Linux Fpga Documentation

The kernel development community

Jun 10, 2024

CONTENTS

CHAPTER

ONE

FPGA DEVICE FEATURE LIST (DFL) FRAMEWORK OVERVIEW

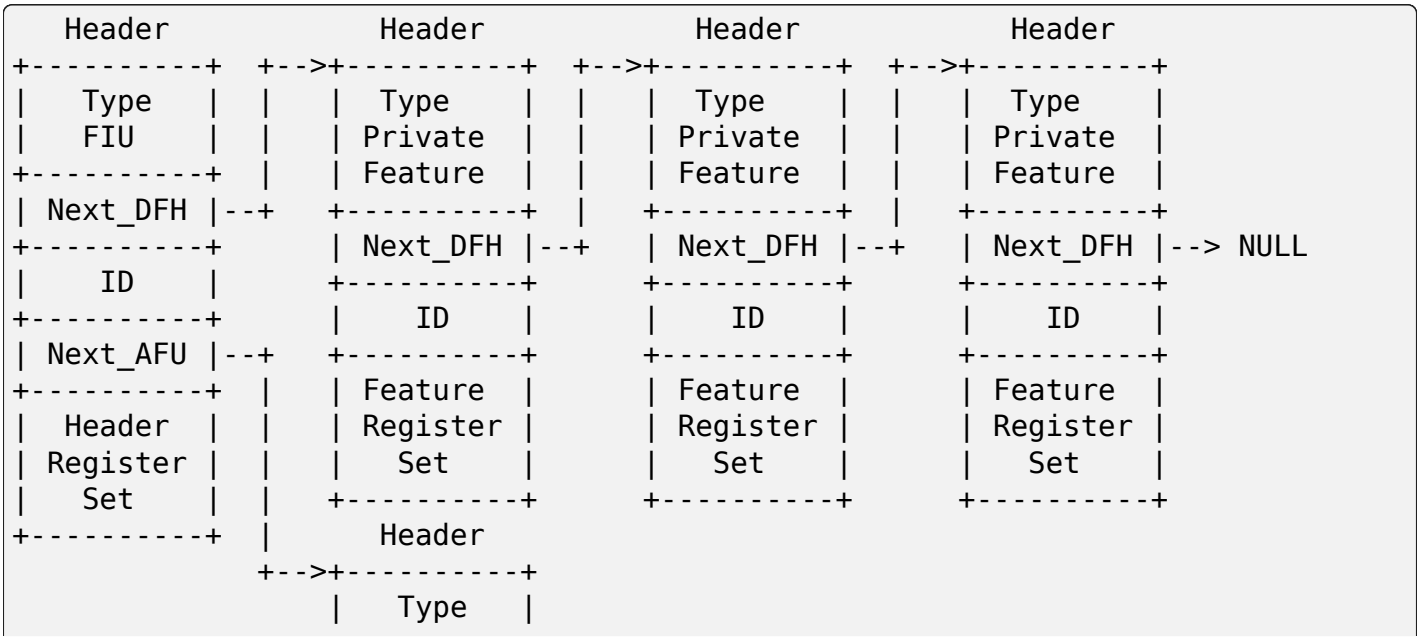
Authors:

- Enno Luebbers <enno.luebbers@intel.com>
- Xiao Guangrong <guangrong.xiao@linux.intel.com>
- Wu Hao <hao.wu@intel.com>
- Xu Yilun <yilun.xu@intel.com>

The Device Feature List (DFL) FPGA framework (and drivers according to this framework) hides the very details of low layer hardware and provides unified interfaces to userspace. Applications could use these interfaces to configure, enumerate, open and access FPGA accelerators on platforms which implement the DFL in the device memory. Besides this, the DFL framework enables system level management functions such as FPGA reconfiguration.

1.1 Device Feature List (DFL) Overview

Device Feature List (DFL) defines a linked list of feature headers within the device MMIO space to provide an extensible way of adding features. Software can walk through these predefined data structures to enumerate FPGA features: FPGA Interface Unit (FIU), Accelerated Function Unit (AFU) and Private Features, as illustrated below:



```
                  |   AFU    |
                  +----------+
                  | Next_DFH |--> NULL
                  +----------+
                  |   GUID   |
                  +----------+
                  |  Header  |
                  | Register |
                  |   Set    |
                  +----------+
```

FPGA Interface Unit (FIU) represents a standalone functional unit for the interface to FPGA, e.g. the FPGA Management Engine (FME) and Port (more descriptions on FME and Port in later sections).

