Brief Introduction to WebStaX SW Architecture

Application Note

by Allan W. Nielsen

CONFIDENTIAL

AN1184
Rev. APPL-2024.06
2024-06-28

Table of Contents

1. Introduction

This document will offer a brief introduction to what WebStaX is, how it is designed, how it can be customized and integrated. More detailed information can be found in the product spec, the various user guides and configuration guides.

2. Component overview

WebStaX is a collection of many different SW components which is combined into a product targeting the end-user. This collection of SW includes both third-party and internal developed SW, and it include both proprietary and open-source SW.

This means that the WebStaX product include a fair amount of SW developed for this specific purpose, but it also include third-party SW that has been carefully selected, integrated, in some cases patches, and tested as a single product.

When new versions of WebStaX (updates) is being released it typically updates all components used in WebStaX, and it offer a mechanism to install the upgrades. Each release of WebStaX includes a list of all third-party SW included, and specify the versions and license terms for each third-party SW packet.

The following subsections lists all the main components that constitutes the WebStaX product.

2.1. Board Support Packet (BSP)

The BSP provides almost all third-party components that are needed. This includes both development tools needed to build the executable and third-party components needed on target. Example of development tools are: cross-compiler, cmake, linker, automake/autoconf etc. Example of target components are Linux kernel, libc, net-snmp, dropbear, busybox etc.

MSCC provides a BSP that is designed and optimized for MSCC reference boards and the WebStaX application software. The BSP is distributed both as sources and binaries. The sources are needed for customers who want/need to change the BSP, while the binary BSP can be used if no changes are required.

More details can be found in UG1068.

2.2. MESA/Unified-API

The MESA/Unified-API is a library used to access the switching/phy hardware. The API is included as part of the application SW. Customers who are building a product based on one of the WebStaX variants will automatically be using the API included in the WebStaX source package.

The API provides an abstraction layer such that the application does not need to be aware of the register set provided by a single chip, and allow the application to support many different switch cores.

More details can be found in UG1070.

2.3. Application

The WebStaX product family includes three different application packages: `WebStaX`, `SMBStaX` and `IStaX`. The three packages have different feature sets and different licensing terms. This document will not be focusing on the individual packages, but it will assume that one of the three packages is being used. When referring to "MSCC-Application", "application" or "switch application" then it is one of these three packages.

The application is a collection of SW modules that can be divided into the following categories:

Control modules

Allow the user to apply static configuration and/or query status. Such modules will typically use the MESA library to configure the switch-HW accordingly. Examples of such modules includes: `port`, `acl`, `vlan`.

Protocol modules

Allow the user to enable/configure protocols that can interact with other network equipment. Such modules will typically listen for certain network packets, and transmit packets accordingly to the protocols specification. Example of such modules includes: `gvrp`, `lldp`, `ptp`.

Interface modules

Provides management interface that can be used to configure and/or query status of switch. Example of such modules include: `cli`, `ssh`, `snmp`, `json-rpc`, `web`.

Infrastructure modules

This is a set of module that is not directly visible to the user, but provides some infrastructure facilities that is used by SW running on the switch. Example of such modules is: `basics` (algorithms and containers), `critd` (mutex and dead lock detection), `trace` (debug facilities).

The application SW is organized in such way that each module can be enabled/disabled in the build process, and new modules can easily be added.

> ? Some of the application modules have internal dependencies which must be considered when enabled/disabled modules. Not all combinations has been tested, and certain combinations may lead to compilation errors.

2.4. Flash images

A flash image is a binary image that may be burned to the `NOR` flash using a programmer. The flash images include partition table for the `NOR` flash, boot-loader, bring-up image (Linux kernel, stage1 file system, stage2 minimal). A given flash image may only be used on the specific board it is designed for.

3. Integration with Linux

This section will provide a brief overview of the system architecture, and go into some details on how WebStaX is integrated with the Linux OS. The image below illustrates the overall system architecture.

