that in MySQL would be to simply delete the user account. That's a little awkward if FMX is keeping records of activity against individual user accounts (which it doesn't do …yet). It would not be possible to dereference entries listed against an account that has been destroyed. By making an `optician` level privilege, there's another solution. The account can be rendered useless by withdrawing `optician` privileges, but it would not have to be deleted. And you thought I was joking in the first paragraph where I postulated a user account with absolutely no access privileges!

2.2 User Account Tables

In keeping with the structure of this document, the FMX tables used to support user account management will be described here. Unfortunately this is one place where the document structure doesn't work very well. To more fully understand these tables, it's best to read the following table descriptions in conjunction with the user account management commands provided by the FMX script.

2.2.0 Table **access**

```
mysql> describe access;
+--------+-----------------------------------------+------+-----+----------+-------+
| Field  | Type                                    | Null | Key | Default  | Extra |
+--------+-----------------------------------------+------+-----+----------+-------+
| role   | enum('logician','physician','technician') |      | PRI | logician |       |
| name   | varchar(16)                             |      | PRI |          |       |
| access | set('select','insert','update','delete')  |      |     |          |       |
+--------+-----------------------------------------+------+-----+----------+-------+
```

Figure 3 Description of the user account table `access`

```
mysql> select * from access;
+------------+------------------+-----------------------------+
| role       | name             | access                      |
+------------+------------------+-----------------------------+
| logician   | fileseq          | select,insert,update        |
| logician   | logical          | select,insert,update        |
| logician   | module           | select,insert,update        |
| logician   | numbers          | select,update               |
| logician   | resolved         | select,insert,update        |
| logician   | simple           | select,insert,update        |
| logician   | unresolved       | select,insert,update        |
| logician   | visible          | select,insert,update        |
| physician  | physical         | select,insert,update,delete |
| physician  | physlog          | select,insert               |
| technician | cpu              | select,insert,update,delete |
| technician | crate2board      | select,insert,update,delete |
| technician | instrument       | select,insert,update,delete |
| technician | instrument2crate | select,insert,update,delete |
| technician | object           | select,insert,update,delete |
| technician | sib              | select,insert,update,delete |
+------------+------------------+-----------------------------+
```

Figure 4 Dump of the user account table `access`

This table is to compensate for a curious omission in the MySQL product. It doesn't support the idea of abstract "access control lists". Given this feature, I could define (once) the privileges

associated being a logician, physician or technician, then give those privileges as a named block to user accounts. Lacking this feature, the `access` table provides a poor man's version. When user accounts are created (or modified), the FMX script executes a list of SQL "grant" (or "revoke") commands as specified by this table.

2.2.1 Table **`request`**

```
mysql> describe request;
+------------+------------------+------+-----+---------------------+----------------+
| Field      | Type             | Null | Key | Default             | Extra          |
+------------+------------------+------+-----+---------------------+----------------+
| request    | int(10) unsigned |      | PRI | NULL                | auto_increment |
| received   | datetime         |      |     | 0000-00-00 00:00:00 |                |
| id         | varchar(16)      |      | UNI |                     |                |
| name       | varchar(64)      |      |     |                     |                |
| mail       | varchar(64)      |      |     |                     |                |
| pass       | varchar(41)      |      |     |                     |                |
| new        | enum('Y','N')    |      |     | Y                   |                |
| logician   | enum('','Y','N') |      |     |                     |                |
| physician  | enum('','Y','N') |      |     |                     |                |
| technician | enum('','Y','N') |      |     |                     |                |
+------------+------------------+------+-----+---------------------+----------------+
```

Figure 5 Description of the user account table `request`

The `request` table is used to register requests to create or modify user accounts. Such requests are registered by the role based `barbican` account. This is the only table that the `barbican` account can write to (it's the only table the `barbican` account can even *see*, with the exception of "select" access to the `name` column in FMX's `user` table, and that's only to avoid duplicating account names).

One thing to note is that the password column (`pass`) is already encoded before it is stored in this table. Passwords are never passed between the database client and server "in clear", and the fact that someone could use the `barbican` account to access this table would not yield access to another applicant's password.

My remaining worry about this table is that someone could use the `barbican` account to discover an applicant's e-mail address (stored in column `mail`). I would hate for this table to turn into a source of spam.

2.2.2 Table **`user`**

```
mysql> describe user;
+------------+---------------+------+-----+---------+-------+
| Field      | Type          | Null | Key | Default | Extra |
+------------+---------------+------+-----+---------+-------+
| id         | varchar(16)   |      | PRI |         |       |
| name       | varchar(64)   |      |     |         |       |
| mail       | varchar(64)   |      |     |         |       |
| logician   | enum('Y','N') |      |     | Y       |       |
| physician  | enum('Y','N') |      |     | Y       |       |
| technician | enum('Y','N') |      |     | Y       |       |
| optician   | enum('Y','N') |      |     | Y       |       |
+------------+---------------+------+-----+---------+-------+
```

Figure 6 Description of the user account table `user`

(Actual contents of this table not presented for obvious reasons).

Clearly I was suffering from an imagination deficit the day I defined this table. The table name `user` is just plain confusing. All MySQL databases already contain a table called `user` (in the "system" database, called, surprise, `mysql`). Oh well!

FMX's `user` table can be thought of as extending the information stored in MySQL's `user` table. FMX's `user` table stores a user's e-mail address, the user's set of privileges and the user's "trivial" name (that doesn't mean the user is trivial … besides, it's the user that gets to define this column!).

3 Hardware Topology Tables

One of the difficulties encountered by FSW in the early days of using embedded file systems was that the file systems never sat still! With very few flight equivalent embedded system crates and boards, it was common to shotgun systems by taking a board out of one crate and putting it in another. This made mincemeat of any system that recorded file uploads against an *instrument*, so an early design decision (implemented as far back as the first incarnation of FMX), was to record file uploads against *boards*. In this way, it didn't matter if a board moved between crates, the records of what files were on it moved with it.

That solved the problem of wandering boards, but it moved the problem elsewhere. When file operations are performed, people want to say "upload file <foo> to the test-bed's primary SIU, lower TFFS EEPROM bank, directory <bar>. People do *not* want to say "upload file <foo> to the board identified by the property control number `GLAT0927`". This is what drove the implementation of hardware topology descriptions.

3.0 Hardware Topology Enumerations

Hardware topology descriptions use four enumerations tables:

3.0.0 Enumeration Table **`tag`**

```
mysql> describe tag;
+---------+--------------+------+-----+---------+-------+
| Field   | Type         | Null | Key | Default | Extra |
+---------+--------------+------+-----+---------+-------+
| tag     | tinyint(4)   |      | PRI | 0       |       |
| name    | varchar(16)  |      | UNI |         |       |
| vxworks | enum('Y','N') |      |     | Y       |       |
+---------+--------------+------+-----+---------+-------+
```

Figure 7 Description of enumeration table `tag`

```
mysql> select * from tag;
+-----+-----------+---------+
| tag | name      | vxworks |
+-----+-----------+---------+
|   0 | mv2304    | Y       |
|   1 | mcp750    | Y       |
|   2 | rad750    | Y       |
|   3 | linux-gcc | N       |
|   4 | sun-gcc   | N       |
+-----+-----------+---------+
```

Figure 8 Dump of enumeration table `tag`

This table is actually more general purpose than it placement in the hardware topology section might imply. The only use hardware topology makes of the `tag` table is to identify:

- CPU hardware architectures (i.e. is the target CPU an mv2304 or a rad750).
- Crate backplane types (i.e. does the target crate have a VME or a cCPI crate).

Clearly, hardware topology does not take advantage of the `vxworks` column of the table.

3.0.1 Enumeration Table **node**

```
mysql> describe node;
+---------+-------------+------+-----+---------+-------+
| Field   | Type        | Null | Key | Default | Extra |
+---------+-------------+------+-----+---------+-------+
| node    | tinyint(4)  |      | PRI | 0       |       |
| name    | varchar(16) |      | UNI |         |       |
| address | tinyint(4)  |      | UNI | 0       |       |
+---------+-------------+------+-----+---------+-------+
```

Figure 9 Description of enumeration table `node`

```
mysql> select * from node;
+------+------+---------+
| node | name | address |
+------+------+---------+
|    0 | siu0 |      34 |
|    1 | siu1 |      35 |
|    2 | epu0 |      36 |
|    3 | epu1 |      37 |
|    4 | epu2 |      38 |
|    5 | vsc  |       0 |
+------+------+---------+
```

Figure 10 Dump of enumeration table `node`

Once again, this table sees wider service than just hardware descriptions. The column `address` for instance has only one very narrow function (generating a CPU's "compound serial number"), and is not used in hardware topology descriptions.

3.0.2 Enumeration Table **instrument**

```
mysql> describe instrument;
+------------+---------------------------------+------+-----+---------+-------+
| Field      | Type                            | Null | Key | Default | Extra |
+------------+---------------------------------+------+-----+---------+-------+
| instrument | tinyint(4)                      |      | PRI | 0       |       |
| name       | varchar(16)                     |      | UNI |         |       |
| towers     | enum('real','simulated','none') |      |     | real    |       |
| host       | int(10) unsigned                |      |     | 0       |       |
| location   | varchar(64)                     |      |     |         |       |
| comment    | text                            |      |     |         |       |
+------------+---------------------------------+------+-----+---------+-------+
```

Figure 11 Description of the fields in enumeration table `instrument`

```
mysql> select * from instrument;
+------------+---------+-----------+------+--------------+--------------------------+
| instrument | name    | towers    | host | location     | comment                  |
+------------+---------+-----------+------+--------------+--------------------------+
|          1 | mordor  | real      |    2 | NRL          | The LAT instrument       |
|          2 | gondor  | simulated |    1 | SLAC 84 B101 | Test-bed                 |
|          3 | rohan   | none      |    1 | NRL          |                          |
|          4 | arnor   | none      |    1 | SLAC 84 B101 | Uber-test-stand          |
|          5 | orthanc | real      |    1 | SLAC 84 B101 | Multi-crate (with tower) |
|          6 | shire   | none      |    1 | SLAC 84 B101 | SIU flight spare         |
+------------+---------+-----------+------+--------------+--------------------------+
```

Figure 12 Dump of enumeration table `instrument`

The fields in this table require a little more explanation:

Field	Description
`instrument`	The instrument index.
`name`	The instrument name.
`towers`	The type of towers attached to this instrument's front-end. Can be one of `real`, `simulated` or `none`. This is an obscure part of the hardware description and a full explanation for this field will be deferred to section 14. For now, suffice it to say that having this information can avoid the pratfall of loading the wrong set of timing constants for the available tower hardware.
`host`	The identity of the controlling database. This is a complicated subject, involving FMX's ability to run spread over replicating databases on different hosts. Discussion of this topic is deferred to section 6.1.
`location`	A free descriptive field.
`comment`	A free descriptive field.

Table 3 Extended description of the fields in enumeration table `instrument`

At this point, I can just hear everyone waiting for me to explain away those instrument names!

In fact FSW has a long tradition of using a "Middle Earth" theme for naming objects in the test laboratory. It probably started as a pun on the file format of code modules produced by the VxWorks cross-compilation suite. These are written in "Extensible Linker Format", or "ELF" files for short. Various embedded system CPUs started being assigned IP names like `lat-elf1`, `lat-elf2`, … . FSW also needed Sun workstations to talk to those embedded systems over serial lines. Inevitably these "elf bosses" were given the names of senior elves: `lat-elrond`,

lat-arwen and lat-cirdan. The whole thing just snow-balled from there. When we figured out that we needed a “forest” of computers to drive the FES, they were christened lat-ent1, lat-ent2, …, and were put under the management of the boss ent, lat-fangorn. (actually, that was a mistake, Fangorn is the name of the forest, not the name of the boss ent (which is Treebeard I believe)). When linux boxes started invading, they became lat-hobbit1, lat-hobbit2, … . Showing a little bias no doubt, any Windows based PCs were given names in the series lat-orc1, lat-orc2, … (funny; we don’t have any of those any more).

So when it came time to find names for our various instruments, I continued with the theme. Prior to that time, the test-stands had names like North-East (meaning the test-stand in the North-East corner of the room). Not being equipped with a compass in my head, these names always confused me, but the fact that they were geographical in nature probably led me to think about geographical names from Middle Earth. Hence instrument names based on the large geographical areas described in Tolkien’s “Lord of the Rings”.

And now you know the rest of the story.

3.0.3 Enumeration Table **object**

FMX recognizes only three types of objects: CPUs, SIBs and crates. CPUs and SIBs are included because they are the only boards capable of supporting a file system. The reason for including crates will become apparent when we start building up descriptions of full instruments.

The values for the primary key for this table, object, are taken directly from the property control database and are unique.

You might notice a definite trend here. The column providing the primary key for a table is almost always named after the table itself. The database doesn’t demand this. I just find it convenient!

```
mysql> describe object;
+--------+--------------------------+------+-----+---------+-------+
| Field  | Type                     | Null | Key | Default | Extra |
+--------+--------------------------+------+-----+---------+-------+
| object | varchar(8)               |      | PRI |         |       |
| name   | enum('crate','cpu','sib') |      |     | crate   |       |
+--------+--------------------------+------+-----+---------+-------+
```

Figure 13 Description of the object table

```
mysql> select * from object;
+----------+-------+
| object   | name  |
+----------+-------+
| GLAT0094 | crate |
| GLAT0838 | cpu   |
| GLAT0840 | sib   |
| GLAT0845 | cpu   |
.          .       .
.          .       .
| GLAT9004 | crate |
| GLAT9005 | crate |
| GLAT9006 | crate |
+----------+-------+
```

Figure 14 Partial dump of enumeration table object

A full dump of the object table is rather long and not very informative, so only a subset has been shown here:

3.1 Hardware Objects, Extended Attributes

Each of the object types recognized by FMX has extended attributes provided by other tables:

Object Type	Extended Attributes Provided By Table
cpu	cpu
sib	sib
crate	crate2board

Table 4 Tables providing extended object attributes

3.1.0 Extended Attributes Of Object Type **cpu**

```
mysql> describe cpu;
+-------+-------------+------+-----+---------+-------+
| Field | Type        | Null | Key | Default | Extra |
+-------+-------------+------+-----+---------+-------+
| cpu   | varchar(8)  |      | PRI |         |       |
| tag   | tinyint(4)  |      |     | 0       |       |
| sn    | tinyint(4)  |      |     | 0       |       |
| ip    | varchar(32) |      |     |         |       |
+-------+-------------+------+-----+---------+-------+
```

Figure 15 Description of the extended attributes table cpu

```
mysql> select * from cpu;
+----------+-----+----+-----------+
| cpu      | tag | sn | ip        |
+----------+-----+----+-----------+
| GLAT0838 |   2 | 32 |           |
| GLAT0845 |   2 | 33 |           |
| GLAT1076 |   2 | 34 |           |
| GLAT1077 |   2 | 35 |           |
| GLAT1131 |   2 | 36 |           |
| GLAT1660 |   2 | 37 |           |
| GLAT2515 |   2 | 12 |           |
| GLAT2516 |   2 | 14 |           |
| GLAT2517 |   2 | 15 |           |
| GLAT2518 |   2 | 17 |           |
| GLAT2519 |   2 | 18 |           |
| GLAT2521 |   2 | 13 |           |
| GLAT8001 |   0 | 64 | lat-elf10 |
| GLAT8002 |   0 | 65 | lat-elf11 |
| GLAT8003 |   0 | 66 | lat-elf12 |
| GLAT8004 |   0 | 67 | lat-elf9  |
| GLAT8005 |   0 | 68 | lat-elf21 |
| GLAT8006 |   0 | 69 | lat-elf1  |
| GLAT8007 |   0 | 70 | lat-elf5  |
+----------+-----+----+-----------+
```

Figure 16 Dump of extended attributes table cpu

Field	Description
cpu	CPU's object number as it appears in the object table.
tag	An index from the tag enumeration table. In this context, the tag index is used to indicate the architecture of the CPU (e.g. one of mv2304, mcp750 or rad750).
sn	Serial number. Used as part of a CPU's "compound serial number". Simply BAE's manufacturing number for rad750 boards. Made up for COTS boards.
ip	IP address. Only relevant for CPUs that have access to Ethernet and have the capability to store the VxWorks "boot string" in on-board non-volatile RAM (i.e. this excludes rad750). An unusual field in that it can be NULL (the board does not have access to Ethernet), but if it's not NULL, it must be unique.

Table 5 Extended description of the extended attributes table cpu

3.1.1 Extended Attributes Of Object Type **sib**

```
mysql> describe sib;
+-------+-------------+------+-----+---------+-------+
| Field | Type        | Null | Key | Default | Extra |
+-------+-------------+------+-----+---------+-------+
| sib   | varchar(8)  |      | PRI |         |       |
| ip    | varchar(32) |      |     |         |       |
+-------+-------------+------+-----+---------+-------+
```

Figure 17 Description of the extended attributes table sib

```
mysql> select * from sib;
+----------+-----------+
| sib      | ip        |
+----------+-----------+
| GLAT0840 |           |
| GLAT0847 |           |
| GLAT0879 |           |
| GLAT0925 | lat-elf2  |
| GLAT0967 | lat-elf24 |
| GLAT0972 | lat-elf4  |
| GLAT1103 | lat-elf23 |
| GLAT1705 | lat-elf3  |
| GLAT2205 |           |
| GLAT2206 |           |
| GLAT2207 |           |
| GLAT2208 |           |
| GLAT2209 |           |
| GLAT2210 |           |
| GLAT2212 |           |
| GLAT2319 | lat-elf25 |
| GLAT2510 |           |
+----------+-----------+
```

Figure 18 Dump of the extended attributes table sib

Field	Description
sib	SIB's object number as it appears in the object table.

ip	IP address. Only relevant for systems that have access to Ethernet and have the capability to store the VxWorks “boot string” in off-board non-volatile RAM (i.e. a rad750 working in cooperation with a SIB). An unusual field in that it can be NULL (the board does not have access to Ethernet), but if it’s not NULL, it must be unique.

Table 6 Extended description of the extended attributes table sib

Quick aside on the ip column in tables cpu and sib.

When working with embedded systems using the VxWorks file and debugging toolset across an Ethernet connection, the embedded CPU needs to store a small amount of information about its network connection in some non-volatile location. This is known as the boot string. On our COTS boards, the CPU board itself has a small amount of NVRAM where the CPU can store this string. Unfortunately the only non-volatile location available on a rad750 is the SUROM. This did not seem like a very safe place to store the boot string! Instead, a rad750 operating in cooperation with a SIB is allowed to store the boot string in the EEPROM of the SIB. This works very nicely until a SIB gets divorced from its rad750, at which point the IP information travels with the SIB, not with the CPU!

If you want to get even more obtuse, consider what happens when an mcp750 CPU is used in conjunction with a SIB board (yes you can do that). In this case the CPU always uses the boot string from its private NVRAM. Any boot string in the SIB is ignored.

In fact, the only use FMX makes of IP information is to identify a target CPU. This is one place where FMX’s FSW heritage is evident. In the early days of FMX, we often identified CPUs as “the CPU on the other end of this IP address”. Many commands actually took the IP name or address as a parameter. The IP style of identifying a target is still supported in FMX, but it is now more common to use the target CPU’s instrument name and node name.

At this point you might have expected to see a description of the extended attributes of a crate. The table that provides those attributes, crate2board, is in fact dual purposed, so its description is deferred to the next section.

