A case study into improving Network device driver quality by design

Anjali Singhai Jain
Kevin Scott
anjali.singhai@intel.com
kevin.c.scott@intel.com

Abstract

In this paper we discuss some of the key software design decisions which were made to facilitate driver development and validation prior to receiving silicon. We discuss engineering collaboration across the hardware, software and firmware teams as partners in the generation of driver code, as well as using driver synthesis to generate code.

The goal of the project was three fold: 1) Develop drivers with a significant number of code paths which are easily exercisable. 2) Allow both open source and closed source drivers to share resources and obtain higher quality code as a result. 3) Allow as much code as possible to be Open Source.

We discuss the lessons learned on what and how to share between drivers, and how to develop the code such that it can be used seamlessly by driver teams with various needs (Linux, Windows, FCoE, iSCSI, DCB, RDMA, silicon firmware modules, etc.).

Biography

Kevin Scott is a Network Software Engineer who has been with Intel's LAN Access Division for 12 years.

Anjali Singhai Jain is a Network software Engineer at Intel Inc., currently working at the Intel site in Portland, Oregon. Over the past 8 years, she has been involved in software development for tools, early prototype libraries and network device drivers for Intel's Networking division. Anjali as part of tools team has mastered in software design for validation and writing OS agnostic modules. She has been actively involved in using her learning's in her current position as Linux network device driver developer.

Anjali has an M.S. in Computer Science from Portland State University.

1 Introduction

In general, as computing systems have advanced, the sophistication and complexity of the software and hardware that comprise them has increased. This general trend is mirrored in Intel's network controllers. As feature lists in new hardware grows, the ability to write the various pieces of software which control, validate and test the hardware become more complex. This increase in complexity also generally means that development times (Time to Market) is greater. Methods to reduce development time for a new product with uncompromised quality must be identified.

This paper discusses one such code development mandate, which we call "Shift Left". The Shift Left mandate is a corporate wide effort to address time to market challenges.

We will focus this paper on the following road-blocks to our Shift Left goals:

- How to increase the scope of sharing code across platforms, environments, and teams/functionality (Firmware, Hardware, Silicon Validation, Software).
- Splitting Hotpath (code traversed repeatedly during packet processing) and non-Hotpath code and keeping the later common.
- Designing an OS abstraction layer to allow more code to be shared.
- Auto-generation of hardware definitions for use by drivers and tools i.e. driver synthesis.
- Co-development with other software groups.
- Ability to validate software paths prior to FPGA arrival.
- Software device emulation.

