Linux Customizations

Application Note

CONFIDENTIAL

AN1163
Rev. APPL-2024.06
2024-06-28

This document will present a collection of options to customize the firmware images managed by WebStaX software project (SW) used in ENT switch products. All facilities presented here do not require source code access for the SW but do require a Board Support Package (BSP), and can be used to customize without recompiling the WebStaX image. This document will mostly focus on simple customization facilities. For more advanced customization, knowledge of embedded Linux and C programming is required.

The document will first present a set of customization facilities that can be used and combined. After this it will provide a few use cases with examples on how the different facilities may be used.

Table of Contents

1. Facilities

This section includes a number of different facilities that can be used to customize a firmware image for a product in the WebStaX family.

1.1. Modular Firmware Images

Modular Firmware Images (MFI) is a new image type that is being introduced with the Linux based version of the WebStaX family. The modular firmware architecture is designed to be flexible and allow the user to replace and append various components without the need to recompile/rebuild all the components.

The MFI file consists of two parts, a `stage1` and a `stage2`. `stage1` as a minimum must include a `kernel` and a `initrd` section. `stage2` may include a number of root file system elements. `stage1` must be accessible by the boot-loader. `stage2` on the other hand is loaded by the `initrd` application after the `kernel` has booted the system. This may be used to locate the root filesystem (which is the majority of the image file) in the NAND flash not accessible by the boot-loader.

`stage2` of the firmware image includes parts that are being used to build the final root file system. Each root file system element is **just** `xz` compressed `tarball` file with a small header. The `initrd` application in the `stage1` section, will iterate through the root file system elements, and extract each of the associated tarballs into a ram file system. The file system entries will be processed in the order they are located in the file, meaning that a later element can overwrite files in an earlier element.

The script called `mfi.rb` is being provided as part of the BSP, and it should be used to build, alter and/or inspect the MFI files. Start by validating that the `mfi.rb` is installed and able to run:

```
$ mfi.rb --help
Usage: mfi [global-options] [<command> [command-specific-options]]
    -i, --input  <file>              Firmware image to read. New image is created if
not specified.
    -o, --output  <file>             Firmware image to write.
    -k, --public-key  <file>         Use public key for validation.
    -v, --verbose  <lvl>             Set verbose level.
    -d, --dump                       Dump the inventory of TLV(s) in the firmware
image.
    -V, --version                    Print the version of this program and exit.

Commands:
  help         Prints this help message.
  stage1       Inspect or alter the stage1 area of the firmware image.
  bootloader   Inspect or alter any bootloader tlv in the firmware image.
  rootfs       Inspect or alter the rootfs tlv(s) in the firmware image.
```

The `mfi.rb` script must be included in the search path. This script is distributed as part of the BSP, and is installed at: `/opt/mscc/mscc-brsdk-mips-vXX.YY/stage1/x86_64-linux/usr/bin/mfi.rb` where `XX.YY` represents the version of the BSP. This section will assume it to be part of the search path.

1.1.1. Firmware File Format

Note: This section documents the binary file format used by MFI. This is provided as background information, most people should be able to just use the `mfi.rb` tool to inspect, read and write `mfi` files.

MFI uses a binary file format. The file starts with the `stage1` part, the optional `stage2` follows immediately after `stage1`.

Both `stage1` and `stage2` are using the following `typedef` for little endian integers:

```
    typedef uint32_t mscc_le_u32;
```

The file format of `stage1` and `stage2` is documented below.

Stage 1

The binary `stage1` header is defined by the following:

```
typedef struct mscc_firmware_vimage {
    mscc_le_u32 magic1;         // 0xedd4d5de
    mscc_le_u32 magic2;         // 0x987b4c4d
    mscc_le_u32 version;        // 0x00000001

    mscc_le_u32 hdrlen;         // Header length
    mscc_le_u32 imglen;         // Total image length (stage1)

    char machine[32];           // Machine/board name
    char soc_name[32];          // SOC family name
    mscc_le_u32 soc_no;         // SOC family number

    mscc_le_u32 img_sign_type;  // Image signature algorithm. TLV has
                                // signature data

    // After <hdrlen> bytes;
    // struct mscc_firmware_vimage_tlv tlvs[0];
} mscc_firmware_vimage_t;
```

