SW Introduction to WebStaX under Linux

User Guide

UG1068
Rev. 4.3.0
2017-12-15

Confidential

1. Introduction

This document will provide an introduction to the WebStaX Application software (SW) from a SW developer perspective. The intention of this document is to briefly cover what new SW developers need to know, if they are to create a product based on the WebStaX source package. This will include general architecture description, integration options, development environment, customization, installation, and more. The purpose of this is to get the new user started, and not to provide an complete reference.

Starting from the 4.00.01 release, the WebStaX SW stack is running on top of Linux instead of eCos. This document will only cover the newer generation of WebStaX SW which is running on Linux.

Table of Contents

1.1. Audience

The intended audience for this document is SW developers who need to build and/or change the WebStaX source code.

1.2. Prerequisites

This document assumes the reader possesses the following skills/resources:

1. Fluent in C/C++ and to some extent Makefiles. HTML/CSS/JS is required to change the web interface.
2. Root access to a recent Linux development environment, and fairly experienced in working with a Linux shell.
   a. Building new images from the sources (including boot-loader, BSP and application) requires a 64-bit Linux machine with at least 8GB of RAM, 50GB of disk space and 4-8 CPU cores. This document uses Ubuntu 16.04LTS as reference. Ubuntu 14.04LTS is also compatible.
   b. Access to a TFTP and/or HTTP server that can be used for SW upgrades.
   c. RS232 terminal to access the target (needed to debug without IP connectivity).
3. MSCC APPL source package (version 4.00.01 or newer) and the corresponding binary BSP. To change the BSP the BSP source package is also needed.
4. A MSCC reference board supported by the 4.x release (new users are advised to start with a supported reference board and then move to custom boards when the basic environment is configured correctly).

2. Component overview

The WebStaX SW stack consists of a number of different components. All components are needed to build a working WebStaX product. This section will give an overview of the different components and explain what role they fulfil. Projects may need to change/replace one or more of components to support new board types, customization and different integration models.

2.1. 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 host 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. Building the BSP from sources can take a fair amount of time (especially if running in a virtual machine or on old hardware), and MSCC therefore recommends to start out with the binary BSP and use that until modifications are needed.

> The binary BSP is compiled for the internal CPU (little endian MIPSr2). If an alternative CPU is being used, then the BSP needs to be compiled for that CPU.

2.2. API

The API is a library which is 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.

2.3. Application

The WebStaX product family includes four different application packages: `WebStaX`, `SMBStaX`, `IStaX` and `CEServices`. The four packages have different feature sets, and different licensing terms. This document will not be focusing a lot on the individual packages, but it will assume that one of the four packages is being used. When referring to "MSCC-Application", "application" or "switch application" then it is one of these four packages. All examples in this document will be using the `SMBStaX` packages, but all the procedures covered in this document are the same for all packages.

> Customers may choose not to use the application provided by MSCC, but instead use an existing application or write their application from scratch. Such a project will only be using the API, and will need the dedicated API release. This option is out of scope for this document.

2.4. Boot-loader

The boot-loader provides the first SW that is running at the target when it is powered on. The boot-loader is responsible for configuring the CPU, memory controller, loading the Linux kernel into memory and other.

The boot-loader provided by MSCC is based on RedBoot, with a number of patches applied on top. The boot-loader is being distributed both as binary and source, and the binary is built for specific reference boards.

Even the boot-loader is made generic, some custom boards may need to update it. These updates need to be made in the RedBoot sources, and new binaries need to be created from the sources.

> Customers may choose to use alternative boot-loaders, but they will need to add support for the `MFI` image format that is being used by the switch application.

2.5. 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.

Most flash images are *boot-strap images* which means that they include a minimal image that provides just enough functionality to perform a SW upgrade over the network.

> Most reference boards use both the `NOR` and `NAND` flash to store the kernel and root-file system. The `NAND` flash can not be burned using a programmer, and the `NOR` flash is not big enough to include the full application, which is why a boot-strap image is needed. The boot-strap image is small enough to fit into the `NOR` flash and provide enough functionality to perform a SW upgrade over the network to a *full* application (`WebStaX`, `SMBStaX`, `IStaX` or `CEServices`). When doing a SW upgrade the installation process will split the image and will utilize both the `NOR` and `NAND` flash.

3. Integration options

The WebStaX product family is very flexible, and it offers a number of different integration options. This section will describe the most common options that should be considered for new projects.

3.1. MSCC-Application or API-Only

The API is bundled and is part of the switch application packages, but it also exists in a stand-alone package. Customers can therefore choose to use one of the application packages that already include the API, or they can choose to go with the stand-alone API package.

Following is some of the characteristics of a projects based on one of the MSCC-Application variants vs. API-Only projects:

MSCC-Application

- Provides a complete turnkey-like application with cli/web/snmp management interfaces.
- Complete high-level JSON-RPC interface.
- Implements many L2/L3 protocols (the set of protocols depend on the variant).
- Proprietary license.

API-Only

- Driver like functionality
    - Does not implement any protocols and does not perform any *network I/O*.

- C-library which must be instantiated by an application.
- Permissive license (MIT).

This document will only focus on projects that use one of the application variants.

3.2. Internal CPU or External CPU

The MSCC switch chips include an internal MIPS 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 MIPS 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 alternative BSP can be designed from scratch.

3.2.1. 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 (and configure) one of the switch ports as the CPU port. This is called a `NPI` (Node Processor Interface) port. This section documents some of the pros and cons; the details depend on the switch chip and can be found in the data sheets.

3.2.1.1. CPU frames over PCI-E

Frames can be extracted and injected by reading/writing registers exposed in the switch core. This register access is typically done over PCI-E but can in theory also be done using other physical interfaces.

This approach will require a kernel driver (or user-space application using the `tun` / `tap` facilities) to implement a `NIC` interface that will read/write from/to the CPU queue registers. The `NIC` driver must expose the frames as is, including the internal frame header. The `MUX` driver will connect to this `NIC` interface and decode the `ifh` header. An interrupt will indicate when there are frames to be read.

The advantages of this approach is that it is simple, it does not require any dedicated hardware, and it does not consume one of the switch ports. The downside is that the frame-flow may affect the CPU performance as the CPU is being used to read/write the frames.

3.2.1.2. Dedicated `NPI` port

The switch core can be configured to dedicate one of the switch ports as `NPI` port. This means that the frame flow between the CPU and switch-core is a normal ethernet connection.

> Some chips are in some configurations using the `FCS` to carry certain information. This may be an issue if using a dedicated `NPI` port to implement the frame flow between switch core and host CPU.

This approach will require that the host CPU has a free `MAC` interface that can be used to connect to the `NPI` port of the switch-core. The `MAC` interface must be supported by a Linux `NIC` driver.

The advantages of this approach is that the performance depends on the host `MAC` interface (and the associated driver) which may be better than running over PCI-E. The downside is that this solution consumes a switch port from the switch core that cannot be used for anything else, and it requires a free `MAC` on the host CPU.

3.3. MSCC-BSP or custom BSP

A BSP which provides a tool-chain, host tools, and various target libraries/applications is required to build the MSCC switch application. MSCC encourages customers to use the MSCC-BSP as it is designed for and tested with the MSCC Ethernet products. But customers are welcome to use an alternative BSP or create one from scratch. Typical arguments for using alternative BSPs are strong preference to other embedded distributions like Yocto, T2-SDK, Gentoo, etc, or having an existing BSP with support for an external CPU which is intended for the project.

MSCC is in general not supporting customers in integrating the MSCC application into custom BSP's - customers that choose this path must therefore be able to do this on their own. Customers that choose to design their own BSP need to look at the MSCC BSP to get the list of packages and patches used by the MSCC switch application.

4. Brief system architecture

This section will provide a brief overview of the system architecture. The section will focus on how the MSCC switch application has been integrated with the Linux system, and on how third-party components may interact. The image below illustrates the overall system architecture.

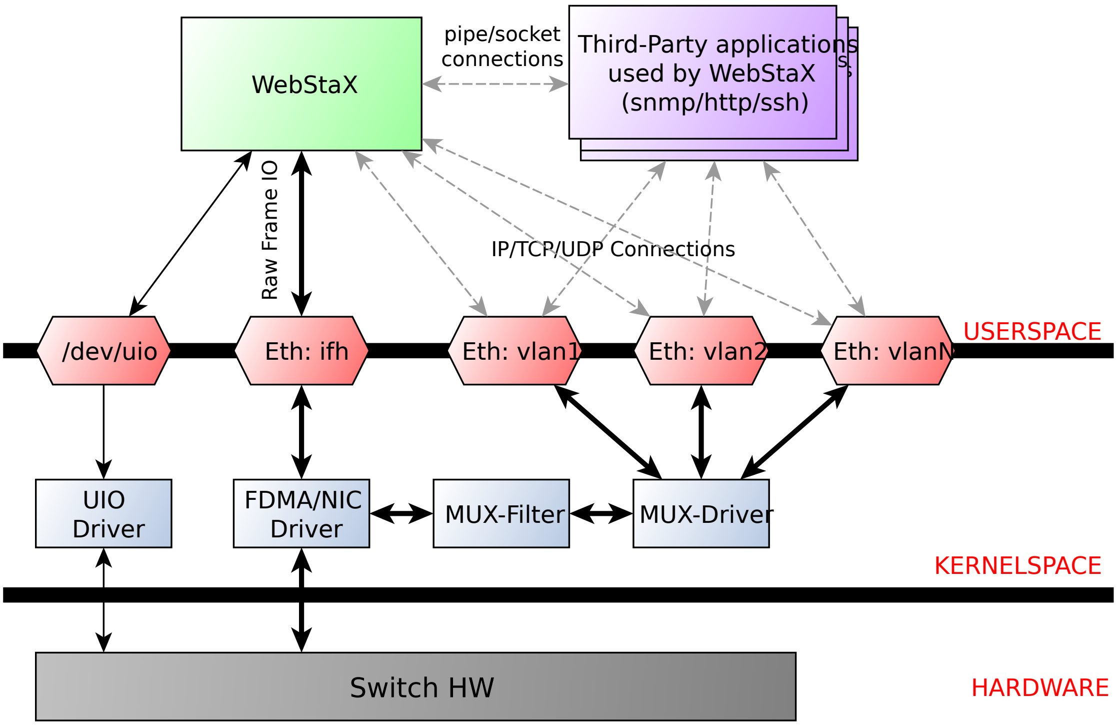

Figure 1. Overall system architecture

The green box labelled WebStaX is the MSCC switch application, it can be any of the supported variants (`WebStaX`, `SMBStaX`, `IStaX` or `CEServices`). 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, and 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.

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.

4.1. 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.

> The FDMA driver included in the MSCC BSP only supports the internal CPU. Projects that uses an external CPU need to provide a NIC driver that will connect the CPU queue in the switching hardware with a Linux network interface.

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. 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`.

A system without any configuration will not have any IP interface, but only the `ifh` interface that exposes the raw frames. When a `VLAN interface` is created, a corresponding Linux network interface is created by the `MUX-Driver`.

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.

4.2. 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`, `IStaX` or `CEServices`). 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` master 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.

More advanced configuration of various system services is covered in [AN1163].

4.3. Boot sequence

The boot-sequence of a WebStaX system differs a bit from what is seem in most *general purpose* Linux systems. The 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:

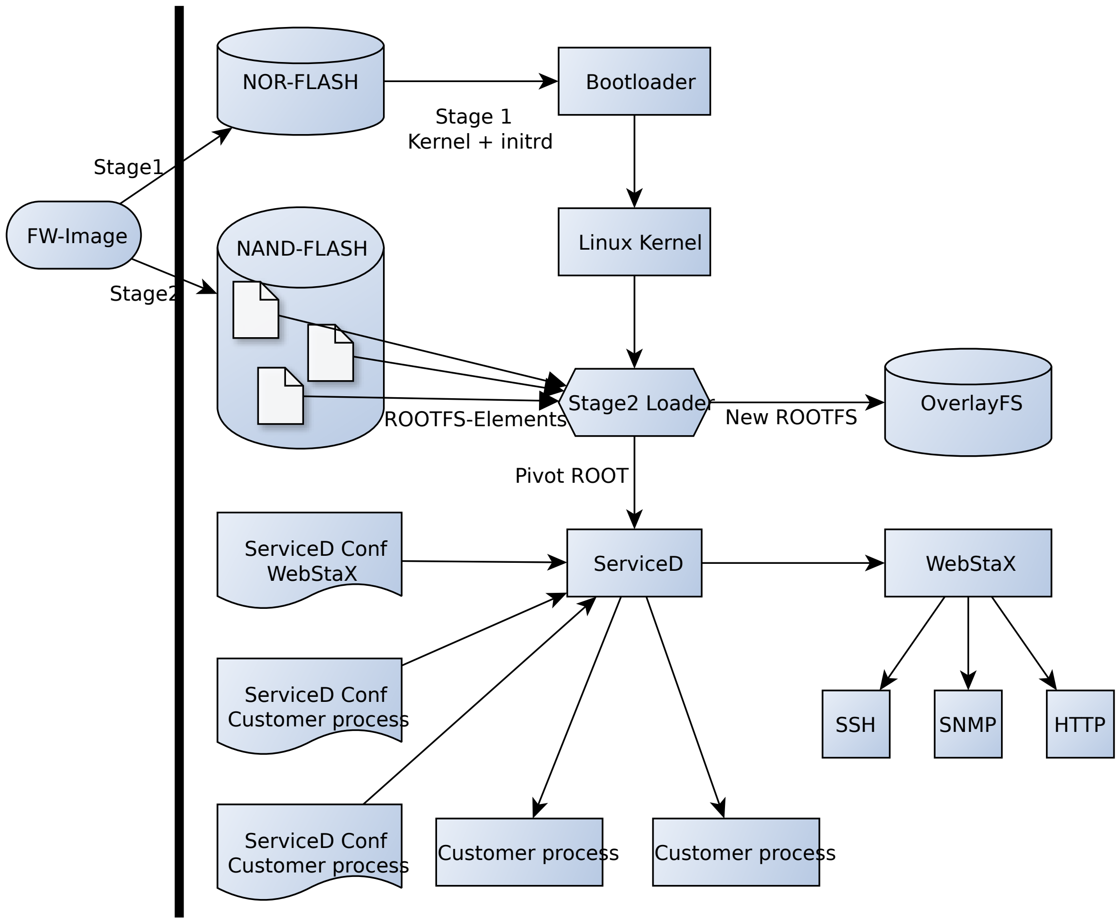

Figure 2. Boot process

4.3.1. Image split

The image format used in WebStaX is called `mfi` and it is designed to allow using both the `NOR` and `NAND` flash to store firmware images. `Redboot` does not have the required drivers to read from the `NAND` flash, meaning that the Linux kernel must be stored in `NOR` flash. When the kernel is booted, it will run the `stage2` loader (also from `NOR`) which will mount the `NAND` flash, do a `pivot_root` and use the `NAND` flash as the root file system from this point on.