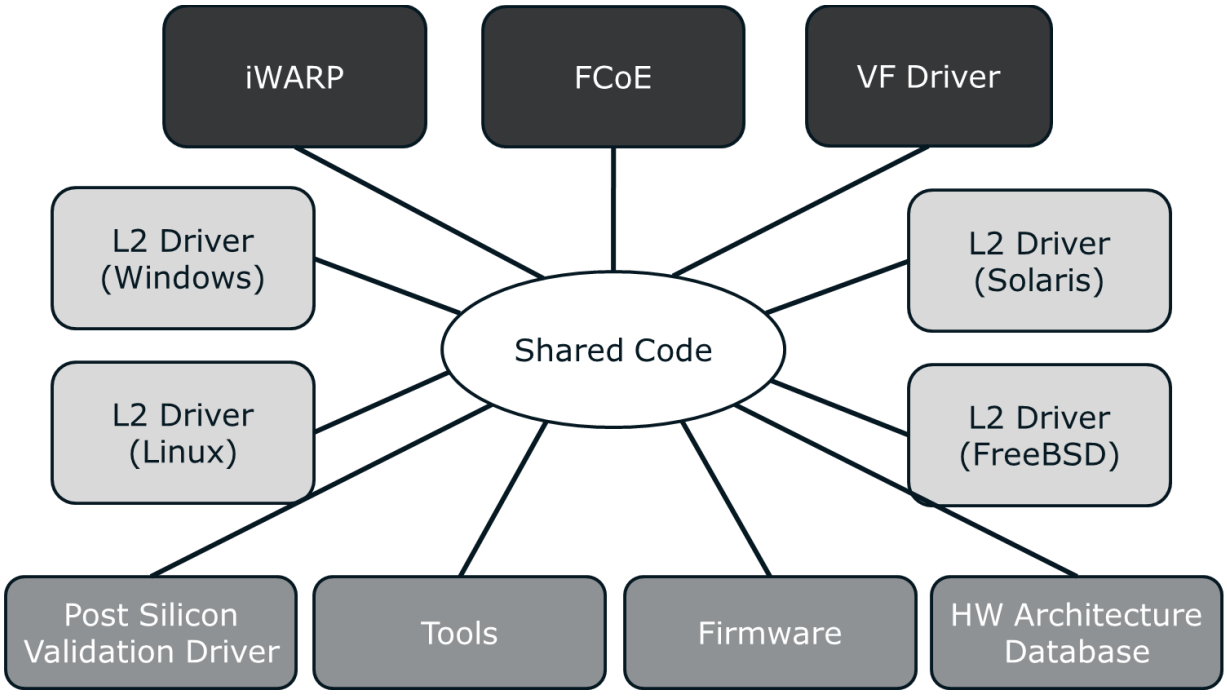

Figure : Various components that share code.

Figure 1 shows the different drivers, tools and components that share huge chunks of code. This was made possible due to early shared code design, keeping the consumer needs in mind and identifying the OS abstraction APIs needed for the same.

2 Design Philosophy

Intel has adopted the Shift Left mandate for its LAN products in an effort to release products in a timely manner; by moving project milestones earlier (to the left) in their lifetime. As silicon complexity has grown, due to supporting technologies such as SR-IOV (Single Root I/O Virtualization), DCB (Data Center Bridging), FCoE (Fibre Channel over Ethernet) and RDMA (Remote Direct Memory Access) among others, the overall product development timeline has remained the same or even shrunk.

Additionally, in order to remain competitive in the market, it is required that the time between the arrival of first silicon to the time that silicon is available to customers, is as short as possible. This effectively mandates that much of the software development and validation occurs prior to first silicon arrival.

The Shift Left mandate has introduced new complexities in the product definition / software life cycle model:

- The hardware architecture specification may not be complete prior to the beginning of software development. There are numerous “details” which are not fully understood or fleshed out at this stage, so how can the software respond to this challenge?
- Different software components will be in development at the same time ex: firmware and OS base drivers. This forces components which are being actively developed at the same time to function together at some level.
- The software has to be validated on something prior to FPGA (Field Programmable Gate Array) arrival. Since the FPGA may arrive close to the arrival of first silicon, this leaves little room for validation post FPGA arrival.

This single mandate has forced teams to rethink, redesign and innovate how they develop software. Converting some serial tasks into parallel tasks and identify new partners to indirectly implement and validate their drivers.

Intel LAN software has historically utilized shared code, source code which is used by two or more components, to develop various drivers and tools. The amount of code which was shared varied, as some teams needed to develop their components before the shared code was functionally complete. The time constraints, as viewed by individual teams, would lead to purpose built code and ultimately, a duplication of effort.

In order to address this, the shared code had to start development earlier, grow in scope and be used by a wider developer audience.

2.1 Automatic Code Generation

One method to allow for earlier code development is to combine the act of defining hardware specifications by the architecture teams and the generation of code which can be used to manipulate the definition i.e. the auto-generation of code.

Previous Intel LAN products have used little, if any auto-generated code. All definitions for register offsets, masks, shifts, etc. were manually coded by a developer, who obtained the information from an Engineering Architecture Specification (EAS). The EAS in turn is written by a team of silicon and software architects, over a long period of time. During this time, things such as register definitions are modified and lead to errors in the shared code.

The Intel Networking Group is now using the tool which architects use to define the silicon (RTL), and develop the EAS, to generate code which is directly useable in the shared code. This all but eliminates

one of the historically more problematic issues of code development; the manual task of translating an EAS into shared code.

Another benefit of auto-generated code is the consistency of style. Since the shared code is part of the Linux kernel, avoiding submission issues because of simple style guideline errors can greatly speed upstream submission. On projects with many developers, and varying experience with the style guidelines, auto-generated code makes a lot of sense.

For reference, the Linux base driver for a future networking product contains over 13,000 lines of auto-generated code and roughly 60,000 lines of total driver code. So, currently, approximately 25% of the base is auto-generated. If this amount of code were generated manually, it would have resulted in many individual code patches, and the overhead associated with the generation and validation of such patches. This is a significant savings in time and effort.