Immediately after the `stage1` header is a number of `stage1` Type Length Value (TLV). Each `TLV` follows the following format (no padding between `TLVs`):

```
typedef struct mscc_firmware_vimage_tlv {
    mscc_le_u32 type;      // TLV type (mscc_firmware_image_stage1_tlv_t)
    mscc_le_u32 tlv_len;   // Total length of TLV (hdr, data, padding)
    mscc_le_u32 data_len;  // Data length of TLV
    u8          value[0];  // Blob data
} mscc_firmware_vimage_tlv_t;
```

The following enum documents the list of `TLVs` supported:

```
typedef enum {
    MSCC_STAGE1_TLV_KERNEL      = 0,
    MSCC_STAGE1_TLV_SIGNATURE   = 1,
    MSCC_STAGE1_TLV_INITRD      = 2,
    MSCC_STAGE1_TLV_KERNEL_CMD  = 3,
    MSCC_STAGE1_TLV_METADATA    = 4,
    MSCC_STAGE1_TLV_LICENSES    = 5
} mscc_firmware_image_stage1_tlv_t;
```

The following enum documents the list of signatures supported:

```
typedef enum {
    MSCC_FIRMWARE_IMAGE_SIGNATURE_MD5    = 1,
    MSCC_FIRMWARE_IMAGE_SIGNATURE_SHA256 = 2,
    MSCC_FIRMWARE_IMAGE_SIGNATURE_SHA512 = 3,
} mscc_firmware_image_signature_t;
```

The signature covers the whole `stage1` `TLVs`. This `digital signature scheme` is calculated based on `OpenSSL` `RSA`.

Stage 2

The `stage2` part does not have a common header, it is "just" a sequence of `stage2 TLVs`. The `stage2 TLV` header looks like this:

```
typedef struct mscc_firmware_vimage_s2_tlv {
    mscc_le_u32 magic1;    // 0xa7b263fe
    mscc_le_u32 type;      // TLV type (mscc_firmware_image_stage2_tlv_t)
    mscc_le_u32 tlv_len;   // Total length of TLV (hdr, data, padding)
    mscc_le_u32 data_len;  // Data length of TLV
    mscc_le_u32 sig_type;  // Signature type
                           // (mscc_firmware_image_signature_t)
    u8          value[0];  // Blob data
} mscc_firmware_vimage_stage2_tlv_t;
```

In contrast to `stage1 TLVs`, then `stage2 TLVs` embed the signature directly into each `TLV`. This means that the `stage2 TLVs` is signed individually. The binary layout is as follows:

```
                                                    20 + data_len
0       4      8        12         16          20      |           tlv_len
+-------+------+---------+----------+-----------+------+-----------+
| magic | type | tlv_len | data_len | sige_type | data | signature |
+-------+------+---------+----------+-----------+------+-----------+
```

The supported `stage2 TLV` types are documented by the following enumeration (currently only `ROOTFS` is supported):

```
typedef enum {
    MSCC_STAGE2_TLV_ROOTFS = 2,
} mscc_firmware_image_stage2_tlv_t;
```

Root file system element `ROOTFS`

The root file system element `TLV` type is 2. The root file system element is optional and may be repeated, meaning that a given firmware image may include between zero and **N** of these elements. The data content of this `TLV` is a new `TLV` area using the following header:

```
0      4        8   Length + 8
+------+--------+------+
| Type | Length | Data |
+------+--------+------+
```

Following is the list of `sub-tlv` types supported by the root file system elements:

1. **Name:** An ASCII encoded string with the name of this element.
2. **Version:** An ASCII encoded string with the version information of this element.
3. **License terms:** An ASCII encoded string with the license terms of this element.

4. **PreExec** An executable that is being invoked before the `tar` archive is extracted into the root file system. **Not implemented in current release**.
5. **Content:** A `xz` compressed `tar` archive with the root file system content.
6. **PostExec:** An executable that is being invoked after the `tar` archive is extracted into the root file system. **Not implemented in current release**.