Accelerated Function Unit (AFU) represents an FPGA programmable region and always connects to a FIU (e.g. a Port) as its child as illustrated above.

Private Features represent sub features of the FIU and AFU. They could be various function blocks with different IDs, but all private features which belong to the same FIU or AFU, must be linked to one list via the Next Device Feature Header (Next_DFH) pointer.

Each FIU, AFU and Private Feature could implement its own functional registers. The functional register set for FIU and AFU, is named as Header Register Set, e.g. FME Header Register Set, and the one for Private Feature, is named as Feature Register Set, e.g. FME Partial Reconfiguration Feature Register Set.

This Device Feature List provides a way of linking features together, it's convenient for software to locate each feature by walking through this list, and can be implemented in register regions of any FPGA device.

1.2 Device Feature Header - Version 0

Version 0 (DFHv0) is the original version of the Device Feature Header. All multi-byte quantities in DFHv0 are little-endian. The format of DFHv0 is shown below:

```
+---------------------------------------------------------------------+
|63 Type 60|59 DFH VER 52|51 Rsvd 41|40 EOL|39 Next 16|15 REV 12|11 ID 0| 0x00
+---------------------------------------------------------------------+
|63                                GUID_L                           0| 0x08
+---------------------------------------------------------------------+
|63                                GUID_H                           0| 0x10
+---------------------------------------------------------------------+
```

- Offset 0x00
  - Type - The type of DFH (e.g. FME, AFU, or private feature).
  - DFH VER - The version of the DFH.
  - Rsvd - Currently unused.
  - EOL - Set if the DFH is the end of the Device Feature List (DFL).

  - Next - The offset in bytes of the next DFH in the DFL from the DFH start, and the start of a DFH must be aligned to an 8 byte boundary. If EOL is set, Next is the size of MMIO of the last feature in the list.
  - REV - The revision of the feature associated with this header.
  - ID - The feature ID if Type is private feature.

- Offset 0x08
  - GUID_L - Least significant 64 bits of a 128-bit Globally Unique Identifier (present only if Type is FME or AFU).
- Offset 0x10
  - GUID_H - Most significant 64 bits of a 128-bit Globally Unique Identifier (present only if Type is FME or AFU).

1.3 Device Feature Header - Version 1

Version 1 (DFHv1) of the Device Feature Header adds the following functionality:

- Provides a standardized mechanism for features to describe parameters/capabilities to software.
- Standardize the use of a GUID for all DFHv1 types.
- Decouples the DFH location from the register space of the feature itself.

All multi-byte quantities in DFHv1 are little-endian. The format of Version 1 of the Device Feature Header (DFH) is shown below:

```
+-------------------------------------------------------------------------+
|63 Type 60|59 DFH VER 52|51 Rsvd 41|40 EOL|39 Next 16|15 REV 12|11 ID 0| 0x00
+-------------------------------------------------------------------------+
|63                                  GUID_L                             0| 0x08
+-------------------------------------------------------------------------+
|63                                  GUID_H                             0| 0x10
+-------------------------------------------------------------------------+
|63                   Reg Address/Offset                        1|  Rel  0| 0x18
+-------------------------------------------------------------------------+
|63        Reg Size       32|Params 31|30 Group    16|15 Instance      0| 0x20
+-------------------------------------------------------------------------+
|63 Next    35|34RSV33|EOP32|31 Param Version 16|15 Param ID           0| 0x28
+-------------------------------------------------------------------------+
|63                 Parameter Data                                        0| 0x30
+-------------------------------------------------------------------------+

                              ...

+-------------------------------------------------------------------------+
|63 Next    35|34RSV33|EOP32|31 Param Version 16|15 Param ID           0|
+-------------------------------------------------------------------------+
|63                 Parameter Data                                        0|
+-------------------------------------------------------------------------+
```