3.2 Hardware Mapping Tables

3.2.0 Hardware Mapping Table **crate2board**

```
mysql> describe crate2board;
+-------+--------------------------+------+-----+---------+-------+
| Field | Type                     | Null | Key | Default | Extra |
+-------+--------------------------+------+-----+---------+-------+
| crate | varchar(8)               |      | PRI |         |       |
| tag   | tinyint(4)               |      |     | 0       |       |
| ctdb  | enum('sib','pmc','none') |      |     | sib     |       |
| cpu   | varchar(8)               | YES  | MUL | NULL    |       |
| sib   | varchar(8)               | YES  | MUL | NULL    |       |
+-------+--------------------------+------+-----+---------+-------+
```

Figure 19 Description of the hardware mapping table crate2board

```
mysql> select * from crate2board;
+----------+-----+------+----------+----------+
| crate    | tag | ctdb | cpu      | sib      |
+----------+-----+------+----------+----------+
| GLAT0094 |   0 | none | GLAT8002 | NULL     |
| GLAT0853 |   0 | pmc  | GLAT8007 | NULL     |
| GLAT0892 |   0 | pmc  | GLAT8004 | NULL     |
| GLAT1101 |   0 | none | GLAT8003 | NULL     |
| GLAT1268 |   0 | pmc  | GLAT8005 | NULL     |
| GLAT1278 |   0 | pmc  | GLAT8006 | NULL     |
| GLAT1333 |   0 | pmc  | GLAT8001 | NULL     |
| GLAT2513 |   2 | sib  | GLAT2516 | GLAT2207 |
| GLAT2522 |   2 | none | GLAT2518 | GLAT2206 |
| GLAT2523 |   2 | none | GLAT2519 | GLAT2209 |
| GLAT2524 |   2 | none | GLAT2521 | GLAT2210 |
| GLAT2525 |   2 | sib  | GLAT2515 | GLAT2212 |
| GLAT2527 |   2 | sib  | GLAT2517 | GLAT2208 |
| GLAT9001 |   2 | sib  | GLAT1077 | GLAT1103 |
| GLAT9002 |   2 | none | GLAT0845 | GLAT2319 |
| GLAT9003 |   2 | none | GLAT1660 | GLAT0967 |
| GLAT9004 |   2 | sib  | GLAT1076 | GLAT0925 |
| GLAT9005 |   2 | none | GLAT1131 | GLAT1705 |
| GLAT9006 |   2 | none | GLAT0838 | GLAT0972 |
+----------+-----+------+----------+----------+
```

Figure 20 Dump of the hardware mapping table `crate2board`

Field	Description
`crate`	Crate's object number as it appears in the `object` table.
`tag`	An index from the `tag` enumeration table. In this context, the tag index is used to indicate the architecture of the crate backplane (e.g. one of `VME` or `cCPI`).
`ctdb`	The support the *crate* offers for 1553 communications. 1553 communications can be supported by either a mezzanine card in a COTS CPU, or by the 1553 capabilities of a SIB. This is an obscure part of the hardware description. Suffice it to say that having this information can avoid the pratfall of loading the wrong software driver for the available 1553 hardware. This will be covered in more detail in the section 14.
`cpu`	CPU's object number as it appears in the `object` table.
`sib`	SIB's object number as it appears in the `object` table.

Table 7 Extended description of the hardware mapping table `crate2board`

Apart from the descriptive elements (`tag` and `ctdb`) the purpose of this table should be self-evident. It simply associates CPU and SIB boards with a crate. Note that this structure forbids having more than one CPU or SIB board associated with a crate (though it is perfectly legal, and might even make sense, to have *no* boards associated).

3.2.1 Hardware Mapping Table **instrument2crate**

```
mysql> describe instrument2crate;
+------------+------------+------+-----+---------+-------+
| Field      | Type       | Null | Key | Default | Extra |
+------------+------------+------+-----+---------+-------+
| instrument | tinyint(4) |      | MUL | 0       |       |
| node       | tinyint(4) |      |     | 0       |       |
| crate      | varchar(8) |      | PRI |         |       |
+------------+------------+------+-----+---------+-------+
```

Figure 21 Description of the hardware mapping table `instrument2crate`

```
mysql> select * from instrument2crate;
+------------+------+----------+
| instrument | node | crate    |
+------------+------+----------+
|          1 |    0 | GLAT2527 |
|          1 |    1 | GLAT2513 |
|          1 |    2 | GLAT2522 |
|          1 |    3 | GLAT2523 |
|          1 |    4 | GLAT2524 |
|          2 |    1 | GLAT9001 |
|          2 |    2 | GLAT9002 |
|          2 |    3 | GLAT9003 |
|          2 |    5 | GLAT1268 |
|          4 |    0 | GLAT9004 |
|          4 |    2 | GLAT9005 |
|          4 |    3 | GLAT9006 |
|          4 |    5 | GLAT1278 |
|          5 |    0 | GLAT1333 |
|          5 |    2 | GLAT0094 |
|          5 |    3 | GLAT1101 |
|          5 |    5 | GLAT0892 |
|          6 |    0 | GLAT2525 |
|          6 |    5 | GLAT0853 |
+------------+------+----------+
```

Figure 22 Dump of the hardware mapping table `instrument2crate`

Field	Description
`instrument`	An index from the `instrument` enumeration table.
`node`	An index from the `node` enumeration table.
`crate`	Crate's object number as it appears in the `object` table.

Table 8 Extended description of the hardware mapping table `instrument2crate`

Once again, the purpose of this table should be self-evident. It simply maps crates to computer nodes on given instruments.

3.3 Hardware Topology Table Integrity

In the preceding descriptions, I have not spent any time annotating some of the more obscure elements in the table definitions, elements such as "primary keys", "unique field(s)" and so on. All such definitions are there to support table integrity. If you go back now and examine those

constraints, you will find that they disallow (by database rule) a large variety of instrument misconfigurations, including:

- The same CPU being mapped to two different crates.
- Multiple CPUs being mapped to the same crate.
- An instrument having more than one node called “EPU1”.

This type of integrity enforcement is a theme throughout FMX. If I could find a way for the database to protect its integrity using database rules, I did so. Not only do such rules build user confidence in the integrity of the database, but by placing the rules at the lowest possible level, the database protects itself against whatever tool is used to access it and does not rely on the accessing tool not to do anything stupid.

Having said that, some integrity rules are harder to enforce than others! There are one or two places in the FMX database where the database needs help from the tool that’s manipulating it to ensure integrity. I hope to annotate any such cases as I go along.

3.4 Hardware Topology Display

The hardware topology tables see a lot of action in the FMX script, translating instrument/node pairs to boards and vice versa. For the moment, the only display based on these tables is in response to the command `fmx describe instrument <instrument name>` (and no, I haven’t described that command yet, but I think the syntax makes the purpose of the command pretty clear!). Here’s an example:

```
flora04:apw> fmx describe instrument arnor

arnor:
    Comment ..... Uber-test-stand
    Location .... SLAC 84 B101
    FMX host..... glastlnx06.slac.stanford.edu
    Towers ...... none

    siu0:
        Crate identifier ............... GLAT9004
        Crate type ..................... cPCI
        1553 provided by ............... sib
        Hardware address on EBM ........ 0x22

        cpu:
            CPU object number .......... GLAT1076
            CPU type ................... rad750
            IP address ................. <undefined>
            Compound serial number  .... 0x22220004

        sib:
            SIB object number .......... GLAT0925
            IP address ................. lat-elf2

    epu0:
        Crate identifier ............... GLAT9005
        Crate type ..................... cPCI
        1553 provided by ............... none
        Hardware address on EBM ........ 0x24

        cpu:
            CPU object number .......... GLAT1131
            CPU type ................... rad750
            IP address ................. <undefined>
            Compound serial number  .... 0x24240104

        sib:
            SIB object number .......... GLAT1705
            IP address ................. lat-elf3

    epu1:
        Crate identifier ............... GLAT9006
        Crate type ..................... cPCI
        1553 provided by ............... none
        Hardware address on EBM ........ 0x25

        cpu:
            CPU object number .......... GLAT0838
            CPU type ................... rad750
            IP address ................. <undefined>
            Compound serial number  .... 0x20250204

        sib:
            SIB object number .......... GLAT0972
            IP address ................. lat-elf4
```

```
        vsc:
            Crate identifier ............... GLAT1278
            Crate type ..................... VME
            1553 provided by ............... pmc

            cpu:
                CPU object number .......... GLAT8006
                CPU type ................... mv2304
                IP address ................. lat-elf1
```

Figure 23 Hardware topology dump for instrument `arnor`

4 Logical File Tables

Ok you've had it pretty easy up to now. User account a management is always a "my eyes glaze over" topic, and the hardware topology tables don't really contain any sophisticated logic. The free ride is over, it's time to buckle down to hard-core FMX.

The first thing to establish is the concept of file *types* and how they map to (currently) just three file *groups,* where each file *group* receives different processing.

4.0 Logical Table **`filetype`**

```
mysql> describe filetype;
+-----------+-----------------------------------+------+-----+---------+-------+
| Field     | Type                              | Null | Key | Default | Extra |
+-----------+-----------------------------------+------+-----+---------+-------+
| filetype  | smallint(6)                       |      | PRI | 0       |       |
| name      | varchar(16)                       |      | UNI |         |       |
| directory | varchar(16)                       | YES  | MUL | NULL    |       |
| member    | enum('simple','compound','module') | YES  |     | NULL    |       |
| comment   | text                              |      |     |         |       |
+-----------+-----------------------------------+------+-----+---------+-------+
```

Figure 24 Description of the logical table `filetype`

```
mysql> select * from filetype;
+----------+--------------+-----------+----------+-----------------------------------------+
| filetype | name         | directory | member   | comment                                 |
+----------+--------------+-----------+----------+-----------------------------------------+
|        0 | system       | sys       | NULL     | (reserved for system functions)         |
|        1 | vxw          | vxw       | module   | VxWorks RTOS image                      |
|        2 | relocateable | rel       | module   | Relocateable code module                |
|        3 | cdm          | cdm       | module   | CDM format configuration file           |
|        4 | hsk          | hsk       | simple   | Housekeeping configuration files        |
|        5 | lci          | lci       | simple   | Charge injection configuration/script file |
|        6 | latc         | latc      | simple   | Instrument configuration file           |
|        7 | fof          | fof       | compound | File of files                           |
|        8 | sbs          | sbs       | compound | Secondary boot script                   |
|        9 | sbm          | sbm       | module   | Secondary boot module                   |
|       10 | pbc          | pbc       | module   | Primary boot code                       |
|       11 | ltc          | ltc       | simple   | Thermal control configuration files     |
|       12 | serial       | ser       | simple   | Serial number (embedded in SUROM)       |
+----------+--------------+-----------+----------+-----------------------------------------+
```

Figure 25 Dump of the logical table `filetype`

Field	Description
`filetype`	The primary (unique) index for the table. Follows the tradition of naming the primary key after the table name.
`name`	A “trivial” name for the file type. Must be unique.
`directory`	The directory name where such files are stored. This directory name is used both for creating storage locations in the ground based “file repository”, and as a “pseudo” directory name to refer to directories on embedded file systems. Must be unique.
`member`	The processing group the file type maps to. Only three processing groups are defined (so far).
`comment`	A free comment field.

Table 9 Extended description of the hardware mapping table `instrument2crate`

Notes:

- The distinction between `name` and `directory` is historical. As the FMX design has evolved the utility of this distinction has become less and less. Maybe some brave soul will one day upgrade FMX to eliminate the distinction completely.
- File type `sbm` is somewhat of a curiosity. An `sbm` is, in fact, a relocateable module. It’s only distinguishing feature is that an `sbm` must always have an entry point called `SBC_init()`. It’s valuable for FMX to maintain the distinction however, because an `sbm` module can be meaningfully uploaded to a greater variety of locations (a non-`sbm` relocateable module does not do very well when it’s used as a secondary boot module).
- The statement “the processing of all file types associated with a given file group is identical” is not strictly true. There are subtle differences in the processing of, for instance, file types `relocateable` (a relocateable code module) and `vxw` (an absolute code module). However, those differences are so slight compared to the bulk processing of the file group that it’s much easier to have one processing chain per file group and implement file type based code branches to accommodate the differences between the members of the group.

To capture information about different file types, FMX uses two tables common to all file groups and then additional tables specific to a file group. The mappings are as follows:

Table Name	Table Usage
`logical, visible`	Common to all file types
`simple`	Specific to file group `simple`
`module`	Specific to file group `module`
`unresolved, resolved, fileseq`	Specific to file group `compound`

Table 10 Inventory of tables supporting logical files

Starting with the common tables:

4.1 Common Tables

4.1.0 Logical Table **`logical`**

```
mysql> describe logical;
+----------+------------------+------+-----+---------------------+-------+
| Field    | Type             | Null | Key | Default             | Extra |
+----------+------------------+------+-----+---------------------+-------+
| logical  | int(10) unsigned |      | PRI | 0                   |       |
| external | int(10) unsigned |      |     | 0                   |       |
| filetype | smallint(6)      |      |     | 0                   |       |
| nickname | varchar(8)       |      |     |                     |       |
| locked   | enum('Y','N')    |      |     | Y                   |       |
| lockedBy | varchar(64)      |      |     |                     |       |
| lockedAt | datetime         |      |     | 0000-00-00 00:00:00 |       |
| added    | datetime         |      |     | 0000-00-00 00:00:00 |       |
+----------+------------------+------+-----+---------------------+-------+
```

Figure 26 Description of the logical table `logical`

```
mysql> select * from logical;
+---------+------------+----------+----------+--------+------------------------+
| logical | external   | filetype | nickname | locked | lockedBy               |
+---------+------------+----------+----------+--------+------------------------+
|       1 | 4294967295 |        2 | cdm      | N      | flora01-apw-6977       |
|       2 | 4294967295 |        2 | cpu_db_s | N      | flora01-apw-7001       |
|       3 | 4294967295 |        2 | lem_db_s | N      | flora01-apw-7025       |
|       4 | 4294967295 |        2 | rad750_r | N      | flora01-apw-7049       |
|       5 | 4294967295 |        2 | pbs      | N      | flora01-apw-7073       |
|       6 | 4294967295 |        2 | msg_mt   | N      | flora01-apw-7103       |
|       7 | 4294967295 |        2 | msg_prin | N      | flora01-apw-7127       |
|       8 | 4294967295 |        2 | imm      | N      | flora01-apw-7151       |
|       9 | 4294967295 |        2 | ccsds_pk | N      | flora01-apw-7175       |
|      10 | 4294967295 |        2 | itc      | N      | flora01-apw-7199       |
|      11 | 4294967295 |        2 | sumt_rt_ | N      | flora01-apw-7247       |
.         .            .          .          .        .                        .
.         .            .          .          .        .                        .
|     308 | 4294967295 |        8 | siu_ddt  | N      | lat-licos01-glast-27519 |
|     309 | 4294967295 |        8 | siu_ddt  | N      | lat-licos01-glast-27618 |
|     310 | 4294967295 |        8 | epu_ddt  | N      | lat-licos01-glast-27655 |
|     311 | 4294967295 |        3 | ltc_data | N      | flora04-apw-295        |
|     312 | 4294967295 |        8 | siu_tvac | N      | flora04-apw-411        |
|     313 | 4294967295 |        8 | siu_tvac | N      | lat-hobbit5-jana-26597 |
+---------+------------+----------+----------+--------+------------------------+
```

Figure 27 Partial dump of the logical table `logical`

Field	Description
`logical`	The primary (unique) index for the table. This is the *logical file key* spoken of if the introductory section. This is the central trick of FMX. A unique logical file key is assigned to a file when it's added to the database, and all other tables use the logical file key to cross-reference their rows back to this table. It's also the number planted into the file header of an uploadable file.
`external`	From the FMX perspective, a free field. It was implemented so that another tool (MOOT for instance) could plant a number significant to MOOT in the FMX database, thus allowing back references to the rows in MOOD tables from a logical record retrieved from FMX.
`filetype`	Discussed above.
`nickname`	A very nearly free field. All files uploaded to embedded file systems are prefixed with a file header. The file header provides space for an arbitrary eight byte ASCII string (actually it doesn't even have to be ASCII, it could be any eight bytes). This area has been dubbed the file's "nickname" (after all, it's pretty short!), and FMX captures the nickname planted into the file header here.
`locked`	The result of early paranoia. Please see notes.
`lockedBy`	The result of early paranoia. Please see notes.
`lockedAt`	The result of early paranoia. Please see notes.
`added`	Timestamp (in UTC time) for when the file was added to FMX.

Table 11 Extended description of the logical table `logical`

Notes:

- I have lopped off most rows and the timestamp columns in Figure 27 to maintain clarity.

- Being the central trick for FMX, this is one of the first tables I implemented. I was determined that there would be no way the table could get corrupted. In my zeal, I introduced all sorts of safety gizmos, including these hand held row locking columns with back traces to the node/user/process taking the lock. With hindsight, this was overkill. MySQL is fully functional database (particularly so when tables are specified to use the InnoDB storage engine) and there were easier, more “databasey” ways to achieve the same result. With a little bit of effort all the lock columns could be removed and replaced with a proper auto-increment column and “for update” clauses in some SQL queries. Given that the current system, ugly as it is, achieves the desired goal, reworking it has never risen to the top of the priority list.

4.1.1 Logical Table **visible**

```
mysql> describe visible;
+---------+------------------+------+-----+---------------------+-------+
| Field   | Type             | Null | Key | Default             | Extra |
+---------+------------------+------+-----+---------------------+-------+
| visible | int(10) unsigned |      | PRI | 0                   |       |
| size    | int(10) unsigned |      |     | 0                   |       |
| added   | datetime         |      |     | 0000-00-00 00:00:00 |       |
| name    | text             |      | MUL |                     |       |
| file    | longblob         |      |     |                     |       |
+---------+------------------+------+-----+---------------------+-------+
```

Figure 28 Description of the logical table `visible`

```
mysql> select visible,size,name from visible;
+---------+-------+-------------------------------------------------------------+
| visible | size  | name                                                        |
+---------+-------+-------------------------------------------------------------+
|       1 | 44275 | rel/CDM/V0-2-2/mcp750/cdm/libcdm.o                          |
|       2 |  2011 | rel/CDM/V0-2-2/mcp750/cdm/cdm.f                             |
|       3 | 44275 | rel/CDM/V0-2-2/mv2304/cdm/libcdm.o                          |
|       4 |  2010 | rel/CDM/V0-2-2/mv2304/cdm/cdm.f                             |
|       5 | 44275 | rel/CDM/V0-2-2/rad750/cdm/libcdm.o                          |
|       6 |  2011 | rel/CDM/V0-2-2/rad750/cdm/cdm.f                             |
|       7 | 19638 | rel/CPU_DB/V0-2-1/mcp750/cpu_db_server/libcpu_db_server.o   |
|       8 |   829 | rel/CPU_DB/V0-2-1/mcp750/cpu_db_server/cpu_db_server.f      |
|       9 | 19638 | rel/CPU_DB/V0-2-1/mv2304/cpu_db_server/libcpu_db_server.o   |
.         .       .                                                             .
.         .       .                                                             .
|    1521 | 12242 | cdm/LTC_DB/V1-0-2/linux-gcc/ltc_data/libltc_data.so         |
|    1522 | 19724 | cdm/LTC_DB/V1-0-2/mcp750/ltc_data/libltc_data.o             |
|    1523 |   885 | cdm/LTC_DB/V1-0-2/mcp750/ltc_data/ltc_data.f                |
|    1524 | 19724 | cdm/LTC_DB/V1-0-2/mv2304/ltc_data/libltc_data.o             |
|    1525 |   884 | cdm/LTC_DB/V1-0-2/mv2304/ltc_data/ltc_data.f                |
|    1526 | 19724 | cdm/LTC_DB/V1-0-2/rad750/ltc_data/libltc_data.o             |
|    1527 |   885 | cdm/LTC_DB/V1-0-2/rad750/ltc_data/ltc_data.f                |
|    1528 | 34468 | cdm/LTC_DB/V1-0-2/sun-gcc/ltc_data/libltc_data.so           |
|    1529 |  3676 | sbs/FTS/V0-1-7/B0-6-9/siu.fmx.tvac                          |
|    1530 |   456 | sbs/FTS/V0-1-7/B0-6-9/siu.fmx.tvac.0000313.f                |
+---------+-------+-------------------------------------------------------------+
```

Figure 29 Partial dump of the logical table `visible`

Field	Description

Field	Description
visible	The primary (unique) index for the table.
size	Size of the file.
added	Timestamp (in UTC time) for when the file was added to this table.
name	File name. This is the file name as at appears in the host file system. Expressed as a full path name relative to some root directory. Where an FMX installation is set up to replicate, each database would have its own associated visible file space.
file	The file. Literally!

Table 12 Extended description of the logical table visible

Notes:

- I have lopped off most rows in Figure 29 to maintain clarity.
- Interestingly, there is no field named logical in this table. That is because the different file groups produce different numbers of records in the visible table. To map between the logical table and the visible table, it's necessary to map through the group specific tables.
- That last entry, file, really is the file! There is a common misconception that the visible file tree maintained on the host's filing system is the file repository. It isn't. The whole visible file tree could be blown away and then completely recreated from the files here in the visible table. Once again, this dates to the original implementation of FMX. The database was used as a poor man's file transfer system, so that the visible file tree could be recreated on a number of hosts from a single database. Be aware that this is completely the opposite of how MOOD is structured.

That file column can have other unintended side effects! Be very careful when running SQL "select" statements against this table. The query:

```
mysql> select * from visible where name='<some_file_name>';
```

will print rather more than you intended!