If the firmware image includes more than one root file system elements, then the content of those is being merged. If the same file(s) is present in multiple archives, then it is the content from the last archive that wins.

1.1.2. Tool Support `mfi.rb`

The `MFI` command line tool is used to construct and inspect the firmware images. It also supports appending/replacing the contents to an existing firmware image. It is written in `Ruby`, hence it should be working across platforms (`WIN`, `OSX` and `LINUX`) as long as `Ruby` is supported and properly installed.

`MFI` tool includes a number of different submodules for different operations on the firmware images.

```
$ mfi -help
Usage: mfi [global-options] [<command> [command-specific-options]]
    -i, --input  <file>              Firmware image to read. New image is created if
not specified.
    -o, --output  <file>             Firmware image to write.
    -k, --public-key  <file>         Use public key for validation.
    -v, --verbose  <lvl>             Set verbose level.
    -d, --dump                       Dump the inventory of TLV(s) in the firmware
image.
    -V, --version                    Print the version of this program and exit.
    -c, --collect-sha  <file>        Collect sha's to file when doing dump.

Commands:
  help         Prints this help message.
  stage1       Inspect or alter the stage1 area of the firmware image.
  bootloader   Inspect or alter any bootloader tlv in the firmware image.
  rootfs       Inspect or alter the rootfs tlv(s) in the firmware image.
```

Note: `MFI` tool currently supports submodules are shown below, more submodules might be added in the future.

- stage1
- bootloader
- rootfs

Add submodule name for further help information on a specific submodule.

```
$ ## mfi <submodule> -help
$ mfi stage1 -help
Usage: stage1 [options]
    -a, --kernel-get <file>          Extract the kernel from the firmware image, and
write it to <file>.
    -b, --kernel-set <file>          Update the kernel blob in the firmware image
with the raw content of <file>.
    -c, --initrd-get <file>          Extract the initrd from the firmware image and
write it to <file>.
    -d, --initrd-set <file>          Update the initrd blob in the firmware image
with the raw content of <file>.
    -e, --metadata-get <file>        Extract the metadata blob from the firmware
image and write it to <file>.
    -f, --metadata-set <file>        Update the metadata blob in the firmware image
with the raw content of <file>.
    -m, --machine <string>           Set the machine string in the image..
    -w, --soc-name <string>          Set the soc-name string in the image..
    -n, --soc-no <string>            Set the soc-no string in the image..
    -k, --kernel-command <string>    Set the kernel command line in the image..
    -l, --license-terms <file>       Update the licenses blob in the firmware image
with the raw content of <file>.
    -s, --sign-data <type> <keyfile> Sign data with (RSA) key
```

Let’s take a look at how `stage1` is normally composed.

```
mfi.rb -o <output_file_name>.mfi stage1 \
       --kernel-set <path_to_kernel_file> \
       --initrd-set <path_to_initrd_file> \
       --kernel-command "init=/usr/bin/stage1-loader loglevel=4" \
       --metadata-set <path_to_metadata_file> \
       --license-terms <path_to_licensedata_file> \
       --machine <machine_name> \
       --soc-name <soc_name> \
       --soc-no <soc_number>
```

In the example above, `--kernel-command` can also be adjusted as needed of course. Since `stage1` is the first component in the firmware image, there is no input file (annotated as `-i`).

Upon execution, `<output_file_name>.mfi`, with all `stage1` components in place, will be created.

The other submodules like `rootfs` is added in the same manner. `<new_output_file_name>.mfi` could be same as the input file.

```
mfi.rb -i <output_file_name>.mfi -o <new_output_file_name>.mfi rootfs\
       --action append \
       --name "rootfs" \
       --version "SDK_VERSION" \
       --file <path_to_rootfs_file>.tar.xz
```

1.2. ServiceD

ServiceD is a service manager. The switch application and other user applications are all considered as services which will be spawned, monitored and managed by ServiceD.

1.2.1. Service Configuration File

Each service is spawned according to its configuration file, located in the `/etc/mscc/service/` folder. The configuration file follows certain formats.