- Offset 0x00
  - Type - The type of DFH (e.g. FME, AFU, or private feature).
  - DFH VER - The version of the DFH.
  - Rsvd - Currently unused.
  - EOL - Set if the DFH is the end of the Device Feature List (DFL).
  - Next - The offset in bytes of the next DFH in the DFL from the DFH start, and the start of a DFH must be aligned to an 8 byte boundary. If EOL is set, Next is the size of MMIO of the last feature in the list.
  - REV - The revision of the feature associated with this header.
  - ID - The feature ID if Type is private feature.
- Offset 0x08
  - GUID_L - Least significant 64 bits of a 128-bit Globally Unique Identifier.
- Offset 0x10
  - GUID_H - Most significant 64 bits of a 128-bit Globally Unique Identifier.
- Offset 0x18
  - Reg Address/Offset - If Rel bit is set, then the value is the high 63 bits of a 16-bit aligned absolute address of the feature's registers. Otherwise the value is the offset from the start of the DFH of the feature's registers.
- Offset 0x20
  - Reg Size - Size of feature's register set in bytes.
  - Params - Set if DFH has a list of parameter blocks.
  - Group - Id of group if feature is part of a group.
  - Instance - Id of feature instance within a group.
- Offset 0x28 if feature has parameters
  - Next - Offset to the next parameter block in 8 byte words. If EOP set, size in 8 byte words of last parameter.
  - Param Version - Version of Param ID.
  - Param ID - ID of parameter.
- Offset 0x30
  - Parameter Data - Parameter data whose size and format is defined by version and ID of the parameter.

1.4 FIU - FME (FPGA Management Engine)

The FPGA Management Engine performs reconfiguration and other infrastructure functions. Each FPGA device only has one FME.

User-space applications can acquire exclusive access to the FME using open(), and release it using close().

The following functions are exposed through ioctls:

- Get driver API version (DFL_FPGA_GET_API_VERSION)
- Check for extensions (DFL_FPGA_CHECK_EXTENSION)
- Program bitstream (DFL_FPGA_FME_PORT_PR)
- Assign port to PF (DFL_FPGA_FME_PORT_ASSIGN)
- Release port from PF (DFL_FPGA_FME_PORT_RELEASE)
- Get number of irqs of FME global error (DFL_FPGA_FME_ERR_GET_IRQ_NUM)
- Set interrupt trigger for FME error (DFL_FPGA_FME_ERR_SET_IRQ)

More functions are exposed through sysfs (/sys/class/fpga_region/regionX/dfl-fme.n/):

Read bitstream ID (bitstream_id)
bitstream_id indicates version of the static FPGA region.

Read bitstream metadata (bitstream_metadata)
bitstream_metadata includes detailed information of static FPGA region, e.g. synthesis date and seed.

Read number of ports (ports_num)
one FPGA device may have more than one port, this sysfs interface indicates how many ports the FPGA device has.

Global error reporting management (errors/)
error reporting sysfs interfaces allow user to read errors detected by the hardware, and clear the logged errors.

Power management (dfl_fme_power hwmon)
power management hwmon sysfs interfaces allow user to read power management information (power consumption, thresholds, threshold status, limits, etc.) and configure power thresholds for different throttling levels.

Thermal management (dfl_fme_thermal hwmon)
thermal management hwmon sysfs interfaces allow user to read thermal management information (current temperature, thresholds, threshold status, etc.).

Performance reporting
performance counters are exposed through perf PMU APIs. Standard perf tool can be used to monitor all available perf events. Please see performance counter section below for more detailed information.

1.5 FIU - PORT

A port represents the interface between the static FPGA fabric and a partially reconfigurable region containing an AFU. It controls the communication from SW to the accelerator and exposes features such as reset and debug. Each FPGA device may have more than one port, but always one AFU per port.

1.6 AFU

An AFU is attached to a port FIU and exposes a fixed length MMIO region to be used for accelerator-specific control registers.

User-space applications can acquire exclusive access to an AFU attached to a port by using open() on the port device node and release it using close().

The following functions are exposed through ioctls:

- Get driver API version (DFL_FPGA_GET_API_VERSION)
- Check for extensions (DFL_FPGA_CHECK_EXTENSION)
- Get port info (DFL_FPGA_PORT_GET_INFO)
- Get MMIO region info (DFL_FPGA_PORT_GET_REGION_INFO)
- Map DMA buffer (DFL_FPGA_PORT_DMA_MAP)
- Unmap DMA buffer (DFL_FPGA_PORT_DMA_UNMAP)
- Reset AFU (DFL_FPGA_PORT_RESET)
- Get number of irqs of port error (DFL_FPGA_PORT_ERR_GET_IRQ_NUM)
- Set interrupt trigger for port error (DFL_FPGA_PORT_ERR_SET_IRQ)
- Get number of irqs of UINT (DFL_FPGA_PORT_UINT_GET_IRQ_NUM)
- Set interrupt trigger for UINT (DFL_FPGA_PORT_UINT_SET_IRQ)

DFL_FPGA_PORT_RESET:
reset the FPGA Port and its AFU. Userspace can do Port reset at any time, e.g. during DMA or Partial Reconfiguration. But it should never cause any system level issue, only functional failure (e.g. DMA or PR operation failure) and be recoverable from the failure.