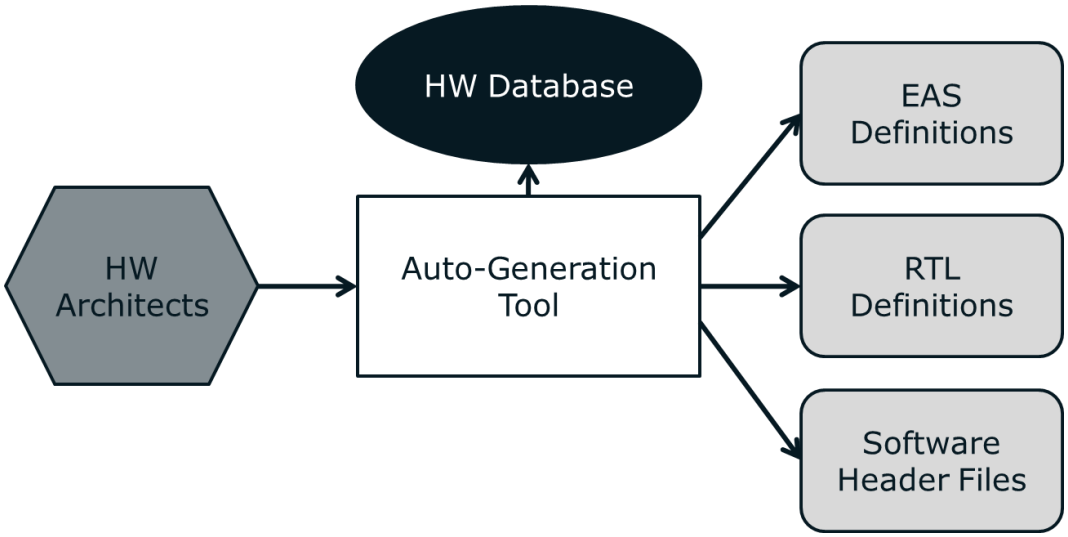

Figure : Auto Code Generation

Figure 2 describes the actions of the code auto generation tool. The tool is used by the hardware architects to define the contents of the definitions database. The tool uses this single database to produce content for the EAS, C header files, RTL code etc. The tool also tracks revision history and can be used across various generation of products.

2.2 Code Re-Use

Code duplication has historically been a problem, as different components must meet widely different timeline requirements. For example, in order to test and validate FPGAs destined for software developers, the silicon validation team would write a custom driver (SV driver), which by its nature was similar to a high level OS driver (Linux or Windows), yet the code developed could have been used in the shared code.

With the Shift Left mandate, developers across disciplines agree to work with a single code base, derived from the architect's specification, and this code in turn is used in numerous tools and drivers.

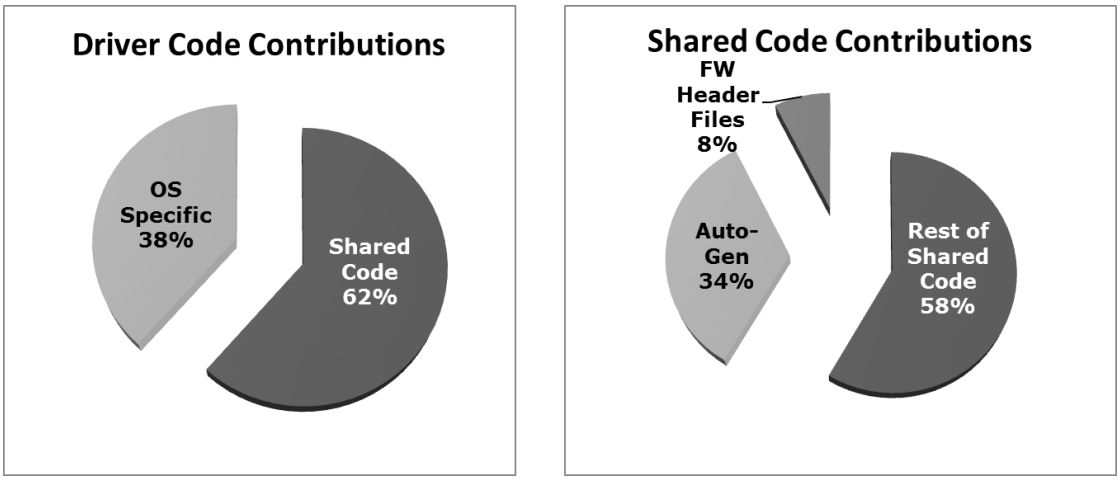

Figure : Shared code pieces vs. OS specific code in Linux Device driver.