The following is an example of a ServiceD configuration file:

```
# Start of config file
# Comment line starts with '#'
# This is an example configuration file called
# /etc/mscc/service/switch_app.service
name = switch_app
type = service
on_error = reboot
env = LD_PRELOAD=/lib/btsig.so
env = FOO=bar
env = Hello=world
# depend =
cmd = /usr/bin/switch_app
ready_file = /tmp/switch_app.ready
```

Notice the LD_PRELOAD line above: It causes the btsig shared object to be loaded when the application (here "switch_app") and any of its descendant processes execute. btsig registers a signal handler for a range of signals for the program it hooks into. If one of the signals handled by btsig is raised by the program, btsig will print a stack backtrace along with the command line and thread/process IDs. The list of signals captured by btsig is: SIGINT, SIGQUIT, SIGILL, SIGFPE, SIGSEGV, SIGBUS, SIGPWR, SIGUSR1, and SIGUSR2. If either of the three latter is raised (e.g. with "kill -USR1 <pid>"), execution will continue after the stack backtrace has been printed. Otherwise it will exit with EXIT_FAILURE. The program that btsig hooks into may install its own handler for one or more of these signals. This will override btsig's handler. Two more environment variables control how btsig works. The first, BTSIG_OUT_FILE, if set, will cause btsig to write to this file instead of writing to stdout. The second, BTSIG_ERR_FILE, if set, will cause btsig to write to this file rather than to stderr.

1.2.2. Appearance and Requirements

The following table lists all the fields that is allowed in a ServiceD configuration file. ServiceD will **NOT** start a service that has an faulty configuration.

Fields	Appearance & Requirements	Description
`name`	Must appear once	Name of service (only used for debugging and logging).

Fields	Appearance & Requirements	Description
`type`	Allow once	Type of service. Allowed values are `service` and `oneshot`. `service` is a long running process that by default will be auto-restarted by ServiceD, if it exits. Its ready file will be also deleted until it is ready again. `oneshot` process will be seen as ready and its ready file, if any, will be created, if its process executes and exits normally with return code 0. Otherwise it will, by default, be respawned until it exits normally. The default behavior of the actions taken when a service exits, can be altered with the `on_error` field.
`on_error`	Allow once	Controls what happens if or when a ServiceD-controlled process terminates unexpectedly. Valid values for this field are `respawn`, `reboot`, and `ignore`. `respawn` (which is the default if not specified) causes a process to be re-started. `reboot` causes the system to be rebooted. `ignore` does nothing. The semantics of "process terminates unexpectedly" differ bewteen a long-lived and a one-shot process: For a long-lived process, the `on_error` action is taken whether or not the process got terminated by a signal or exited itself with any return code. For a one-shot process, the `on_error` action is taken only if the process terminates by a signal or exits with a non-zero return code.
`env`	Allow zero or more	Specify environment variables to be set for the given service. Must follow syntax: `key=val`
`cmd`	Must appear once	Specify the command to invoke.
`depend`	Allow zero or more	A list of services, this service depends on. The service will not be started before all its dependencies are ready.
`ready_file`	Allow once	Specifies a file that the service can use to flag that it is ready. The ServiceD application will poll the availability of the file, and the service is not considered ready until the `ready_file` exists. This is only used when `type` is `service`.

Fields	Appearance & Requirements	Description
`serviced_profile`	Allow once	Profile of service. Any string is allowed. Only services with the targeted profile will be spawned by `ServiceD`. `webstax` is the default profile if nothing is specified in the service configuration file. `debug` can be speficied in the kernel command line so as to start the linux shell for debug purpose.