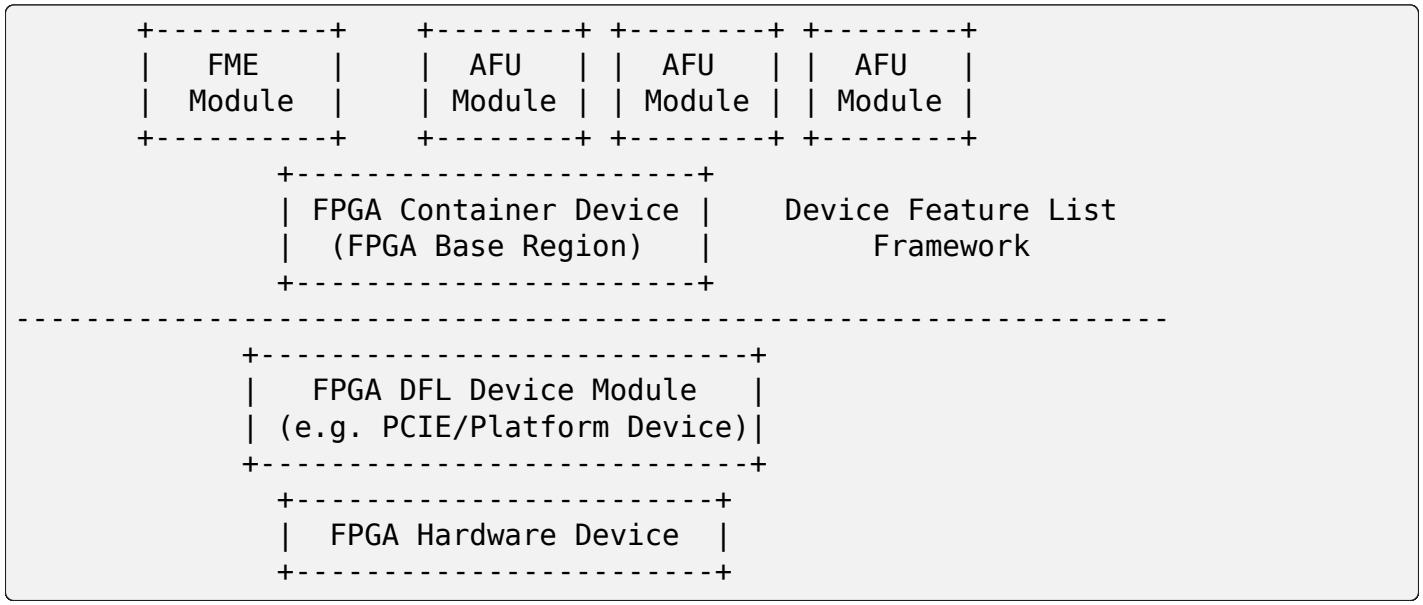
User-space applications can also mmap() accelerator MMIO regions.

More functions are exposed through sysfs: (/sys/class/fpga_region/<regionX>/<dfl-port.m>/):

Read Accelerator GUID (afu_id)
afu_id indicates which PR bitstream is programmed to this AFU.

Error reporting (errors/)
error reporting sysfs interfaces allow user to read port/afu errors detected by the hardware, and clear the logged errors.

1.7 DFL Framework Overview



DFL framework in kernel provides common interfaces to create container device (FPGA base region), discover feature devices and their private features from the given Device Feature Lists and create platform devices for feature devices (e.g. FME, Port and AFU) with related resources under the container device. It also abstracts operations for the private features and exposes common ops to feature device drivers.

The FPGA DFL Device could be different hardware, e.g. PCIe device, platform device and etc. Its driver module is always loaded first once the device is created by the system. This driver plays an infrastructural role in the driver architecture. It locates the DFLs in the device memory, handles them and related resources to common interfaces from DFL framework for enumeration. (Please refer to drivers/fpga/dfl.c for detailed enumeration APIs).