This design is a bit different from what is seen in many embedded systems where the entire root file system is placed in the `initrd` section loaded by the boot-loader. The reason for this design is that the `NAND` flash is significantly cheaper than the `NOR` flash, and splitting the image into both `NOR` and `NAND` will lower the BOM cost.

This means that the actual boot process starts already when the image is being installed, because the installation process must split the image file and burn the `kernel` + `initrd` section to the `NOR` flash, and also burn the remaining part to `NAND` flash. The SW upgrade facilities, which are part of the MSCC-Application, will take care of that automatically. This is illustrated at the left side in Boot process image.

When the system is powered on, the boot-loader will initialize the hardware and load the `kernel` + `initrd` into memory, and start the Linux kernel. When the Linux kernel is up and running, it will look for an executable called `stage2-loader` (part of the `stage1` part of the BSP) in the `initrd` area, and invoke that process as `PID` 1. The `stage2-loader` loader will mount the `NAND` flash, and look for the corresponding `stage2` section of the current firmware image in the `NAND`. After finding it, it will iterate through its contents, 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 `stage2-loader` will use the `pivot_root` to replace the existing root with the newly prepared root file system.

4.3.2. `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.

For more details on the `mfi` format and `ServiceD` read the [AN1163] document.

5. Installing SW on a target

This section describes how to install SW into a target, whether that is a 'fresh' installation, i.e. install on a target with an empty NOR device, or a SW upgrade of an existing installation. The two processes are different, hence they are covered separately.

5.1. Installing SW from scratch - How to flash a board

If the device has no SW installed in it already, e.g. empty `NOR` or if a SW upgrade is not possible (e.g. upgrade from an eCos version to this Linux release), then the device needs to be flashed with a flash image. Flash images is part of the normal WebStaX release, and can be used with the reference boards. To build custom flash images see section: Custom flash images.

The clean installation is a two step process; first the `NOR` memory of the device needs to be flashed with a proper binary image (the *flash image*) that will bring the device into a *bring-up* state with basic network connectivity and then the device needs to be bootstrapped with the final *full* application (`WebStaX`, `SMBStaX`, `IStaX` or `CEServices`).

5.1.1. Flashing the NOR with a flash image

In order to flash the `NOR` memory of the device, a flash memory programmer is required. In order to generate the following guidelines and examples, the **FORTE** (http://www.asix.net/prg_forte.htm) memory programmer from **ASIX** (http://www.asix.net/) was used. Other memory programmers will work as well, but covering their installation methods is out of the scope of this document.

Things you will need:

- A board as a target
- A flash memory programmer (**FORTE** (http://www.asix.net/prg_forte.htm) is recommended, **PRESTO** (http://www.asix.net/prg_presto.htm) is slower but works too)
- A PC running Windows
- Universal Programmer tool (downloaded from **ASIX.net** (http://www.asix.net/dwnld_up.htm)) installed
- Binary flash image for the specific board

Once all the hardware is in place and all the drivers and software is installed, go ahead and start the ASIX UP program. You will then be prompted with the following screen that allows you to select and connect to your programmer. Please select the programmer you are using and the proper port. If you check "Always use this S/N", then you will no longer see this initial screen, but you can always select programmers from within UP (Options > Select Programmer, or press Shift + F4 on the keyboard).

Figure 3. Start-up screen of ASIX UP

Next, you have to select the NOR device you are about to program. If it is the first time you start UP you will also see the following screen where you can select the device you want to program. Otherwise, you will be redirected to the main screen of the software. You can always select another device through Device > Select device (or press F4 on the keyboard).

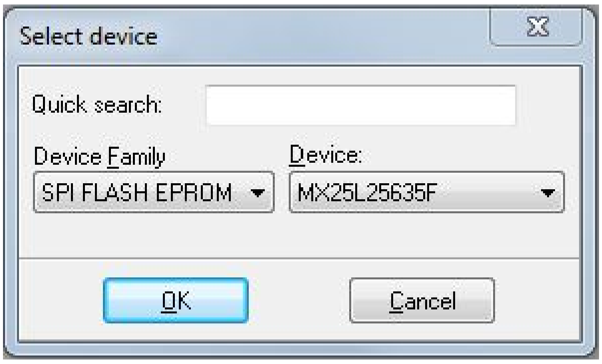

Figure 4. Flash device selection screen of ASIX UP

The Device Family should be *SPI FLASH EPROM*, and the Device ID depends on the respective NOR flash the target is equipped with. Below you can see a table indicating a few of the NOR flash devices that can be found on the MSCC reference boards.

> The table is only listing a few of the most commonly NOR flashes that are used in the MSCC reference boards. Make sure to check the NOR Part No. on your device before performing the flash procedure that is outlined in this section.

Table 1. Flash memory table

MSCC reference board (family name)	NOR Flash Part No.	Device name in ASIX UP	Binary image name
Serval-1	25P28V6P	M25P128	linux-serval1-16mb-256kb.bin
Serval-1	MX25L12835FMI	MX25L12835F	linux-serval1-16mb-64kb.bin
Caracal-1	25P28V6P	M25P128	linux-caracal1-16mb-256kb.bin
Serval-2	MX25L25635F	MX25L25635F	linux-serval2-32mb-64kb.bin
Jaguar-2	MX25L25635F	MX25L25635F	linux-jaguar2-cu8sfp16-32mb-64kb.bin
Serval-T	MX25L25635FMI	MX25L25635F	linux-servalt-32mb-64kb.bin

As seen from the table above, the Device name in ASIX UP is not always the same or even similar to the Device Part No., and in that case the Device name can usually be derived by the data-sheet of the `NOR` device. This will be necessary for applications where customers create their own board.

For MSCC reference boards though, the above table also provides the mapping to the appropriate binary flash image. The APPL-4.0 package (`WebStaX`, `SMBStaX`, `IStaX` or `CEServices`) contains the directory **flash-images** where binary flash images can be found for all MSCC reference boards. The right-most column of the above table indicates the right image for each reference board.

With that in mind, open the right binary through File > Open (or press Ctrl + O on the keyboard). You are now in the main screen of the program and you should see something similar to the following:

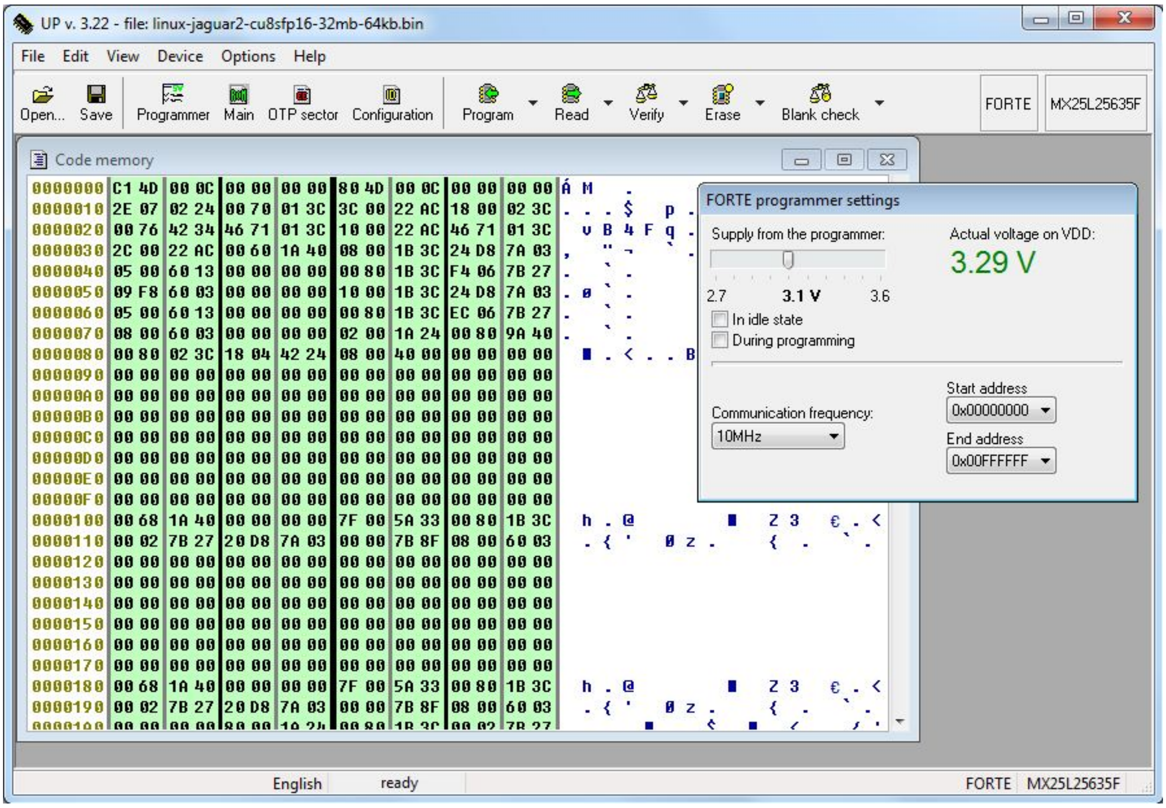

Figure 5. Main screen of ASIX UP

The version of UP along with the loaded binary image can be seen on the top-left corner, while the memory programmer (in this case FORTE) along with the selected Device can be seen on the top-right corner. One thing to notice here is the voltage of the flash device, and the expected value for MSCC ref. boards is something in the range of 3.1V - 3.3V.

The first time you use the software, you can also set your preferred program setting under Options > Program settings (Shift + F10). Those will be kept across. We suggest to check the "Do not perform blank check after erasing" option if you want to speed-up the process. The next figure shows a possible configuration:

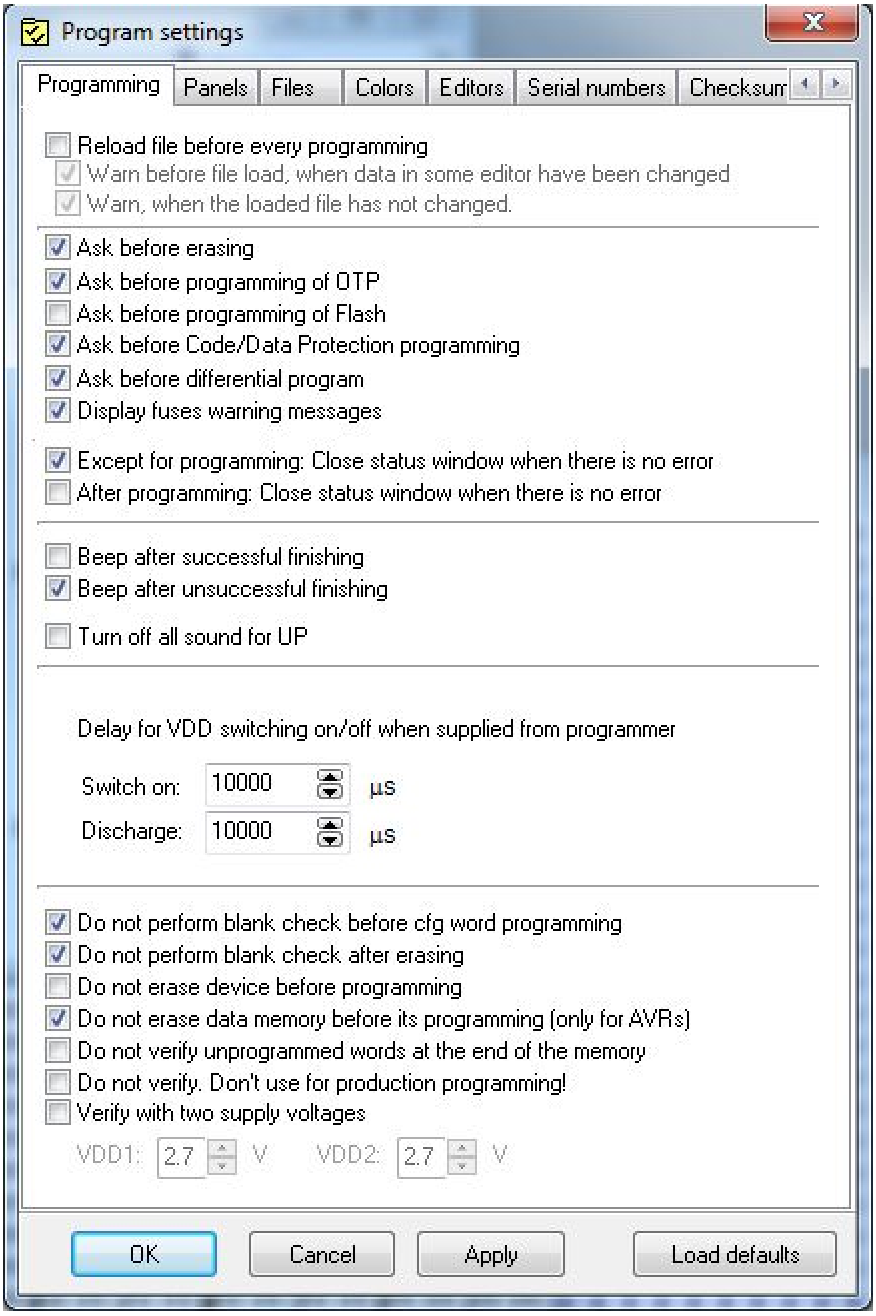

Figure 6. Program settings screen of ASIX UP

You are ready to flash your device now, so click on Device > Program > Program all except OTP sector / Program all (depending on the NOR, you might be presented with more than one option). The process should start (you might get a confirmation pop-up first) and you will see some progress bars.