1.2.3. Booting ServiceD profiles

To boot an alternative ServiceD profile, break the system in RedBoot, load the linux partition, and add `serviced_profile=debug` to the Linux kernel command string (assuming you want to boot the debug profile). See example below:

```
Serval Reference board detected (VSC7418 Rev. B).

RedBoot(tm) bootstrap and debug environment [ROMRAM]
Non-certified release, version 1_19-dec88fb - built 12:47:33, May  2 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 [0x800292c0-0x87fafffc available]
FLASH: 0x40000000-0x40ffffff, 64 x 0x40000 blocks
== Executing boot script in 3.000 seconds - enter ^C to abort
^C
RedBoot> ^C
RedBoot> fis load -x linux
MD5 signature validated
Stage1: 0x80100000, length 4490700 bytes
Initrd: 0x80600000, length 188416 bytes
Kernel command line: init=/usr/bin/stage2-loader loglevel=4
RedBoot> exec -c "init=/usr/bin/stage2-loader loglevel=4 serviced_profile=debug"
```

The system will now boot into a Linux shell.

1.3. JSON-IPC

The Switch Application includes a JSON Inter-Process Communication (IPC) module providing an IPC service to other applications. This IPC can be used for two purposes:

- The User Application can send JSON requests and receive corresponding JSON responses for normal configuration or monitoring purposes.

- The User Application can add/delete registrations for event notifications. If a registered event occurs, the Switch Application will send a JSON notification message to the User Application.

```
+-----------+       +-----------+
|  Switch   |<----->|    User   |
|Application|       |Application|
+-----------+       +-----------+
```

1.3.1. IPC Message Format

The JSON IPC has the following properties:

- The IPC uses a Unix domain socket bound to `/var/run/json_ipc.socket`.
- The exchanged messages consist of two parts:
  - Length: 4 byte data length field in native CPU endianness.
  - Data: JSON message with the length above. JSON notification registration is done using these method names:
    - Add registration: "jsonIpc.config.notification.add".
    - Delete registration: "jsonIpc.config.notification.del".

```
0        4     4 + length
+--------+--------------+
| length | JSON Message |
+--------+--------------+
```

1.3.2. JSON Message Examples

The following examples only show the JSON message part of the JSON IPC message, i.e., the length field is not included. First a normal request-response communication is shown (get system information):

```
User Application -> Switch Application:
{"method":"systemUtility.config.systemInfo.get",
 "params":[],
 "id":"json_ipc"}

Switch Application -> User Application:
{"id":"json_ipc",
 "error":null,
 "result":{"Hostname":"my-switch",
           "Contact":"",
           "Location":""}}
```

Next, an event registration (port status update), an event notification (link up) and an event deregistration (port status update) is shown:

```
User Application -> Switch Application:
{"method":"jsonIpc.config.notification.add",
 "params":["port.status.update"],
 "id":"json_ipc"}

Switch Application -> User Application:
{"method":"port.status.update",
 "id":null,
 "params":[{"event-type":"modify",
            "key":"Gi 1/1",
            "val":{"Link":false,
                   "Fdx":true,
                   "Fiber":false,
                   "Speed":"speed1G",
                   "SFPType":"none",
                   "SFPVendorName":"",
                   "SFPVendorPN":"",
                   "SFPVendorRev":"",
                   "LossOfSignal":false,
                   "TxFault":false,
                   "Present":false,
                   "SFPVendorSN":""}}]}

User Application -> Switch Application:
{"method":"jsonIpc.config.notification.del",
 "params":["port.status.update"],
 "id":"json_ipc"}
```

1.4. Boot-time Configuration

The Switch Application includes a number of features, which can be disabled at boot-time. This may be done if a given feature is not desired or if it is preferred to implement the feature outside the Switch Application. The following features can currently be disabled:

- CLI via console port
- CLI via Telnet
- CLI via SSH
- SNMP
- Web handlers
- Web server

The boot-time configuration is done in `/etc/switch.conf` on the system. This file does not exist by default, but may be added to the image using an MFI file. The format of the file is JSON-based as shown below. In this example, SSH and Web handlers are disabled at boot-time.

```
{
  "cli":{
    "enable":true,
  },
  "ssh":{
    "enable":false,
  },
  "snmp":{
    "enable":true
  },
  "telnet":{
    "enable":true
  },
  "web":{
    "enable":true,
    "handlers":false
  }
}
```

2. Use Cases

2.1. Custom Web

The Web pages are added as a TLV section by default. Therefore it is very easy to customize Web by replacing the default Web by a customized one.