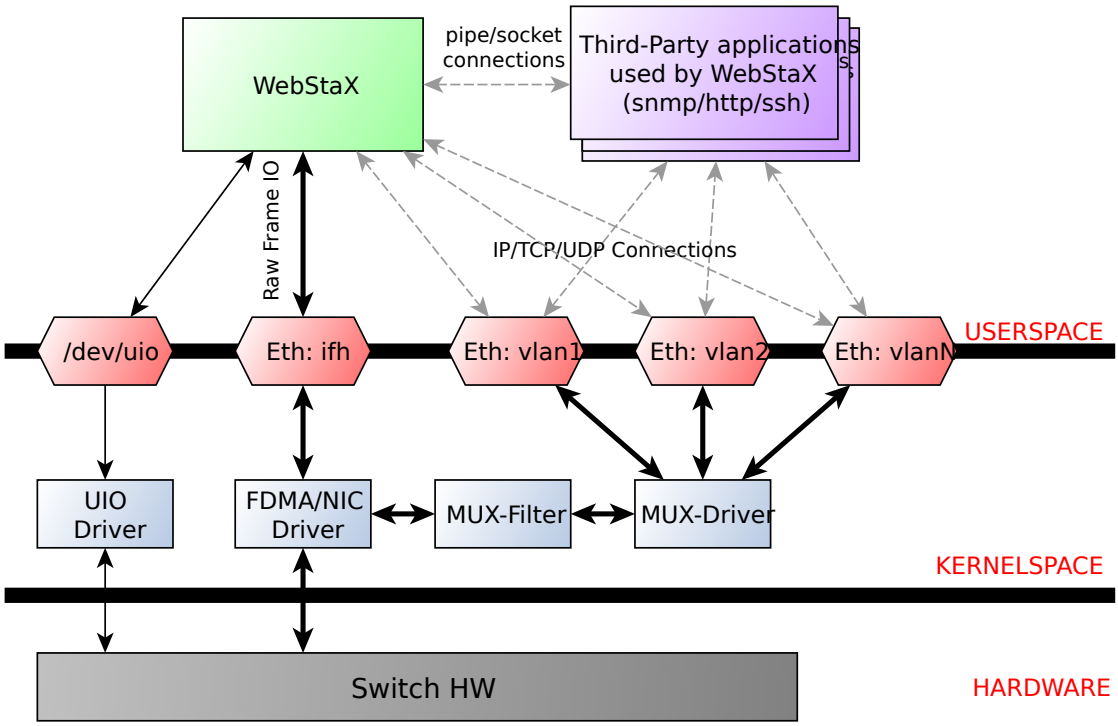

Figure 1. Overall system architecture

The green box labelled WebStaX is the MSCC switch application, it can be any of the supported variants (`WebStaX`, `SMBStaX` or `IStaX`). The switch application is running as a long-lived normal user-space process (as root), and it is interacting with the switch registers through the `uio` driver. The WebStaX application includes an instance of the API (linked in as a library). The application must be the exclusive owner of the API and switch registers.

?

This means that no other process is allowed to instantiate the API and alter the switch registers in HW, this must go through the API instance already created by the application. Other process can communicate with the WebStaX application and access the indirectly access the API through the WebStaX process).

The `uio` kernel space driver is a simple kernel module which does two things; 1) exposes the entire register region of the switch hardware, and 2) exposes all interrupts from the switch HW. The `uio` kernel module is provided by the Linux kernel (part of the BSP) and allows user-space applications, like WebStaX, to gain access to HW registers and interrupts from user-space. This is achieved by a `mmap` of the register region from the user-space application.

3.1. Frame flow

Besides from configuring the switch registers in HW, the application also implements a number of protocols (which may influence the switch configurations). To implement these protocols the application needs to inject frames into the switch core, and it needs to extract frames that have been redirected to the CPU (either because it was send to the MAC address of the CPU, or because an ACL rule has captured the frame). To implement this frame-flow the Linux kernel in the BSP provides a FDMA driver which can inject/extract to/from the CPU queue in the switching hardware.

Frames that are injected/extracted to/from the CPU queue are prefixed with an extra header that carries various side-band information related to the frame (front port, classified VLAN, ACL rule number, time stamp etc.). The content of the header is chip dependent and the content is specified in the data sheet of the switching chip. This information is needed by the application to implement most of the L2 protocols, but it also causes a problem when the frame is being processed through the Linux IP stack (as the kernel does not know inter-frame-header). To solve this, received frames are being exposed both on a Linux network interface called `ifh` (short for *interface frame header*) and to the `MUX-Filter` (see figure Overall system architecture).