The FPGA Management Engine (FME) driver is a platform driver which is loaded automatically after FME platform device creation from the DFL device module. It provides the key features for FPGA management, including:

a) Expose static FPGA region information, e.g. version and metadata. Users can read related information via sysfs interfaces exposed by FME driver.

b) Partial Reconfiguration. The FME driver creates FPGA manager, FPGA bridges and FPGA regions during PR sub feature initialization. Once it receives a DFL_FPGA_FME_PORT_PR ioctl from user, it invokes the common interface function from FPGA Region to complete the partial reconfiguration of the PR bitstream to the given port.

Similar to the FME driver, the FPGA Accelerated Function Unit (AFU) driver is probed once the AFU platform device is created. The main function of this module is to provide an interface for userspace applications to access the individual accelerators, including basic reset control on port, AFU MMIO region export, dma buffer mapping service functions.

After feature platform devices creation, matched platform drivers will be loaded automatically to handle different functionalities. Please refer to next sections for detailed information on functional units which have been already implemented under this DFL framework.

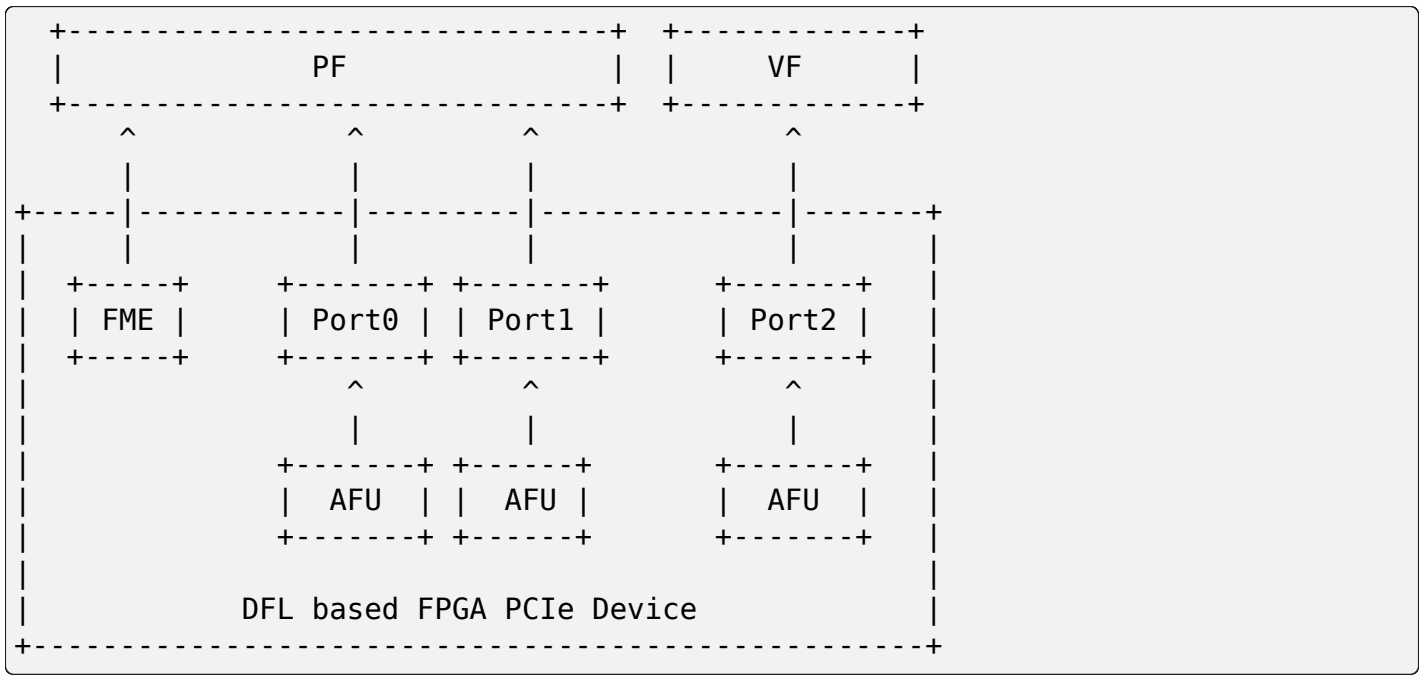
1.8 Partial Reconfiguration

As mentioned above, accelerators can be reconfigured through partial reconfiguration of a PR bitstream file. The PR bitstream file must have been generated for the exact static FPGA region and targeted reconfigurable region (port) of the FPGA, otherwise, the reconfiguration operation will fail and possibly cause system instability. This compatibility can be checked by comparing the compatibility ID noted in the header of PR bitstream file against the compat_id exposed by the target FPGA region. This check is usually done by userspace before calling the reconfiguration IOCTL.