In this use case, a simple HTML file capable of showing port status via JSON interface will be demonstrated.

2.1.1. Create, Compress New Web

Execute the following script to create a custom web file.

custom_web.sh

```
mkdir -p custom-web/var/www/webstax/
cd custom-web/var/www/webstax/
cat > custom_web.html <<\EOF
<!DOCTYPE html>
<html>
  <head>
    <title>Test test test...</title>
    <script src="jquery-2.1.4.min.js" type="text/javascript"
charset="utf-8"></script>
  </head>
  <body><div id = "port0" >PORT    ---    LINK STATUS</div>
    <script type="text/javascript">
     function port_status_cb(d) {
       for (var i = 0; i < d["result"].length; ++i) {
         var newdiv = "port"+(i + 1);
         $('#port' + i).append($('<div id='+newdiv+'></div>'));
         $('#port' + (i+1)).html(d["result"][i]["key"] + "     ---     " +
d["result"][i]["val"]["Link"]);}}
     $(function () {$.post("/json_rpc",
"{\"method\":\"port.status.get\",\"params\":[],\"id\":\"jsonrpc\"}")
                    .done(function(data){ console.log(data);
port_status_cb(data);}); });
    </script>
  </body>
</html>
EOF
wget https://code.jquery.com/jquery-2.1.4.min.js
cd ../../../
```

There are many Web handlers provided by the two below.

- `FastCGI`
- MSCC switch application

```
mkdir -p etc
cat > etc/switch.conf <<\EOF
{
    "web":{
        "enable":true,
        "handlers":false
    }
}
EOF
tar -cf custom_web.tar var/ etc/
xz --check=none custom_web.tar
```

Now a new file `custom_web.tar.xz` should be available for replacement.

2.1.2. Replace

Before replacement, we also need to inspect the MFI image for the index of the default Web TLV section.

```
/<mfi script dir>/mfi.rb \
    -i <some dir>/<targeted switch>.mfi \
    -d
```

Example output could be:

```
Stage1
  Version:1
  Magic1:0xedd4d5de, Magic2:0x987b4c4d, HdrLen:92, ImgLen:1660224
  Machine:luton10, SocName:luton26, SocNo:2, SigType:1
  Tlv Type:Kernel(0), Data Length:1454304
  Tlv Signature(1),   Data Length:16 (validated)
  Tlv Initrd(2),      Data Length:188416
  Tlv KernelCmd(3),   Data Length:38
  Tlv Metadata(4),    Data Length:188
  Tlv License(5),     Data Length:17096
Stage2 - Index:0
  Tlv FsElement(2), Data Length:2415915
  MD5(1), Length:16 Data: 364cdb5a4b227211477adfa7653ab817 (validated)
    Name                : rootfs
    Content file name   : /opt/mscc/mscc-brsdk-mips-v01.50/stage2/smb/rootfs.tar.xz
    Content file length : 2415828
Stage2 - Index:1
  Tlv FsElement(2), Data Length:2942598
  MD5(1), Length:16 Data: e9ed05b6f04d7aeac4373db6f109eef7 (validated)
    Name                : vtss
    Content file name   : vtss-rootfs.tar.xz
    Content file length : 2942552
Stage2 - Index:2
  Tlv FsElement(2), Data Length:442733
  MD5(1), Length:16 Data: dd2f8256504737d628213ee51a134fe1 (validated)
    Name                : vtss-web-ui
    Content file name   : vtss-www-rootfs.tar.xz
    Content file length : 442676
```

From the log above, Index 2 is the default Web TLV section.

Now we can replace the default Web TLV index (2) with the newly generated web file `custom_web.tar.xz`.

```
/<mfi script dir>/mfi.rb \
    -i <some dir>/<targeted switch>.mfi \
    -o <targeted switch>_custom_web.mfi rootfs \
    --index 2 \
    --action replace \
    --name "custom_web" \
    --file <some dir>/custom_web.tar.xz
```

At this point, a image named `<targeted switch>_custom_web.mfi` will be generate. Issue the commands below to inspect it again.

```
/<mfi script dir>/mfi.rb \
    -i <some dir>/<targeted switch>_custom_web.mfi \
    -d
```

Example output is shown as below. And pay attention to the Index 2 which is now replaced by the customized web.