4.2 Simple Tables

4.2.0 Logical Table **simple**

```
mysql> describe simple;
+---------+------------------+------+-----+---------+-------+
| Field   | Type             | Null | Key | Default | Extra |
+---------+------------------+------+-----+---------+-------+
| logical | int(10) unsigned |      | PRI | 0       |       |
| import  | int(10) unsigned |      | UNI | 0       |       |
| export  | int(10) unsigned |      | UNI | 0       |       |
+---------+------------------+------+-----+---------+-------+
```

Figure 30 Description of the logical table simple

```
mysql> select * from simple;
+---------+--------+--------+
| logical | import | export |
+---------+--------+--------+
|      84 |    553 |    554 |
|      85 |    555 |    556 |
|     300 |   1502 |   1503 |
|     301 |   1504 |   1505 |
|     302 |   1506 |   1507 |
|     303 |   1508 |   1509 |
|     304 |   1510 |   1511 |
|     305 |   1512 |   1513 |
|     306 |   1514 |   1515 |
|     307 |   1516 |   1517 |
+---------+--------+--------+
```

Figure 31 Dump of the logical table `logical`

Field	Description
`logical`	The file's logical key.
`import`	The index in the visible table where the "import" file is stored.
`export`	The index in the visible table where the "export" (prepared for flight) file is stored.

Table 13 Extended description of the logical table `simple`

Notes:

- Simple is right! This is a very simple case!

4.3 Module Tables

4.3.0 Logical Table **`module`**

```
mysql> describe module;
+-------------+------------------+------+-----+---------+-------+
| Field       | Type             | Null | Key | Default | Extra |
+-------------+------------------+------+-----+---------+-------+
| logical     | int(10) unsigned |      | MUL | 0       |       |
| import      | int(10) unsigned |      | PRI | 0       |       |
| export      | int(10) unsigned | YES  | MUL | NULL    |       |
| package     | varchar(32)      |      | MUL |         |       |
| version     | varchar(32)      |      |     |         |       |
| constituent | varchar(32)      |      |     |         |       |
| tag         | tinyint(4)       |      |     | 0       |       |
+-------------+------------------+------+-----+---------+-------+
```

Figure 32 Description of the logical table `module`

```
mysql> select * from module;
+---------+--------+--------+---------+---------+---------------+-----+
| logical | import | export | package | version | constituent   | tag |
+---------+--------+--------+---------+---------+---------------+-----+
|       1 |      3 |      4 | CDM     | V0-2-2  | cdm           |   0 |
|       1 |      1 |      2 | CDM     | V0-2-2  | cdm           |   1 |
|       1 |      5 |      6 | CDM     | V0-2-2  | cdm           |   2 |
|       2 |      9 |     10 | CPU_DB  | V0-2-1  | cpu_db_server |   0 |
|       2 |      7 |      8 | CPU_DB  | V0-2-1  | cpu_db_server |   1 |
|       2 |     11 |     12 | CPU_DB  | V0-2-1  | cpu_db_server |   2 |
|       3 |     15 |     16 | LEM_DB  | V0-1-1  | lem_db_server |   0 |
|       3 |     13 |     14 | LEM_DB  | V0-1-1  | lem_db_server |   1 |
|       3 |     17 |     18 | LEM_DB  | V0-1-1  | lem_db_server |   2 |
|       4 |     21 |     22 | RAD750  | V1-3-4  | rad750_reboot |   0 |
|       4 |     19 |     20 | RAD750  | V1-3-4  | rad750_reboot |   1 |
|       4 |     23 |     24 | RAD750  | V1-3-4  | rad750_reboot |   2 |
|       5 |     27 |     28 | PBS     | V2-10-7 | pbs           |   0 |
|       5 |     25 |     26 | PBS     | V2-10-7 | pbs           |   1 |
|       5 |     29 |     30 | PBS     | V2-10-7 | pbs           |   2 |
.         .        .        .         .         .               .     .
|     283 |   1467 |   1468 | THS     | V1-5-0  | ths           |   0 |
|     283 |   1465 |   1466 | THS     | V1-5-0  | ths           |   1 |
|     283 |   1469 |   1470 | THS     | V1-5-0  | ths           |   2 |
|     284 |   1471 |   1472 | THS     | V1-5-0  | ths4ddt       |   2 |
|     311 |   1524 |   1525 | LTC_DB  | V1-0-2  | ltc_data      |   0 |
|     311 |   1522 |   1523 | LTC_DB  | V1-0-2  | ltc_data      |   1 |
|     311 |   1526 |   1527 | LTC_DB  | V1-0-2  | ltc_data      |   2 |
|     311 |   1521 |   NULL | LTC_DB  | V1-0-2  | ltc_data      |   3 |
|     311 |   1528 |   NULL | LTC_DB  | V1-0-2  | ltc_data      |   4 |
+---------+--------+--------+---------+---------+---------------+-----+
```

Figure 33 Partial dump of the logical table `module`

Field	Description
`logical`	The file's logical key.
`import`	The index in the visible table where the "import" file is stored.
`export`	The index in the visible table where the "export" (prepared for flight) file is stored.
`package`	The CMX package name.
`version`	The CMX version number of the package.
`constituent`	The CMX constituent name.
`tag`	The target architecture for the module (an index from table `tag`).

Table 14 Extended description of the logical table `module`

Notes:

- Now things start to get interesting!
- I have lopped off most rows in Figure 23 to maintain clarity.
- When dealing with modules, FMX once again betrays its FSW heritage by dealing, not in simple filenames as the `simple` files do, but in an abstraction. The abstraction comes from CMX, and is the ability to completely identify a code module by specifying:
    - Package name
    - Package version number

    - Constituent name
    - Target architecture
- Given a CMX “file” identification, FMX can capture that “file” for all relevant architectures. Each architecture gets a separate row in the `module` table.
- For pure code modules, FMX will try to capture a copy of the module for each embedded system architecture, and prepare a corresponding “prepared for flight” version of each one. One “add” request for a pure code module can thus result in:
    - One entry in the `logical` table.
    - Three entries in the `module` table (one per embedded system architecture).
    - Six entries in the `visible` table (times two for “import” and “export” files).
- In the special case of CDM modules, there’s a good chance that these will be needed in ground based software when data is processed. For CDM modules, FMX also attempts to capture the CMX’s `sun-gcc` and `linux-gcc` tags for those modules (though FMX does not go to try to prepare the sun and linux versions for flight!). Even so, adding a CDM module to FMX can add:
    - One entry in the `logical` table.
    - Five entries in the `module` table (one per architecture).
    - Eight entries in the `visible` table.

4.4 Compound Tables

Better fasten your seat belt. Turbulence ahead.

First of all, what *is* a compound file. A compound file is a file that contains references to other files. There are currently two such (recognized) file types:

- `fof` File of files.
- `sbs` Secondary boot script.

A `fof` is literally that. It’s a file whose whole content is simply a list of other files. The mechanism is completely general, though the only use that’s been made so far of `fofs` is the binding together of a series of LATC (LAT Configuration) files.

An `sbs` is a little more sophisticated. It’s in fact a list of modules to load during secondary boot and a list of calls to get the system up and running. Secondary boot scripts are a second home for FSW developers and there will be a section later on dedicated to their syntax and usage. For the purposes of this section, their only feature of note is that each “load” line in a secondary boot script refers to another file, in this case, a code module.

The difficulty is that those file references are ideally phrased as references to other files already extant on an embedded file system. At the moment when a compound file is added to FMX, the exact locations/names of the files referred to are not necessarily known. Indeed the files might not even have been loaded.

To accommodate this, compound files go through a two stage process. When a compound file is added to FMX, it is stored in its *unresolved* form. It stays like this until a request is made to upload the compound file to a target. At that point, the FMX script attempts to resolve all the file references to files already listed as present and available on the target. If that resolution process is successful, FMX creates a new *resolved* file and keeps records of what it did in the specialized compound file tables.

Well nearly true. In fact, the FMX script can detect if the resolution exactly matches a previous effort to resolve the file. If it does, it simply returns directions to the existing resolution, and does not create a new resolution file.

In terms of the FMX database, the biggest difficulty in all of this was how to track all these resolution files. Consider that for members of file group `simple`, the mapping from import file to export file is a simple one-to-one. Even for code modules the mapping is strictly bounded; there can never be more than five import files and three export files for one logical import. These are easy mappings to represent in a database.

For compound files, the mapping is open-ended. A single import file may map to any number of export files because the content of the export file is governed by the exact contents of the target file system at the time the resolution is attempted.

There's also a somewhat more philosophical difficulty involving what is meant by a resolved file.

As a matter of practicality, embedded file systems are currently being run as a redundant pair, with the contents of the lower EEPROM bank being mirrored in the upper EEPROM bank. When a compound file is resolved, the resolution process needs direction about which devices it is allowed to search in order to find resolutions. In keeping with the use of upper and lower EEPROM banks being used as a mirrored pair, the standard methodology has become to resolve a compound file once against the contents of the lower bank (and then to upload that resolution to the lower bank), and then resolve it again against the contents of the upper bank (and then upload this second resolution to the upper bank). Assuming that the banks are true mirrors, there's nothing logically different between the compound file in the lower bank and the compound file in the upper bank. On the other hand, should that mirroring ever break down (e.g. a file gets corrupted), then it might be valuable to know which specific resolution was in use.

The FMX tables remember the physical distinction between alternate resolutions of a compound file by assigning each new resolution its own *logical file key*. (Well I did warn you that this was going to get complicated!). This logical file key is also inserted in the uploadable file header. Thus if the distinction of the resolution is important, the logical key from the file header points to a specific resolution. If this distinction is unimportant, the key returned from the file header can be run backwards through the `resolved` table to identify the unresolved, *parent* file.

OK, I've probably confused you enough by now. Here are the tables.

4.4.0 Logical Table **`unresolved`**

```
mysql> describe unresolved;
+---------+------------------+------+-----+---------+-------+
| Field   | Type             | Null | Key | Default | Extra |
+---------+------------------+------+-----+---------+-------+
| logical | int(10) unsigned |      | PRI | 0       |       |
| import  | int(10) unsigned |      | UNI | 0       |       |
+---------+------------------+------+-----+---------+-------+
```

Figure 34 Description of the logical table `unresolved`

```
mysql> select * from unresolved;
+---------+--------+
| logical | import |
+---------+--------+
|      86 |    557 |
|      87 |    558 |
|     128 |    799 |
|     129 |    800 |
|     147 |    907 |
|     148 |    908 |
|     171 |   1032 |
|     172 |   1033 |
|     187 |   1100 |
|     188 |   1101 |
|     195 |   1118 |
|     196 |   1119 |
|     231 |   1298 |
|     232 |   1299 |
|     243 |   1322 |
|     244 |   1323 |
|     260 |   1396 |
|     261 |   1397 |
|     269 |   1407 |
|     270 |   1408 |
|     291 |   1493 |
|     292 |   1494 |
|     294 |   1496 |
|     295 |   1497 |
|     296 |   1498 |
|     312 |   1529 |
|     344 |   1716 |
|     345 |   1717 |
|     346 |   1718 |
|     347 |   1719 |
+---------+--------+
```

Figure 35 Dump of the logical table `unresolved`

Field	Description
`logical`	The file's logical key (this is the *parent*, *unresolved* file)
`import`	The index in the visible table where the "import" file is stored.

Table 15 Extended description of the logical table `unresolved`

Notes:

- Get deep enough into a database and everything turns into numbers!
- When a compound file is imported, just three records are generated:
  - One record in table `logical` (for the *parent*, unresolved file)
  - One record in table `unresolved`.
  - One record in table `visible`.

4.4.1 Logical Table **resolved**

```
mysql> describe resolved;
+----------+------------------+------+-----+---------+-------+
| Field    | Type             | Null | Key | Default | Extra |
+----------+------------------+------+-----+---------+-------+
| parent   | int(10) unsigned |      |     | 0       |       |
| logical  | int(10) unsigned |      | PRI | 0       |       |
| export   | int(10) unsigned |      | UNI | 0       |       |
| searched | varchar(16)      |      |     |         |       |
| resolved | varchar(16)      |      |     |         |       |
+----------+------------------+------+-----+---------+-------+
```

Figure 36 Description of the logical table `resolved`

```
mysql> select * from resolved;
+--------+---------+--------+-----------+----------+
| parent | logical | export | searched  | resolved |
+--------+---------+--------+-----------+----------+
|    147 |     149 |    909 | /ee0,/ee1 | /ee0     |
|    171 |     173 |   1034 | /ee0,/ee1 | /ee0     |
|    171 |     175 |   1036 | /ee1      | /ee1     |
|    172 |     174 |   1035 | /ee0      | /ee0     |
|    172 |     176 |   1037 | /ee1      | /ee1     |
.        .         .        .           .          .
|    344 |     349 |   1721 | /ee1,/ee0 | /ee1     |
|    345 |     352 |   1724 | /ee1,/ee0 | /ee1     |
|    346 |     348 |   1720 | /ee1,/ee0 | /ee1     |
|    347 |     351 |   1723 | /ee1,/ee0 | /ee1     |
+--------+---------+--------+-----------+----------+
```

Figure 37 Partial dump of the logical table `resolved`

Field	**Description**
`parent`	The file's logical key (this is the *parent*, *unresolved* file)
`logical`	The file's logical key (this is the *child*, *resolved* file)
`export`	The index in the visible table where the "export" file is stored.
`searched`	Ordered list of devices to search during the resolution process.
`resolved`	List of devices that provided file resolutions.

Table 16 Extended description of the logical table `resolved`

Notes:

- I have lopped off most rows in Figure 37 to maintain clarity.
- The resolved table keeps records of the devices *searched* during resolution.
- The resolved table keeps records of the devices that *provided* resolutions.
- If an attempt at resolution results in new file, the following records are created
    - One record in table `logical` (for the new *child* resolution).
    - One record in table `resolved`.
    - One record in table `visible`.
    - Several records in table `fileseq` (see next section).

4.4.2 Logical Table **fileseq**

```
mysql> describe fileseq;
+----------+------------------+------+-----+---------+-------+
| Field    | Type             | Null | Key | Default | Extra |
+----------+------------------+------+-----+---------+-------+
| logical  | int(10) unsigned |      | PRI | 0       |       |
| sorted   | int(10) unsigned |      | PRI | 0       |       |
| physical | int(10) unsigned |      |     | 0       |       |
+----------+------------------+------+-----+---------+-------+
```

Figure 38 Description of the logical table `fileseq`

```
mysql> select * from fileseq;
+---------+--------+---------------+
| logical | sorted | physical      |
+---------+--------+---------------+
.         .        .               .
|     313 |     64 | 40C0003E      |
|     314 |      0 | 60800001      |
|     314 |      1 | 6080010F      |
|     314 |      2 | 608000D4      |
|     314 |      3 | 60800004      |
|     314 |      4 | 60800062      |
|     314 |      5 | 60800006      |
|     314 |      6 | 60800007      |
|     314 |      7 | 60800008      |
|     314 |      8 | 60800009      |
|     314 |      9 | 60800077      |
|     314 |     10 | 60800111      |
|     314 |     11 | 60800063      |
|     314 |     12 | 6080000C      |
|     314 |     13 | 60800064      |
|     314 |     14 | 608000FD      |
|     314 |     15 | 6080000F      |
|     314 |     16 | 60800010      |
|     314 |     17 | 608000D0      |
|     314 |     18 | 608000D1      |
|     314 |     19 | 608000E3      |
|     314 |     20 | 608000E4      |
|     314 |     21 | 608000E2      |
|     314 |     22 | 60800016      |
|     314 |     23 | 60800017      |
|     314 |     24 | 60800018      |
|     314 |     25 | 60800118      |
|     314 |     26 | 6080001A      |
|     314 |     27 | 6080011B      |
|     314 |     28 | 60800068      |
|     314 |     29 | 6080001D      |
|     314 |     30 | 6080009C      |
|     314 |     31 | 6080009D      |
|     314 |     32 | 60800102      |
|     314 |     33 | 608000FF      |
|     314 |     34 | 608000FE      |
|     314 |     35 | 60800113      |
|     314 |     36 | 60800112      |
|     314 |     37 | 60800110      |
|     314 |     38 | 60800115      |
|     314 |     39 | 60800114      |
|     314 |     40 | 608000C1      |
|     314 |     41 | 608000CF      |
|     314 |     42 | 608000D9      |
|     314 |     43 | 60800101      |
|     314 |     44 | 60C0010C      |
|     314 |     45 | 60C0002E      |
|     314 |     46 | 60C0002F      |
|     314 |     47 | 60C00030      |
|     314 |     48 | 60C00074      |
|     314 |     49 | 60C00032      |
```

```
|      314 |      50 | 60C000D3       |
|      314 |      51 | 60C000DA       |
|      314 |      52 | 60C000DB       |
|      314 |      53 | 60C000DC       |
|      314 |      54 | 60C00034       |
|      314 |      55 | 60C000B7       |
|      314 |      56 | 60C00076       |
|      314 |      57 | 60C00137       |
|      314 |      58 | 60C00038       |
|      314 |      59 | 60C00039       |
|      314 |      60 | 60C0003A       |
|      314 |      61 | 60C0003B       |
|      314 |      62 | 60C0003C       |
|      314 |      63 | 60C0003D       |
|      314 |      64 | 60C0003E       |
|      348 |       0 | 60800001       |
.          .         .                .
+----------+---------+----------------+
```

Figure 39 Partial dump of the logical table `fileseq`

Field	Description
`logical`	The file's logical key (this is the *child*, *resolved* file)
`sorted`	A key to preserve the order of the physical file resolutions.
`physical`	The resolved file reference.

Table 17 Extended description of the logical table `fileseq`

Notes:

- I have lopped off most rows in Figure 37 to maintain clarity, but the pattern of exclusion is a little different. I have retained the "last" record for logical file 313, all the records for logical file 314, and the first record for logical file 348.
- FMX believes that the files in a compound file are an *ordered* list. When recording a resolution, it therefore keeps a "sorting" parameter (the field `sorted`).
- I printed the last column in hex because these are (finally) the physical file names that appear in telecommands/telemetry and are the mandated form to refer to files in the embedded file systems. A `physical` number like this is translated (algorithmically) to a file name string on board. Given the algorithm, these numbers are far easier to interpret when viewed in hex (and these types of file ID numbers will be discussed more extensively in the next section).

If you're still with me, congratulations! You've now made it through the most confusing piece of the database. There's one more twiddly bit in the physical file tables, but nothing as bad as dealing with compound files.

5 Physical File Tables

The end of the previous section introduced for the first time the idea of a physical file number, or *file ID*. Now would be a good time to go into that in more detail (the defining document for this material is the FILE package traveler).

There is very little space in a telecommand to put a string representing a file name (and string handling in some of the available tools is not always the best). Thus file *names* are never used in either telecommands of telemetry. Instead each file is referred to with a 32-bit number.

That solves the compactness problem in telecommands, but the (TFFS section) of the on board file systems really are file systems! They expect to use file names! This is accommodated by defining an algorithm to turn the 32 bit number into a string (and, if syntactically feasible, vice-versa). The 32 bit number is bit-field encoded as follows (bit 0 is most significant bit):

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Device				“Directory”						“File name”																					

Table 18 Format of a file ID

The three bit device is translated directly into a device string:

Device number	Device name string
0	`/boot`
1	`/ram`
2	`/ee0`
3	`/ee1`
4	`/usr0`
5	`/usr1`
6	
7	

Table 19 File devices as defined in flight software

The seven bit directory field is translated directly to a numeric string preceded by the letter `d`, producing strings like `d012`. The twenty-two bit file name field is similarly translated to a numeric string, this time prefixed by the letter `f`, producing strings like `f0000123`. Thus the hex ID `0x40C00028` would translate to the string `/ee0/d003/f0000040`.

OK, the telecommands are happy (very compact) and file system is happy (has file names). The only party left out in the cold is the human! (Humans are not very instinctive dealing with file references like `0x40C00028` and `/ee0/d003/f0000040`).

FMX helps out a little here. Starting with one simplification, FMX tries to present a very uniform view of the file system (though unfortunately, those file names are still in the form `f0000123`).

The simplifying assumption is to take FMX concepts and map them onto parts of the file name. When constructing an upload file ID, FMX puts the file type into the seven bits of the directory field, and puts the logical file key into the twenty-two bit file name field. Thus all files of file type 2 (relocateable modules) always go into the on-board directory 2, all files of file type 4 (housekeeping) go into the on-board directory 4.

FMX also tries to regularize the names and "appearance" of on-board devices. This was in part to end a couple of bad language habits that many in FSW were prone to (including me). We would often refer to "uploading a file to RAM". The point of confusion with that statement was that during primary boot, it is indeed possible to upload a file to RAM … meaning precisely that … RAM. Unfortunately, it's also possible to upload a file to "RAM" after secondary boot, except that more often than not, "RAM" in that case meant the `/ram` disk.

The other verbal trick was to use the device names for the two TFFS partitions, `/ee1` and `/ee0`, interchangeably with references to the upper and lower EEPROM banks. They're not quite the same thing! We've divided each EEPROM bank into two pieces: a memory mapped segment where we keep specialized secondary boot files, and a (larger) segment formatted as a true file system (the latter being the `/ee1` and `/ee0` devices). The problem was that we never really had a word to refer separately to the memory mapped sections.