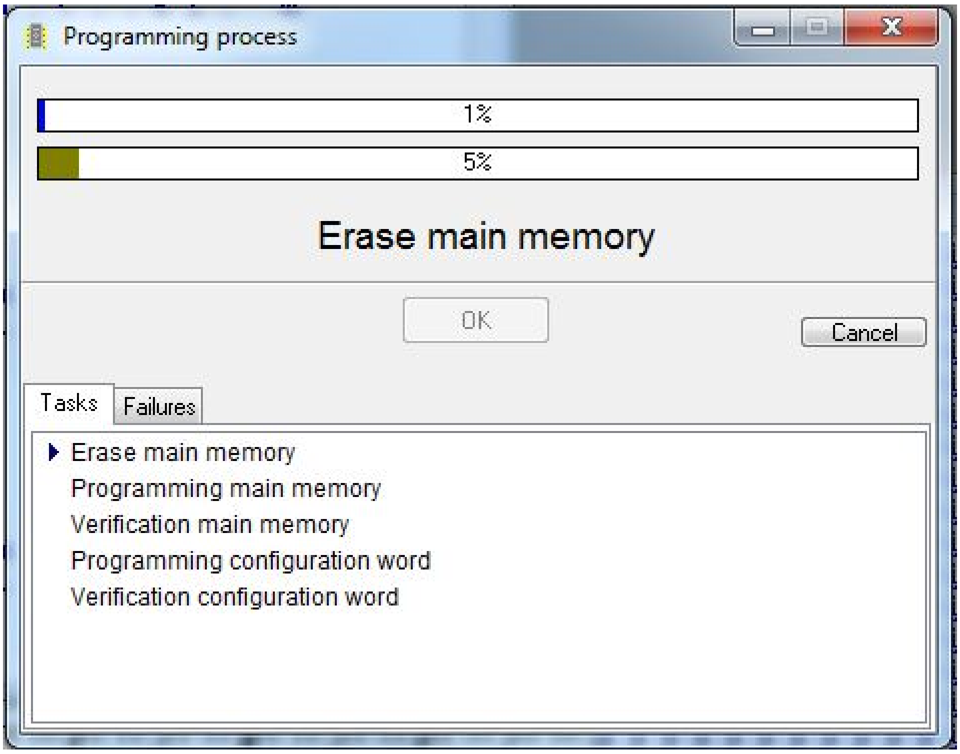

Figure 7. Programming process screen of ASIX UP

When the process is finished, you should get the following screen and no errors or warnings.

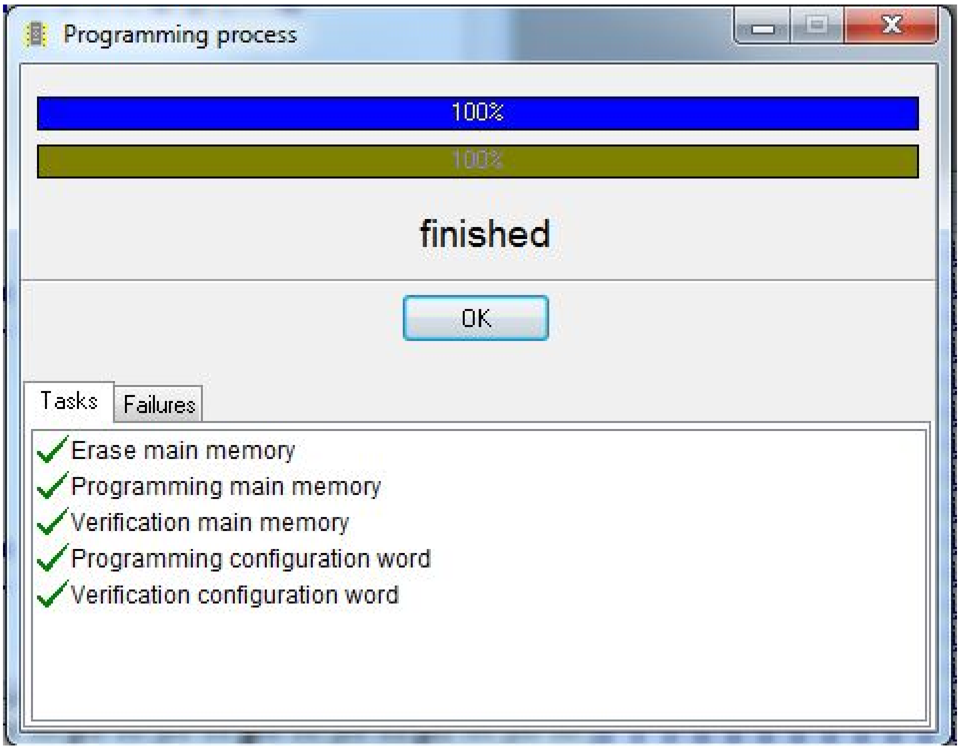

Figure 8. Programming process screen of ASIX UP - successful programming

The device is now flashed with a bring-up image, but before it can be put to use one final step is required. The MAC address of the board has been reset, and the board will pick-up a random MAC address the next time it powers up. You need to change that by making a RS232 connection to the device and issuing the following commands on ICLI (Industrial Command Line Interface): *platform debug allow* and *debug board mac <mac-address>*. Then reboot the device and the flashing process is complete.

> The MAC address given is the device `BASE` address. You implicitly should reserve the next *N* addresses for the device as well. *N* depends on the number of physical ports on the device in question.

Let's take a Serval-1 reference board. Here's the output from the device's first boot after the flash process:

```
+M25PXX : Init device with JEDEC ID 0xC22018.
Serval Reference board detected (VSC7418 Rev. B).

RedBoot(tm) bootstrap and debug environment [ROMRAM]
Non-certified release, version 1_19-5f9ed7e - built 13:31:17, Jun 17 2016

Copyright (C) 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009
Free Software Foundation, Inc.
RedBoot is free software, covered by the eCos license, derived from the
GNU General Public License. You are welcome to change it and/or distribute
copies of it under certain conditions. Under the license terms, RedBoot's
source code and full license terms must have been made available to you.
Redboot comes with ABSOLUTELY NO WARRANTY.

Platform: VCore-III (MIPS32 24KEc) SERVAL
RAM: 0x80000000-0x88000000 [0x800292c0-0x87fdfffc available]
FLASH: 0x40000000-0x40ffffff, 256 x 0x10000 blocks
== Executing boot script in 3.000 seconds - enter ^C to abort
RedBoot> diag -p
RedBoot> fis load -x linux
MD5 signature validated
Stage1: 0x80100000, length 4311824 bytes
Initrd: 0x80600000, length 188416 bytes
Kernel command line: init=/usr/bin/stage2-loader loglevel=4
RedBoot> exec
Now booting linux kernel:
 Base address 0x80080000 Entry 0x80100000
 Cmdline : init=/usr/bin/stage2-loader loglevel=4
 Active fis: linux
[    0.884288] vcfw_uio vcfw_uio: UIO driver loading
[    0.889189] vcfw_uio vcfw_uio: Invalid memory resource
[    0.894392] iounmap: bad address   (null)
00:00:01 Stage 1 booted
00:00:01 Using device: /dev/mtd7
00:00:10 Mounted /dev/mtd7
00:00:10 Loading stage2 from NOR flash partition 'linux'
00:00:12 Overall: 11669 ms, ubifs = 9590 ms, rootfs 2016 ms of which xz = 0 ms of
which untar = 0 ms
Starting application...
Using existing mount point for /switch/
W conf 00:00:16 65/conf_board_start#385: Warning: MAC address not set, using random:
02-00-c1-75-c2-83
Press ENTER to get started
```

The device has selected a random MAC address after the flash process. We now use the debug command for setting the board's MAC address and then reboot.

```
# platform debug allow

WARNING: The use of 'debug' commands may negatively impact system behavior.
Do not enable unless instructed to. (Use 'platform debug deny' to disable
debug commands.)

NOTE: 'debug' command syntax, semantics and behavior are subject to change
without notice.

# debug board mac 00-01-C1-00-C9-90
# reload cold
% Cold reload in progress, please stand by.
Rebooting system...
# Umount done.[  166.748728] VcoreIII I2C: Disabling with active transfer pending

[  166.784052] reboot: Restarting system
```

5.1.2. Bootstrapping

After the `NOR` has been flashed with the steps outlined in the previous section, the flash is partitioned, the boot-loader, Linux kernel and initramfs are installed and for MSCC reference boards a bring-up application is present which allows for basic network connectivity. Note that at this stage the NAND flash still needs to be formatted and partitioned before it is put into use. The bootstrap option that is part of the bring-up application will seamlessly take care of that, plus perform a SW upgrade to the selected APPL-4.0 package (`WebStaX`, `SMBStaX`, `IStaX` or `CEServices`).

In order to demonstrate how the bootstrapping process works, we take the example of a Serval-1 reference board that has been flashed using the method explained in Flashing the NOR with a flash image.

Things needed to perform the bootstrap:

- The bootstrap option is only available through the ICLI management interface, therefore a terminal connection to the device is required.
- Basic network connectivity from/to the device is also needed since we are going to be downloading one of the APPL-4.0 packages into the device.
- An APPL-4.0 SW image (`WebStaX`, `SMBStaX`, `IStaX` or `CEServices`). Customers can build this image themselves through the build system and by following the process explained in section Setting up development environment. For reference boards however, the released package already contains images for all MSCC reference boards. These can be found in `/bin/` of the respective release package. For this example we will be using a SMBStaX image taken from `SMBStaX-4.00.01/bin/smb_serval/smb_serval.mfi`
- An HTTP or TFTP server for distributing the above image.

Having all that in place, we simply log in to the device through ICLI and issue the *debug firmware bootstrap <url>* command.

> You may need to set-up IP configuration properly on device to upgrade bootstrap firmware.

```
Press ENTER to get started
# platform debug allow

WARNING: The use of 'debug' commands may negatively impact system behavior.
Do not enable unless instructed to. (Use 'platform debug deny' to disable
debug commands.)

NOTE: 'debug' command syntax, semantics and behavior are subject to change
without notice.

# debug firmware bootstrap http://10.10.130.147:8080/smb_serval.mfi
Fetching...
looking up 10.10.130.147
connecting non-blocking to 10.10.130.147:8080
connection: Success
requesting http://10.10.130.147:8080/smb_serval.mfi
Bootstrap ubi starts...
ubiformat: mtd7 (nand), size 134217728 bytes (128.0 MiB), 1024 eraseblocks of 131072
bytes (128.0 KiB),
min. I/O size 2048 bytes
libscan: scanning eraseblock 1023 -- 100 % complete
ubiformat: 1024 eraseblocks have valid erase counter, mean value is 3
ubiformat: formatting eraseblock 1023 -- 100 % complete
Bootstrap ubi done ok.
Writing primary image
Erasing flash ... done
Programming flash ... done
  Done
Writing backup image
Erasing flash ... done
Programming flash ... done
  Done
Rebooting ...
[ 7757.970262] VcoreIII I2C: Disabling with active transfer pending
[ 7758.003073] reboot: Restarting system
...
// Skipping standard boot output
...
Press ENTER to get started

Username: admin
Password:
# show version

MAC Address      : 00-01-c1-00-c9-90
Previous Restart : Cold

System Contact   :
System Name      :
System Location  :
System Time      : 1970-01-01T00:17:19+00:00
System Uptime    : 00:17:19


Bootloader
```

```
----------
Image           : RedBoot (bootloader)
Version         : version 1_19-5f9ed7e
Date            : 13:31:17, Jun 17 2016

Active Image
------------
Image           : linux (primary)
Version         : Version 4.00.01
Date            : 2016-07-06T11:23:33+02:00
Upload filename  : smb_serval.mfi

Backup Image
------------
Image           : linux.bk (backup)
Version         : Version 4.00.01
Date            : 2016-07-06T11:23:33+02:00
Upload filename  : smb_serval.mfi

------------------
SID : 1
------------------
Chipset ID       : VSC7418
Board Type       : Serval PCB106
Port Count       : 11
Product          : Vitesse SMBStaX Switch
Software Version : SMBStaX (standalone) Version 4.00.01 Build 272
Build Date       : 2016-07-06T11:23:33+02:00
Code Revision    : 82e4c3f
```

The bootstrap process has now formatted and partitioned the NAND flash, plus installed the selected Application image to both NOR and NAND. The same image is both the active and the backup image of the device.

5.2. Upgrading SW from within an existing installation

Performing a SW upgrade from within APPL-4.0 is similar to the bootstrap process, with the difference that it is supported by all management interfaces. I.e. ICLI, Web, JSON-RPC and SNMP. For simplicity reasons, this document will only focus on the ICLI interface. The rest of the prerequisites listed in the bootstrap section (Basic Upgrade Requirements) remain the same.

After having all the above in place, simply log in to the device and issue the *firmware upgrade <url>* command as also seen in the example below:

```
Press ENTER to get started

Username: admin
Password:
# firmware upgrade http://10.10.130.147:8080/smb_serval.mfi
Fetching...
looking up 10.10.130.147
connecting non-blocking to 10.10.130.147:8080
connection: Success
requesting http://10.10.130.147:8080/smb_serval.mfi
Got 8936650 bytes
Starting flash update - do not power off device!
Erasing flash...done
Programming flash...done
Swapping images...done
Restarting, please wait...Umount failed: D[  450.658665] VcoreIII I2C: Disabling
with active transfer pending
evice or resource busy, retry with force
Umount failed again: Device or resource busy!!!
[  450.695431] reboot: Restarting system
...
//Skipping standard boot output
...
Press ENTER to get started

Username: admin
Password:
# show version

MAC Address      : 00-01-c1-00-c9-90
Previous Restart : Cold

System Contact   :
System Name      :
System Location  :
System Time      : 1970-01-01T00:01:43+00:00
System Uptime    : 00:01:43


Bootloader
----------
Image            : RedBoot (bootloader)
Version          : version 1_19-5f9ed7e
Date             : 13:31:17, Jun 17 2016

Active Image
------------
Image            : linux (primary)
Version          : dev-build by vkosteas@soft-dev10 2016-08-19T14:19:11+02:00
Config:smb_serval SDK:v02.32-smb
Date             : 2016-08-19T14:19:11+02:00
Upload filename  : smb_serval.mfi

Backup Image
------------
```

```
Image            : linux.bk (backup)
Version          : Version 4.00.01
Date             : 2016-07-06T11:23:33+02:00
Upload filename  : smb_serval.mfi

------------------
SID : 1
------------------
Chipset ID       : VSC7418
Board Type       : Serval PCB106
Port Count       : 11
Product          : Vitesse SMBStaX Switch
Software Version : SMBStaXdev-build by vkosteas@soft-dev10 2016-08-19T14:19:11+02:00
Config:smb_serval SDK:v02.32-smb
Build Date       : 2016-08-19T14:19:11+02:00
Code Revision    : a506391+
```

As seen from the example above, after the upgrade is complete the uploaded image has taken its place as active image (in this case a development build was used for the test), while the previously active image is now the new backup image.