1.9 FPGA virtualization - PCIe SRIOV

This section describes the virtualization support on DFL based FPGA device to enable accessing an accelerator from applications running in a virtual machine (VM). This section only describes the PCIe based FPGA device with SRIOV support.

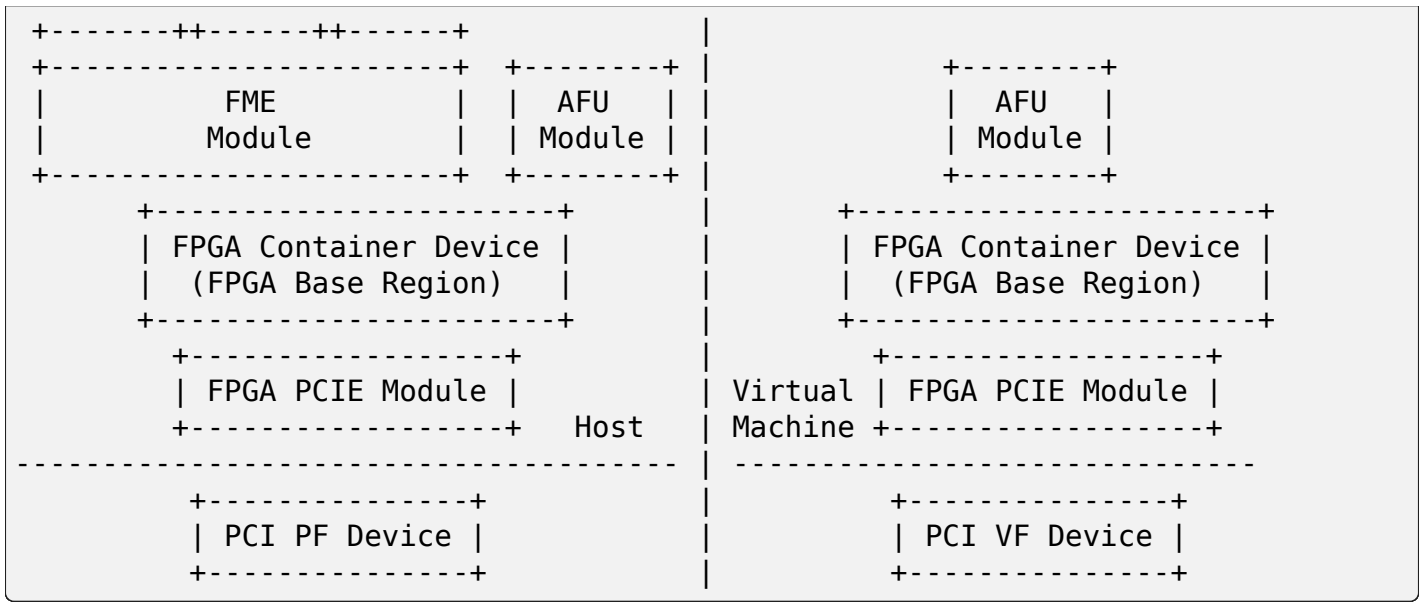
Features supported by the particular FPGA device are exposed through Device Feature Lists, as illustrated below:



FME is always accessed through the physical function (PF).

Ports (and related AFUs) are accessed via PF by default, but could be exposed through virtual function (VF) devices via PCIe SRIOV. Each VF only contains 1 Port and 1 AFU for isolation. Users could assign individual VFs (accelerators) created via PCIe SRIOV interface, to virtual machines.

The driver organization in virtualization case is illustrated below:



FPGA PCIe device driver is always loaded first once an FPGA PCIe PF or VF device is detected. It:

- Finishes enumeration on both FPGA PCIe PF and VF device using common interfaces from DFL framework.
- Supports SRIOV.

The FME device driver plays a management role in this driver architecture, it provides ioctls to release Port from PF and assign Port to PF. After release a port from PF, then it's safe to expose this port through a VF via PCIe SRIOV sysfs interface.

To enable accessing an accelerator from applications running in a VM, the respective AFU's port needs to be assigned to a VF using the following steps:

1. The PF owns all AFU ports by default. Any port that needs to be reassigned to a VF must first be released through the DFL_FPGA_FME_PORT_RELEASE ioctl on the FME device.
2. Once N ports are released from PF, then user can use command below to enable SRIOV and VFs. Each VF owns only one Port with AFU.