So FMX has invented some "virtual" devices. Note that, with the exception of `/ram`, `/ee0`, and `/ee1`, these are names that have no real existence in the FSW running on board. Where necessary, FMX "reverse translates" into something the on-board software understands. The real pay-off for this extra work comes at the command line when using the FMX script. Many of the exchanges (commands and responses) are couched in terms of more human-readable file names. The difference isn't too noticeable for the well known devices (`/ram`, `/ee0`, `/ee1`):

In telecommand	**FSW on-board software**			**In FMX interactions**		
File ID	Device	Directory	File	File Key	File Type	Appears as
`0x4020011b`	`/ee0`	`d002`	`f0000283`	`283`	`rel`	`/ee0/rel/f0000283`
`0x61000055`	`/ee1`	`d004`	`f0000085`	`85`	`hsk`	`/ee1/hsk/f0000085`
`0x2030003e`	`/ram`	`d003`	`f0000062`	`62`	`cdm`	`/ram/cdm/f0000062`

Table 20 Examples of on-board file IDs (for files on TFFS devices)

but becomes more apparent when talking about files in the memory mapped sections:

In telecommand	**FSW on-board software**			**In FMX interactions**		
File ID	Device	Directory	File	File Key	File Type	Appears as
`0x00000000`	`/boot`	`d000`	`f0000000`	`90`	`vxw`	`/mem/vxw/f0000090`
`0x00000001`	`/boot`	`d000`	`f0000001`	`259`	`sbm`	`/mem/sbm/f0000259`
`0x00000002`	`/boot`	`d000`	`f0000002`	`286`	`sbs`	`/mem/sbs/f0000286`
`0x00000003`	`/boot`	`d000`	`f0000003`	`90`	`vxw`	`/mm0/vxw/f0000090`
`0x00000004`	`/boot`	`d000`	`f0000004`	`259`	`sbm`	`/mm0/sbm/f0000259`

0x00000005	/boot	d000	f0000005	286	sbs	/mm0/sbs/f0000286
0x00000006	/boot	d000	f0000006	90	vxw	/mm1/vxw/f0000090
0x00000007	/boot	d000	f0000007	259	sbm	/mm1/sbm/f0000259
0x00000008	/boot	d000	f0000008	286	sbs	/mm1/sbs/f0000286

Table 21 Examples of on-board file IDs (for files on memory mapped devices)

At which point, I should put down the definition of devices in FMX.

5.0 Physical Table **device**

```
mysql> describe device;
+--------+--------------------------------+------+-----+----------+-------+
| Field  | Type                           | Null | Key | Default  | Extra |
+--------+--------------------------------+------+-----+----------+-------+
| device | tinyint(4)                     |      | PRI | 0        |       |
| name   | varchar(8)                     |      | UNI |          |       |
| type   | enum('volatile','persistent') |      |     | volatile |       |
| board  | enum('cpu','sib')              |      |     | cpu      |       |
+--------+--------------------------------+------+-----+----------+-------+
```

Figure 40 Description of the physical table `device`

```
mysql> select * from device;
+--------+------+------------+-------+
| device | name | type       | board |
+--------+------+------------+-------+
|      1 | /ram | volatile   | cpu   |
|      2 | /ee0 | persistent | sib   |
|      3 | /ee1 | persistent | sib   |
|      4 | /mem | volatile   | cpu   |
|      5 | /mm0 | persistent | sib   |
|      6 | /mm1 | persistent | sib   |
|      7 | /rom | persistent | cpu   |
+--------+------+------------+-------+
```

Figure 41 Dump of the physical table `device`

Field	Description
`device`	The device index.
`name`	The device name.
`type`	Device persistence model (`volatile` or `persistent`).
`board`	Type of board where the device is located (`cpu` or `sib`).

Table 22 Extended description of the physical table `device`

Notes:

- FMX maintains extra information about the persistence model of a device in order to, it was planned, simplify the “scrubbing” of volatile devices when crate power was removed. To date, very little use has been made of volatile devices (or more accurately, when used, they are assumed to start from the empty state). FMX currently makes little or no use of the persistence information.
- `/mm0` is the memory mapped area of the lower EEPROM bank.

- `/mm1` is the memory mapped area of the upper EEPROM bank.
- `/mem` refers to the three RAM regions pre-allocated during primary boot where a non-persistent copy of the following three files (respectively) can be stored:
    - VxWorks RTOS.
    - Secondary boot module.
    - Secondary boot script.
- `/rom` was introduced for completeness. It refers to the 256 kByte "SUROM" on the rad750 board. To date there is very little support for this.

5.1 Physical File Tables

I know I've used a lot of words hammering on about devices, but it was a lack of understanding of the different nature of various devices that led to a sizeable error in versions of FMX prior to V3-0-0. Fixing that error involved changing the definitions of the tables about to be described, so it's worth taking a moment to see how the error came about.

Early versions of FMX only dealt with the `/ee0` and `/ee1` devices. Contents of the memory mapped sections were considered sufficiently slow moving that they could be controlled by hand, and no-one at that time was interested in using `/ram`. Given the simplification that FMX makes, identifying the directory with the file type and the file ID with the logical file key, it seemed impossible that a file could be accidentally "clobbered" (except by another copy of itself). Thus when recording file uploads, the physical tables assumed that the combination of the board to which it was being uploaded, the device and the logical file key was a "good" primary key (must be unique), and that violations of this key would indicate that an attempt was being made to overwrite a file (albeit with another copy of itself).

Doesn't work for memory mapped files! The following is a description of a real occurrence:

On 2006-05-06 the secondary boot script with logical file key 266 (B0-6-8) was loaded onto the device `/mm0` of SIU1 (on the LAT instrument).

On 2006-05-23 the secondary boot script with logical file key 286 (B0-6-9) was loaded onto the device `/mm0` of SIU1 (on the LAT instrument).

The files carried different logical file keys, so FMX assumed that it was impossible for one to clobber the other. At this point, FMX records showed that there were two files loaded on the section of `/mm0` dedicated to holding the secondary boot script. That section can only hold one file. Not good.

That forced the introduction of the `directory` and `fileID` fields in the physical file tables. When dealing with proper file systems (`/ram`, `/ee0`, and `/ee1`), the `directory` is mapped, as previously described, to the file type, and the file's `fileID` is mapped, as previously described, to the logical file key.

For memory mapped files however, there is a fixed mapping:

File type	Description	directory	fileID
vxw	VxWorks RTOS image	0	0
sbm	Secondary boot module	1	0
sbs	Secondary boot script	2	0

Table 23 Special `directory` and `fileID` mappings for memory mapped devices

By placing a uniqueness constraint on the combination of the board being uploaded to, the device, the directory and the file ID, it is no longer possible to have two files in the same place at the same time!

So on to the physical file tables:

5.1.0 Physical Table **physlog**

```
mysql> describe physlog;
+-----------+----------------------------------+------+-----+---------------------+-------+
| Field     | Type                             | Null | Key | Default             | Extra |
+-----------+----------------------------------+------+-----+---------------------+-------+
| logical   | int(10) unsigned                 |      |     | 0                   |       |
| board     | varchar(8)                       |      | PRI |                     |       |
| device    | tinyint(4)                       |      |     | 0                   |       |
| directory | tinyint(4)                       |      |     | 127                 |       |
| fileID    | int(10) unsigned                 |      |     | 4294967295          |       |
| command   | enum('commit','corrupt','delete') |      |     | commit              |       |
| logged    | datetime                         |      |     | 0000-00-00 00:00:00 |       |
| sequence  | int(10) unsigned                 |      | PRI | 1                   |       |
+-----------+----------------------------------+------+-----+---------------------+-------+
```

Figure 42 Description of the physical table `physlog`

```
mysql> select * from physlog;
+---------+----------+--------+-----------+--------+---------+---------------------+----------+
| logical | board    | device | directory | fileID | command | logged              | sequence |
+---------+----------+--------+-----------+--------+---------+---------------------+----------+
.         .          .        .           .        .         .                     .          .
|     314 | GLAT0925 |      6 |         2 |      0 | delete  | 2006-07-17 22:18:22 |      652 |
|     349 | GLAT0925 |      6 |         2 |      0 | commit  | 2006-07-17 22:18:22 |      653 |
|     311 | GLAT2207 |      2 |         3 |    311 | commit  | 2006-07-17 22:23:57 |      439 |
|     317 | GLAT1705 |      3 |         2 |    317 | commit  | 2006-07-17 22:24:37 |      268 |
|     286 | GLAT2207 |      5 |         2 |      0 | delete  | 2006-07-17 22:24:57 |      440 |
|     350 | GLAT2207 |      5 |         2 |      0 | commit  | 2006-07-17 22:24:57 |      441 |
|     325 | GLAT1705 |      3 |         2 |    325 | commit  | 2006-07-17 22:25:12 |      269 |
|     324 | GLAT1705 |      3 |         2 |    324 | commit  | 2006-07-17 22:25:25 |      270 |
|     323 | GLAT1705 |      3 |         2 |    323 | commit  | 2006-07-17 22:25:42 |      271 |
|     342 | GLAT1705 |      3 |         2 |    342 | commit  | 2006-07-17 22:25:55 |      272 |
|     343 | GLAT1705 |      3 |         2 |    343 | commit  | 2006-07-17 22:26:03 |      273 |
|     341 | GLAT1705 |      3 |         2 |    341 | commit  | 2006-07-17 22:26:30 |      274 |
|     338 | GLAT1705 |      3 |         2 |    338 | commit  | 2006-07-17 22:26:54 |      275 |
|     337 | GLAT1705 |      3 |         2 |    337 | commit  | 2006-07-17 22:27:17 |      276 |
|     320 | GLAT1705 |      3 |         2 |    320 | commit  | 2006-07-17 22:27:33 |      277 |
|     327 | GLAT1705 |      3 |         2 |    327 | commit  | 2006-07-17 22:28:12 |      278 |
|     328 | GLAT1705 |      3 |         2 |    328 | commit  | 2006-07-17 22:28:41 |      279 |
|     331 | GLAT1705 |      3 |         2 |    331 | commit  | 2006-07-17 22:28:56 |      280 |
|     321 | GLAT1705 |      3 |         2 |    321 | commit  | 2006-07-17 22:29:28 |      281 |
|     335 | GLAT1705 |      3 |         2 |    335 | commit  | 2006-07-17 22:29:42 |      282 |
|     322 | GLAT1705 |      3 |         2 |    322 | commit  | 2006-07-17 22:30:08 |      283 |
|     315 | GLAT1705 |      3 |         3 |    315 | commit  | 2006-07-17 22:30:18 |      284 |
|     316 | GLAT1705 |      3 |         3 |    316 | commit  | 2006-07-17 22:30:27 |      285 |
|     318 | GLAT1705 |      3 |         3 |    318 | commit  | 2006-07-17 22:30:37 |      286 |
|     339 | GLAT1705 |      3 |         2 |    339 | commit  | 2006-07-17 22:30:47 |      287 |
|     340 | GLAT1705 |      3 |         3 |    340 | commit  | 2006-07-17 22:30:56 |      288 |
|     351 | GLAT1705 |      3 |         8 |    351 | commit  | 2006-07-17 22:31:15 |      289 |
|     288 | GLAT1705 |      6 |         2 |      0 | delete  | 2006-07-17 22:32:20 |      290 |
|     352 | GLAT1705 |      6 |         2 |      0 | commit  | 2006-07-17 22:32:20 |      291 |
|     317 | GLAT0972 |      3 |         2 |    317 | commit  | 2006-07-17 22:34:54 |      266 |
|     325 | GLAT0972 |      3 |         2 |    325 | commit  | 2006-07-17 22:35:30 |      267 |
.         .          .        .           .        .         .                     .          .
+---------+----------+--------+-----------+--------+---------+---------------------+----------+
```

Figure 43 Partial dump of the physical table `physlog`

Field	Description
`logical`	The uploaded file's logical key
`board`	Board identifier for the target board.
`device`	Device index.
`directory`	Directory index (usually file type except as noted above).
`fileID`	File ID (usually the logical file key except as noted above).
`command`	Operation being record (one of `commit`, `delete`, or `corrupt`).
`logged`	Timestamp for the operation.
`sequence`	Sequence number to disambiguate identical timestamps (but see notes).

Table 24 Extended description of the physical table `physlog`

Notes:

- The `physlog` table records every file transaction against all instruments (at least, all those that it's told about!). Unsurprisingly, this is the biggest table in FMX (FMX has currently allocated ~350 logical file keys, for which there are ~3500 `physlog` records).
- FMX has continued to meet the design goal of recording uploads against boards, rather than against instruments. It is quite possible, for instance, to transfer a SIB between two instruments in the lab. and have the FMX database keep track.
- The `corrupt` operation is probably useless (to date, no corrupt records have been recorded). The original idea was that should an EEPROM fault develop, we could mark the file sitting on that fault "corrupt". The file would play no further part in operations, but would hide the bad location. Cute idea, but probably won't work because:
    - TFFS is allowed (even encouraged) to move files around to provide "wear leveling". There's no guarantee that the file will stay still.
    - With the more recent policy of keeping complete and independent operational copies on each of the EEPROM banks, a corrupted file cannot be substituted by its sibling in the opposite bank.
- There's more to the sequence number than meets the eye. This is due to the fact that in a replicating database scenario, two different database daemons could write to this table at the same time. It is essential under those circumstances that they do not write conflicting records. Because a board can only be mapped to one instrument at a time, and because an instrument can only be under the control of one database daemon at a time, the combination of the board number and an incrementing sequence number provides a "good" table key (i.e. guarantees unique rows).

5.1.1 Physical Table **`physical`**

```
mysql> describe physical;
+-----------+-------------------------------+------+-----+------------+-------+
| Field     | Type                          | Null | Key | Default    | Extra |
+-----------+-------------------------------+------+-----+------------+-------+
| logical   | int(10) unsigned              |      |     | 0          |       |
| board     | varchar(8)                    |      | PRI |            |       |
| device    | tinyint(4)                    |      | PRI | 0          |       |
| directory | tinyint(4)                    |      | PRI | 127        |       |
| fileID    | int(10) unsigned              |      | PRI | 4294967295 |       |
| state     | enum('committed','corrupted') |      |     | committed  |       |
+-----------+-------------------------------+------+-----+------------+-------+
```

Figure 44 Description of the physical table `physical`

```
mysql> select * from physical;
+---------+----------+--------+-----------+--------+-----------+
| logical | board    | device | directory | fileID | state     |
+---------+----------+--------+-----------+--------+-----------+
.         .          .        .           .        .           .
|     170 | GLAT2210 |      3 |         3 |    170 | committed |
|     204 | GLAT2210 |      3 |         3 |    204 | committed |
|     210 | GLAT2210 |      3 |         3 |    210 | committed |
|     218 | GLAT2210 |      3 |         3 |    218 | committed |
|     219 | GLAT2210 |      3 |         3 |    219 | committed |
|     220 | GLAT2210 |      3 |         3 |    220 | committed |
|     249 | GLAT2210 |      3 |         3 |    249 | committed |
|     250 | GLAT2210 |      3 |         3 |    250 | committed |
|     267 | GLAT2210 |      3 |         3 |    267 | committed |
|      90 | GLAT2210 |      5 |         0 |      0 | committed |
|     259 | GLAT2210 |      5 |         1 |      0 | committed |
|     285 | GLAT2210 |      5 |         2 |      0 | committed |
|      90 | GLAT2210 |      6 |         0 |      0 | committed |
|     259 | GLAT2210 |      6 |         1 |      0 | committed |
|     285 | GLAT2210 |      6 |         2 |      0 | committed |
|       1 | GLAT2212 |      3 |         2 |      1 | committed |
|       4 | GLAT2212 |      3 |         2 |      4 | committed |
|       7 | GLAT2212 |      3 |         2 |      7 | committed |
|       8 | GLAT2212 |      3 |         2 |      8 | committed |
|       9 | GLAT2212 |      3 |         2 |      9 | committed |
|      22 | GLAT2212 |      3 |         2 |     22 | committed |
|      23 | GLAT2212 |      3 |         2 |     23 | committed |
|      24 | GLAT2212 |      3 |         2 |     24 | committed |
|      26 | GLAT2212 |      3 |         2 |     26 | committed |
.         .          .        .           .        .           .
+---------+----------+--------+-----------+--------+-----------+
```

Figure 45 Partial dump of the physical table `physical`

Field	Description
`logical`	The uploaded file's logical key
`board`	Board identifier for the target board.
`device`	Device index.
`directory`	Directory index (usually file type except as noted above).
`fileID`	File ID (usually the logical file key except as noted above).
`state`	File state (one of `committed`, or `corrupted`).

Table 25 Extended description of the physical table `physical`

Notes:

- The `physical` table records the *current* state of all embedded file systems. This table is in fact redundant with the `physlog` table (the `physlog` table can give you the state of a file system at any given time in history. If that time is "now", you get the `physical` table). My concern was that the physlog table would grow very large and that "running the table" to get the most often sought after piece of information (what's the state now), would be prohibitively inefficient. As it stands, very few files have been deleted from on-board file systems, so the `physical` table is currently almost as large as the `physlog` table.

I was a little worried about keeping redundant information in the database, which probably explains the command:

```
fmx check physical …
```

which will be described later.

6 Special Database Topics

6.0 The Last Of The Tables

There are just two tables that haven't yet been described.

6.0.0 Table **numbers**

```
mysql> describe numbers;
+-------+--------------------------+------+-----+---------+-------+
| Field | Type                     | Null | Key | Default | Extra |
+-------+--------------------------+------+-----+---------+-------+
| name  | enum('logical','visible') |      | PRI | logical |       |
| value | int(10) unsigned         |      |     | 0       |       |
+-------+--------------------------+------+-----+---------+-------+
```

Figure 46 Description of the special table `numbers`

```
mysql> select * from numbers;
+---------+-------+
| name    | value |
+---------+-------+
| logical |   354 |
| visible |  1726 |
+---------+-------+
```

Figure 47 Dump of the special table `numbers`

This table dates from the same period as table `logical`, when I was being extremely paranoid about database corruption. Should the `logical` (and `visible`) tables ever be converted to more "databasey" methods, this table could be scrapped.

I was concerned about a little more than database integrity. I was also worried about running out of numbers. On-board file IDs are limited to twenty-two bits (about 4 million). Auto-increment numbers in a database have a nasty habit of "skipping" whole chunks of numbers (or at least, an Oracle database I once worked on had that habit … it's an efficiency dodge to keep a series of auto-increment numbers readily available in memory and works very well unless the database is stopped, at which point all the cached numbers are lost). One of the things that my paranoid technique achieved was that it never lost numbers. With hindsight I don't know why I was so worried. Over a 10-year lifetime, we'd have to chew up over 1000 logical file keys a day before

getting near the 4 million limit (and MySQL database operations have been *extremely* stable … the main FMX database has been up for over 100 days, so it would never have lost any cached auto-increment numbers during that period).

6.0.1 Table **global**

```
mysql> describe global;
+--------+---------------------------------+------+-----+-------------+-------+
| Field  | Type                            | Null | Key | Default     | Extra |
+--------+---------------------------------+------+-----+-------------+-------+
| status | enum('development','production') |      |     | development |       |
+--------+---------------------------------+------+-----+-------------+-------+
```

Figure 48 Description of the special table `global`

```
mysql> select * from global;
+------------+
| status     |
+------------+
| production |
+------------+
```

Figure 49 Dump of the special table `global`

The latest addition to the table menagerie. This is a very simple “singleton” table, provided for people like Joanne who do development work against both the production FMX database and the development FMX database, and need to be able to detect the difference (the “select” in the above example was run from the production FMX database!).

6.1 Database Replication

One of the more difficult goals to achieve with FMX was the desire to “take the LAT on the road” (referring to its travels to the Naval Research Laboratory and then Spectrum Astro). The difficulty is that while on the road, the LAT is under the control of a completely local computing environment (the “mobile rack”), and that there may not always be connectivity from the mobile rack back to SLAC.

The solution adopted has been to replicate the FMX database on the mobile rack. MySQL has built-in facilities to do database replication, even across an unreliable communications medium. All the trickiness however is tied to that word “replication”.

Database replication should not be confused with database mirroring.

Database mirroring is far harder to achieve because the databases involved act as peers. The heart of the problem is what to do when one of the members wants to perform what should be an “atomic” operation (e.g. allocate a new unique number from an auto-incrementing series on a database table that’s shared by all). Solutions often involve a lot of communications between the interested parties along the lines of “Get ready, I am about to …”, “Here I go…”, “OK I’m done…”. There’s also lots of jargon that goes with it, like “two-phase commit”.