> The upgrade process outlined above can not be used in order to upgrade an existing customer or MSCC reference board from an eCos installation to a Linux installation. A binary flash image (MSCC provided or customer provided) needs to be flashed in the device first, according to the process explained in Installing SW from scratch - How to flash a board.

6. Setting up development environment

Working with the source code raises some requirements to the development environment. This section will provide instructions on how to set-up a development machine based on x86_64 Ubuntu 16.04LTS installation. Other (recent) Linux distributions can be used, but that is not supported by MSCC. Setting up the development environment requires `root` access through the `sudo` command.

First step is to install a bunch of required packages using the package system provided by Ubuntu:

```
$ sudo apt-get install bc build-essential bzip2 coreutils cpio findutils gawk git grep gzip libc6-i386 libcrypt-openssl-rsa-perl libncurses5-dev patch perl python ruby sed squashfs-tools tcl tar wget libyaml-tiny-perl libcgi-fast-perl
```

Next step is to download and install the binary BSP. This example will be using `2017.02-066` as example, but future releases may depend on newer versions. Section: Customizing the BSP shows the steps to determine which BSP version a given WebStaX release expects to use. The steps below will download, install and test that the installed binaries work:

```
$ cd
$ wget -q http://mscc-ent-open-source.s3-eu-west-1.amazonaws.com/public_root/bsp/mscc-brsdk-mips-2017.02-066.tar.gz
$ sudo mkdir -p /opt/mscc
$ sudo tar xf mscc-brsdk-mips-2017.02-066.tar.gz -C /opt/mscc
$ /opt/mscc/mscc-brsdk-mips-2017.02-066/stage2/smb/x86_64-linux/usr/bin/mipsel-linux-gcc --version
mipsel-linux-gcc.br_real (Buildroot 2016.05-git) 5.3.0
Copyright (C) 2015 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

The final step is to extract the WebStaX sources and build them using the newly installed tool-chain. The WebStaX sources (`WebStaX`, `SMBStaX`, `IStaX` or `CEServices`) are proprietary SW and distribution may differ from customer to customer. Contact your Microsemi support to get instructions on how to get access to the sources.

The outcome of a compilation is a so-called MFI file, which is the image format used by WebStaX-based SW.

For `IStaX` and `CEServices` packages, there are individual per-target build configurations (makefiles), whereas for `WebStaX` and `SMBStaX` packages, there is only one single build configuration makefile covering all supported targets. The following two sections show how to extract and build MFI files for the two flavors.

6.1. Using single-target build configuration makefiles

The snippet below assumes that the `IStaX` variant is used, and that the source archive `IStaX.tgz` has already been downloaded to the home directory.

```
$ cd
$ mkdir istax                                # Create a folder
$ tar -C istax -xf IStaX.tgz                 # Extract the sources
$ cd istax/build                             # Enter the building catalog
$ ln -s configs/istax_serval_tep.mk config.mk # Select configuration to build
$ make -j8                                   # Build the sources
...
$ ls obj/*.mfi # Test that an 'mfi' file was produced
obj/istax_serval_tep.mfi
```

6.2. Using multi-target build configuration makefiles

Starting with version 4.3.0, an application compiled once may be used to create MFI files for multiple targets. This is currently only supported for `WebStaX` and `SMBStaX` packages. The `bringup_multi.mk`, `web_multi.mk` and `smb_multi.mk` are the only build configuration makefiles distributed with these packages.

The snippet below assumes that the `SMBStaX` variant is used, and that the source archive `SMBStaX.tgz` has already been downloaded to the home directory.

```
$ cd
$ mkdir smbstax                       # Create a folder
$ tar -C smbstax -xf SMBStaX.tgz      # Extract the sources
$ cd smbstax/build                    # Enter the building catalog
$ ln -s configs/smb_multi.mk config.mk # Select configuration to build
$ make -j8                            # Build the sources
...
$ ls obj/*.mfi # Test that a range of 'mfi' files were produced
obj/smb_caracal1.mfi  obj/smb_caracal2.mfi    obj/smb_caracal_lite.mfi
obj/smb_jr2_24.mfi    obj/smb_jr2_24_aqr.mfi  obj/smb_jr2_48.mfi
...
```

Use `make list_subtargets` to show the MFI files that a multi target configuration makefile creates, like this:

```
$ cd smbstax/build
$ make list_subtargets
Targets: smb_caracal1.mfi smb_caracal2.mfi smb_caracal_lite.mfi ...
```

Finally, since all MFI files are linked every time a simple `make` is issued, it may be desirable only to build the MFI files that are really needed. In order to do so, follow these steps, which assumes that we are interested only in `smb_caracal2.mfi`:

```
$ cd smbstax/build
$ rm -f obj/*.mfi           # Remove any old MFI files to emphasize our point
$ make -j8 smb_caracal2.mfi # Build only this MFI file
...
$ ls obj/*.mfi              # Test that it's created
obj/smb_caracal2.mfi
```

7. Customizing SW

This section will document how to build the various SW components from sources, and how to change the corresponding sources. The section is intended as a *getting started* guide, and it will focus on documenting work-flow of MCSS developed components (third-party components like Buildroot and RedBoot is documented by the upstream projects).

7.1. Customizing the BSP

The BSP is used to cross-compile the majority of all the third-party components used in the application. Some projects may want to add other third-party components and use those in their customizations of the software. The easiest way to do that, is to customize the BSP provided by MSCC. The BSP is distributed both in binary and source format. To customize the BSP, the BSP sources are needed and must be downloaded.

First step is figure out which version of the BSP matches the application release. To do that, go to the folder with the application sources (this example will use the `SMBStaX` variant in version 4.00.01). The BSP version is specified in `build/make/paths-brsdk.mk` in a variable called `MSCC_SDK_VERSION`. Here is how to find the associated BSP version:

```
$ cd ~/webstax2
$ cat build/make/paths-brsdk.mk  | grep "MSCC_SDK_VERSION "
MSCC_SDK_VERSION       ?= 2017.02-066
```

This tells us that BSP version `2017.02.066` belongs to the given SW release. Download using a browser from: http://mscc-ent-open-source.s3-website-eu-west-1.amazonaws.com or from the command line:

```
$ cd
$ wget -q http://mscc-ent-open-source.s3-eu-west-1.amazonaws.com/public_root/bsp/mscc-brsdk-source-2017.02-066.tar.gz
$ tar -xf mscc-brsdk-source-2017.02-066.tar.gz
```

Before starting to customize the BSP, it is a good idea to check that it compiles without any modifications. Building all stages of the BSP requires a number of steps (and time). The building process is automated by the `./board/vtss/make_binary_release.rb` script.

Here is how to build the BSP for a MIPS target (expect this to take from 20 minutes and up to several hours depending CPU/RAM/Disk resources):

```
$ cd mscc-brsdk-source-2017.02-066
$ ./board/vtss/make_binary_release.rb --arch mips --stage2 smb,minimal --parallel --version 2017.02-066.01
```

Lots of warnings is printed on the screen when compiling the BSP. These are warnings in third-party code, and can be ignored.

For more options in the `./board/vtss/make_binary_release.rb` script, please use `./board/vtss/make_binary_release.rb --help`.

If the build completes successfully, then it will store the resulting binary BSP in the current folder. Lets see if it exists:

```
$ ls
CHANGES
COPYING
Config.in
Config.in.legacy
MSCC-README
Makefile
Makefile.legacy
README
arch
board
boot
configs
dl
docs
fs
linux
mscc-brsdk-mips-2017.02-066.01.tar.gz
output-mips-stage1
output-mips-stage2_minimal
output-mips-stage2_smb
package
support
system
toolchain
```

As we can see above, the build script has packed the binary BSP in `mscc-brsdk-mips-2017.02-066.01.tar.gz`. The three folders `output-mips-*` are the workspaces used by `buildroot`, we will use then later when changing the BSP.

The BSP is now ready to be installed in `/opt/mscc/` and used by the application.

7.1.1. BSP Stages

Before starting to alter the BSP, some background knowledge on the use of stages will be needed. The `./board/vtss/make_binary_release.rb` will build both `stage1` and `stage2` images.

The `stage1` image includes Linux kernels for all supported targets, and the `stage2-loader` which is used to load the `MFI` image and change root to the `NAND` flash. Only the kernels in the `stage1` image are chip dependent.

The `stage2` image include everything but the kernel, it is not chip dependent (only CPU architecture dependent), and it exists in the following variants: `smb`, `minimal` and `debug`. The `smb` variant is being used in all production images (`WebStaX`, `SMBStaX`, `IStaX` and `CEServices`), the `minimal` is only used for bringup/bootstrap images, and the `debug` variant includes the same packages as `smb` but all packages are compiled with debug info.

This means, that in order to add new packages to the image, then changes in `stage2/smb` are required. To add support for new boards/CPUs or to alter the Linux kernel configuration, then changes in `stage1` are required.

7.1.2. Adding a package

This step will assume that the BSP which belongs to the application has been built already, if not then go to section Customizing the BSP and follow the steps there.

The easiest way to alter the packages included in the various BSP stages/variants, is to use the `make menuconfig` configuration tool which is part of `buildroot`. To do this, navigate to the `output-mips-xxx` where `xxx` represents the stage and variant.

As an example, lets add the `iproute2` package to the `stage2_smb` variant:

```
$ cd mscc-brsdk-source-2017.02-066
$ make O=output-mips-stage2_smb menuconfig
```

Use the `curses` menu to navigate to: `Target packages` then `Networking applications`, and now select the `iproute2` package. Exit the configuration tool (remember to save at the end), and build the specific stage to see that it works like expected (call `make` with the `O=xxx` options, but without the `menuconfig` target):

> The configuration changes are only stored in the `output-mips-stage2_smb` folder which will disappear when doing a clean build (which is default in `make_binary_release.rb`). To persist the changes copy the `output-mips-stage2_smb/.config` to `configs/mscc_stage2_smb_defconfig`.

```
$ make O=output-mips-stage2_smb
```

This will re-build the `stage2_smb` variant, and include `iproute2` and all dependencies of `iproute2`.

> The BSP is using `buildroot`, to learn more about build root read the upstream documentation at https://buildroot.org/.

The `iproute2` tool has now been cross compiled for the MIPS CPU, and it is installed in the `output-mips-stage2_smb` folder. But to use this along with the build system used by the application, then it needs to be packed into a BSP package. To do that, we will use the `make_binary_release.rb` script, but this time the `--no-build` option is added to avoid a complete rebuild (actually nothing will be built, it will just make a BSP package, this will only take a few minutes):

```
$ ./board/vtss/make_binary_release.rb --arch mips --stage2 smb,minimal --parallel --version 2017.02-066.01 --no-build
```

To persist the configuration changes, copy the `output-mips-stage2_smb/.config` to `configs/mscc_stage2_smb_defconfig`:

```
$ cp output-mips-stage2_smb/.config configs/mscc_stage2_smb_defconfig
```

The new build including the `iproute2` package is now available in mscc-brsdk-mips-2017.02-066.01.tar.gz.

7.1.3. Using the new BSP

To use the newly install BSP it needs to be installed, and the application needs to be linked with the new BSP.

Installing the new BSP is simply a matter of extracting it in `/opt/mscc`:

```
$ cd mscc-brsdk-source-2017.02-066.01
$ sudo tar -xf mscc-brsdk-mips-2017.02-066.01.tar.gz -C /opt/mscc/
```

To use the new BSP, either set the environment variable `MSCC_SDK_VERSION` or update the default setting in `build/make/paths-brsdk.mk`. In this example we will use the environment variable:

```
$ cd ~/webstax2/build
$ rm -rf obj config.mk # always do a clean build when changing BSP
$ ln -s configs/smb_caracal1.mk config.mk
$ MSCC_SDK_VERSION=2017.02-066.01 make
Using toolchain: /opt/mscc/mscc-brsdk-mips-2017.02-066.01 - mips - smb
...
```

The first line in the output of the make script, is printing what BSP it is pointing to. Use this to double check that it has picked up the newly built BSP.

The resulting `mfi` files will include the `iproute2` command in the `debug system shell`. Try install the image on a target device, and see if the `ip` command works as expected:

```
Press ENTER to get started

Username: admin
Password:
# platform debug allow
# debug system shell
/ # ip --help
/ # ip --help
Usage: ip [ OPTIONS ] OBJECT { COMMAND | help }
       ip [ -force ] -batch filename
where  OBJECT := { link | address | addrlabel | route | rule | neighbor | ntable |
                   tunnel | tuntap | maddress | mroute | mrule | monitor | xfrm |
                   netns | l2tp | fou | tcp_metrics | token | netconf }
       OPTIONS := { -V[ersion] | -s[tatistics] | -d[etails] | -r[esolve] |
                    -h[uman-readable] | -iec |
                    -f[amily] { inet | inet6 | ipx | dnet | mpls | bridge | link } |
                    -4 | -6 | -I | -D | -B | -0 |
                    -l[oops] { maximum-addr-flush-attempts } | -br[ief] |
                    -o[neline] | -t[imestamp] | -ts[hort] | -b[atch] [filename] |
                    -rc[vbuf] [size] | -n[etns] name | -a[ll] | -c[olor]}
```

7.2. Customizing the Linux Kernel

This step will assume that the BSP which belongs to the application has been built already, if not then go to section Customizing the BSP and follow the steps there.

The Linux Kernel is part of the images in `stage1` of the BSP, so its customization will also take place in the `stage1` using the kernel `menuconfig`.

However, the kernel is target specific, and so the customizations must take place in the target specific directory. Let's take the example of the `jaguar2c` family:

```
$ cd mscc-brsdk-source-2017.02-066
$ cd output-mips-stage1/build/mscc-linux-jaguar2c-4b8fb7f/
$ make ARCH=mips CROSS_COMPILE=../../host/usr/bin/mipsel-linux- menuconfig
```

Using the `curses` menu, we can now make target specific customizations to the kernel. But before going any further, some brief background on where the above configurations are stored:

Just like with the `stage2` customization example that was presented in Adding a package, the saved configuration from `menuconfig` is stored in a temporary `.config` file inside the current directory; this will be erased the next time a clean make is performed. The permanent configurations are stored in two places: `mscc-brsdk-source-2017.02-066.01/board/vtss/common/linux.config` where global kernel options are kept and `mscc-brsdk-source-2017.02-066.01/package/mscc-linux-jaguar2c/linux.cfg` where target specific kernel options are kept. When configuring `stage1` the `linux.config` global options are applied first, while the `linux.cfg` options are appended later on top of the global.

Take care that since the target specific options are applied in the end, they will overwrite any contradicting options from the global configuration. E.g. if an option such as `CONFIG_NET` is disabled globally, but still enabled in a target specific file, then network support will still be enabled for that target.

Because the `.config` options file that is auto-generated by `menuconfig` will also contain more options, we recommend not to copy the contents of this file, but instead manually edit the contents of `mscc-brsdk-source-2017.02-066.01/board/vtss/common/linux.config` and `mscc-brsdk-source-2017.02-066.01/package/mscc-linux-jaguar2c/linux.cfg`. That will ensure that only the needed options will be retained.

Let's demonstrate how to add the option of overriding the default kernel command line. Using the `curses` menu, navigate to: `Kernel hacking` then select the `Built-in kernel command line` option and then use `Help` to display the name of the option. This provides us with the name of the `CONFIG_` options we need to set in order to enable the given selection. Exit `menuconfig` without saving any changes, and then append the new option to `linux.cfg` for `jaguar2c` (we only want to change it for this target) and make a new build of the kernel only:

```
// Append the "CONFIG_" option to the target specific config file
$ echo "CONFIG_CMDLINE_BOOL=y" >> ../../../package/mscc-linux-jaguar2c/linux.cfg
// Delete the buildroot stamp in order to trigger a new build without rebuilding the
entire stage1
$ rm output-mips-stage1/build/mscc-linux-jaguar2c-4b8fb7f/.stamp_configured
// Call for a partial build of stage1
$ make O=output-mips-stage1/
```

Next, we make a new package of the BSP that contains the newly built kernel for `jaguar2c`, using the way described in Adding a package:

```
$ ./board/vtss/make_binary_release.rb --arch mips --stage2 smb,minimal --parallel
--version 2017.02-066.01 --no-build
```

7.3. Customizing RedBoot

RedBoot is currently the only boot-loader supported by MSCC. All reference boards come with a pre-installed boot-loader, and all releases include a binary boot-loader image for each of the supported reference boards. This is normally sufficient when just using the reference boards (or custom boards that are compatible with the reference boards).

But some projects need to patch the boot-loader (often because they want to change the output printed, due to changes to the hardware that need to be handled by RedBoot or in order to implement features that can only be done in the boot-loader). In such cases it is necessary to build the boot-loader from sources, patch in the required changes and do a new boot-loader release for the given project.

> Customers are welcome to use other boot-loaders such as `uBoot`, but MSCC does not provide a working reference design based on `uBoot`. Supporting alternative boot-loaders is therefore considered out-of-scope for this document.

7.3.1. Installing required tools

RedBoot is a part of eCos, and it is therefore also using the tool-chain from eCos (not the same tool-chain as the one provided by the BSP). First step in building RedBoot from source is therefore to make sure that the required tools are installed:

```
$ ls /opt/ecos
$ ls /opt/vtss-cross-ecos-mips32-24kec-v2
```

If the `/opt/ecos` does not exists then follow the steps below to install it:

```
$ wget -q http://mscc-ent-open-source.s3-eu-west-1.amazonaws.com/public_root/ecos-toolchain/ecos.tar.bz2
$ sudo tar -xf ecos.tar.bz2 -C /opt
```

If the `/opt/vtss-cross-ecos-mips32-24kec-v2` does not exists then follow the steps below to install it:

```
$ wget -q http://mscc-ent-open-source.s3-eu-west-1.amazonaws.com/public_root/ecos-toolchain/vtss-cross-ecos-mips32-24kec-v2.tar.bz2
$ sudo tar -xf vtss-cross-ecos-mips32-24kec-v2.tar.bz2 -C /opt
```

Check that the required tools are installed and working correctly by invoking one of the tools provided:

```
$ /opt/ecos/ecos-2.0/tools/bin/ecosconfig --version
ecosconfig 2.0 (May  9 2003 09:45:47)
Copyright (c) 2002 Red Hat, Inc.
$ /opt/vtss-cross-ecos-mips32-24kec-v2/bin/mipsel-vtss-elf-gcc --version
mipsel-vtss-elf-gcc (crosstool-NG 1.20.0 - vtss-eCos-toolchain-v2) 4.9.1
Copyright (C) 2014 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

7.3.2. Building RedBoot from sources

Next step is to find the RedBoot sources. They are distributed as a `mscc-redboot-xxxxxxx.tar.gz` (where `xxxxxxx` is the version number) file along with the normal application releases. This example will be using version `5f9ed7e`. Start by extracting the respective tar-ball:

```
$ tar xf mscc-redboot-5f9ed7e.tar.gz
```

The RedBoot build script assumes that the sources reside in a git repository, which therefore must be initialized:

```
$ cd mscc-redboot-5f9ed7e
$ git init
Initialized empty Git repository in ~/mscc-redboot-5f9ed7e/.git/
$ git add .
$ git commit -m "Initial import of version 5f9ed7e"
[master (root-commit) 291b5ac] Initial import of version 5f9ed7e
```

After this, RedBoot is ready to build. Use the `mscc-build.rb` to build the corresponding configuration for your device (expect this to take from a few minutes and up to one hour):

```
$ ./mscc-build.rb --verbose --parallel --machines serval1
```

The `--machines <family_name>` parameter of the script allows for selecting the right chip family among the available options. Options valid for MSCC Application 4.00.01 are: `luton26`, `serval1`, `jaguar2` and `servalt`.

For more options in the `mscc-build.rb` script, please use `./mscc-build.rb --help`.

When the build has completed, then the newly built images are available in the `images` folder (in the above case, only `serval1` image is present):

```
$ ls -la images/
total 984
drwxr-xr-x  2 anielsen epdeng2   4096 Aug 19 10:10 .
drwxr-xr-x 12 anielsen epdeng2   4096 Aug 19 10:10 ..
-rw-r--r--  1 anielsen epdeng2 151128 Aug 19 10:10 redboot-serval1.img
```

7.3.3. Changing the RedBoot sources

RedBoot is now ready for customization. The main sources are found in `packages/redboot/current/src/`, and a good starting point is to read: `packages/redboot/current/src/main.c`.

7.3.4. Installing a new bootloader

To try out the new boot-loader, install it on a target device using the `debug firmware bootloader <url>` command:

```
# platform debug allow
# debug firmware bootloader http://some.ip.address/redboot-machine.img
```

7.4. Customizing the Application

The majority of the WebStaX functionality is implemented in the application, and customizing the application is therefore an important section of this document. This section will document different strategies on how the application can be customized. Many projects may benefit from combining the different customization facilities.

7.4.1. External process

The easiest way to customize the application is to add an external process that will communicate with the existing switch application as it is. How to add new processes (internal developed or third-party) is covered in [AN1163]. [AN1163] also covers how to configure an existing application without having to rebuild it.

7.4.2. Build configurations

A build configuration provides a high-level configuration of the build system with information such as chip type and CPU-architecture. For `IStaX` and `CEServices` packages, each build target must have its own build configuration makefile, which must be located in the `./build/configs/` directory. For `WebStaX` and `SMBStaX` packages, there are only multi-target build configuration makefiles included by default though single-target build configuration files are still supported. These configuration build files are also located in the `./build/configs/` directory. See also Using multi-target build configuration makefiles.

In the following, the home directory is assumed to be the extraction directory of the application source files.

7.4.2.1. Using a build configuration

In order to use a build configuration and build SW for a given target, we first create a `config.mk` symbolic link to the respective configuration (e.g. `bringup_multi.mk`):

```
$ cd build
$ ln -s configs/bringup_multi.mk config.mk
$ make -j8
"Using binary API from .../bin/mips"
Using toolchain: .../mscc-brsdk-mips-2017.02-066 - mscc-brsdk-mips-2017.02-066 -
minimal
[CXX] ../../vtss_appl/board/led.cxx
[CXX] ../../vtss_appl/board/monitor.cxx
[CXX] ../../vtss_appl/board/interrupt.cxx
[CXX] ../../vtss_appl/board/board_subjects.cxx
...
Linking bringup_serval_t.elf
Generate bringup_caracal1.app-rootfs
Generate bringup_caracal2.app-rootfs
Generate bringup_jr2_48.app-rootfs
Generate bringup_ocelot_10.app-rootfs
Generate bringup_jr2_24_aqr.app-rootfs
Generate bringup_jr2_24.app-rootfs
Generate bringup_ocelot_8.app-rootfs
Generate bringup_serval2.app-rootfs
Generate bringup_serval.app-rootfs
Build bringup_caracal1.mfi
Build bringup_caracal2.mfi
Build bringup_ocelot_10.mfi
Generate bringup_serval_t.app-rootfs
Build bringup_jr2_24_aqr.mfi
Build bringup_jr2_24.mfi
make[1]: Leaving directory '.../build/obj'
```

7.4.2.2. Customizing build configurations

Customers implementing their own boards must create a build configuration and add it to the `build/configs/` directory.

The build configuration is used to control the following:

1. The selected Switch API chipset.
2. The choice of MEBA implementation.

   MEBA is the target board application interface (See UG1069 for more information on MEBA).
3. Linux target OS platform name.
4. The choice of kernel-mode board driver.

   This is a kernel-mode board driver that sets up I2C muxing, SFP to port number mapping and exposes board-specific SPI devices.
5. The application flavor (`WebStaX`, `SMBStaX`, `IStaX` or `CEServices`) and possible customizations to these pre-defined profiles.

Currently, there are two different types of board configuration layouts:

1. `single`

The configuration file will only build an image for a single board. This is typically used for CEServices and IStaX profiles.

2. `multi`

   The configuration file will build multiple board images. This is used for all other configurations.

7.4.2.2.1. Defining target configurations

Customizing both `single` and `multi` flavors will require defining a custom target by means of the `DefineNamedTarget` macro.

Example:

```
$(eval $(call DefineNamedTarget,acme,vsc7418_afiot,serval1,acme,nulldrv))
```

This defines the named target board profile `acme`, consisting of the following properties. You can see the list of target profiles in the file `./build/make/templates/targets.in`, along with possible values of the different entities.

In the example, the five parameter values are:

1. Profile name (`acme`).

   A file called `./build/make/meba/meba_<profile_name>.json` defines the MEBA layer configuration.

2. Profile API target (`vsc7418_afiot`).

   This is a named target configuration from the MESA Switch API.

3. Linux kernel target name (`serval1`).

   This selects the Linux kernel used in the MFI image.

4. MEBA target name (`acme`).

   The MEBA library is taken from the API, but may also be customized (see later). The reference drivers are in the MEBA `./vtss_api/meba/src/` directory.

5. MEBA linux kernel driver (`nulldrv`).

   This controls I2C muxes and SFP to port number mappings. The `nulldrv` is a dummy driver. The drivers are in the MEBA `./vtss_api/meba/linux_kernel_modules/` directory. You may use one of the existing drivers if they match your board or you may make your own. In the latter case, this parameter should match the directory name accordingly.

The MEBA layer JSON configuration file typically defines thes board name, but may have additional arbitrary `<key,value>` pairs that can be read from the C-code located in `vtss_api/meba/src/<your_meba_library>/`.

An example of the `acme` MEBA layer configuration file could be:

build/make/meba/meba_acme.json

```
{
    "meba" : {
        "board":"Acme MK2",
        "type":"123",
        "target":"0x7418"
    },
    {"dummy":"end" }
}
```

Once you have a target configuration, you can add this to the list of configurations that will be built by the `DefineTargetByName` macro. This takes just two parameters.

1. The name of the build

   This will control the name of the image file produced: `<name>.mfi`.

2. The name of the target profile (the name from a `DefineNamedTarget` call.)

You may have one or more calls to `DefineNamedTarget`. Each call will just add to the list of configurations that will be built.

7.4.2.2.2. Customizing MEBA layer

As the MEBA layer is the glue layer between the application and your target board, this is most likely to needing customization.

When you are creating a new target configuration, you are providing the MEBA interface name. If you just change the name, the build system will search for a pre-compiled MEBA layer with that name. While it is possible to add a new binary MEBA layer, you will typically be better off by providing the layer as source code, which will get compiled and combined into your build image on the fly.

To do this, you can use the `Custom/MebaSrc_<name>` "make" variable. This variable (with `<name>` matching your custom MEBA layer name) should be a list of source files which can be compiled into a MEBA library. The files can be located anywhere in the source tree, typically relative to the `$(TOPABS)` predefined variable.

Example:

```
Custom/MebaSrc_acme := $(TOPABS)/vtss_appl/meba/meba_acme.c
```

Here, the `acme` MEBA layer will be created from a single source file, and it will be available for creating custom target configurations with the `DefineNamedTarget` macro.

7.4.2.2.3. Controlling application modules

The functionality of the switch application is defined by two factors

1. The application main profile.

   This will be `WebStaX`, `SMBStaX`, `IStaX` or `CEServices`. Each profile is licensed separately, so not all may be available to you.

2. Specifically added or omitted modules.

   Modules added may be one defined by you, or a module normally used in another profile.

The profile is normally selected in the second last line in the configuration file. You should be able to locate the profile name (all caps), and you can change this according to your desire and what profiles you have licensed.