The `MUX-Filter` will *see* all frames being received by the CPU queue in the switching hardware. The driver will decode the frame header to see which classified VLAN a given frame belongs to, and if such an interface exists, then the switch dependent frame header is popped and the frame is being processed by the Linux IP stack. The `MUX-Filter` is configured by the user-space application using the `netlink` protocol, and this configuration channel allows the application to dynamically create and delete IP interfaces that correspond to a VLAN domain. These kinds of interfaces are being referred to as `VLAN interfaces`.

This design allows the user-space applications to implement various L2 protocols and have access to all the side-band data collected by the switch-core, and it also allows existing Linux applications to do various socket operations (IP, UDP and TCP) without changing these applications.

3.2. System services

The WebStaX application will listen on a number of TCP/UDP ports, and it will spawn a number of third-party services. The list of TCP/UDP ports and third-party services depends on the variant (`WebStaX`. `SMBStaX` or `IStaX`). Examples of listing ports are TCP port 23 which the application listens on in order to implement telnet. Examples of third-party services are `hiawatha`, which is being used as web-server and `net-snmp` as `SNMP` main agent.

External services needed by the WebStaX application are automatically started by the application itself. The application also offers configuration hooks that can stop a given service if the user does not wish to use it.

3.3. Boot sequence

The boot-sequence of a WebStaX system differs a bit from what is seen in most *general purpose* Linux systems. There are two main reasons for these differences: a) The system starts by booting from `NOR` and when the kernel is up, it mounts the `NAND` flash as its root file system; b) The system uses a custom `init` process called `ServiceD`.

The following illustrates the boot-process of a WebStaX system:

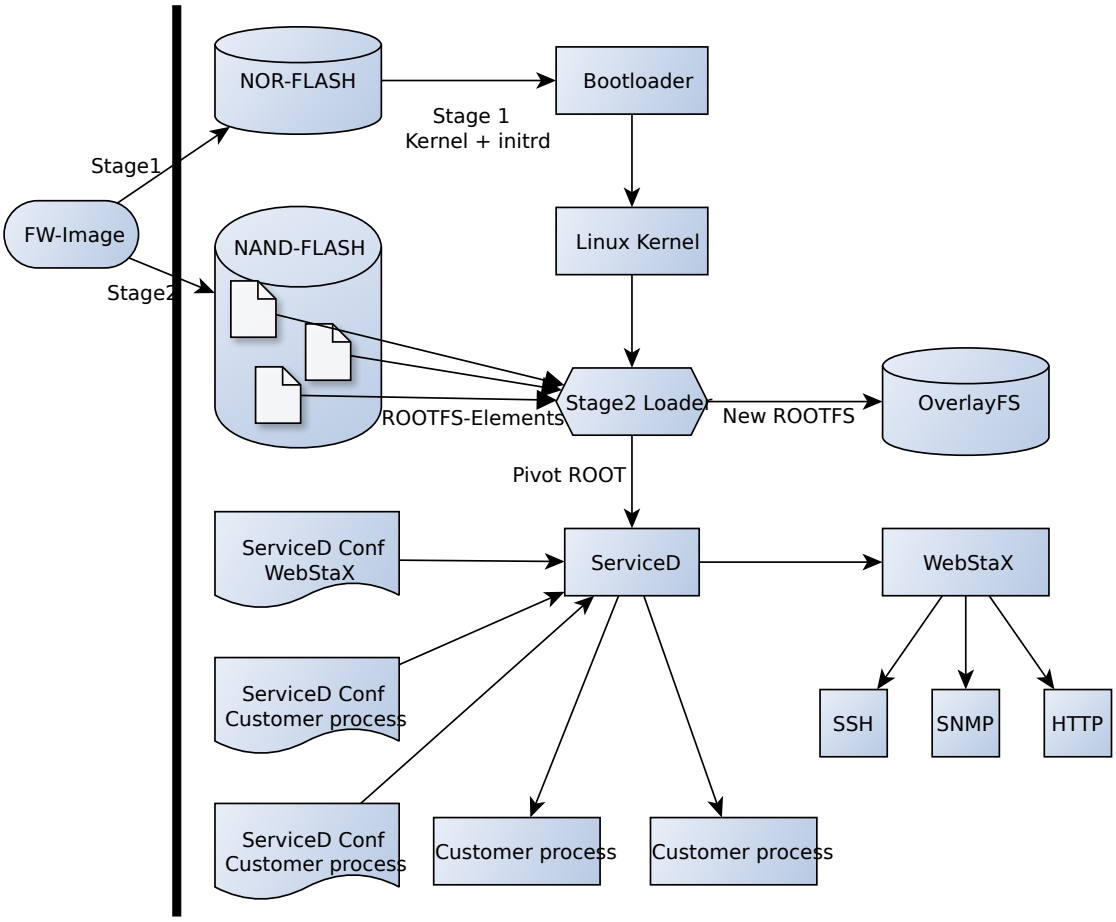

Figure 2. Boot process

3.3.1. The modular image format (`mfi`)

The image format used in WebStaX is called `mfi` (modular firmware images) and it is designed such that more file-system images can be appended. When the system is booted the union of all the appended filesystems is presented as the root-file system.