Figure 3 shows the percentage split of different contributors to the Linux device driver. Left chart shows percentage of Total shared code vs. OS specific code. Right chart shows what percentage of shared code consists of Auto-generated code, FW code and the rest. Shared code between OS drivers, validation drivers and Tools has become a major portion of the overall device driver.

In order for code-reuse to have a meaningful impact, we focused on two major areas:

- OS abstraction APIs to aid with keeping code common
- Identifying what should be part of common code and what should remain in OS specific files

2.2.1 OS Abstraction

Historically, the shared code could be considered a very thin abstraction layer between software and hardware, in that it consisted of fundamental hardware/software interface mechanisms. To increase the role of shared code, we have abstracted roles which were normally handled by OS specific drivers:

- Allocation of both physical and virtual memory structures
- Memory manipulation functions
- Code Synchronization: spinlocks
- Delays
- Debug mechanisms (debug prints and driver logging mechanisms)

Here is an example of a set of OS Abstraction APIs as declared in the shared code. The definitions are filled out by each of the OS drivers in the OS specific files. This is a good example to illustrate how we designed the APIs to meet various drivers' needs to allocate the memory from different pools depending on its use. Another driver could chose to completely ignore the type and allocate from a common pool.

Code Abstraction Example:

```
/* Memory allocation types */
enum i40e_memory_type {
        i40e_mem_arq_buf = 0,          /* ARQ indirect command buffer */
        i40e_mem_asq_buf = 1,
        i40e_mem_atq_buf = 2,          /* ATQ indirect command buffer */
        i40e_mem_arq_ring = 3,             /* ARQ descriptor ring */
        i40e_mem_atq_ring = 4,             /* ATQ descriptor ring */
        i40e_mem_pd = 5,                   /* Page Descriptor */
        i40e_mem_bp = 6,                   /* Backing Page - 4KB */
```

```
        i40e_mem_bp_jumbo = 7,              /* Backing Page - > 4KB */
        i40e_mem_reserved
};
/* prototype for functions used for dynamic memory allocation */
enum i40e_status_code i40e_allocate_dma_mem(struct i40e_hw *hw, struct i40e_dma_mem
*mem,
                                            enum i40e_memory_type type, u64 size, u32 alignment);
enum i40e_status_code i40e_free_dma_mem(struct i40e_hw *hw, struct i40e_dma_mem *mem);
enum i40e_status_code i40e_allocate_virt_mem(struct i40e_hw *hw,  struct i40e_virt_mem
*mem, u32 size);
enum i40e_status_code i40e_free_virt_mem(struct i40e_hw *hw,  struct i40e_virt_mem *mem);
```