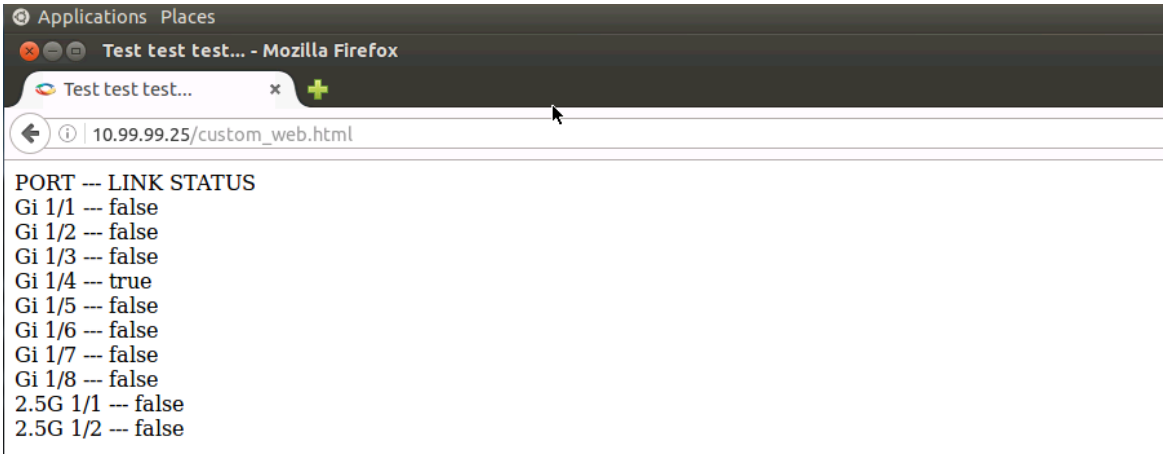
```
Stage1
  Version:1
  Magic1:0xedd4d5de, Magic2:0x987b4c4d, HdrLen:92, ImgLen:1660224
  Machine:luton10, SocName:luton26, SocNo:2, SigType:1
  Tlv Type:Kernel(0), Data Length:1454304
  Tlv Signature(1),   Data Length:16 (validated)
  Tlv Initrd(2),      Data Length:188416
  Tlv KernelCmd(3),   Data Length:38
  Tlv Metadata(4),    Data Length:188
  Tlv License(5),     Data Length:17096
Stage2 - Index:0
  Tlv FsElement(2), Data Length:2415915
  MD5(1), Length:16 Data: 364cdb5a4b227211477adfa7653ab817 (validated)
    Name                : rootfs
    Content file name   : /opt/mscc/mscc-brsdk-mips-v01.50/stage2/smb/rootfs.tar.xz
    Content file length : 2415828
Stage2 - Index:1
  Tlv FsElement(2), Data Length:2942598
  MD5(1), Length:16 Data: e9ed05b6f04d7aeac4373db6f109eef7 (validated)
    Name                : vtss
    Content file name   : vtss-rootfs.tar.xz
    Content file length : 2942552
Stage2 - Index:2
  Tlv FsElement(2), Data Length:488963
  MD5(1), Length:16 Data: 233048efebce4424331f09ccd8b4c448 (validated)
    Name                : custom-web
    Content file name   : /home/wjin/custom_web.tar.xz
    Content file length : 488900
```

Load the new image onto the device and the desired HTML files should be accessible now.

We could check it via the commands below.

```
# platform debug allow
# debug system shell
/ # ls -la /var/www/webstax/ | grep custom_web
-rw-r--r--    1 root     root            98 Jan  1 00:00
# custom_web.html
```

Log in the device and open the customized Web page, we could see the port status:



2.2. Quagga Integration

`Quagga` is a routing software suite supporting most of the main routing protocols such as RIP, OSPF, etc. It contains several daemons, one for each protocol, and one called `zebra` for interface declaration and static routing. In this use case, we will spawn `zebra` as a static routing daemon via `ServiceD`.