   ```
   echo N > $PCI_DEVICE_PATH/sriov_numvfs
   ```

3. Pass through the VFs to VMs
4. The AFU under VF is accessible from applications in VM (using the same driver inside the VF).

Note that an FME can't be assigned to a VF, thus PR and other management functions are only available via the PF.

1.10 Device enumeration

This section introduces how applications enumerate the fpga device from the sysfs hierarchy under /sys/class/fpga_region.

In the example below, two DFL based FPGA devices are installed in the host. Each fpga device has one FME and two ports (AFUs).

FPGA regions are created under /sys/class/fpga_region/:

```
/sys/class/fpga_region/region0
/sys/class/fpga_region/region1
/sys/class/fpga_region/region2
...
```

Application needs to search each regionX folder, if feature device is found, (e.g. “dfl-port.n” or “dfl-fme.m” is found), then it’s the base fpga region which represents the FPGA device.

Each base region has one FME and two ports (AFUs) as child devices:

```
/sys/class/fpga_region/region0/dfl-fme.0
/sys/class/fpga_region/region0/dfl-port.0
/sys/class/fpga_region/region0/dfl-port.1
...

/sys/class/fpga_region/region3/dfl-fme.1
/sys/class/fpga_region/region3/dfl-port.2
/sys/class/fpga_region/region3/dfl-port.3
...
```

In general, the FME/AFU sysfs interfaces are named as follows:

```
/sys/class/fpga_region/<regionX>/<dfl-fme.n>/
/sys/class/fpga_region/<regionX>/<dfl-port.m>/
```

with ‘n’ consecutively numbering all FMEs and ‘m’ consecutively numbering all ports.

The device nodes used for ioctl() or mmap() can be referenced through:

```
/sys/class/fpga_region/<regionX>/<dfl-fme.n>/dev
/sys/class/fpga_region/<regionX>/<dfl-port.n>/dev
```

1.11 Performance Counters

Performance reporting is one private feature implemented in FME. It could supports several independent, system-wide, device counter sets in hardware to monitor and count for performance events, including “basic”, “cache”, “fabric”, “vtd” and “vtd_sip” counters. Users could use standard perf tool to monitor FPGA cache hit/miss rate, transaction number, interface clock counter of AFU and other FPGA performance events.

Different FPGA devices may have different counter sets, depending on hardware implementation. E.g., some discrete FPGA cards don’t have any cache. User could use “perf list” to check which perf events are supported by target hardware.

In order to allow user to use standard perf API to access these performance counters, driver creates a perf PMU, and related sysfs interfaces in /sys/bus/event_source/devices/dfl_fme* to describe available perf events and configuration options.

The “format” directory describes the format of the config field of struct perf_event_attr. There are 3 bitfields for config: “evtype” defines which type the perf event belongs to; “event” is the identity of the event within its category; “portid” is introduced to decide counters set to monitor on FPGA overall data or a specific port.

The “events” directory describes the configuration templates for all available events which can be used with perf tool directly. For example, fab_mmio_read has the configuration “event=0x06,evtype=0x02,portid=0xff”, which shows this event belongs to fabric type (0x02), the local event id is 0x06 and it is for overall monitoring (portid=0xff).

Example usage of perf:

```
$# perf list |grep dfl_fme

dfl_fme0/fab_mmio_read/                                 [Kernel PMU event]
<...>
dfl_fme0/fab_port_mmio_read,portid=?/                   [Kernel PMU event]
<...>

$# perf stat -a -e dfl_fme0/fab_mmio_read/ <command>
or
$# perf stat -a -e dfl_fme0/event=0x06,evtype=0x02,portid=0xff/ <command>
or
$# perf stat -a -e dfl_fme0/config=0xff2006/ <command>
```

Another example, fab_port_mmio_read monitors mmio read of a specific port. So its configuration template is “event=0x06,evtype=0x01,portid=?”. The portid should be explicitly set.