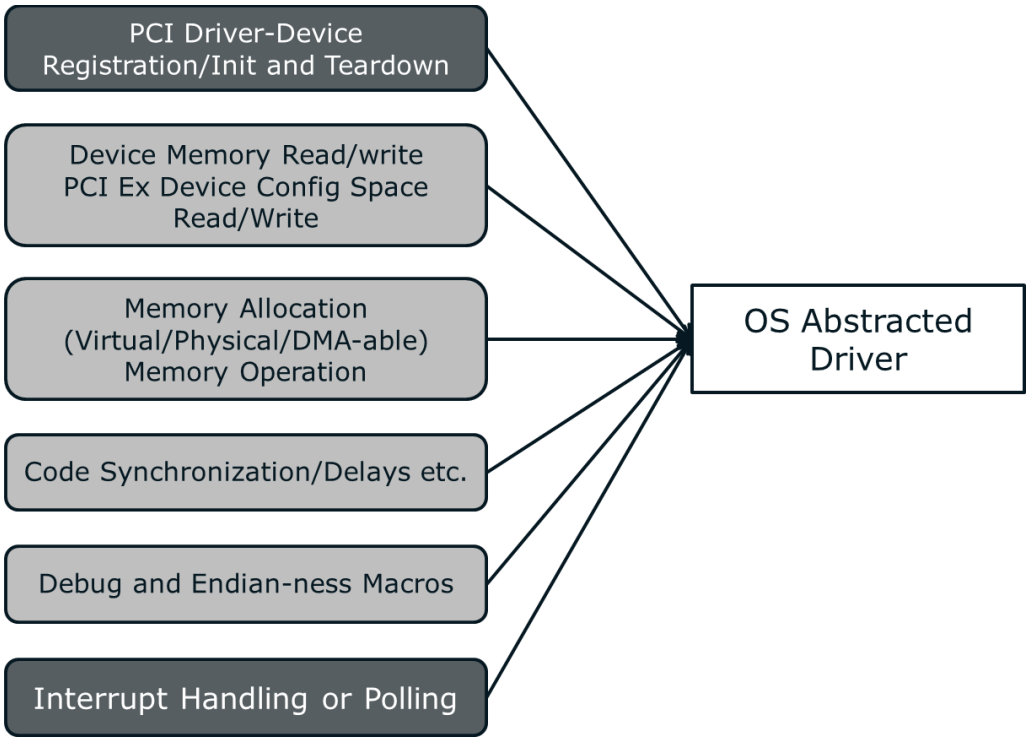

Figure : OS Abstraction parts.

Figure 4: This figure takes a leap into the future. Blocks in the middle are abstracted as of now in our driver. If we were to consider a completely OS abstracted driver for various reasons in the future, we would have to come up with OS abstraction APIs for the top and the bottom block.

2.2.2 Hotpath vs. Non-Hotpath

The shared code primarily focuses on code (Non-Hotpath) which is run outside of Interrupt Service Routines (ISRs) and Deferred Procedure Calls (DPCs) i.e. the Hotpath. We term the Interrupt or the DPC code as Hotpath since it is not desirable that the driver; these portions of the code is where all of the packet processing happens in the driver and any code here impacts the total throughput of the driver. Keeping Hotpath code out of shared code allows for OS specific drivers to tune it in the most efficient way to achieve relevant performance characteristics. On the other hand, Initialization codes, configuration code, tear down and reset code is non-Hotpath and is very Hardware focused, these make the best candidates for Shared code.

2.3 Co-development

The shared code is simply too complicated and large for any single team to develop. Historically, the various teams have worked and developed in a fairly compartmentalized manner, coming together and sharing details only when there is some sort of problem; often an emergency. Now, the insights and perspectives of the various teams have a single point of focus. Features which may initially be needed by only one team, and written in team specific code in the past, are now shared, so that when any team needs to use said feature, they simply have to incorporate what is already there. Features, workarounds, errata do not need to be re-discovered.

2.3.1 Firmware/Software team as a model for co-development

Similar to the hardware specific definitions being owned by the hardware team, the firmware definitions are owned by the firmware team and used by both the firmware and OS driver teams. Though not auto-generated, the firmware and software teams now use identical code (Shared Firmware Header files) to develop their separate drivers. Since there is a single owner (the firmware team), the code is always accurate with the latest specification, and can't be misinterpreted by the OS driver teams.

The firmware header files generation follows the below rules:

- Shared code style guideline.
- Patches generated solely by firmware team.
- Reviewed by software shared code team.
- Maintained by software shared code team.
- Consumers are both software and firmware teams.

3 Testing Scope

Shared code has historically been written with OS base drivers in mind, so the code paths which are developed and exercised are generally limited to operations only the base drivers would consider valid. With multiple teams developing the shared code, the scope of testing and validation is widened. Code paths which may not have been exercised previously are now exposed and testable.

3.1 Early validation partners and tools

3.1.1 Shared Code use by Post-Silicon Validation Drivers

Prior to the OS driver teams obtaining an FPGA or first silicon, the silicon validation teams validate hardware behavior on a silicon validation driver, which is based on the Linux driver. As this driver is developed with the shared code, very early feedback on the health of the code is obtained. In addition to this early feedback, the testing methodology of the silicon validation team is very different than that of the OS driver teams. The Silicon validation tests are heavily automated and randomly generated, designed to exercise various hardware boundaries and extremes; paths the OS driver code would only exercise in rare cases. This increases overall shared codes robustness.