The model in database replication is easier. One database is the master and all others are slaves. All changes (inserts, deletes, updates) occur on the master and then replicate down to the slaves, which are, ideally, read only. This system of replication is perfectly adequate for, for instance, replicating a database across many slave machines and then spreading the load of

read requests across the many slaves.

MySQL 4.1 (which is what we're running) provides database replication. It does not provide database mirroring. Database mirroring is advertised as a new feature in MySQL 5.0 and higher.

Given what I'm about to describe, anyone interested in upgrading FMX to MySQL 5.1?

Almost all FMX "file add" activities occur for files produced at SLAC (new code modules, new configuration files, …), so for the logical side of FMX, the replication model with the SLAC database as the master and the mobile rack database as the slave works very well.

Problems start on the physical side.

Each instrument wants to make its own records in the physical side of the database (typically to record things like new file uploads). Unfortunately, the LAT instrument is running against the FMX database on the mobile rack whereas all other instruments are running against the FMX database at SLAC. This breaks the "standard" replication model, requiring simultaneous write access to the physical tables on both the master and slave databases.

Clearly, this has to have been resolved, because we're running with two databases! The major elements of the solution were:

- Bi-directional replication. Both databases act as both master and slave. New records on the SLAC database are replicated on the mobile rack. New records on the mobile rack are replicated at SLAC.
- *Very* careful design of the physical (and hardware topology) tables. All physical tables have a column to capture the *board* for which the row data is being recorded. By the rules of the hardware topology tables, a board can only ever be associated with one instrument. If all the databases work from an agreed table that lists which database is controlling which instrument, and if write access to the physical tables is only permitted on the database "hosting" the instrument, it is in fact impossible for writes on different databases to collide.

This is the explanation for the mysterious `host` column previously (and only vaguely) described in Table 3. It's also the opportunity to describe one of the last undefined tables.

6.1.0 Table **`host`**

```
mysql> describe host;
+--------+------------------+------+-----+---------+-------+
| Field  | Type             | Null | Key | Default | Extra |
+--------+------------------+------+-----+---------+-------+
| host   | int(10) unsigned |      | PRI | 0       |       |
| name   | varchar(128)     |      | UNI |         |       |
| master | enum('Y','N')    |      |     | Y       |       |
| proxy  | varchar(128)     | YES  |     | NULL    |       |
| port   | int(10) unsigned | YES  |     | NULL    |       |
+--------+------------------+------+-----+---------+-------+
```

Figure 50 Description of table `host`

```
mysql> select * from host;
+------+------------------------------+--------+-----------+------+
| host | name                         | master | proxy     | port |
+------+------------------------------+--------+-----------+------+
|    1 | glastlnx06.slac.stanford.edu | Y      | NULL      | NULL |
|    2 | lat-dmz01                    | N      | lat-dmz01 | 8205 |
+------+------------------------------+--------+-----------+------+
```

Figure 51 Dump of table `host`

Field	Description
`host`	Host index.
`name`	Host IP name (most of the time!)
`master`	Flag to indicate if this is the FMX master database (logical side operations are only allowed on the master database).
`proxy`	See notes.
`port`	See notes.

Table 26 Extended description of table `host`

Notes:

- It's never easy. In addition to all the messing about to set up bi-directionally replicating databases, you then have to throw in the fact that the mobile rack has stunted network facilities, and operates through firewalls to talk to SLAC (actually a fully fledged, hardware based VPN).

  The result is that from the mobile rack computers, it's not possible to connect to the SLAC FMX database using the usual `--host=<host> --port=<port>` technique. The connection is made through a pre-positioned `ssh` tunnel on the local (`lat-dmz01`) computer. The extra columns in this table (`proxy` and `port`), allow FMX to identify the communications method that it needs to employ to establish a connection to the master database.

7 The FMX Script

The current FMX script consists of ~6000 lines of Perl, and was written *extremely* fast. To get it out the door all the faster, it plagiarizes anything and everything it could lay its hands on. In some places that worked out well (e.g. it took all its command line parsing from CMX); in other places, not so well (the script is far too linear for my taste, and with care could probably be reduced to half its current size while making it more maintainable).

I shall leave any further critique to a section at the end where I will detail “FMX Future”. For now, let’s worry about what the FMX script does in its current form.

The FMX script is a Unix command line driven tool. It is stored and maintained in the CMX package FMX, but has no compileable constituents. It does however depend on a number of external tools. The most obvious is the ability to access a MySQL database, but it also (depending on command) requires access to commercial tools (e.g. the WindRiver cross-compilation version of the `binutils` suite), and other FSW tools (e.g. the file header manipulation routines in package FILE).

7.0 Command Prerequisites

To access the FMX database via the FMX script, the user needs three things:

- An FMX database account.
  - See next section for how to apply.
- A *protected* `.my.cnf` file to direct the MySQL connection.
  - Protection schemes will also be outlined in the next section.
- An environment variable, `FMX_C_FDB`, pointing to the root of FMX’s “visible file tree”.
  - For people using the standard FSW login at SLAC, this variable is set automatically.
  - For others, the current setting of the environment variable (at SLAC) would be:

```
> setenv FMX_C_FDB /afs/slac/g/glast/fmx
```

7.1 Command Syntax

Command line parsing was ripped off in toto from the CMX script. For a complete description of the command line syntax, please see the description in the CMX manual.

There remains one subtle (or maybe not so subtle) between the `cmx` command and the `fmx` command line. The `cmx` command is invoked via an alias (for a variety of not very interesting technical reasons). On the up-side, this means that the `cmx` command line is not subject to Unix expansion (you can put a `*` in a `cmx` command and Unix will not try to do a file expansion on it). On the downside, `cmx` commands are hard to pipe (the command `cmx show hierarchy > ~/my.file` will not work … you have to spawn the command and capture the output from the spawned command like this: `(cmx show hierarchy) >~/my.file`).

The `fmx` command is not invoked via an alias, so reverse the logic of the comments about `cmx`. `fmx` commands are easy to pipe, but the command line is subject to Unix expansion (so be prepared to use a lot of quotation marks!).

7.2 Command Groups

In general, `fmx` commands map quite directly onto the underlying database tables. The commands will therefore be broken up into subject areas (Enumeration Commands, User Accounts, Hardware Topology Descriptions, …) and placed in separate sections.

8 Enumeration Commands

These are just a set of utility commands to access some of the enumerations in the FMX database. They are most easily demonstrated with simple terminal dumps.

8.0 fmx show devices

`fmx show devices [--[no]title]`

- `[--[no]title]`

  Suppress the column titles. Provided so that an automated tool would find it easier to parse the command output. Default is `--title`.

```
flora05:apw> fmx show devices
Idx  Name  Type        Board
---  ----  ----------  -----
1    /ram  volatile    cpu
2    /ee0  persistent  sib
3    /ee1  persistent  sib
4    /mem  volatile    cpu
5    /mm0  persistent  sib
6    /mm1  persistent  sib
7    /rom  persistent  cpu
```

Figure 52 Output from the `fmx show devices` command

The same thing with the titles suppressed:

```
flora05:apw> fmx show devices --notitle
1  /ram  volatile    cpu
2  /ee0  persistent  sib
3  /ee1  persistent  sib
4  /mem  volatile    cpu
5  /mm0  persistent  sib
6  /mm1  persistent  sib
7  /rom  persistent  cpu
```

Figure 53 Output from the `fmx show devices` command (titles suppressed)

8.1 fmx show filetypes

```
fmx show filetypes [--[no]title]
```

- `[--[no]title]`

  Suppress the column titles. Provided so that an automated tool would find it easier to parse the command output. Default is `--title`.

```
flora05:apw> fmx show filetypes
Idx  Name          Dir   Group     Comment
---  ------------  ----  --------  ----------------------------------------
0    system        sys   <none>    (reserved for system functions)
1    vxw           vxw   module    VxWorks RTOS image
2    relocateable  rel   module    Relocateable code module
3    cdm           cdm   module    CDM format configuration file
4    hsk           hsk   simple    Housekeeping configuration files
5    lci           lci   simple    Charge injection configuration/script file
6    latc          latc  simple    Instrument configuration file
7    fof           fof   compound  File of files (unresolved)
8    sbs           sbs   compound  Secondary boot script (unresolved)
9    sbm           sbm   module    Secondary boot module
10   pbc           pbc   module    Primary boot code
11   ltc           ltc   simple    Thermal control configuration files
12   serial        ser   simple    Serial number (embedded in SUROM)
```

Figure 54 Output from the `fmx show filetypes` command

8.2 fmx show hosts

```
fmx show hosts [--[no]title]
```

- `[--[no]title]`

  Suppress the column titles. Provided so that an automated tool would find it easier to parse the command output. Default is `--title`.

```
flora05:apw> fmx show hosts
Idx  Name                          Master
---  ----------------------------  ------
1    glastlnx06.slac.stanford.edu  Y
2    lat-dmz01                     N
```

Figure 55 Output from the `fmx show hosts` command

8.3 fmx show instruments

```
fmx show instruments [--[no]title]
```

- `[--[no]title]`

  Suppress the column titles. Provided so that an automated tool would find it easier to parse the command output. Default is `--title`.

```
flora05:apw> fmx show instruments
Idx  Name     Towers     HostID  Location      Comment
---  -------  ---------  ------  ------------  ------------------------
1    mordor   real       2       NRL           The LAT instrument
2    gondor   simulated  1       SLAC 84 B101  Test-bed
3    rohan    none       1       NRL
4    arnor    none       1       SLAC 84 B101  Uber-test-stand
5    orthanc  real       1       SLAC 84 B101  Multi-crate (with tower)
6    shire    none       1       SLAC 84 B101  SIU flight spare
```

Figure 56 Output from the `fmx show instruments` command

9 User Account Commands

There currently only four user account management commands. They are distinguished by the fact that they are the only commands in FMX that interact with the user. Three of the commands are privileged (i.e. can only be accessed by the `magician` role account, where the interaction is to interrogate for the `magician` password). The fourth command is the account application, and will prompt the user to create the account password.

The only prerequisites for an account application are:

- That the `fmx` command be accessible from the command line.
- That the session has correctly defined the environment variable `FMX_C_FDB`.

9.0 fmx create user

```
fmx create user <username>
    --name=<name>
    --mail=<mail>
    --logician
    --physician
    --technician
```

- `<username>` MySQL user account name (usually your Unix login ID)
- `--name=<name>` A “trivial” name (something more helpful than `<username>!`).
- `--mail=<mail>` E-mail address
- `--logician` Request write access to `logical` tables
- `--physician` Request write access to `physical` tables
- `--technician` Request write access to `hardware` tables

The command will prompt the user to define (and confirm) a user password, and then repeat back the details of the application. If the details are satisfactory, the user replies `continue`, and the application is registered in the database.

At this point, the database should mail the `magician` account keeper to inform him/her that there’s an account application outstanding. Unfortunately, the FMX script is not that sophisticated yet, so the applicant should find someone with `magician` privileges (currently `apw` and `blee`) to let them know.

9.1 fmx show requests

`fmx show requests`

Prints a list of currently outstanding user account requests. The command will prompt the user for the `magician` password.

9.2 fmx approve

`fmx approve <username>`

Approves the account application by user `<username>`. The command will prompt the user for the `magician` password.

It would be useful for the FMX script to auto-generate a mail message back to the applicant. Unfortunately it doesn't do that yet.

9.3 fmx deny

`fmx deny <username>`

Denies the account application by user `<username>`. The command will prompt the user for the `magician` password.

Once again it would be useful for the FMX script to auto-generate a mail message back to the applicant. Unfortunately it doesn't do that yet.

9.4 Once A User Account Is Approved

Once approved, the applicant will start accessing the FMX database using a private account. To provide the information necessary to complete connections to the database, the user needs to set up a `.my.cnf file`. The contents of the file are very simple:

```
[client]

# You can use this area to define MySQL defaults for other databases

[fmx]

# The FMX script will use definitions in this stanza

database = fmx
user     = <your username>
password = <your password>
```

Figure 57 Example `.my.cnf` file

The contents of the file are simple, but do contain a password, so the file itself must be carefully protected. That can be tricky. MySQL expects to find this file in your home directory. You might be able to protect it in that location on some file systems (make the file read/write to the owner, but unreadable/unwriteable to anyone else), but that doesn't work too well at SLAC where home directories are usually in AFS file space, and AFS provides protection at the *directory* level.

The simplest solution I've come up with was to place the `.my.cnf` file inside the `.ssh` directory in my home directory. The `.ssh` directory is traditionally heavily protected (though it does no harm to further protect the `.my.cnf` file using more traditional file protections). To satisfy

MySQL search criteria, I then created a soft link called `.my.cnf` file in my home directory which pointed to the real copy in the `.ssh` directory.

10 Hardware Commands

This is going to be a very short section. There are no commands in FMX to alter hardware topologies!

The stubs are in place. You will find help for commands like:

- `fmx insert …`
- `fmx remove …`

And an appropriate syntax is not hard to imagine, but the commands are not implemented.

For the moment, the only way to alter hardware topologies is to use the `mysql` tool to connect directly to the database and change the records by hand.

The one command that *is* implemented `fmx describe <instrument>` which prints a dump of

an instrument's topology. This has already been described in section 3.4.

11 Logical Commands

There is basically only one command in FMX that directly affects the logical tables. The command is “add”. It comes in several flavours depending on the file *group* to which the file being added belongs, but within a file group, the syntax is consistent. Only three commands will therefore be described, one for each file group. A placeholder will be put in the command definition with the following allowed substitutions:

Placeholder name	Placeholder for file types…
simple	hsk, lci, latc, ltc, serial
module	vxw, relocateable, cdm, sbm, pbc
compound	fof, sbs

Table 27 File type to file group mappings

11.0 fmx add <simple>

```
fmx     add <simple> <file>
      [ --[no]compress ]
      [ --nickname=<nick> ]
        --path=<path>
      [ --xref=<xref> ]
```

- `<simple>`

  One of the <simple> file types detailed in Table 27.

- `<file>`

  File to be added to FMX.

- `--[no]compress`

  Request file compression (or not). The default is `--compress`.

- `--nickname=<nick>`

  The nickname (eight letters maximum) to put in the file header of the uploadable file. If present, must define a string. If omitted, defaults to the file’s (unqualified) name. If that’s longer than eight letters, the command is rejected.

- `--path=<path>`

  Mandatory qualifier. Path name to a location in the FMX visible file tree where the file being added (and its derived products) should be stored. Intended audience for this qualifier is another tool (e.g. MOOT/MOOD) that wants to use the directory structure in the FMX visible file tree as an organizational principle.

  Note that FMX "owns" the first directory name in the tree (the name corresponding to file type). An example of `--path` usage is given below.

- `--xref=<xref>`

  A user qualifier (FMX makes no use of it). If present, `<xref>` must be an integer value. If omitted, the default is 0xffffffff ($2^{32} - 1$). Intended audience for this qualifier is another tool (e.g. MOOT/MOOD) that needs FMX to store a back reference.

File and path names are a little confusing. An example is more illustrative. The command:

```
flora02:apw> fmx add lci ~/foo.bar --path=my/personal/directory …
```

Figure 58 Example of an `fmx add <simple>` command

Would capture/generate the following two files:

```
$FMX_C_FDB/lci/my/personal/directory/foo.bar
$FMX_C_FDB/lci/my/personal/directory/foo.bar.f
```

Figure 59 Files captured by an `fmx add <simple>` command

11.1 fmx add <module>

```
fmx    add <module> <package> <constituent>
      [ --production | --version=<version> ]
      [ --tags=<tags> ]
```

- `<module>`

  One of the <module> file types detailed in Table 27.

- `<package>`

  CMX package name of file to add to FMX.

- `<constituent>`

  CMX constituent name of file to add to FMX.

- `--production | --version=<version>`

  Specify what version of the package/constituent to add. Mutually exclusive qualifiers. Defaults to `--production`.

- `--tags=<tags>`

  List of tags (in the CMX sense) to import. Defaults to all known tags processable *by the current host*.

Be very careful about that last statement. Constituents targeted at embedded system architectures cannot be fully processed by a Linux host.

Modules added to FMX in this way are written to systematically names directories in the FMX visible file tree. Once again an example is the easiest way to explain:

```
flora02:apw> fmx add relocateable PBS pbs –version=V2-10-9
```

Figure 60 Example of an `fmx add <module>` command

Would capture/generate the following six files:

```
$FMX_C_FDB/rel/PBS/V2-10-9/mcp750/pbs/libpbs.o
$FMX_C_FDB/rel/PBS/V2-10-9/mcp750/pbs/pbs.f
$FMX_C_FDB/rel/PBS/V2-10-9/mv2304/pbs/libpbs.o
$FMX_C_FDB/rel/PBS/V2-10-9/mv2304/pbs/pbs.f
$FMX_C_FDB/rel/PBS/V2-10-9/rad750/pbs/libpbs.o
$FMX_C_FDB/rel/PBS/V2-10-9/rad750/pbs/pbs.f
```

Figure 61 Files captured by an `fmx add <module>` command

In a slight variation, FMX also captures host shareables for CDM modules. The command:

```
flora02:apw> fmx add cdm LCI_DB lci_data –version=V1-0-2
```

Figure 62 Example of an `fmx add cdm` command

Would capture/generate the following eight files:

```
$FMX_C_FDB/cdm/LCI_DB/V1-0-2/linux-gcc/lci_data/liblci_data.so
$FMX_C_FDB/cdm/LCI_DB/V1-0-2/mcp750/lci_data/liblci_data.o
$FMX_C_FDB/cdm/LCI_DB/V1-0-2/mcp750/lci_data/lci_data.f
$FMX_C_FDB/cdm/LCI_DB/V1-0-2/mv2304/lci_data/liblci_data.o
$FMX_C_FDB/cdm/LCI_DB/V1-0-2/mv2304/lci_data/lci_data.f
$FMX_C_FDB/cdm/LCI_DB/V1-0-2/rad750/lci_data/liblci_data.o
$FMX_C_FDB/cdm/LCI_DB/V1-0-2/rad750/lci_data/lci_data.f
$FMX_C_FDB/cdm/LCI_DB/V1-0-2/sun-gcc/lci_data/liblci_data.so
```

Figure 63 Files captured by an `fmx add cdm` command

11.2 fmx add <compound>

```
fmx     add <compound> <file>
      [ --nickname=<nick> ]
        --path=<path>
```

- `<compound>`

  One of the <compound> file types detailed in Table 27.

- `<file>`

  File to be added to FMX.

- `--nickname=<nick>`

  The nickname (eight letters maximum) to put in the file header of the uploadable file. If

present, must define a string. If omitted, defaults to the file's (unqualified) name. If that's longer than eight letters, the command is rejected.

- `--path=<path>`

  Mandatory qualifier. Path name to a location in the FMX visible file tree where the file being added (and its derived products) should be stored. Intended audience of this qualifier is another tool (e.g. MOOT/MOOD) that wants to use the directory structure in the FMX visible file tree as an organizational principle.

  Note that FMX "owns" the first directory name in the tree (the name corresponding to file type). An example of `--path` usage is given below.

File and path names are a little confusing. An example is more illustrative. The command:

```
flora02:apw> fmx add sbs ~/foo.bar --path=FTS/V0-1-7/B0-0-0 …
```

Figure 64 Example of an `fmx add <compound>` command

Would capture/generate the following file:

```
$FMX_C_FDB/sbs/FTS/V0-1-7/B0-0-0/foo.bar
```

Figure 65 Files captured by an `fmx add <compound>` command

11.3 fmx import

This is the grand-daddy of all "add" commands. Don't be fooled by the apparently simple syntax, this is an extremely powerful command.

- `fmx import <build_name> [--[no]dry-run]`
- `<build_name>`

  A named flight software build (a string like `Bnn-nn-nn`).

- `[--[no]dry-run]`

  Controls whether the command actually performs the command or simply reports on what it would do. The default is `--dry-run`.

This command may only be executed from the CMX instance named in the `<build_name>`. The command will check for this before execution. It also checks that it is running on a host that can execute VxWorks utilities (which effectively limits this command to Sun hosts).

This is a multi-step process. The command:

- Looks up the FMX template scripts in the FTS package in the named build. The list of scripts scanned is a global list in the FMX script, but currently consists of:
  - `siu.fmx.template`
  - `epu.fmx.template`
  - `siu.fmx.ddt.burn-siu`
  - `siu.fmx.ddt.burn-epu`
  - `epu.fmx.ddt.burn-epu`

- Scans all these scripts to construct a unique list of code modules required by the scripts (thus a constituent like PBS/pbs, which appears in all the scripts, only appears once in the final list).
- Resolves the version number for each constituent on the list from the named build.
- Interrogates FMX records to see if the members of the list, down to the version number, are already present in FMX. If the module is already present, it's eliminated from the list.
- Executes the appropriate `fmx add ...` commands to add any members of the list still remaining.
- Resolves any `prod` indirections in the original FMX scripts to specific version numbers.
- Executes the appropriate `fmx add ...` commands to add the resolved FMX scripts to the database.