The specific adding and omission of individual modules are controlled by two `make` variables:

- Custom/AddModules
- Custom/OmitModules

You can add a line setting each of these variables (between the first and the last line) as desired. Individual module names are separated by space. You can check the result by executing `make show_modules`.

Note that not all modules may be able to be removed individually, but may tie to other modules. Also be sure to re-compile after changing module configuration (`make clean`)

7.4.2.2.4. Customizing preprocessor variables

You can add custom preprocessor variables by using the `Custom/Defines` make variable. This can enable certain features in default modules (refer to other configuration files) or behavior of modules in your own modules. You set CPP preprocessor options and all application code will have these compiler options added.

Example:

```
Custom/Defines := -DMY_OPTION=1
```

7.4.2.2.5. Customizing single image configuration makefile

Below is the example configuration file `ce_serval_tep.mk`:

```
include $(BUILD)/make/templates/linuxSwitch.in
$(eval $(call DefineTargetByName,ce_serval_tep,serval_tep))
$(eval $(call linuxSwitch/ServalT,CESERVICES,STANDALONE,SERVAL_TEP,brsdk,mips))
$(eval $(call linuxSwitch/Build))
```

The above file does not contain a `DefineNamedTarget` macro - at least it seems. It uses the `serval_tep` name, but that is a profile name that comes with the default set of profiles, which can be found in `./build/make/templates/targets.in`. So, to customize this build file you would add your own target profile (as described earlier), and then use that in the `DefineTargetByName` macro.

For example:

```
1 include $(BUILD)/make/templates/linuxSwitch.in
2 $(eval $(call DefineNamedTarget,acme,vsc7435_aqr,servalt,servalt,acme))
3 $(eval $(call DefineTargetByName,ce_acme,acme))
4 $(eval $(call linuxSwitch/ServalT,CESERVICES,STANDALONE,SERVAL_TEP,brsdk,mips))
5 $(eval $(call linuxSwitch/Build))
```

The above example creates a configuration file to build `ce_acme.mfi` (line 3), using the original configuration of ce_serval_tep (line 2), but renaming the board name by using the previously defined `./build/make/meba/meba_acme.json` file. Also, it will use the `acme` MEBA kernel module driver (directory name `acme`).

7.4.2.2.6. Customizing multi image configuration file

Let us look at an example file:

```
1 include $(BUILD)/make/templates/linuxSwitch.in
2 Custom/MebaSrc_acme := $(TOPABS)/vtss_appl/meba/meba_acme.c
3 $(eval $(call DefineNamedTarget,acme,vsc7418_afiot,serval1,acme,nulldrv))
4 $(eval $(call DefineTargetByName,smb_acme,acme))
5 $(eval $(call linuxSwitch/Multi,SMBSTAX,brsdk,mips))
6 $(eval $(call linuxSwitch/Build))
```

While the example above uses the general layout of the "multi" image configurations, only one image is produced (only one `DefineTargetByName` call).

The above example defines a new MEBA layer `acme` (line 2), which is used in the target configuration of the same name (line 3), and again used to compile an SMBStaX image (line 5) called `smb_acme.mfi` (line 4).

7.4.3. Adding a custom module to the Application

This section provides a detailed description of how to add a new software module to the Application. An example hello_world module will be created along with a trace message indicating proper execution of the module. Finally, the module will be added to all management interfaces.

For the rest of the section, it is assumed that the home directory is the extraction point of the application sources, see Extract APPL sources.

7.4.3.1. Creating a makefile

Each application module has each own makefile, located in `build/make/`. Therefore we create a new file `build/make/module_hello_world.in` with the below content:

build/make/module_hello_world.in

```
MODULE_ID_hello_world := 143 # VTSS_MODULE_ID_HELLO_WORD
DIR_hello_world := $(DIR_APPL)/hello_world
OBJECTS_hello_world := hello_world.o
$(OBJECTS_hello_world): %.o: $(DIR_hello_world)/%.cxx
    $(call compile_cxx,$(MODULE_ID_hello_world), $@, $<)

INCLUDES += -I$(DIR_hello_world)
```

The first character of the `$(call compile_c, $@, $<)` line must be tab and not spaces, per makefile rules.

7.4.3.2. Creating a source directory

Switch application modules are placed in different directories inside `/vtss_appl/`. To add the new module's sources simply create a new directory called `hello_world` and start adding source and header files into it:

```
$ cd vtss_appl/
$ mkdir hello_world
$ vim hello_world/hello_world_api.h
...
$ vim hello_world/hello_world.cxx
...
```

The external header `hello_world_api.h` is necessary since it contains the declaration of the module's `init` function:

vtss_appl/hello_world/hello_world_api.h

```
#ifndef _HELLO_WORLD_API_H_
#define _HELLO_WORLD_API_H_

/* Initialize module */
vtss_rc hello_world_init(vtss_init_data_t *data);

#endif /* _HELLO_WORLD_API_H_ */
```

And the `hello_world` program which is using the standard initialization function template, with a single `printf` statement:

vtss_appl/hello_world/hello_world.cxx

```
#include "main.h"

/* Initialize module */
vtss_rc hello_world_init(vtss_init_data_t *data)
{
  vtss_isid_t isid = data->isid;
  vtss_rc rc = VTSS_OK;

  switch (data->cmd) {
    case INIT_CMD_INIT:
      printf("%s\n", "Hello World!");
      break;
    case INIT_CMD_START:
      break;
    case INIT_CMD_CONF_DEF:
      break;
    case INIT_CMD_MASTER_UP:
      break;
    case INIT_CMD_MASTER_DOWN:
      break;
    case INIT_CMD_SWITCH_ADD:
      break;
    case INIT_CMD_SWITCH_DEL:
      break;
    default:
      break;
  }
  return rc;
}
```

7.4.3.3. Adding the module to the build

With the makefile and the module's sources present, we now add the new module to the build by appending it to the build configuration as described in Build configurations:

```
Custom/AddModules := tod post adt_7476_api hello_world
```

And next we call the module's `init` function through the application's main (`/vtss_appl/main/main.cxx`):

vtss_appl/main/main.cxx

```
#ifdef VTSS_SW_OPTION_HELLO_WORLD
#include "hello_world_api.h"
#endif
```

The `init` call must be placed inside the `initfun` struct.

vtss_appl/main/main.cxx

```
static struct {
    vtss_rc           (*func)(vtss_init_data_t *data);
    const char        *name;
    vtss_tick_count_t max_callback_ticks;
    init_cmd_t        max_callback_cmd;
} initfun[] = {
#ifdef VTSS_SW_OPTION_HELLO_WORLD
    INITFUN(hello_world_init)
#endif
}
```

Assign the module a unique module ID, that has to be added in 2 places:

vtss_appl/include/vtss/appl/module_id.h

```
/** Module IDs
 * !!!!! IMPORTANT !!!!!
 * ---------------------
 * When adding new module IDs, these MUST be added at the end of the current
 * list. Also module IDs MUST NEVER be deleted from the list.
 * This is necessary to ensure that the Msg protocol can rely on consistent
 * module IDs between different SW versions.
 */
enum {
    /* Switch API */
    VTSS_MODULE_ID_API_IO                =   0,  /* API I/O Layer */
    VTSS_MODULE_ID_API_CI                =   1,  /* API Chip Interface Layer */
    VTSS_MODULE_ID_API_AI                =   2,  /* API Application Interface Layer */
    VTSS_MODULE_ID_SPROUT                =   3,  /* SPROUT (3) */
    VTSS_MODULE_ID_MAIN                  =   4,
    ...
    VTSS_MODULE_ID_HELLO_WORLD           = 143,

    /*
     * INSERT NEW MODULE IDS HERE. AND ONLY HERE!!!
     *
     * REMEMBER to add a new entry in the module id database on our twiki
     * before adding the entry here!!!
     *
     * Assign the module ID number from the database to the enum value here
     * like shown in VTSS_MODULE_ID_DHCP_SERVER above.
     * This will allow for 'holes' in the enum ranges on different products/branches.
     *
     * REMEMBER ALSO TO ADD ENTRY IN \vtss_appl\util\vtss_module_id.c\vtss_module_names[] !!!
     * REMEMBER ALSO TO ADD ENTRY IN \vtss_appl\util\vtss_module_id.c vtss_priv_lvl_groups_filter[] !!!
     */

    /* Last entry, default */
    VTSS_MODULE_ID_NONE
};
```

vtss_appl/util/vtss_module_id.cxx

```
#include "vtss_module_id.h"

/* These module name will shown as privilege group name.
   Please don't use space in module name, use under line instead of it.
   The module name can be used as a command keyword. */
const char * const vtss_module_names[VTSS_MODULE_ID_NONE + 1] =
{
    [VTSS_MODULE_ID_API_IO]          /*   0 */ = "obsolete_api_io",
    [VTSS_MODULE_ID_API_CI]          /*   1 */ = "api_cil",
    [VTSS_MODULE_ID_API_AI]          /*   2 */ = "api_ail",
    [VTSS_MODULE_ID_SPROUT]          /*   3 */ = "sprout",
    [VTSS_MODULE_ID_MAIN]            /*   4 */ = "main",
    ...
    [VTSS_MODULE_ID_HELLO_WORLD]     /* 143 */ = "Hello_World",

    /* Add new module name above it. And please don't use space
       in module name, use underscore instead. */
    [VTSS_MODULE_ID_NONE]                      = "none"
};
```

Perform a **make** and test that the new module is included in the build:

```
$ touch vtss_appl/hello_world/hello_world.cxx
$ make -C build
Using toolchain: /opt/mscc/mscc-brsdk-mips-2017.02-066 - mips - smb
...
[CXX] ../../vtss_appl/hello_world/hello_world.cxx
...
```

And then upgrade the device with the newly built .mfi image. Check that the new module is added successfully by observing the `Hello World!` message:

```
00:00:01 Stage 1 booted
00:00:01 Using device: /dev/mtd7
00:00:09 Mounted /dev/mtd7
00:00:09 Loading stage2 from RAM
00:00:10 Stage2 ends at 0x76c4ece6, offset 00874ce6
00:00:10 Overall: 9553 ms, ubifs = 8029 ms, rootfs 1454 ms of which xz = 0 ms of which untar = 0 ms
Starting application...
Using existing mount point for /switch/
Hello World!

Press ENTER to get started
```

7.4.3.4. Adding management interfaces

The MSCC application stack includes four different management interface (`ICLI`, `SNMP`, `JSON-RPC` and `Web`). When adding new modules, it is often necessary to add new commands (or objects) in the management interfaces. Most modules provided by MSCC are fully supported on all management interfaces, but projects that are adding new modules only need to implement the interfaces they need. This section covers basic examples of how to add a new management interface to a module, e.g. for a new custom module. We will be improving on top of the existing `hello_world` custom module.

7.4.3.4.1. `ICLI`

`ICLI` is the command line interfaces that users are presented with when logging into using `rs232`, `telnet` or `ssh`. This section will show how to create a simple `ICLI` command.

`ICLI` commands are traditionally implemented inside the module directory, and called `<module_name>.icli`.

A very simple `ICLI` file could look something like this:

vtss_appl/hello_world/hello_world.icli

```
CMD_BEGIN
COMMAND = hello world
PRIVILEGE = ICLI_PRIVILEGE_15
CMD_MODE = ICLI_CMD_MODE_EXEC

CODE_BEGIN
{
    (void)icli_session_self_printf("Hello world\n");
}
CODE_END
CMD_END
```

To include the `ICLI` file in the build job use the `add_icli` function as shown below (❶):

make/module_hello_world.in

```
DIR_hello_world := $(DIR_APPL)/hello_world
OBJECTS_hello_world := hello_world.o
$(eval $(call add_icli,$(MODULE_ID_hello_world),$(DIR_hello_world)/hello_world.icli)) ❶
$(OBJECTS_hello_world): %.o: $(DIR_hello_world)/%.cxx
        $(call compile_cxx, $@, $<)

INCLUDES += -I$(DIR_hello_world)
```

Register the ICLI command for the module:

vtss_appl/hello_world/hello_world.cxx

```
#include "main.h"
extern "C" int hello_world_icli_cmd_register();
..
..
  switch (data->cmd) {
    case INIT_CMD_INIT:
      T_W("hello world! (init)\n");
      hello_world_icli_cmd_register();
```

7.4.3.4.2. `Web`

The Web GUI is comprised of the following two elements:

- Static elements - HTML pages, style sheets, and graphic files.
- Dynamic elements - Dynamic data retrieved by the static HTML pages – representing state of configuration data. These are implemented in so-called page handlers that can be found in the following module directory for each module: `vtss_appl/<module>/<module>_web.c`. When creating a custom module, the respective `module_web.c` handler needs to be implemented, along with the html pages for the new module. HTML pages are usually located in `vtss_appl/<module>/html/*.htm`. When a new page is created, it has to be added in the web GUI by listing the page in `vtss_appl/web/menu_default.c`.

Simply modifying the style sheet and the graphic resource files directly can, to a large extent, change the graphic look of the Web GUI. Style sheets are located in `vtss_appl/web/html/lib/*.css`. Graphics files are located in `vtss_appl/web/html/images/`. Both GIF and PNG formats are used. If changing graphic files, their sizes should be retained.

Finally, the web logo can be also customized by changing the icons `vtss_appl/web/html/logo.gif` and `vtss_appl/web/html/favicon.ico`.

7.4.3.4.3. `SNMP` and `JSON-RPC`

This section will provide a simple example on how to expose objects in a private `MIB` and on the `JSON-RPC` interface.

> The expose framework is not suitable for implementing public `MIBs`. Public `MIBs` are implemented using the `mib2c` tool provided by the `net-snmp` project. Using `mib2c` is out-of-scope for this document.

> The `Expose` framework will derive the `MIB` or JSON specification from the implementation, and not the implementation from the specification. This may be different from other frameworks.