3.1.2 Hardware emulation: KVM-QEMU

Historically, OS drivers could only be run once FPGA devices arrived and were programmed and tested by the silicon validation team. However, with the Shift Left mandate, this is no longer reasonable. In order to exercise OS driver code prior to FPGA, we have emulated our network device in software using KVM-QEMU framework provided with standard Linux kernel. Though this only emulates a small set of

total device features, it allows the OS drivers to exercise critical code paths: initialization, transmit, receive, DPC/interrupt, etc. It also makes possible testing other components of the driver stack, such as protocol drivers and user mode components which may leverage features in the base driver.

Advanced offloads and features are areas for improvement going forward.

4 Linux Specific Changes to Improve quality

In our software model, there are many different drivers that interact at various levels with the silicon.

4.1 Common Client Interface Model

Earlier versions of our base drivers and silicon had many different ways to interact with protocol drivers; some co-existed with the base driver, and others resided in their own modules using a proprietary interface. In our new driver model, we have devised a generic interface between the base driver and the protocol drivers. This helps reduce code, complexity and provides easy ways to scale and test.

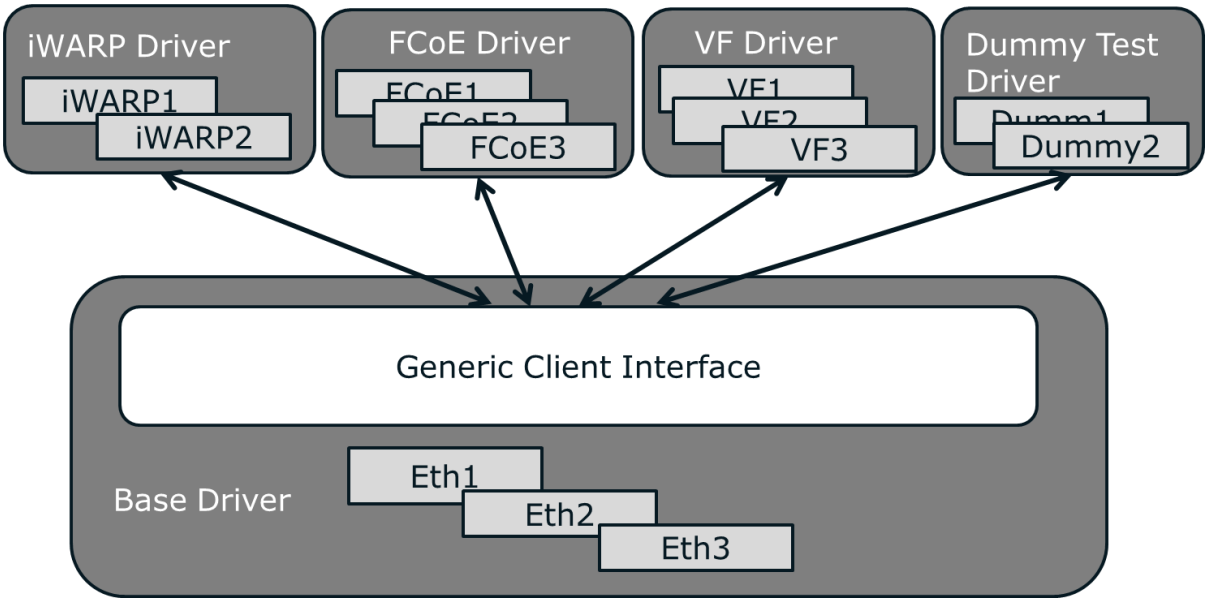

Figure : Client Interface Model in Linux

Figure 5 describes the Generic Client Interface that is implemented in the base driver and exposed among all protocol drivers that need the base driver services. Base driver exposes the various PCI devices present in the system that the driver supports as individual HW Interfaces (ex. eth0, eth1 etc.). The protocol (Client) drivers each in turn have instances corresponding to the base driver Interfaces. Each Client instance is tied to its respective base driver interface through the generic client Interface layer. iWARP is a name given to the RDMA driver in this case. VF drivers are the Virtual function drivers that use SR-IOV to boost VM Network performance.

4.2 debugfs

We now use the debugfs support built in Linux kernel in our driver for:

- Driver validation support during development and project releases.