As I said, a very powerful command!

From the description, it should also be clear that this is a very single-minded command. Its sole purpose is to add any *new* code modules in a named build to the FMX database logical side, along with the revised scripts in the named build.

The command is also useful in its reporting mode to investigate just how many new constituents are involved before a build is constructed. The CMX instance named `slac` always represents the most up-to-date (production level) code available. The command:

```
flora02:apw> fmx import slac
```

Figure 66 Example of an `fmx import` command (in report only mode)

prints a report of all the constituents in the `slac` instance which are not already available in the FMX database.

12 Physical Commands

The physical commands use verbs like `upload` and `commit`. It should be made clear from the start that these commands (and others described in this section) do not in fact `upload` or `commit` (or `corrupt` or `delete`) anything. The command `fmx upload ...` should be interpreted as a query to the database along the lines of "if I were to upload file such-and-such to instrument/node/device so-and-so, where can I find the right file in FMX's visible file tree, and what file ID should I give the file on the target embedded file system?". Similarly, `fmx commit ...` is really a request to FMX along the lines of "please record the fact that file such-and-such has just been committed to instrument/node/device so-and-so".

While these commands never touch an embedded file system, they should nevertheless be used with caution. Many of them (e.g. `commit`, `delete`) do create records in the database, which could result in the database having inaccurate records of the state of embedded file systems.

They were originally written to test the "file life-cycle", and I had intended that once I had that working, I would withdraw the commands (you may notice in the response to `fmx help` that these commands are listed as "pure fakery"!). Since then, other tools have interfaced with them, so I can no longer do that.

Note however, that there is some leverage in the user account privileges. Only accounts with privilege `physician` are allowed to write to the physical tables (and then only for instruments controlled by the host they are talking to). Access to `physician` privilege should probably be limited as far as possible!

The physical commands also suffer from the same excess of flavours as the logical commands. If anything, it's even worse because there are now a variety of verbs to map against a variety of file types. To avoid populating each individual member of the matrix, I will continue to the file type placeholders shown in Table 27. I will also add need a "verb" placeholder:

Verb placeholder name	**Placeholder for verbs…**
`verb`	`commit, corrupt, delete`

Table 28 Definition of placeholder <verb>

12.0 Commit, Corrupt, and Delete Commands

12.0.0 fmx <verb> <simple>

```
fmx     <verb> <simple> <file>
        --device=<device>
      [ --epoch=<seconds> | --logged=<timestamp> ]
        --instrument=<instrument> --node=<node> | --ip=<ip>
```

- `<verb>`

  One of `commit`, `corrupt` or `delete`.

- `<simple>`

  One of the <simple> file types detailed in Table 27.

- `<file>`

  Record the `<verb>` of file `<file>` in FMX. `<file>` is expressed in relation to the FMX file tree, with the first directory (the file type directory) omitted. In other words (and assuming a file type of `lci`), expressed relative to `$FMX_C_FDB/lci/`

- `--device=<device>`

  A device name from FMX's standard set (`/mm0`, `/ee1`, …).

- `--epoch=<seconds>`

  Specify a timestamp to associate with the physical log records. `<seconds>` is the "seconds since LAT epoch". Provided as an easy method for a scripting environment to pluck a timestamp out of a piece of telemetry and pass the timestamp to FMX.

  Exclusive with the `--logged` method of qualifying a timestamp.

  Please note the warning at the end of the command description.

- `--logged=<timestamp>`

  Specify a timestamp to associate with the physical log records created. `<timestamp>` must be expressed in MySQL's timestamp format: `'YYYY-MO-DD HH:MM:SS'`

  Exclusive with the `--epoch` method of specifying a timestamp.

  Please note the warning at the end of the command description.

- `--instrument=<instrument> --node=<node>`

  Use the instrument/node method to identify the target CPU. `<instrument>` and `<node>` are keywords defined by the FMX database. `<instrument>` is an open ended list currently consisting of `mordor`, `gondor`, `arnor`, `rohan`, and `orthanc`. `<node>` is a closed list consisting of `siu0`, `siu1`, `epu0`, `epu1`, and `epu2`.

  Exclusive with the IP method of specifying the target.

- `--ip=<ip>`

  Use the IP method to identify the target CPU. `<ip>` can be specified as either the IP name or the IP address of the target CPU. For obvious reasons, this does not work for targets that don't have ethernet cards (like the real LAT instrument).

  Exclusive with the instrument/node method of specifying the target.

Whether specified via `--epoch` or `--logged`, FMX will sanity check the timestamp. Timestamps in the future are rejected, as are any attempts to specify a time that precedes the most recent record for the file question.

If both are omitted, the timestamp defaults to the current wall-clock time (expressed in UTC).

12.0.1 fmx <verb> <module>

```
fmx     <verb> <module> <package> <constituent>
        --device=<device>
      [ --epoch=<seconds>] | --logged=<timestamp> ]
        --instrument=<instrument> --node=<node> | --ip=<ip>
        --version=<version>
```

- `<verb>`

  One of `commit`, `corrupt` or `delete`.

- `<module>`

  One of the <module> file types detailed in Table 27.

- `<package>`

  CMX package name.

- `<constituent>`

  CMX constituent name.

- `--device=<device>`

  A device name from FMX's standard set (`/mm0`, `/ee1`, …).

- `--epoch=<seconds>`

  Specify a timestamp to associate with the physical log records. `<seconds>` is the "seconds since LAT epoch". Provided as an easy method for a scripting environment to pluck a timestamp out of a piece of telemetry and pass the timestamp to FMX.

  Exclusive with the `--logged` method of qualifying a timestamp.

  Please note the warning at the end of the command description.

- `--logged=<timestamp>`

Specify a timestamp to associate with the physical log records created. <timestamp> must be expressed in MySQL's timestamp format: 'YYYY-MO-DD HH:MM:SS'

Exclusive with the --epoch method of specifying a timestamp.

Please note the warning at the end of the command description.

- --instrument=<instrument> --node=<node>

  Use the instrument/node method to identify the target CPU. <instrument> and <node> are keywords defined by the FMX database. <instrument> is an open ended list currently consisting of mordor, gondor, arnor, rohan, and orthanc. <node> is a closed list consisting of siu0, siu1, epu0, epu1, and epu2.

  Exclusive with the IP method of specifying the target.

- --ip=<ip>

  Use the IP method to identify the target CPU. <ip> can be specified as either the IP name or the IP address of the target CPU. For obvious reasons, this does not work for targets that don't have ethernet cards (like the real LAT instrument).

  Exclusive with the instrument/node method of specifying the target.

- --version=<version>

  CMX version number.

Whether specified via --epoch or --logged, FMX will sanity check the timestamp. Timestamps in the future are rejected, as are any attempts to specify a time that precedes the most recent record for the file question.

If both are omitted, the timestamp defaults to the current wall-clock time (expressed in UTC).

12.0.2 fmx <verb> <compound>

```
fmx    <verb> <compound> <file>
       --device=<device>
     [ --epoch=<seconds> | --logged=<timestamp> ]
       --instrument=<instrument> --node=<node> | --ip=<ip>
       --search=<device_list>
```

- <verb>

  One of commit, corrupt or delete.

- <compound>

  One of the <compound> file types detailed in Table 27.

- <file>

  Record the <verb> of file <file> in FMX. <file> is expressed in relation to the FMX

file tree, with the first directory (the file type directory) omitted. In other words (and assuming a file type of `fof`), expressed relative to `$FMX_C_FDB/fof/`

- `--device=<device>`

  A device name from FMX's standard set (`/mm0`, `/ee1`, …).

- `--epoch=<seconds>`

  Specify a timestamp to associate with the physical log records. `<seconds>` is the "seconds since LAT epoch". Provided as an easy method for a scripting environment to pluck a timestamp out of a piece of telemetry and pass the timestamp to FMX.

  Exclusive with the `--logged` method of qualifying a timestamp.

  Please note the warning at the end of the command description.

- `--logged=<timestamp>`

  Specify a timestamp to associate with the physical log records created. `<timestamp>` must be expressed in MySQL's timestamp format: `'YYYY-MO-DD HH:MM:SS'`

  Exclusive with the `--epoch` method of specifying a timestamp.

  Please note the warning at the end of the command description.

- `--instrument=<instrument> --node=<node>`

  Use the instrument/node method to identify the target CPU. `<instrument>` and `<node>` are keywords defined by the FMX database. `<instrument>` is an open ended list currently consisting of `mordor`, `gondor`, `arnor`, `rohan`, and `orthanc`. `<node>` is a closed list consisting of `siu0`, `siu1`, `epu0`, `epu1`, and `epu2`.

  Exclusive with the IP method of specifying the target.

- `--ip=<ip>`

  Use the IP method to identify the target CPU. `<ip>` can be specified as either the IP name or the IP address of the target CPU. For obvious reasons, this does not work for targets that don't have ethernet cards (like the real LAT instrument).

  Exclusive with the instrument/node method of specifying the target.

- `--search=<device_list>`

  A comma separated list of devices from FMX's standard set (`/mm0`, `/ee1`, …). This is the list (and order) of devices used to complete the resolution of file references in the compound file.

Whether specified via `--epoch` or `--logged`, FMX will sanity check the timestamp. Timestamps in the future are rejected, as are any attempts to specify a time that precedes the most recent record for the file question.

If both are omitted, the timestamp defaults to the current wall-clock time (expressed in UTC).

The fact that the device search list has to respecified when using `fmx <verb> <compound>` commands is a historical accident (and certainly very inconvenient). One rather immediate improvement to FMX would be an alternative set of `fmx <verb> <compound>` commands that allowed the use of the resolved logical file key as the parameter.

12.1 Populate And Upload Commands

For “simple” and “module” file types, the upload command is relatively straightforward. The same is not true when the same sort of logic is applied to “compound” file types. As a result the command `fmx upload <compound>` has been split into two parts. The `fmx populate <compound>` command deals with the uploading of the files a compound file refers to, while the `fmx upload <compound>` command deals with uploading the compound file itself.

12.1.0 fmx upload <simple>

```
fmx     upload <simple> <file>
        --device=<device>
      [ --instrument=<instrument> --node=<node> | --ip=<ip> ]
```

- `<simple>`

  One of the <simple> file types detailed in Table 27.

- `<file>`

  Interrogate FMX for how to upload file `<file>`. `<file>` is expressed in relation to the FMX file tree, with the first directory (the file type directory) omitted. In other words (and assuming a file type of `lci`), expressed relative to `$FMX_C_FDB/lci/`

- `--device=<device>`

  A device name from FMX’s standard set (`/mm0`, `/ee1`, …).

- `--instrument=<instrument> --node=<node>`

  Use the instrument/node method to identify the target CPU. `<instrument>` and `<node>` are keywords defined by the FMX database. `<instrument>` is an open ended list currently consisting of `mordor`, `gondor`, `arnor`, `rohan`, and `orthanc`. `<node>` is a closed list consisting of `siu0`, `siu1`, `epu0`, `epu1`, and `epu2`.

  Exclusive with the IP method of specifying the target.

- `--ip=<ip>`

  Use the IP method to identify the target CPU. `<ip>` can be specified as either the IP name or the IP address of the target CPU. For obvious reasons, this does not work for targets that don’t have ethernet cards (like the real LAT instrument).

  Exclusive with the instrument/node method of specifying the target.

If successful, FMX will respond with a line like:

```
FMX: Upload physical=0x41000054 file=hsk/LHK/V6-4-8/LHKCFSCH/LHKCFSCH.f
```

Figure 67 Example output from an `fmx upload <simple>` command

12.1.1 fmx upload <module>

```
fmx     upload <module> <package> <constituent>
        --device=<device>
      [ --instrument=<instrument> --node=<node> | --ip=<ip> ]
        --version=<version>
```

- `<module>`

  One of the <module> file types detailed in Table 27.

- `<package>`

  CMX package name.

- `<constituent>`

  CMX constituent name.

- `--device=<device>`

  A device name from FMX's standard set (`/mm0`, `/ee1`, …).

- `--instrument=<instrument> --node=<node>`

  Use the instrument/node method to identify the target CPU. `<instrument>` and `<node>` are keywords defined by the FMX database. `<instrument>` is an open ended list currently consisting of `mordor`, `gondor`, `arnor`, `rohan`, and `orthanc`. `<node>` is a closed list consisting of `siu0`, `siu1`, `epu0`, `epu1`, and `epu2`.

  Exclusive with the IP method of specifying the target.

- `--ip=<ip>`

  Use the IP method to identify the target CPU. `<ip>` can be specified as either the IP name or the IP address of the target CPU. For obvious reasons, this does not work for targets that don't have ethernet cards (like the real LAT instrument).

  Exclusive with the instrument/node method of specifying the target.

- `--version=<version>`

  CMX version number.

If successful, FMX will respond with a line like:

```
FMX: Upload physical=0x4080011c file=rel/THS/V1-5-0/rad750/ths4ddt/ths4ddt.f
```

Figure 68 Example output from an `fmx upload <module>` command

12.1.2 fmx populate <compound>

```
fmx     populate <compound> <file>
        --device=<device>
      [ --instrument=<instrument> --node=<node> | --ip=<ip> ]
        --search=<device_list>
```

- `<compound>`

  One of the <compound> file types detailed in Table 27.

- `<file>`

  Interrogate FMX for how to upload the files *referred to* by file `<file>`. `<file>` is expressed in relation to the FMX file tree, with the first directory (the file type directory) omitted. In other words (and assuming a file type of `sbs`), expressed relative to `$FMX_C_FDB/sbs/`

- `--device=<device>`

  A device name from FMX's standard set (`/mm0`, `/ee1`, ...).

- `--instrument=<instrument> --node=<node>`

  Use the instrument/node method to identify the target CPU. `<instrument>` and `<node>` are keywords defined by the FMX database. `<instrument>` is an open ended list currently consisting of `mordor`, `gondor`, `arnor`, `rohan`, and `orthanc`. `<node>` is a closed list consisting of `siu0`, `siu1`, `epu0`, `epu1`, and `epu2`.

  Exclusive with the IP method of specifying the target.

- `--ip=<ip>`

  Use the IP method to identify the target CPU. `<ip>` can be specified as either the IP name or the IP address of the target CPU. For obvious reasons, this does not work for targets that don't have ethernet cards (like the real LAT instrument).

  Exclusive with the instrument/node method of specifying the target.

- `--search=<device_list>`

  A comma separated list of devices from FMX's standard set (`/mm0`, `/ee1`, ...). This is the list (and order) of devices used to complete the resolution of file references in the compound file.

If successful, this command will respond with a list of files that must be uploaded to the target CPU before the "compound" file can be successfully resolved. The responses assume that any missing files should be uploaded to the first device listed in `<device_list>`. A typical response might look like:

```
FMX: Upload physical=0x40c0014d file=cdm/LIM_DB/V0-5-0/rad750/lim_data/lim_data.f
FMX: Upload physical=0x40c00076 file=cdm/LSM_DB/V1-0-1/rad750/lsm_data/lsm_data.f
FMX: Upload physical=0x40c00137 file=cdm/LTC_DB/V1-0-2/rad750/ltc_data/ltc_data.f
FMX: Upload physical=0x40c00038 file=cdm/MEM_DB/V0-0-0/rad750/mem_data/mem_data.f
FMX: Upload physical=0x40c00039 file=cdm/MON_DB/V0-0-0/rad750/mon_data/mon_data.f
FMX: Upload physical=0x40c0003a file=cdm/MPP_DB/V0-0-0/rad750/mpp_data/mpp_data.f
FMX: Upload physical=0x40c0003b file=cdm/MSG_DB/V0-0-2/rad750/msg_data/msg_data.f
FMX: Upload physical=0x40c0003c file=cdm/PBC_DB/V0-0-0/rad750/pbc_data/pbc_data.f
FMX: Upload physical=0x40c0003e file=cdm/PIG_DB/V2-0-0/rad750/pig_data/pig_data.f
```

Figure 69 Example output from an `fmx populate <compound>` command

12.1.3 fmx upload <compound>

```
fmx    upload <compound> <file>
       --device=<device>
     [ --[no]create ]
       --instrument=<instrument> --node=<node> | --ip=<ip>
       —-search=<device_list>
```

- `<compound>`

  One of the <compound> file types detailed in Table 27.

- `<file>`

  Interrogate FMX for how to upload file `<file>`. `<file>` is expressed in relation to the FMX file tree, with the first directory (the file type directory) omitted. In other words (and assuming a file type of `sbs`), expressed relative to `$FMX_C_FDB/sbs/`

- `--[no]create`

  Control the creation of a new compound file resolution, if no existing resolution matches. The default is `–-nocreate`.

- `--device=<device>`

  A device name from FMX's standard set (`/mm0`, `/ee1`, …).

- `--instrument=<instrument> --node=<node>`

  Use the instrument/node method to identify the target CPU. `<instrument>` and `<node>` are keywords defined by the FMX database. `<instrument>` is an open ended list currently consisting of `mordor`, `gondor`, `arnor`, `rohan`, and `orthanc`. `<node>` is a closed list consisting of `siu0`, `siu1`, `epu0`, `epu1`, and `epu2`.

  Exclusive with the IP method of specifying the target.

- `--ip=<ip>`

  Use the IP method to identify the target CPU. `<ip>` can be specified as either the IP name or the IP address of the target CPU. For obvious reasons, this does not work for targets that don't have ethernet cards (like the real LAT instrument).

  Exclusive with the instrument/node method of specifying the target.

- `--search=<device_list>`

  A comma separated list of devices from FMX's standard set (`/mm0`, `/ee1`, …). This is the list (and order) of devices used to complete the resolution of file references in the compound file.

If successful, FMX will respond with a line like:

```
FMX: Upload physical=0x00000005 file=sbs/FTS/V0-1-7/B0-6-9/siu.fmx.template.0000286.f
```

Figure 70 Example output from an `fmx upload <compound>` command

12.2 Query Commands

The query commands return a well defined string, giving the on-board identity and status of the queried file. The string has the format (where the state can be one of `committed`, `corrupted`, of `deleted`):

```
FMX: Query physical=0x20800062 state=committed
```

Figure 71 Response to an `fmx query` commands

12.2.0 fmx query <simple>

```
fmx     query <simple> <file>
        --device=<device>
        --instrument=<instrument> --node=<node> | --ip=<ip>
```

- `<simple>`

  One of the <simple> file types detailed in Table 27.

- `<file>`

  Record the `<verb>` of file `<file>` in FMX. `<file>` is expressed in relation to the FMX file tree, with the first directory (the file type directory) omitted. In other words (and assuming a file type of `lci`), expressed relative to `$FMX_C_FDB/lci/`

- `--device=<device>`

  A device name from FMX's standard set (`/mm0`, `/ee1`, …).

- `--instrument=<instrument> --node=<node>`

  Use the instrument/node method to identify the target CPU. `<instrument>` and `<node>` are keywords defined by the FMX database. `<instrument>` is an open ended list currently consisting of `mordor`, `gondor`, `arnor`, `rohan`, and `orthanc`. `<node>` is a closed list consisting of `siu0`, `siu1`, `epu0`, `epu1`, and `epu2`.

  Exclusive with the IP method of specifying the target.

- `--ip=<ip>`

  Use the IP method to identify the target CPU. `<ip>` can be specified as either the IP name or the IP address of the target CPU. For obvious reasons, this does not work for targets that don't have ethernet cards (like the real LAT instrument).

Exclusive with the instrument/node method of specifying the target.

12.2.1 fmx query <module>

```
fmx     query <module> <package> <constituent>
        --device=<device>
        --instrument=<instrument> --node=<node> | --ip=<ip>
        --version=<version>
```

- `<module>`

  One of the <module> file types detailed in Table 27.

- `<package>`

  CMX package name.

- `<constituent>`

  CMX constituent name.

- `--device=<device>`

  A device name from FMX's standard set (`/mm0`, `/ee1`, …).

- `--instrument=<instrument> --node=<node>`

  Use the instrument/node method to identify the target CPU. `<instrument>` and `<node>` are keywords defined by the FMX database. `<instrument>` is an open ended list currently consisting of `mordor`, `gondor`, `arnor`, `rohan`, and `orthanc`. `<node>` is a closed list consisting of `siu0`, `siu1`, `epu0`, `epu1`, and `epu2`.

  Exclusive with the IP method of specifying the target.

- `--ip=<ip>`

  Use the IP method to identify the target CPU. `<ip>` can be specified as either the IP name or the IP address of the target CPU. For obvious reasons, this does not work for targets that don't have ethernet cards (like the real LAT instrument).

  Exclusive with the instrument/node method of specifying the target.

- `--version=<version>`

  CMX version number.