Its usage of perf:

```
$# perf stat -a -e dfl_fme0/fab_port_mmio_read,portid=0x0/ <command>
or
$# perf stat -a -e dfl_fme0/event=0x06,evtype=0x02,portid=0x0/ <command>
or
$# perf stat -a -e dfl_fme0/config=0x2006/ <command>
```

Please note for fabric counters, overall perf events (fab_*) and port perf events (fab_port_*) actually share one set of counters in hardware, so it can’t monitor both at the same time. If this set of counters is configured to monitor overall data, then per port perf data is not supported. See below example:

```
$# perf stat -e dfl_fme0/fab_mmio_read/,dfl_fme0/fab_port_mmio_write,\
                                                 portid=0/ sleep 1

Performance counter stats for 'system wide':

               3      dfl_fme0/fab_mmio_read/
 <not supported>      dfl_fme0/fab_port_mmio_write,portid=0x0/

     1.001750904 seconds time elapsed
```

The driver also provides a “cpumask” sysfs attribute, which contains only one CPU id used to access these perf events. Counting on multiple CPU is not allowed since they are system-wide counters on FPGA device.

The current driver does not support sampling. So “perf record” is unsupported.

1.12 Interrupt support

Some FME and AFU private features are able to generate interrupts. As mentioned above, users could call ioctl (DFL_FPGA_*_GET_IRQ_NUM) to know whether or how many interrupts are supported for this private feature. Drivers also implement an eventfd based interrupt handling mechanism for users to get notified when interrupt happens. Users could set eventfds to driver via ioctl (DFL_FPGA_*_SET_IRQ), and then poll/select on these eventfds waiting for notification. In Current DFL, 3 sub features (Port error, FME global error and AFU interrupt) support interrupts.

1.13 Add new FIUs support

It’s possible that developers made some new function blocks (FIUs) under this DFL framework, then new platform device driver needs to be developed for the new feature dev (FIU) following the same way as existing feature dev drivers (e.g. FME and Port/AFU platform device driver). Besides that, it requires modification on DFL framework enumeration code too, for new FIU type detection and related platform devices creation.

1.14 Add new private features support

In some cases, we may need to add some new private features to existing FIUs (e.g. FME or Port). Developers don’t need to touch enumeration code in DFL framework, as each private feature will be parsed automatically and related mmio resources can be found under FIU platform device created by DFL framework. Developer only needs to provide a sub feature driver with matched feature id. FME Partial Reconfiguration Sub Feature driver (see drivers/fpga/dfl-fme-pr.c) could be a reference.

Please refer to below link to existing feature id table and guide for new feature ids application.
https://github.com/OPAE/dfl-feature-id

1.15 Location of DFLs on a PCI Device

The original method for finding a DFL on a PCI device assumed the start of the first DFL to offset 0 of bar 0. If the first node of the DFL is an FME, then further DFLs in the port(s) are specified in FME header registers. Alternatively, a PCIe vendor specific capability structure can be used to specify the location of all the DFLs on the device, providing flexibility for the type of starting node in the DFL. Intel has reserved the VSEC ID of 0x43 for this purpose. The vendor specific data begins with a 4 byte vendor specific register for the number of DFLs followed 4 byte Offset/BIR vendor specific registers for each DFL. Bits 2:0 of Offset/BIR register indicates the BAR, and bits 31:3 form the 8 byte aligned offset where bits 2:0 are zero.

```
+-----------------------------+
|31     Number of DFLS      0|
+-----------------------------+
|31     Offset     3|2 BIR  0|
+-----------------------------+
               . . .
+-----------------------------+
|31     Offset     3|2 BIR  0|
+-----------------------------+
```

Being able to specify more than one DFL per BAR has been considered, but it was determined the use case did not provide value. Specifying a single DFL per BAR simplifies the implementation and allows for extra error checking.

1.16 Userspace driver support for DFL devices

The purpose of an FPGA is to be reprogrammed with newly developed hardware components. New hardware can instantiate a new private feature in the DFL, and then present a DFL device in the system. In some cases users may need a userspace driver for the DFL device:

- Users may need to run some diagnostic test for their hardware.
- Users may prototype the kernel driver in user space.
- Some hardware is designed for specific purposes and does not fit into one of the standard kernel subsystems.

This requires direct access to MMIO space and interrupt handling from userspace. The uio_dfl module exposes the UIO device interfaces for this purpose.

Currently the uio_dfl driver only supports the Ether Group sub feature, which has no irq in hardware. So the interrupt handling is not added in this driver.

UIO_DFL should be selected to enable the uio_dfl module driver. To support a new DFL feature via UIO direct access, its feature id should be added to the driver's id_table.

1.17 Open discussion

FME driver exports one ioctl (DFL_FPGA_FME_PORT_PR) for partial reconfiguration to user now. In the future, if unified user interfaces for reconfiguration are added, FME driver should switch to them from ioctl interface.