When booting, the `init` process will iterate through each section in the `mfi` files and mount each root file system element on top of each other by using the `OverlayFS` facilities in the Linux kernel. Once this process is completed, the final root-file system is ready, and the boot process will continue from freshly prepared root file system.

> ? The `mfi` format allow the different root file system elements to be placed in either `NOR` or `NAND` flash.

3.3.2. `ServiceD` as `init process`

At this point the final root file system is ready, and the system can start to initialize all the services that need to be running. The `ServiceD` application is used to perform this task. The `ServiceD` process will read its configuration files (see `ServiceD Conf`

`WebStaX` and `ServiceD Conf Customer process` in Boot process) and spawn (and monitor) the configured services. In a vanilla WebStaX system there will only exist one service called `switch_app` which represents the WebStaX application. When the application is started it will automatic start the set of services it depends on.

`ServiceD` is not the same as `systemd`. `ServiceD` is the `init` process developed by Microsemi and is part of the `mfi` project. See AN1163 for more details.

4. SW Customization Options

WebStax offers a wide range of facilities to allow all kind of customizations. Following is a listing of the often used customization facilities:

4.1. WebStaX application customization

The WebStaX application consists of a number of modules. The building system allows to disable certain modules if needed (inter module dependency exists and needs to be considered when doing so).

Customers can also add modules of their own, and integrate them as part of the existing product. This allow to extend the existing facilities and still provide the end-user with a unified stream line set of interfaces.

4.2. Interface/management customization

All configuration, status and control in the WebStaX application is exposed as a JSON-RPC interface. This JSON-RPC interface can either be accessed via a HTTP(s) connection (allowing remote access/control), or by using an IPC pipe (to control WebStaX from a local process).

This JSON-RPC interface allow creating other interfaces and/or network management systems.

4.3. GUI adjustments/replacement

The WebStaX application include a web front-end written in html/java-script. This interface can be adjusted with other colours, logos etc, or the entire web interface can be replaced with an alternative GUI that uses the JSON-RPC interface.

4.4. Third Party daemons

Third-party daemons can be added to the `mfi` images and started by the `ServiceD-init` process. Such daemons can either be open-source or proprietary. Third-party daemons can use the existing L3 interfaces to communicate with the outside world, and/or us the JSON-IPC facilities to access the switch facilities through the WebStaX application.

4.5. SNMP

By default only MD5 and DES are supported for SNMPv3. To add support for SHA and AES, openssl must be added to the BSP. See [UG1068] "Adding a package" for guidance on how to add a package to the BSP.

5. HW Integration Options

The MSCC switch chips include an internal CPU, which can be used to run the switch application, but it is also possible to do a board design that uses an external CPU instead.

Customers have to choose whether they want to use the internal CPU, or if they prefer an external CPU. Arguments for choosing an external CPU is typically that more CPU resources are needed, or that an alternative CPU architecture is required. The downside of choosing an external CPU is the cost.

Customers that choose to do a project with an external CPU must also provide the BSP for the given project. The MSCC source BSP can be adjusted to support most CPU architectures, or an custom made BSP can be designed from scratch.

The preferred way of reaching the registers from an external CPU is by using PCI-e, and alternatives options do exists (SPI and ethernet).

5.1. Frame flow with an external CPU

Projects using an external CPU need to decide how to implement the frame-flow, between the switch-core and the host CPU. There are two options: either use PCI-Express or dedicate one of the switch ports for the purpose.

6. Alternatives to WebStaX

Customers who for some reasons does not want to use WebStaX are welcome to use an alternative SW-Stack. Such project should integrate the UnifiedAPI/MESA with the alternative SW-Stack, and may optionally be using the BSP.

7. References

UG1068 - SW Introduction to WebStaX under Linux