12.2.2 fmx query <compound>

```
fmx    query <compound> <file>
       --device=<device>
       --instrument=<instrument> --node=<node> | --ip=<ip>
       --search=<device_list>
```

- `<compound>`

  One of the <compound> file types detailed in Table 27.

- `<file>`

  Record the `<verb>` of file `<file>` in FMX. `<file>` is expressed in relation to the FMX file tree, with the first directory (the file type directory) omitted. In other words (and assuming a file type of `fof`), expressed relative to `$FMX_C_FDB/fof/`

- `--device=<device>`

  A device name from FMX's standard set (`/mm0`, `/ee1`, …).

- `--instrument=<instrument> --node=<node>`

  Use the instrument/node method to identify the target CPU. `<instrument>` and `<node>` are keywords defined by the FMX database. `<instrument>` is an open ended list currently consisting of `mordor`, `gondor`, `arnor`, `rohan`, and `orthanc`. `<node>` is a closed list consisting of `siu0`, `siu1`, `epu0`, `epu1`, and `epu2`.

  Exclusive with the IP method of specifying the target.

- `--ip=<ip>`

  Use the IP method to identify the target CPU. `<ip>` can be specified as either the IP name or the IP address of the target CPU. For obvious reasons, this does not work for targets that don't have ethernet cards (like the real LAT instrument).

  Exclusive with the instrument/node method of specifying the target.

- `--search=<device_list>`

  A comma separated list of devices from FMX's standard set (`/mm0`, `/ee1`, …). This is the list (and order) of devices used to complete the resolution of file references in the compound file.

The fact that the device search list has to respecified when using `fmx query <compound>` commands is a historical accident (and certainly very inconvenient). One rather immediate improvement to FMX would be an alternative set of `fmx query <compound>` commands that allowed the use of the resolved logical file key as the parameter.

12.2.3 fmx directory

```
fmx    directory <search>
      [--[no]committed]
      [--[no]corrupted]
      [--[no]file]
      [--size]
      [--total=[<sum_list>]]
```

- `<search>`

A search string, much like a parameter to a unix `ls` command. Elements of the seach list can be wild-carded using shell style wild-carding.

- `<sum_list>`

  Command separated list of the keywords `device` and `directory`. File byte and file count totals will be provided for the requested levels of roll-up. If no keywords are supplied, the default is to provide roll-ups at both device and directory level.

- `--[no]committed]`

  Include (or not) committed files in the list. Default is `--committed`.

- `--[no]corrupted`

  Include (or not) corrupted files in the list. Default is `--nocorrupted`.

- `--[no]file`

  List (or not) each individual file line. Useful to suppress a lot of output, when only the roll-ups are required. Qualifier will be rejected if some form of roll-up has not been requested.

- `--size`

  Print file sizes.

- `--total=<sum_list>`

  Print roll-ups.

The command is most easily described by listing a few examples:

```
flora03:apw> fmx directory /mordor/siu1/ee0/cdm
/mordor/siu1/ee0/cdm/f0000115 : cdm/CPU_DB/V0-2-3/rad750/cpu_siu/libcpu_siu.o
/mordor/siu1/ee0/cdm/f0000268 : cdm/CPU_DB/V0-2-4/rad750/cpu_siu/libcpu_siu.o
/mordor/siu1/ee0/cdm/f0000046 : cdm/CTS_DB/V0-0-0/rad750/cts_data/libcts_data.o
/mordor/siu1/ee0/cdm/f0000047 : cdm/FILE_DB/V0-0-2/rad750/file_data/libfile_data.o
/mordor/siu1/ee0/cdm/f0000048 : cdm/LCI_DB/V1-0-2/rad750/lci_data/liblci_data.o
/mordor/siu1/ee0/cdm/f0000116 : cdm/LCM_DB/V0-1-0/rad750/lcm_data/liblcm_data.o
/mordor/siu1/ee0/cdm/f0000050 : cdm/LCS_DB/V0-0-0/rad750/lcs_siu/liblcs_siu.o
/mordor/siu1/ee0/cdm/f0000144 : cdm/LEM_DB/V0-1-3/rad750/lem_data/liblem_data.o
/mordor/siu1/ee0/cdm/f0000210 : cdm/LEM_DB/V0-1-5/rad750/lem_data/liblem_data.o
/mordor/siu1/ee0/cdm/f0000052 : cdm/LHK_DB/V0-0-0/rad750/lhk_data/liblhk_data.o
/mordor/siu1/ee0/cdm/f0000145 : cdm/LIM_DB/V0-3-0/rad750/lim_data/liblim_data.o
/mordor/siu1/ee0/cdm/f0000183 : cdm/LIM_DB/V0-4-0/rad750/lim_data/liblim_data.o
/mordor/siu1/ee0/cdm/f0000218 : cdm/LPA_DB/V2-3-0/rad750/lpa_db_data/liblpa_db_data.o
/mordor/siu1/ee0/cdm/f0000118 : cdm/LSM_DB/V1-0-1/rad750/lsm_data/liblsm_data.o
/mordor/siu1/ee0/cdm/f0000055 : cdm/LTC_DB/V1-0-0/rad750/ltc_data/libltc_data.o
/mordor/siu1/ee0/cdm/f0000221 : cdm/LTC_DB/V1-0-1/rad750/ltc_data/libltc_data.o
/mordor/siu1/ee0/cdm/f0000311 : cdm/LTC_DB/V1-0-2/rad750/ltc_data/libltc_data.o
/mordor/siu1/ee0/cdm/f0000056 : cdm/MEM_DB/V0-0-0/rad750/mem_data/libmem_data.o
/mordor/siu1/ee0/cdm/f0000057 : cdm/MON_DB/V0-0-0/rad750/mon_data/libmon_data.o
/mordor/siu1/ee0/cdm/f0000058 : cdm/MPP_DB/V0-0-0/rad750/mpp_data/libmpp_data.o
/mordor/siu1/ee0/cdm/f0000059 : cdm/MSG_DB/V0-0-2/rad750/msg_data/libmsg_data.o
/mordor/siu1/ee0/cdm/f0000060 : cdm/PBC_DB/V0-0-0/rad750/pbc_data/libpbc_data.o
/mordor/siu1/ee0/cdm/f0000062 : cdm/PIG_DB/V2-0-0/rad750/pig_data/libpig_data.o
```

Figure 72 Output from `fmx directory` command (simple version)

This is about as simple as it gets. The command asks for a directory listing for files on instrument `mordor`, node `siu1`, device `/ee0` and directory `cdm`. Even with such a tight definition directory listing can get pretty big! It's possible to further constrain the query with a file name. At first sight, that doesn't look very useful. You'd have to know in advance the translation between, say, `f0000062` and `cdm/PIG_DB/V2-0-0/rad750/pig_data/libpig_data.o`. But `fmx show directory` treats the fifth "directory level" in the command in a special way. It can be used to wild-card the "real" file name. So to pick out that `pig_data` file:

```
flora03:apw> fmx directory '/mordor/siu1/ee0/cdm/*pig_data*'
/mordor/siu1/ee0/cdm/f0000062 : cdm/PIG_DB/V2-0-0/rad750/pig_data/libpig_data.o
```

Figure 73 Output from `fmx directory` command (wild-carding a file name)

Well, one more thing had to happen. I used wild card symbols on the fmx command line, so I had to surround the string with quotation marks, but otherwise, it did pretty much what I wanted it to.

The wild-carding applies to anything in the file name string, so:

```
flora03:apw> fmx directory /mordor/siu1/ee0/cdm/*CPU_DB*
/mordor/siu1/ee0/cdm/f0000115 : cdm/CPU_DB/V0-2-3/rad750/cpu_siu/libcpu_siu.o
/mordor/siu1/ee0/cdm/f0000268 : cdm/CPU_DB/V0-2-4/rad750/cpu_siu/libcpu_siu.o
```

Figure 74 Output from `fmx directory` command (wild-carding a different part of the file name)

Wild-cards can be used anywhere:

```
flora03:apw> fmx directory '/mordor/siu1/mm*/*/*'
/mordor/siu1/mm0/sbm/f0000259 : sbm/SBC/V1-3-2/rad750/sbc_nominal/libsbc_nominal.o
/mordor/siu1/mm0/sbs/f0000350 : sbs/FTS/V0-1-7/B0-6-9/siu.fmx.tvac
/mordor/siu1/mm0/vxw/f0000090 : vxw/VXW/V6-11-2/rad750/vxw_flight/vxw_flight.o
/mordor/siu1/mm1/sbm/f0000259 : sbm/SBC/V1-3-2/rad750/sbc_nominal/libsbc_nominal.o
/mordor/siu1/mm1/sbs/f0000353 : sbs/FTS/V0-1-7/B0-6-9/siu.fmx.tvac
/mordor/siu1/mm1/vxw/f0000090 : vxw/VXW/V6-11-2/rad750/vxw_flight/vxw_flight.o
```

Figure 75 Output from `fmx directory` command (wild-carding the device name)

You may have noticed that no two instrument names start with the same letter. That wasn't entirely accidental. If I'm issuing the above command, I get even lazier:

```
flora03:apw> fmx directory '/m*/siu1/mm*/*/*'
/mordor/siu1/mm0/sbm/f0000259 : sbm/SBC/V1-3-2/rad750/sbc_nominal/libsbc_nominal.o
/mordor/siu1/mm0/sbs/f0000350 : sbs/FTS/V0-1-7/B0-6-9/siu.fmx.tvac
/mordor/siu1/mm0/vxw/f0000090 : vxw/VXW/V6-11-2/rad750/vxw_flight/vxw_flight.o
/mordor/siu1/mm1/sbm/f0000259 : sbm/SBC/V1-3-2/rad750/sbc_nominal/libsbc_nominal.o
/mordor/siu1/mm1/sbs/f0000353 : sbs/FTS/V0-1-7/B0-6-9/siu.fmx.tvac
/mordor/siu1/mm1/vxw/f0000090 : vxw/VXW/V6-11-2/rad750/vxw_flight/vxw_flight.o
```

Figure 76 Output from `fmx directory` command (wild-carding the instrument name)

Providing less than four "directories" is a short-hand way of getting (a little) hardware topology information:

```
flora03:apw> fmx directory '/arnor/*'
/arnor/epu0/ee0
/arnor/epu0/ee1
/arnor/epu0/mem
/arnor/epu0/mm0
/arnor/epu0/mm1
/arnor/epu0/ram
/arnor/epu0/rom
/arnor/epu1/ee0
/arnor/epu1/ee1
/arnor/epu1/mem
/arnor/epu1/mm0
/arnor/epu1/mm1
/arnor/epu1/ram
/arnor/epu1/rom
/arnor/siu1/ee0
/arnor/siu1/ee1
/arnor/siu1/mem
/arnor/siu1/mm0
/arnor/siu1/mm1
/arnor/siu1/ram
/arnor/siu1/rom
```

Figure 77 Using `fmx directory` to identify an instrument's hardware configuration

From here on in, you're on your own (though I would advise against):

```
fmx directory '/*/*/*/*/*'
```

12.2.4 fmx purge

```
fmx    purge <search>
      [--[no]file]
       --keep=<keep_list>
      [--size]
      [--total=[<sum_list>]]
```

- `<search>`

  A search string, much like a parameter to a unix `ls` command. Elements of the seach list can be wild-carded using shell style wild-carding.

- `<keep_list>`

  A comma separated list of `<build_range>`. A `<build_range>` is either a single build name (e.g. `B0-7-0`), or a pair of build names separated by a colon (e.g. `B0-6-15:B0-7-0`). In the latter case, the first build number must not be "greater than" the second.

- `<sum_list>`

  Command separated list of the keywords `device` and `directory`. File byte and file count totals will be provided for the requested levels of roll-up. If no keywords are supplied, the default is to provide roll-ups at both device and directory level.

- `--[no]file`

  List (or not) each individual file line. Useful to suppress a lot of output, when only the roll-ups are required. Qualifier will be rejected if some form of roll-up has not been requested.

- `--keep=<keep_list>`

  Used to construct a list of builds, and thence a list of files, that must be retained in order to satisfy the build scripts. Files become candidates for purging by not appearing in any of the scripts corresponding to the “keep” builds.

- `--size`

  Print file sizes.

- `--total=<sum_list>`

  Print roll-ups.

This command is modeled on the `fmx directory` command, but has a very different use. The purpose is to identify files on embedded systems that are no longer needed. A needed file is defined to be any file that appears in the `sbs` scripts for a named set of builds.

Note that `sbs` scripts deal exclusively in filetypes `cdm` and `relocateable`. This command cannot help with purging out other filetypes. It may not even help in purging out `cdm` files in that the system is perfectly capable of loading a `cdm` file at run time, in which case the file may not even be mentioned in an `sbs` script.

13 Special Commands

Once again, a place to stash everything that I've missed out so far!

13.0 fmx check physical

```
fmx     check physical <search>
```

- `<search>`

  Resembles an absolute directory specification with exactly three elements. In order, the three elements refer to instrument, node, and device. Elements can be (crudely) wild-carded.

The purpose of `fmx check physical` is to ensure that the current contents of the `physical` table agree with the contents of the `physlog` table. Both these tables contain large numbers of records, so the `<search>` parameter can be used to reduce the scope of the verification.

To run a check over, very specifically, instrument `mordor`, node `siu1`, device `/ee0`:

```
flora03:apw> fmx check physical /mordor/siu1/ee0
```

Figure 78 Using `fmx check physical` for a single device

The database is currently clean, so it's hard to demonstrate the output from this command. If there are discrepancies, the command prints tables of them (e.g. a file classed as deleted in physical, but committed in physlog).

That was a very confined verification. It's much more common to check all the records for a given instrument (note the use of quotations to stop the Unix command eating the wild-cards):

```
flora03:apw> fmx check physical '/mordor/*/*'
```

Figure 79 Using `fmx check physical` for a complete instrument

Then there's the mother of all verifications:

```
flora03:apw> fmx check physical '/*/*/*'
```

Figure 80 Using `fmx check physical` for everything

13.1 fmx check visible

```
fmx     check visible
      [ --[no]dry-run ]
```

- `--dry-run`

  Control whether the command tries to repair (one variety of the) discrepancies it identifies. Default is `--dry-run` (i.e. don't attempt repairs).

The purpose of `fmx check visible` is to reconcile the files held in the database with the files visible on the (local) host file system. These can diverge in a replication scenario when a file is added at one database and replicated across to second. There's no "trigger" to extract the file from the second database and make it visible on the second host's file system.

`fmx check visible` prepares a report listing:

- All files known to the database, but not visible in the (FMX tree) host file system.
- All files found in the (FMX tree) host file system, not known to the database.

Files in the first category are usually the result of the replication problem just described. If the qualifier `--dry-run` is specified (in its negated form: `--nodry-run`), then in addition to reporting category one files, the command will export a copy of the file(s) to the host file system's visible file tree.

Files in the second category are usually "cruft" and are candidates for deletion. No action is taken on them other than to report them (even if `--nodry-run` is specified).

You might notice the situation of a file being listed in the database but unavailable in the visible file tree doesn't come up too often. That's because the FMX script has one more trick up its sleeve. Any time FMX executes a command that returns the name of a file that *should* be in the visible file tree, it will first check that it *is* in the visible file tree. If it's missing, FMX automatically exports it.

13.2 fmx help

Yes there is online help!

Once again this was ripped off from the CMX implementation and works the same way. Start with `fmx help` for a general overview, then refine your search by providing the verb (e.g. `fmx help show`). Most commands need one more token to figure out which precise command you're asking about (e.g. `fmx help show key`). Which just happens to be the next command…

13.3 fmx show key

This is certainly a show command, but it didn't quite seem to belong with the other show commands (which list database enumerations).

```
fmx     show key <key>
      [ --[no]export ]
      [ --[no]import ]
      [ --tags=<tags> ]
```

- `<key>`

  A logical file key.

- `--[no]export`

  List information about the export file(s). Default is `–noexport`.

- `--[no]import`

  List information about the import file. Default is `–-import`.

- `--tags=<tags>`

  If the key corresponds to a file in file group module, limit the returned information to those CMX tags that appear in the command separated list `<tag_list>`. If the key does not correspond to a file in file group module, the `–-tags` qualifier is (silently) ignored.

`fmx show key` translates a logical file key into more useful information:

```
flora03:apw> fmx show key 353 --export
FMX: (Import) logical=0x00000138 type=sbs file=sbs/FTS/V0-1-7/B0-6-9/siu.fmx.tvac
FMX: (Export) logical=0x00000161 type=sbs file=sbs/FTS/V0-1-7/B0-6-9/siu.fmx.tvac.0000353.f
```

Figure 81 Output from the `fmx show key` command

14 Secondary Boot Scripts

FMX defines its own, very restricted, scripting language. Scripts are used to define how a flight software embedded system CPU should proceed through secondary boot to arrive at applications mode. For this reason they are referred to as secondary boot scripts (`sbs`). The process for using an FMX secondary boot script for use in an embedded system looks something like:

- Edit the plain ASCII FMX secondary boot script.
- Process the ASCII script through a tool to produce a compact binary form.
- Upload the compact binary form to a CPU in primary boot.
    - To a designated temporary area in processor memory for volatile operation.
    - To a designed location in EEPROM for persistent operation.
- Instruct primary boot to go to secondary boot using a special secondary boot module (`SBC_nominal`) that knows how to read and interpret the compact binary FMX script, and pointing to the script file that `SBC_nominal` should interpret.

Apart from this very direct role in secondary boot processing, FMX scripts can also be used as part of the development process. Using a variation on the ASCII to compact binary tool just described, the same secondary boot script can be translated into a VxWorks script. The VxWorks script can be used in either the Tornado shell or the built-in shell. Unfortunately those two shell environments are not completely equivalent, so the tool that translates from FMX script to VxWorks script has to have some external information driven into it, and have some smarts besides!

Both these tools are provided by FMX.

14.0 FMX Secondary Boot Script Structure

The simplest way to describe an FMX secondary boot script is to simply list one, then describe it:

```
#
# --- Configuration database handling and CPU database server
#
load lfs:CDM/prod/cdm
load lfs:CPU_DB/prod/cpu_db_server
load lfs:LEM_DB/prod/lem_db_server
#load cmx:CMX/prod/cmx_asBuiltSpy         # debug
#
# --- Operating system extensions, board support and basic services
#
load lfs:RAD750/prod/rad750_reboot
load lfs:PBS/prod/pbs
load lfs:MSG/prod/msg_mt
load lfs:MSG/prod/msg_print
load lfs:IMM/prod/imm
load lfs:CCSDS/prod/ccsds_pkt
load lfs:ZLIB/prod/zlib_compress
#
# --- Communications
#
load lfs:ITC/prod/itc
#load cmx:ITC/prod/itc_dump               # debug
load lfs:CTDB/prod/sumt_rt_sib           (ctdb == sib)
load lfs:CTDB/prod/sumt_rt_pmc1553       (ctdb == pmc)
load lfs:CTS/prod/cts_lcp_sumt
load lfs:LCBD/prod/lcbd
#load cmx:LCBT/prod/lcbt                  # debug
load lfs:LCS/prod/lcs
#
# --- File system
#
load lfs:FILE/prod/file_upl
load lfs:FILE/prod/file_lcp
#
# --- Instrument configuration utilities
#
load lfs:LEM/prod/lem
load lfs:LEM/prod/lem_lists
#load cmx:LEM/prod/lem_cli                # debug
load lfs:PIG/prod/pig_lcb_init
load lfs:PIG/prod/pig_power
load lfs:PIG/prod/pig_flying
#
# --- Non-instrument-physics applications and support libraries
#
load lfs:MON/prod/mon
load lfs:MEM/prod/mem_base
load lfs:MEM/prod/mem
load lfs:PBC/prod/pbc
load lfs:ATT/prod/att
load lfs:THS/prod/ths
load lfs:LCM/prod/lcm
#load cmx:LCM/prod/lcm_lcp                # debug
#load cmx:LSM/prod/lsm_dump               # debug
load lfs:LSM/prod/lsm
load lfs:LFS/prod/lfs_lcp
```

```
load lfs:LMC/prod/lmc
load lfs:LTC/prod/ltc
load lfs:LHK/prod/lhk_cfg
load lfs:LHK/prod/lhk
#
# --- Instrument physics applications and their support libraries
#
load lfs:LATC/prod/latc_cmn
load lfs:LATC/prod/latc
load lfs:EDS/prod/eds
load lfs:EDS/prod/ebfio
#load cmx:EDS/prod/edsprint               # debug
load lfs:LSE/prod/lsew
load lfs:LDT/prod/encdec
load lfs:LSEC/prod/lsec
load lfs:LCI/prod/lci
load lfs:LPA/prod/lpa_siu
load lfs:LIM/prod/lim
#
# --- CDM configuration databases
#
load lfs:CPU_DB/prod/cpu_siu
load lfs:CTS_DB/prod/cts_data
load lfs:FILE_DB/prod/file_data
load lfs:LCI_DB/prod/lci_data
load lfs:LCM_DB/prod/lcm_data
load lfs:LCS_DB/prod/lcs_siu
load lfs:LEM_DB/prod/lem_data            (towers == real)
load lfs:LEM_DB/prod/lem_data_fes        (towers != real)
load lfs:LHK_DB/prod/lhk_data
load lfs:LIM_DB/prod/lim_data
load lfs:LSM_DB/prod/lsm_data
load lfs:LTC_DB/prod/ltc_data
load lfs:MEM_DB/prod/mem_data
load lfs:MON_DB/prod/mon_data
load lfs:MPP_DB/prod/mpp_data
load lfs:MSG_DB/prod/msg_data
load lfs:PBC_DB/prod/pbc_data
load lfs:PBS_DB/prod/pbs_db_default
load lfs:PIG_DB/prod/pig_data
#
# --- Startup
#
call PBS_configure()
call MSG_configure()
call FPM_initialize()
call RBM_initialize()
call ITC_initialize()
call CTS_initialize()
call LCS_initialize()
call LATC_initialise()
call LCI_initialise()
call CTS_configure()
call LIM_init_with_db()
call LIM_capture_cal_with_db()
call LCS_configure()
```

```
call MON_initialize()
call MON_start_with_db()
call LCM_initialize()
call MEM_init()
call PBC_init()
call LCM_start()
call LSM_initialize()
call LSM_start()
call FILE_initialize()
call LFS_initialize()
call FILE_start()
call LMC_initialize()
call LMC_start()
call LTC_initialize()
call LTC_StartTask()
call LHK_init()
call LHK_start()
call LCI_startTask()
call LPA_siu_init()
call LIM_capture_physics_with_db()
call LIM_start_with_db()
call LPA_siu_start()
```

Figure 82 A full length FMX secondary boot script

14.0.0 General Structure

An FMX secondary boot script consists of a list of software modules to load, followed by a list of routine calls to make. Load and call lines cannot be interleaved. Later sections will dissect the “load” and “call” syntax.