The `Expose` framework is used to expose existing `C/C++` structures/methods on a `JSON-RCP` or `SNMP` interface. We therefore need some structures and methods to work with before the framework can be used. The following header file for the hello world example defines a simple structure with an associated get method (the three `init` functions will be explained later):

vtss_appl/hello_world/hello_world.hxx

```
#ifndef __VTSS_HELLO_WORLDH__
#define __VTSS_HELLO_WORLDH__

#include <main.h>

typedef struct {
    int status;
} hello_world_status_t;

vtss_rc hello_world_status_get(hello_world_status_t *st);

extern "C" void vtss_appl_hello_json_init();
extern "C" void hello_mib_init();
vtss_rc hello_world_init(vtss_init_data_t *data);

#endif
```

When the type definitions are in place, an abstract serialize function needs to be defined. The serialize function is used for both the `JSON` and `SNMP` interface. The serialize function should be placed in a `<module_name>_serializer.hxx` file, and it will look something like this:

vtss_appl/hello_world/hello_world_serializer.hxx

```
#ifndef __VTSS_HELLO_WORLD_SERIALIZER_HXX__
#define __VTSS_HELLO_WORLD_SERIALIZER_HXX__

#include "vtss_appl_serialize.hxx"
#include "hello_world.hxx"

namespace vtss {
namespace appl {
namespace hello_world {
namespace interfaces {
// Defines how the hello_world_status_t is being exposed
struct StatusLeaf {
    // List of parameters for the access methods
    typedef vtss::expose::ParamList<
            vtss::expose::ParamVal<hello_world_status_t *>> P;

    // Serializing the individual arguments
    VTSS_EXPOSE_SERIALIZE_ARG_1(hello_world_status_t &s) {
        // Expose the struct as an "object" when using JSON.
        typename HANDLER::Map_t m =
                h.as_map(vtss::tag::Typename("hello_world_status_t"));

        // Expose the individual fields in the structure
        m.add_leaf(s.status, vtss::tag::Name("status"),
                   vtss::expose::snmp::Status::Current,
                   vtss::expose::snmp::OidElementValue(1),
                   vtss::tag::Description("description"));
    }

    // List all the access methods - only get is needed for read-only objects
    VTSS_EXPOSE_GET_PTR(hello_world_status_get);
};
}  // namespace interfaces
}  // namespace hello_world
}  // namespace appl
}  // namespace vtss

#endif // __VTSS_HELLO_WORLD_SERIALIZER_HXX__
```

When the serialize classes are in place, then the object can be exposed on the `JSON` and/or `SNMP` interfaces. Here is how to create a `JSON` module and expose the `hello_world_status_get` method:

vtss_appl/hello_world/hello_world_json.cxx

```
#include "hello_world_serializer.hxx"
#include "vtss/basics/expose/json.hxx"

using namespace vtss;
using namespace vtss::json;
using namespace vtss::expose::json;
using namespace vtss::appl::hello_world::interfaces;

// Register the methods in the json engine
namespace vtss { void json_node_add(Node *node); }

// Create a name space in the json-spec
static NamespaceNode ns_hello_world("helloWorld");

// Wrapper function to do the registration
extern "C" void vtss_appl_hello_json_init() {
    json_node_add(&ns_hello_world);
}

// Add the structure as a read-only object on the json interface.
// The resulting json method will be called "helloWorld.status.get"
static StructReadOnly<StatusLeaf> l(&ns_hello_world, "status");
```

The same serialize function can now be used to expose the `hello_world_status_t` structure as objects in the SNMP tree. Following is an example showing how to add a new MIB, and expose the structure as read-only objects in the MIB:

vtss_appl/hello_world/hello_world_mib.cxx

```
#include "hello_world_serializer.hxx"

VTSS_MIB_MODULE("helloMib", "HELLO", hello_mib_init, 1000, root, h) {
    h.add_history_element("000000000000Z", "Initial version");
    h.description("Example mib produced by VTSS-Expose");
}

#define NS(VAR, P, ID, NAME) static NamespaceNode VAR(&P, OidElement(ID, NAME))

using namespace vtss;
using namespace expose::snmp;

namespace vtss {
namespace appl {
namespace hello_world {
namespace interfaces {
NS(objects, root, 1, "helloMibObjects");;
NS(hello_status, objects, 2, "helloStatus");;

static StructRO2<StatusLeaf> l(
        &hello_status,
        vtss::expose::snmp::OidElement(1, "helloStatusGlobals")
);

}  // namespace interfaces
}  // namespace hello_world
}  // namespace appl
}  // namespace vtss
```

The generated MIB file can be downloaded from a running target using the URL http://admin:@a.b.c.d/VTSS-HELLO-MIB.mib, where `a.b.c.d` is the IP address of the target.

```
wget http://admin:@a.b.c.d/VTSS-HELLO-MIB.mib
```

The final step is to actually implement the `hello_world_status_get` method, register the `hello_world` module in the `SNMP` and `JSON` trees and to update the make file.

vtss_appl/hello_world/hello_world.cxx

```
#include <hello_world.hxx>

vtss_rc hello_world_status_get(hello_world_status_t *st) {
    st->status = 123;
    return VTSS_RC_OK;
}

vtss_rc hello_world_init(vtss_init_data_t *data) {
    ...
    switch (data->cmd) {
    ...
    case INIT_CMD_INIT:
        ...
        vtss_appl_hello_json_init();  // register the JSON commands
        hello_mib_init();  // register the MIB objects
        break;
    ...
    }

    ...

    return VTSS_RC_OK;
}
```

make/module_hello_world.in

```
DIR_hello_world := $(DIR_APPL)/hello_world

OBJECTS_hello_world :=  \
    hello_world.o \
    $(if $(MODULE_PRIVATE_MIB),hello_world_mib.o) \
    $(if $(MODULE_JSON_RPC),hello_world_json.o)

$(eval $(call add_icli,$(DIR_hello_world)/hello_world.icli))

$(OBJECTS_hello_world): %.o: $(DIR_hello_world)/%.cxx
        $(call compile_cxx, $@, $<)

INCLUDES += -I$(DIR_hello_world)
```

7.4.3.5. Trace system

The trace system allows for modules to printout helpful messages to the console such as errors, warnings or simply debug messages. The trace system is a framework already included in the MSCC application, however new modules have to register to it before they can start using it. The trace system is also configurable per module, with the option to specify which levels of tracing will be active (i.e. shown in the console) at any given time. The trace system features the following trace levels (listed in descending priority):

- Error
- Warning
- Info
- Debug
- Noise
- Racket

The levels are quite intuitive and a module can use any of them after having registered itself to the trace system. The first step in doing that is to assign the module a unique module ID that has to be added as follows:

vtss_appl/util/vtss_module_id.cxx

```
/* In most cases, a privilege level group consists of a single module
   (e.g. LACP, RSTP or QoS), but a few of them contains more than one.
   For example, the "security" privilege group consists of authentication,
   system access management, port security, TTPS, SSH, ARP inspection and
   IP source guard modules.
   The privilege level groups shares the same array of "vtss_module_names[]".
   And use "vtss_priv_lvl_groups_filter[]" to filter the privilege level group which
   we don't need them.
   For a new module, if the module needs an independent privilege level group
   then the filter value should be equal 0. If this module is included by other
   privilege level group then the filter value should be equal 1.
   Set filter value '0' means a privilege level group mapping to a single module
   Set filter value '1' means this module will be filetered in privilege groups
*/
const int vtss_priv_lvl_groups_filter[VTSS_MODULE_ID_NONE+1] =
{
    /*[VTSS_MODULE_ID_API_IO]            0 */ 1,
    /*[VTSS_MODULE_ID_API_CI]            1 */ 1,
    /*[VTSS_MODULE_ID_API_AI]            2 */ 1,
    /*[VTSS_MODULE_ID_SPROUT]            3 */ 1,
    /*[VTSS_MODULE_ID_MAIN]              4 */ 1,
    ...
    /*[VTSS_MODULE_ID_HELLO_WORLD]     143 */ 1,

    /* Hint:
     * For a new module, if the module needs an independent privilege level group
     * then the filter value should be equal 0. If this module is included by other
     * privilege level group then the filter value should be equal 1.
     * Set filter value '0' means a privilege level group mapping to a single module
     * Set filter value '1' means this module will be filetered in privilege groups
     **/

    // LAST ELEMENT ////////////////////////////////////////////////////////////
    /*[VTSS_MODULE_ID_NONE] */               0
};
```

Next, create a new file called `hello_world_trace.h` with the following content:

vtss_appl/hello_world/hello_world_trace.h

```
#ifndef _HELLO_WORLD_TRACE_H_
#define _HELLO_WORLD_H_

#define VTSS_TRACE_MODULE_ID VTSS_MODULE_ID_HELLO_WORLD
#define VTSS_TRACE_GRP_DEFAULT 0 (1)
#define TRACE_HELLO_WORLD_GRP_CNT          1

#endif /* _HELLO_WORLD_TRACE_H_ */
```

On top of the per module trace level, each module can have each own trace group for better trace granularity. The above header defines the needed trace groups, in this case only one group called `default` as seen in (1).

Finally, we update the `hello_world.cxx` program to include the new header, declare the new `default` trace group (2) and register the module to the trace system (1), (3):

vtss_appl/hello_world/hello_world.cxx

```
#include "hello_world_trace.h"
#include "vtss_trace_api.h"

#if (VTSS_TRACE_ENABLED)
static vtss_trace_reg_t trace_reg = ❶
{
    /*.module_id = */VTSS_TRACE_MODULE_ID,
    /*.name      = */"hello_world",
    /*.descr     = */"example"
};

static vtss_trace_grp_t trace_grps[TRACE_HELLO_WORLD_GRP_CNT] = ❷
{
    /*[VTSS_TRACE_HELLO_WORLD_GRP_DEFAULT] = */{
        /*.name      = */"default",
        /*.descr     = */"Default",
        /*.lvl       = */VTSS_TRACE_LVL_WARNING,
        /*.flags     = */1,
    }
};
#endif /* VTSS_TRACE_ENABLED */

/* Initialize module */
vtss_rc hello_world_init(vtss_init_data_t *data)
{
  vtss_isid_t isid = data->isid;
  vtss_rc rc = VTSS_OK;

  if (data->cmd == INIT_CMD_INIT) { ❸
    /* Initialize and register trace resources */
    VTSS_TRACE_REG_INIT(&trace_reg, trace_grps, TRACE_HELLO_WORLD_GRP_CNT);
    VTSS_TRACE_REGISTER(&trace_reg);
  }
  switch (data->cmd) {
    case INIT_CMD_INIT:
      printf("%s\n", "Hello World!");
      T_W("hello world! (init)\n"); ❹
      break;
    case INIT_CMD_START:
      T_W("hello world! (start)\n"); ❺
      break;
    case INIT_CMD_CONF_DEF:
      break;
    case INIT_CMD_MASTER_UP:
      break;
    case INIT_CMD_MASTER_DOWN:
      break;
    case INIT_CMD_SWITCH_ADD:
      break;
    case INIT_CMD_SWITCH_DEL:
      break;
    default:
      break;
  }
```

```
  return rc;
}
```

Finally, we use the trace system to add a couple of warning messages 4, 5 and build the new application. Upgrade the device with the new image and check that two warning messages are emitted during boot as expected:

```
00:00:01 Stage 1 booted
00:00:01 Using device: /dev/mtd7
00:00:09 Mounted /dev/mtd7
00:00:09 Loading stage2 from RAM
00:00:10 Stage2 ends at 0x76fc6ce6, offset 00874ce6
00:00:10 Overall: 9476 ms, ubifs = 7952 ms, rootfs 1453 ms of which xz = 0 ms of
which untar = 0 ms
Starting application...
Using existing mount point for /switch/
Hello World!
W hello_world 00:00:32 66/hello_world_init#37: Warning: hello world! (init)

W hello_world 00:00:34 69/hello_world_init#40: Warning: hello world! (start)


Press ENTER to get started
```

7.4.3.6. Locking

The `critd` module that is always included in any MSCC application helps in protecting critical sections by the use of mutexes and semaphores. Have a look on `vtss_appl/misc/critd_api.h` for more information on the module's interfaces. In this section, we will show how to use the interfaces provided by `critd` in order to secure critical sections inside the new `hello_world` module. We also demonstrate the use of scope locking that eases the lock/unlock process. See the new `hello_world.cxx` program below:

vtss_appl/hello_world/hello_world.cxx

```
#include "hello_world_trace.h"
#include "vtss_trace_api.h"
#include "critd_api.h" ❺

#define VTSS_ALLOC_MODULE_ID VTSS_MODULE_ID_HELLO_WORLD

static critd_t hello_world_crit; ❷

#if (VTSS_TRACE_ENABLED)
static vtss_trace_reg_t trace_reg =
{
    /*.module_id = */VTSS_TRACE_MODULE_ID,
    /*.name      = */"hello_world",
    /*.descr     = */"example"
};

static vtss_trace_grp_t trace_grps[TRACE_HELLO_WORLD_GRP_CNT] =
{
    /*[VTSS_TRACE_HELLO_WORLD_GRP_DEFAULT] = */{
        /*.name      = */"default",
        /*.descr     = */"Default",
        /*.lvl       = */VTSS_TRACE_LVL_WARNING,
        /*.flags     = */1,
    },
    /*[VTSS_TRACE_HELLO_WORLD_GRP_CRIT] = */{ ❶
        /*.name      = */"crit",
        /*.descr     = */"critical regions",
        /*.lvl       = */VTSS_TRACE_LVL_WARNING,
        /*.flags     = */1,
    },
};
#endif /* VTSS_TRACE_ENABLED */

struct Lock { ❻
    Lock(int line) { ❼
      critd_enter(&hello_world_crit, VTSS_TRACE_HELLO_WORLD_GRP_CRIT, VTSS_TRACE_LVL_NOISE, __FILE__, line);
      T_WG(VTSS_TRACE_HELLO_WORLD_GRP_CRIT, "Entering scoped lock\n");
     }
    ~Lock() { ❽
      critd_exit(&hello_world_crit, VTSS_TRACE_HELLO_WORLD_GRP_CRIT, VTSS_TRACE_LVL_NOISE, __FILE__, 0);
      T_WG(VTSS_TRACE_HELLO_WORLD_GRP_CRIT, "Exiting scoped lock\n");
     }
};
#define HELLO_WORLD_CRIT_ASSERT_LOCKED() critd_assert_locked(&hello_world_crit, TRACE_HELLO_WORLD_GRP_CRIT, __FILE__, __LINE__)
#define CRIT_SCOPE() Lock __lock_guard__(__LINE__)

/* Initialize module */
vtss_rc hello_world_init(vtss_init_data_t *data)
{
  vtss_isid_t isid = data->isid;
  vtss_rc rc = VTSS_OK;
```

```

  if (data->cmd == INIT_CMD_INIT) {
    /* Initialize and register trace resources */
    VTSS_TRACE_REG_INIT(&trace_reg, trace_grps, TRACE_HELLO_WORLD_GRP_CNT);
    VTSS_TRACE_REGISTER(&trace_reg);
  }
  switch (data->cmd) {
    case INIT_CMD_INIT:
      critd_init(&hello_world_crit, "hello_world.crit",
VTSS_MODULE_ID_HELLO_WORLD, VTSS_TRACE_MODULE_ID, CRITD_TYPE_MUTEX); ❸
      critd_exit(&hello_world_crit, VTSS_TRACE_HELLO_WORLD_GRP_CRIT,
VTSS_TRACE_LVL_NOISE, __FILE__, __LINE__); ❹

      printf("%s\n", "Hello World!");
      T_W("hello world! (init)\n");
      break;
    case INIT_CMD_START:
      T_W("hello world! (start)\n");
      break;
    case INIT_CMD_CONF_DEF:
      break;
    case INIT_CMD_MASTER_UP:
      { CRIT_SCOPE(); } ❾
      break;
    case INIT_CMD_MASTER_DOWN:
      break;
    case INIT_CMD_SWITCH_ADD:
      break;
    case INIT_CMD_SWITCH_DEL:
      break;
    default:
      break;
  }
  return rc;
}
```