- Driver configuration interface for developers/users/administrators.

Earlier, we would use custom debug macros in our driver, which were not very extensible. They did not provide for any easy configuration interface; module parameters and ethtool were our only choices for driver configuration. Module parameters are not very well received by upstream kernel development team and ethtool has its limitation in terms of what gets accepted as configuration knobs by the different Network silicon vendors that have Open source presence. debugfs in that respect can be programmed to expose anything and everything the driver writer would like exposed for configuration. This has tremendously increased validation scope for our Linux based drivers.

5 Conclusion

Though the development process is not yet complete for our first driver utilizing the development methods we have described is complete, we have been able to successfully address many of the challenges presented by the Shift Left mandate.

1) As technologies grow more complex and the boundaries between software/hardware/firmware blur, the ability to share as much code as possible between them becomes more important. Utilizing tools which allow for the automatic generation of code eliminates many 'lost in translation' errors which would be experienced by various teams in various geographies, and incur a high cost in time to debug, weakening the Shift Left mandate.
2) A small investment of time and effort in designing the right OS abstraction APIs goes a long way in enabling different OS driver teams to share large amounts of code, helping reduce individual team efforts, bugs and lines of code to be maintained. Choosing which code to share should be done prudently. Sharing code in the fast path limits the ability to make OS specific optimizations to the code, and can limit overall driver performance.
3) Early Hardware emulation in software, by driver developers, helps greatly. It provides a mechanism to validate code paths ahead of silicon. By writing code to the hardware specification, software teams can help identify difficult areas to manage or code for. Driver developers also develop a greater understanding of the hardware model they are developing for. Above all else, we were able to find and fix hundreds of driver bugs prior to silicon arrival. We found that this ability alone justified the time required to develop a hardware model.
4) A common abstracted client interface for advanced technologies (RDMA, FCoE, and Virtualization) provided for scalability and code-reuse. This model provides for building blocks that plug on top of base driver in a generic fashion hence providing for easy testing.

6 Future Directions

The current level of automatic code generation is still limited in scope; it covers only register offsets (and ranges), mask and shift definitions. Extending code generation to structure definitions, field values and names (enumerated types or defines) will further reduce the time it takes to develop code and reduce chances for error. This would approximately double the size of present auto generate code in the drivers.

The OS abstraction layer in shared code in the future could allow for simplified driver models with very little OS dependent code. These would be non-performance diagnostic drivers whose goals would be to exercise code flows and features which are not easily tested in the standard OS environment. Another consideration is the development of a driver which is forward compatible, allowing network connectivity without a full featured OS driver. Such a driver could be deployed in the market prior to silicon, enabling the new silicon out of the box for basic functionality at all times.

7 Glossary

iSCSI (Internet Small Computer System Interface) - iSCSI is a protocol to transport SCSI commands over Internet Protocol (IP) based networks, generally used for storage connectivity.

FCoE (Fibre Channel over Ethernet) – Protocol which encapsulates Fibre Channel packets on standard Ethernet based networks.

FPGA (Field Programmable Gate Array) – An integrated circuit which can be repeatedly programmed and modified.

DCB (Data Center Bridging) – A set of Ethernet extensions to help facilitate functionality in data center environments.

SR-IOV (Single Root I/O Virtualization) – A mechanism to allow a Single Root Function (ex: an Ethernet port) to be exposed as multiple physical functions.

RDMA (Remote Direct Memory Access) – A mechanism to allow bus mastering devices (ex: Ethernet controllers) to place data directly into user space, rather than to kernel memory and then copied to user space.

debugfs – Linux file system used for debugging purposes. Allows for a convenient kernel to user space communication mechanism.

EAS (Engineering Architecture Specification) – Document which describes the technical details, specifications, and interfaces of a particular piece of silicon.

SV (Silicon Validation) – Process by which silicon behavior (post production) is compared to expected behavior, as defined, in part, by an EAS.

DPC (Deferred Procedure Call) – Function which processes required work for the Ethernet device outside of the interrupt context.

RTL (Register Transfer Language) – A low level language used to describe the behavior of operations in integrated circuits.

Shared Code – A single code base which is used by multiple software components to help reduce the amount of code duplication.

KVM-QEMU –Kernel-based Virtual Machine- Quick Emulator