For clarity and documentation purposes, blank lines are allowed, and comments can be introduced “unix fashion” (i.e. all characters from a # to end of line are ignored).

Sorry, the syntax does not support line extension to the next line.

14.0.1 Load Syntax

```
load lfs:LMC/prod/lmc
```

Figure 83 Instance of a `load` line in an FMX secondary boot script

The line above was extracted from the full example. The syntax of the line can be “abstracted” as follows:

```
load <source>:<package>/<version>/<constituent>
```

Figure 84 Abstraction of a `load` of a line in an FMX secondary boot script

`load` is a keyword and should be self-explanatory.

The `<package>`, `<version>` and `<constituent>` elements should be familiar to any FSW developer, but there are a couple of wrinkles:

- When converting to a VxWorks script, the constraints on the `<version>` element are very loose. They are the (CMX) conventional `test`, `dev`, `prod`, `Vx-y-z` branch choices.
- When converting to compact binary format for upload, no such ambiguities are allowed. The `<version>` element must state a specific version.

Ah yes. <source>. Now the fun begins.

<source> must one the three keywords cmx, fmx, or lfs. In general terms:

- cmx Resolve this file to a file in CMX.
- fmx Resolve this file to a file stored in FMX.
- lfs Resolve this file to a file stored in the on-board file system.

But that's only the start of the story.

The FMX command that requests the translation from an sbs script to a VxWorks script must specify two things:

- The CPU target for which the VxWorks script is destined.
- The VxWorks shell mode (Tornado or embedded).

Based on those two pieces of information, the translation process will get into some pretty heavy script manipulation:

- Knowing the target, the FMX script knows how to identify the target's architecture, and can thus substitute in the correct mv2304, mcp750, or rad750 string into the directory path for the file being resolved via cmx or fmx.
- If the user is electing to work at the Tornado shell, all modules must be pushed from the Tornado environment (do *not* try to mix loading files from Tornado and by some other means … you will confuse yourself thoroughly). When the user selects the Tornado shell, files are *always* resolved cmx.
- If the target is an mv2304 crate, then there can *never* be a SIB board from which to source files. When the target CPU is an mv2304, all lfs resolutions are converted to fmx resolutions.
- If logical indirections like test, or dev are used to specify the version of a code module, the user must have a viable CMX environment through which to resolve the indirection. Note that test and dev modules are never loaded into FMX, so all test and dev modules are resolved via cmx.

Provided the above restrictions are recognized, the load directives are otherwise independent. It is perfectly legal to mix cmx, fmx, and lfs resolutions in any fashion. In a session being run from the embedded shell (using a VxWorks image like vxw_symbol) and on a SIB equipped target, it is quite possible to load code from three different sources.

14.0.2 Call Syntax

```
    call LCI_initialise()
```

Figure 85 Instance of a call line in an FMX secondary boot script

The line above was extracted from the full example. The syntax of the line can be "abstracted" as follows:

```
    call <routine> [(] [prm0 [,prm1 [prm2, …]]] [)]
```

Figure 86 Abstraction of a call line in an FMX secondary boot script

call is a keyword and should be self-explanatory.

<routine> is the name of the routine to call.

Parameters to the routine can be listed as a comma separated list after the routine name. Enclosing the parameters in parentheses is optional.

- When converting to a VxWorks script, the constraints on the parameters are very loose. If it's legal at the VxWorks shell prompt, it's legal in an FMX script.
- When converting to compact binary format for upload, there can be at most four parameters, and each parameter must be an integer number (expressed in the normal decimal or hex notation). Any unspecified trailing parameters will be set to zero.

And that's about it!

14.0.3 Conditionals

In this latest incarnation, FMX scripts are allowed a very limited amount of conditional behaviour. The secondary boot script at the start of this section contained two examples:

```
    load lfs:CTDB/prod/sumt_rt_sib          (ctdb == sib)
    load lfs:CTDB/prod/sumt_rt_pmc1553      (ctdb == pmc)
```

Figure 87 The conditional `ctdb` in an FMX secondary boot script

and

```
    load lfs:LEM_DB/prod/lem_data           (towers == real)
    load lfs:LEM_DB/prod/lem_data_fes       (towers != real)
```

Figure 88 The conditional `towers` in an FMX secondary boot script

Most developers have probably stumbled into loading the wrong version of the CTDB summit driver when swapping from an `mv2304` test environment to a `rad750` test environment, so the motivation for providing this functionality should be clear!

So far, only two "conditions" are recognized:

- The FMX database includes in its hardware description section whether a crate is capable of 1553 communications via a PMC 1553 board or a SIB 1553 board (or not capable of 1553 communications at all). The keyword for the property is `ctdb` and the value of the keyword is constrained to one of `pmc`, `sib`, or `none`.
- The FMX database includes in its hardware description section whether an instrument is equipped with real or simulated towers (or without towers completely). The keyword for this property is `towers` and the value of the keyword is constrained to one of `real`, `simulated`, or `none`.

Otherwise, the syntax of a conditional is based on the C language (where the language "operators" (like "==″) do not overlap with symbol names).

14.1 FMX Boot Script Examples

14.1.0 FMX Commands Applicable To FMX Boot Scripts

```
fmx   [ connect | list ] [ serial | tornado | xyplex ] <script>
      [ --instrument=<instrument> --node=<node> | --ip=<ip> ]
      [ --[no]interactive ]
      [ --[no]mount=<devices> ]
```

```
[ --port=<port> ]
[ --search=<devices> ]
```

- `[ connect | list ]`

  In both cases, the command starts by translating the specified FMX script into a VxWorks script.

  If the verb is `list`, the command simply dumps the translated VxWorks script at the terminal (or you could pipe it to a file).

  If the verb is `connect`, the command goes on to connect to the target CPU (using the method defined in the next parameter) and runs the VxWorks script.

- `[ serial | tornado | xyplex ]`

  Specify the connection method to the target CPU. As described earlier, the connection method also has great impact on how the FMX script is translated.

Automating a Tornado connection can get very hairy. Does the command have to start a target server or does one already exist? At command completion, should the command tear down the target server? If it should, can it trace the target server (VxWorks does some weird and wonderful things with target servers, particularly in cases where the embedded system has gone casters up).

Expect surprises and you won't be disappointed!

We are no longer running any serial line connections at SLAC, so the `serial` connection method has not been truly tested.

- `--instrument=<instrument> --node=<node>`

  Use the instrument/node method to identify the target CPU. `<instrument>` and `<node>` are keywords defined by the FMX database. `<instrument>` is an open ended list currently consisting of `mordor`, `gondor`, `arnor`, `rohan`, and `orthanc`. `<node>` is a closed list consisting of `siu0`, `siu1`, `epu0`, `epu1`, and `epu2`.

  Exclusive with the IP method of specifying the target.

- `--ip=<ip>`

  Use the IP method to identify the target CPU. `<ip>` can be specified as either the IP name or the IP address of the target CPU. For obvious reasons, this does not work for targets that don't have ethernet cards (like the real LAT instrument).

  Exclusive with the instrument/node method of specifying the target.

- `--[no]interactive`

  Only valid with the verb `connect`. The default is `--interactive`.

Determines whether the script remains in interactive mode with the target shell after executing the VxWorks script.

- `--[no]mount=<devices>`

  Controls whether the script should start by mounting devices. By default the script will mount the maximal set of devices valid for the given target (it is not possible, for instance, to mount devices `/ee0` and `/ee1` on an `mv2304` target). Valid members of `<devices>` are `/ram`, `/ee0`, and `/ee1`.

- `--port=<port>`

  Only valid for the `serial` connection method.

  Specify which host computer serial port should be used to connect to the target.

- `--search=<devices>`

  Not meaningful for the `tornado` connection method.

  When resolving `lfs` files, control the search order for the on-board devices. Valid members of `<devices>` are `/ee0` and `/ee1`. Single device definitions are allowed. The default is `--search=/ee0,/ee1`.

14.1.1 FMX Boot Script Dumps

The following examples are based on the following FMX shard script:

```
#
# --- Loads
#
load lfs:CDM/prod/cdm
load fmx:CPU_DB/prod/cpu_db_server
load cmx:RAD750/prod/rad750_reboot
#
# --- Calls
#
call PBS_configure()
call MSG_configure()
call FPM_initialize()
call RBM_initialize()
```

Figure 89 Base file for the examples of translations of FMX secondary boot scripts

```
     flora01:apw> fmx list xyplex siu.fmx.example --instrument=gondor --node=siu1
     FILE_sysRamCreate( 0, 0x100000 )
     FILE_sysTffsMount( 2 )
     FILE_sysTffsMount( 3 )
     FILE_loadModuleByID 0x40800001
     FILE_loadModuleByName "/afs/slac/g/glast/fmx/rel/CPU_DB/V0-2-
3/rad750/cpu_db_server/cpu_db_server.f"
     ld 0,1,"/afs/slac/g/glast/flight/OS/binary/RAD750/V1-3-4/rad750/rad750_reboot/librad750_reboot.o"
     PBS_configure(  )
     MSG_configure(  )
     FPM_initialize(  )
     RBM_initialize(  )
     flora01:apw>
```

Figure 90 Base FMX secondary boot script translated for `xyplex`

This demonstrates the most varied result. Instrument `gondor` is the test-bed, so it's `rad750` based and has many flight files already loaded, allowing `lfs` resolutions. Nevertheless, the script author has elected to pull the second module from FMX and the third module from CMX. Hence three different sources for the three files.

```
     flora01:apw> fmx list tornado siu.fmx.example --instrument=gondor --node=siu1
     FILE_sysRamCreate( 0, 0x100000 )
     FILE_sysTffsMount( 2 )
     FILE_sysTffsMount( 3 )
     ld 0,1,"/afs/slac/g/glast/flight/CDM/binary/CDM/V0-2-2/rad750/cdm/libcdm.o"
     ld 0,1,"/afs/slac/g/glast/flight/CDM/binary/CPU_DB/V0-2-3/rad750/cpu_db_server/libcpu_db_server.o"
     ld 0,1,"/afs/slac/g/glast/flight/OS/binary/RAD750/V1-3-4/rad750/rad750_reboot/librad750_reboot.o"
     PBS_configure(  )
     MSG_configure(  )
     FPM_initialize(  )
     RBM_initialize(  )
     flora01:apw>
```

Figure 91 Base FMX secondary boot script translated for `tornado`

This is the same script, but this time it's been translated for a Tornado shell. All files must be pushed.

```
     flora01:apw> fmx list xyplex siu.fmx.example --ip=lat-elf10
     FILE_sysRamCreate( 0, 0x100000 )
     FILE_loadModuleByName "/afs/slac/g/glast/fmx/rel/CDM/V0-2-2/mv2304/cdm/cdm.f"
     FILE_loadModuleByName "/afs/slac/g/glast/fmx/rel/CPU_DB/V0-2-
3/mv2304/cpu_db_server/cpu_db_server.f"
     ld 0,1,"/afs/slac/g/glast/flight/OS/binary/RAD750/V1-3-4/mv2304/rad750_reboot/librad750_reboot.o"
     PBS_configure(  )
     MSG_configure(  )
     FPM_initialize(  )
     RBM_initialize(  )
     flora01:apw>
```

Figure 92 Base FMX secondary boot script translated using the `--ip` method to specify a target CPU

This translation has used the `--ip` method of target identification. IP name `lat-elf10` is (currently) attached to an `mv2304` COTS crate. Knowing that an `mv2304` can never have a SIB, FMX has not tried to mount the EEPROM partitions, and has changed the first module from an `lfs` resolution to an `fmx` resolution.

At this point the combinatorics get pretty wild!

14.2 Building Code Blobs With An FMX Boot Script

In version V3-2-0 of FMX, the syntax of a boot script has been extended. The extension supports the loading of code modules through RAM upload during primary boot, and the later accessing of those code modules during secondary boot. None of our COTS crates support this type of uploading during primary boot, so application of this technique only makes sense if the target is a `rad750`.

14.2.0 Script Syntax Extensions For Code Blobs

This has been accomplished by adding a new `<source>` type (called `mem`) in FMX `load` lines. Note that this `<source>` type can only be intermixed with the `lfs` load type. Any attempt to intermix `mem` with `cmx` or `fmx` load types will be rejected by the parser.

This goal of this command is to allow the construction of temporary code blobs that can be run in a “standard” flight environment (i.e. using the `vxw_flight` operating system). This is clearly a “developer only” technique, useful to load “test” and “development” code into environments (like the real instrument), that cannot support other operating systems. The `mem` type therefore follows the syntax of the `cmx` type, and allows the specification of code modules in `--test` and `--dev` (which in turn means that running this command requires a fully fledged CMX environment to resolve `--test` and `--dev` references).

The following is a valid boot script using the `mem` type:

```
#
# --- Loads
#
load mem:CDM/dev/cdm
load lfs:CPU_DB/prod/cpu_db_server
load mem:RAD750/test/rad750_reboot
#
# --- Calls
#
call PBS_configure()
call MSG_configure()
call FPM_initialize()
call RBM_initialize()
```

Figure 93 Example of a boot script to prepare primary boot uploadable code.

14.2.1 Command Extensions For Code Blobs

A new command has been introduced to read the variation on an `sbs` script and turn it into useful products.

14.2.1.0 fmx create binary

```
fmx    create binary <script>
       --code=<code_file>
       --instrument=<instrument> --node=<node> | --ip=<ip>
       --sbs=<sbs_file>
       --search=<device_list>
```

- `<script>`

  Name of the `sbs` script file to process.

- `<code_file>`

  Name of the file into which to place the code blob.

- `--instrument=<instrument> --node=<node>`

  Use the instrument/node method to identify the target CPU. `<instrument>` and `<node>` are keywords defined by the FMX database. `<instrument>` is an open ended list currently consisting of `mordor`, `gondor`, `arnor`, `rohan`, and `orthanc`. `<node>` is a closed list consisting of `siu0`, `siu1`, `epu0`, `epu1`, and `epu2`.

  Exclusive with the IP method of specifying the target.

- `--ip=<ip>`

  Use the IP method to identify the target CPU. `<ip>` can be specified as either the IP name or the IP address of the target CPU. For obvious reasons, this does not work for targets that don't have ethernet cards (like the real LAT instrument).

  Exclusive with the instrument/node method of specifying the target.

- `--search=<device_list>`

  A comma separated list of devices from FMX's standard set (`/mm0`, `/ee1`, …). This is the list (and order) of devices used to complete the resolution of file references in the compound file.

14.2.2 Command Processing For Code Blobs

Processing an `sbs` script containing `mem` directives performs all the same activities normally associated with converting an `sbs` script into compact binary form ready for upload, with the following additions when a `mem` directive is found.

- Perform a CMX resolution of the code module (and bail out if unsuccessful!).
- Perform the file processing steps normally associated with `fmx adding` a code module.
  - Strip unnecessary sections.
  - Compress.
  - Add file header.
  - Save to one side.

- In the binary sbs script, place a “special” file identifier to state that this file should be loaded from RAM. The file identifier also contains a sequence number indicating which file in the sequence of files defined by mem directives to load.

Once `sbs` script file processing is complete, the command constructs a single file by concatenating the code modules put to one side, prefixing the result with a small “directory structure” so that the code modules can be found individually, and finally, prefixing the whole lot with yet another file header.

14.2.3 Using The Code Blob Products

The `fmx create binary` command thus creates a *matched pair* of output files. The “code” file contains all the code modules designated by the `mem` directives in the `sbs` script. The “sbs” file is exactly matched to the code blob, resolving all `lfs` type directives from files already on-board in the TFFS file system, and all `mem` type directives from the code blob.

To use these files, the `rad750` must be in primary boot. In this state, the code blob should be loaded to the RAM slot usually reserved for the secondary boot module (`/mem/d001/f0000000` in FMX parlance, or `/boot/d000/f0000001` in primary boot parlance). The `sbs` file should be loaded to the RAM slot reserved for a secondary boot script (`/mem/d002/f0000000` in FMX parlance or `/boot/d000/f0000002` in primary boot parlance). The `rad750` should then be requested to perform secondary boot, using whatever operating system you choose (though it should be an “autobooting” operating system like `vxw_flight`), a “regular” secondary boot module *from EEPROM space* (i.e. `/mm0/d001/f0000000` or `/mm1/d001/f0000000`), and the secondary boot script just uploaded.

One final detail. So that the secondary boot module knows that it has an extra resource in which to look up files, instruct the secondary boot to, in effect, mount `/mem/sbm`. The bit to do this is immediately adjacent to the bits that control the mounting of the `/ee0` and `/ee1` TFFS file-systems (bit 18 of the 32 bit secondary boot parameter, where the MSB is bit 0).

15 File-Of-Files

A “file-of-files” is special file type used to list file indirections. The syntax for a file-of-files is pretty simple, but a file-of-files is a compound object in the same sense that a secondary boot script is a compound object. This means that a file-of-files is subject to all the deferred, and target specific, resolution processing that’s applied to a secondary boot script.

15.0 File-Of-Files Syntax

The following is a (slight variation on) a file that was used in file-of-files testing:

```
# Blank lines (below) and lines starting with # are ignored

latc/apw/latc.a
nul/apw/latc.b        # whitespace is allowed before an inline comment ...
latc/apw/latc.c# ... but that’s not mandatory
    latc/apw/latc.d   # whitespace is allowed before the filename
latc/apw/latc.a       # a file can appear more than once
nul                   # minimum specification for a null file
```

Figure 94 Example of a “file-of-files”

Notes:

- The basic syntax is simply an ASCII file.
- Standard Unix-style commentary is allowed (with the exception that a line cannot be extended).
- Files are specified relative to the environment variable `$FMX_C_FDB`.
- Whitespace is allowed between the beginning of the line and the file name.
- Processing preserves the order of the files listed.
- The same file can appear more than once.
- The special filetype `nul` (which is *not* a recognized FMX filetype) indicates that a physical file identifier of zero should be placed in the resolved output file. A file-like string can follow the `nul` filetype, but it’s not mandatory (both styles of specifying a `nul` file appear in the example).

15.1 File-Of-Files Restrictions

The file-of-files filetype was originally envisioned for use with LATC configuration files. In a fit of non-generality, the current implementation places restrictions of the filetypes of files that can be referred to in a file-of-files. There's no particularly good reason for the restriction and it could be lifted if anyone comes up with a reason for doing so.

The following filetypes are not allowed in a file-of-files:

- Any type of code module (filetypes `vxw`, `relocateable`, `cdm`, `sbm`, and `pbc`).
- Secondary boot scripts (filetype `sbs`).

Interestingly, a file-of-files (I must stop typing all that … a `fof` file) *is* allowed to refer to another `fof` file, thus forming `fof` trees (and yes, `fof` file processing will reject trees that recurse!)

16 External Tools Required

17 FMX Future