Add the new trace macro and increment the TRACE_HELLO_WORLD_GRP_CNT:

vtss_appl/hello_world/hello_world_trace.h

```
#define VTSS_TRACE_HELLO_WORLD_GRP_CRIT 1
#define TRACE_HELLO_WORLD_GRP_CNT 2

#endif /* _HELLO_WORLD_TRACE_H_ */
```

Although it is entirely optional, we add one more trace group devoted to the critical sections as seen in the updated `trace_grps` ❶ (This is accompanied by a respective change in the `hello_world_trace.h` file to reflect the newly added trace group). We then create a mutex called `hello_world_crit` ❷ which we initialize using the `critd_init()` function as seen in ❸, ❹. Note that we also include the `critd_api.h` ❺.

The mutex is now ready to use, and the `critd_api.h` provides us with the `critd_enter()` and `critd_exit()` interfaces for locking and unlocking the mutex. However, this can be simplified even more with scope locking. In ❻, we use a C++ structure's contructor ❼ and destructor ❽ to do the locking and unlocking for us. Then we simply lock a section by instantiating a variable of this structure ❾. One of the benefits of this method is that unlocking of the mutex will happen automatically when the program goes out of the locked scope.

Finally, make a build using the new `hello_world.cxx` and load it into the device. Check that you see the following output, thus verifying that the scope locking works as expected.

```
00:00:01 Stage 1 booted
00:00:01 Using device: /dev/mtd7
00:00:09 Mounted /dev/mtd7
00:00:09 Loading stage2 from RAM
00:00:10 Stage2 ends at 0x772bfce6, offset 00875ce6
00:00:10 Overall: 9795 ms, ubifs = 8333 ms, rootfs 1391 ms of which xz = 0 ms of
which untar = 0 ms
Starting application...
Using existing mount point for /switch/
Hello World!
W hello_world 00:00:32 66/hello_world_init#64: Warning: hello world! (init)

W hello_world 00:00:34 69/hello_world_init#67: Warning: hello world! (start)

W hello_world/crit 00:00:35 69/Lock#37: Warning: Entering scoped lock

W hello_world/crit 00:00:35 69/~Lock#41: Warning: Exiting scoped lock


Press ENTER to get started
```

7.4.3.7. Frame flow

Most applications that implement L2/L3 protocols will need to set-up some basic frame flow for receiving/transmitting packets as part of their implementation. This section briefly illustrates the key interfaces for setting up frame flow within a custom module.

7.4.3.7.1. Frame reception

The `packet` module is responsible for distributing received frames to the other application modules, therefore a custom module needs to register itself to the `packet` module in order to receive frames. All the needed interfaces for the `packet` module are declared in the library `packet_api.h` which is found in `vtss_appl/packet/packet_api.h`. Let's see an example registration using the existing `hello_world` program:

vtss_appl/hello_world/hello_world.cxx

```
//...
#include "packet_api.h" ❶

//...

vtss_mac_addr_t dmac = {0xff, 0xff, 0xff, 0xff, 0xff, 0xff};

static BOOL hello_world_packet_rx(void *contxt, const uchar *const frm, const mesa_packet_rx_info_t *const rx_info) ❷
{
    // Example of packet processing
    if (!memcmp(&frm[0], dmac, 6)) {
        T_D("hello world just received a frame!\n");
        // Further processing...
        return VTSS_RC_OK;
    }

    // Don't do anything, packet is discarded
    return VTSS_RC_OK;
}

static void hello_world_packet_register(void) ❸
{
    packet_rx_filter_t rx_filter; ❹
    void               *rx_filter_id;
    vtss_rc            rc;

    packet_rx_filter_init(&rx_filter); ❺
    rx_filter.modid = VTSS_MODULE_ID_HELLO_WORLD;
    rx_filter.cb    = hello_world_packet_rx;

    memcpy(rx_filter.dmac, dmac, sizeof(rx_filter.dmac));
    rx_filter.match  = PACKET_RX_FILTER_MATCH_DMAC;
    rx_filter.prio   = PACKET_RX_FILTER_PRIO_NORMAL;

    if ((rc = packet_rx_filter_register(&rx_filter, &rx_filter_id)) != VTSS_RC_OK) { ❻
        T_E("Failed to register packet rx!\n");
    }
}

/* Initialize module */
vtss_rc hello_world_init(vtss_init_data_t *data)
{
  vtss_isid_t isid = data->isid;
  vtss_rc rc = VTSS_OK;

  if (data->cmd == INIT_CMD_INIT) {
    /* Initialize and register trace resources */
    VTSS_TRACE_REG_INIT(&trace_reg, trace_grps, TRACE_HELLO_WORLD_GRP_CNT);
    VTSS_TRACE_REGISTER(&trace_reg);
  }
  switch (data->cmd) {
    case INIT_CMD_INIT:
```

```
      critd_init(&hello_world_crit, "hello_world.crit", VTSS_MODULE_ID_HELLO_WORLD, VTSS_TRACE_MODULE_ID, CRITD_TYPE_MUTEX);
      critd_exit(&hello_world_crit, VTSS_TRACE_HELLO_WORLD_GRP_CRIT, VTSS_TRACE_LVL_NOISE, __FILE__, __LINE__);

      printf("%s\n", "Hello World!");
      T_W("hello world! (init)\n");
      break;
    case INIT_CMD_START:
      T_W("hello world! (start)\n");
      hello_world_packet_register(); ❼
      break;
    case INIT_CMD_CONF_DEF:
      break;
    case INIT_CMD_MASTER_UP:
      { CRIT_SCOPE(); }
      break;
    case INIT_CMD_MASTER_DOWN:
      break;
    case INIT_CMD_SWITCH_ADD:
      break;
    case INIT_CMD_SWITCH_DEL:
      break;
    default:
      break;
  }
  return rc;
}
```

First we need to include the `packet_api.h` header as seen in ❶. Then we need to create a callback function inside our module, which the packet module will call when it needs to provide a frame. This function is seen in ❷ and is responsible for processing the frame inside the custom module. You can see a very basic example code inside the `hello_world_packet_rx()` function. Then we create the `hello_world_packet_register()` function, see ❸, that will do the registration with the packet module. This function starts by declaring a `rx_filter` ❹ which then initializes using the `packet_api.h` function, `packet_rx_filter_init()`, see ❺. Then, we have to configure the packet filter according to the needs, in this case we demonstrate a simple filter that matches on destination MAC address of the received frame. The last step after configuring the filter is to call the `packet_rx_filter_register()` interface (❻) that will register the filter with the `packet` module. The registration must be made during the `INIT_CMD_START` stage as seen in ❼.

7.4.3.7.2. Frame transmission

The `l2proto` module is responsible for transmitting frames created by the application modules, therefore a custom module needs to utilize interfaces that the `l2proto` module provides. All the needed interfaces are declared in the library `l2proto_api.h` which is found in `vtss_appl/l2proto/l2proto_api.h`. Let's see an example frame transmission function using the existing `hello_world` program:

vtss_appl/hello_world/hello_world.cxx

```
//...
#include "l2proto_api.h" ❶

//...

mesa_mac_addr_t dmac = {0xff, 0xff, 0xff, 0xff, 0xff, 0xff};

static void hello_world_tx(uint l2port, void *buffer, size_t size) ❷
{
    vtss_common_bufref_t bufref;
    uchar *buf;

    // Use the existing buffer, or edit it accordingly
    // ...
    /* fill destination MAC */
    memcpy(buffer, dmac, 6);

    buf = (uchar *)vtss_os_alloc_xmit(l2port, size, &bufref); ❸
    if(buf) {
        memcpy(buf, buffer, size);
        (void) vtss_os_xmit(l2port, buf, size, bufref); ❹
    }
}
```

We start by including the `l2proto_api.h` header as seen in ❶. Then we create a very basic transmit function, `hello_world_tx()`, see ❷. This function assumes that we have already created a frame we would like to transmit, that is the argument `buffer`, and we also know the transmit port `l2port` and the frame size `size`. The `buffer` can be already created in another function, but it can always be manipulated as seen inside this sample function, where we appoint a destination MAC address to the frame. The actual frame transmission is a two step process; first we allocate a transmit buffer using the `vtss_os_alloc_xmit()` interface as seen in ❸, and then (after we fill that buffer with the actual frame we have prepared) call the `vtss_os_xmit()` interface ❹ that will now transmit the frame.

7.5. Custom flash images

WebStaX releases include a set of flash images that matches the different reference boards. If a project has changed the flash components, or wishes to use other components in the flash-images, then it is necessary to build custom flash images. This section will provide the instruction on how to do that.

A prerequisite to this is to have a working boot-loader, application SW and knowledge about the HW flash system.

This project uses an internal developed `flash_build` project to build these projects. Start by downloading this project using `git`:

```
$ git clone http://github.com/vtss/flash_builder
$ cd flash_builder
```

The script reads a template which specifies the flash layout, and points to the files that provide the content for each partition.

The upstream `flash_builder` project includes support for all the different reference boards supported by MSCC. Most customer projects do not need to build all these flash images, and the easiest approach is therefore pick the configuration that is *closest* to the given project and use that as a reference. This example will be using a `Jaguar2` board as reference:

```
$ cp templates/linux-jaguar2-cu48-32mb-64kb.txt templates/custom.txt
```

Next step is to customize the template such that it matches the requirements of the project. Here is what the example template looks like:

```
# Flash template: linux-jaguar2-cu48-32mb-64kb
# The first section describe the flash geometry: capacity, blocksize
---
- capacity: 32M
  blocksize: 64K
#
# Subsequent sections describe individual flash sections:
#  - name: The FIS name. 1 to 15 characters
#  - size: Flash section size. Units 'M' or 'K'
#  - flash: Hex address of section
#  - entry: Hex address of execution entrypoint (optional)
#  - memory: Hex address of memory load address (optional)
#  - datafile: File name to load data from (optional)
#
- name: 'RedBoot'
  size: 256K
  flash: 0x40000000
  datafile: artifacts/redboot-jaguar2.img
- name: 'conf'
  size: 64K
  flash: 0x40040000
- name: 'linux'
  size: 16128K
  flash: 0x40050000
  memory: 0x80100000
  entry: 0x80100000
  datafile: artifacts/bringup_jr2_48.mfi
- name: 'linux.bk'
  size: 16128K
  flash: 0x41010000
  memory: 0x80100000
  entry: 0x80100000
  datafile: artifacts/bringup_jr2_48.mfi
- name: 'FIS directory'
  size: 64K
  flash: 0x41fd0000
- name: 'RedBoot config'
  size: 4K
  flash: 0x41fe0000
  datafile: files/fconfig-linux.bin
- name: 'Redundant FIS'
  size: 64K
  flash: 0x41ff0000
```

If changes to the `capacity` and `blocksize` are needed, we suggest to start by picking a template that matches the new `capacity` and `blocksize`. The `templates` directory contains several useful templates. Then you can change the sizes of the flash sections and optionally update the `datafile` pointers (do not change the last three partitions, these are needed by RedBoot). If no template exists that will fit your needs, then the `capacity` and `blocksize` have to be updated manually such that they match the `NOR` flash mounted on the target device.

> The last three sections of the template depend directly on the `capacity` and `blocksize` values. If these values are changed on an existing template, then the `size` and `flash` values of the last three sections have to be altered as well. Unless you know the needed values, we suggest to use one of the existing templates in order to avoid such changes.

When the template is updated, make sure that all `datafile` artifacts exist; the `linux` and `linux.bk` is built from the WebStaX sources (see Using a build configuration), and the `RedBoot` image is built from the RedBoot sources (see Building RedBoot from sources). Move those files into directory called 'artifacts'.

```
$ ls -la artifacts
total 4548
drwxrwxr-x 2 anielsen anielsen    4096 Aug 22 16:20 .
drwxr-xr-x 9 anielsen anielsen    4096 Aug 22 16:21 ..
-rw-rw-r-- 1 anielsen anielsen 4449541 Aug 22 16:39 bringup_jr2_48.mfi
-rw-rw-r-- 1 anielsen anielsen  195056 Aug 22 16:39 redboot-jaguar2.img
```

Final step is to build the custom flash image:

```
$ perl -w ./buildflash.pl --verbose templates/custom.txt
Completed custom
$ ls -lah images
total 33M
drwxrwxr-x 2 anielsen anielsen 4.0K Aug 22 16:41 .
drwxr-xr-x 9 anielsen anielsen 4.0K Aug 22 16:21 ..
-rw-rw-r-- 1 anielsen anielsen  32M Aug 22 16:40 custom.bin
```

The resulting `custom.bin` can now be programmed to the `NOR` flash using a programmer. This is covered in section Flashing the NOR with a flash image.

8. References

[]

AN1163 Linux Customizations