First a `quagga.service` configuration file, telling `ServiceD` how to start and manage it, is necessary. Then we will cross compile and build `zebra` from a `quagga` release. Appending it upon a `WebStaX` release is the last step.

2.2.1. Service Configuration File

A simple 'quagga' service configuration file could be:

quagga.service

```
name = quagga
type = service
depend = switch_app
cmd = /usr/sbin/zebra -f /etc/quagga/zebra.conf -i /tmp/q.pid
```

You will need to put this file under `/tmp/quagga_install/etc/mscc/service` on your development PC, create these directories if needed. It will eventually be compressed into the final MFI image.

NOTE: do not append "-d" or "--daemon" in the "cmd" line, as ServiceD will do it implicitely.

2.2.2. Configure, Compile and Install

Besides the service configuration file, we need to have quagga configured, compiled and installed for the target device.

Configure

Certain environment variables need to be set correctly as the very first step, such as `PATH`, `GCC`, `LD`, etc.

build_quagga.sh

```
export PATH="/opt/mscc/mscc-brsdk-mips-vXX.XX/stage2/smb/x86_64-linux/usr/bin:/usr/
bin:/bin:/usr/local/bin:/usr/local/sbin"

export LD="mipsel-buildroot-linux-uclibc-ld"
export CC="mipsel-buildroot-linux-uclibc-gcc"
export GCC="mipsel-buildroot-linux-uclibc-gcc"
export STRIP="mipsel-buildroot-linux-uclibc-strip"
export CFLAGS=" -I/tmp/quagga_install/usr/include/"
export LDFLAGS=" -L/tmp/quagga_install/usr/lib/"

## Make sure MSCC SDK is in place via checking gcc version
$GCC --version
if [ "$?" -ne 0 ]
then
    echo "PATH is wrong!"
    exit 1
fi
```

A MSCC SDK that is supporting `ServiceD` has to be correctly set in `PATH`.

Like compiling any Linux software that comes with source code you may simply follow the usual procedure, i.e.,

build_quagga.sh

```
./configure \
    --target=mipsel-buildroot-linux-uclibc \
    --host=mipsel-buildroot-linux-uclibc \
    --build=x86_64-unknown-linux-gnu \
    --prefix=/tmp/quagga_install/usr \
    --sysconfdir=/tmp/quagga_install/etc \
    --localstatedir=/tmp \
    --enable-user=root \
    --enable-group=root \
    --program-prefix="" \
    --enable-zebra
 make
 make install
```

As User manual @ **Quagga** (http://www.nongnu.org/quagga/docs/docs-info.html#Sample-Config-File) mentioned, `zebra.conf` file is needed to start `zebra` and it should be placed under `/tmp/quagga_install/etc/quagga` folder.

zebra.conf

```
hostname Router
password zebra
enable password zebra
interface lo
interface sit0
log file zebra.log
```

The very last step of `build_quagga.sh` is to compress all the files for later appending it by mfi.rb script.

build_quagga.sh

```
## Clean up
rm -rf /tmp/quagga_install/usr/share
rm -rf /tmp/quagga_install/usr/include

## Copy service config file, zebra daemon config file
mkdir -p /tmp/quagga_install/etc/mscc/service
cp quagga.service /tmp/quagga_install/etc/mscc/service/.
mkdir /tmp/quagga_install/etc/quagga
cp zebra.conf /tmp/quagga_install/etc/quagga/.

## Compress with the correct dir hierarchy
tar -C /tmp/quagga_install/ -vc usr etc -f quagga.tar
/usr/bin/xz --check=none quagga.tar
```

At this point, there should be a file named `quagga.tar.xz` generated.

2.2.3. Append

This can be automated by utilizing script `append_quagga.sh` below. We assume that the user already has a valid MFI image `<targeted switch>.mfi` for the targeted device and the mfi.rb script is in place as well.

append_quagga.sh

```
/<mfi script dir>/mfi.rb \
    -i <targeted switch>.mfi \
    -o <targeted switch>_quagga.mfi rootfs \
    --action append \
    --name "quagga_zebra" \
    --file /<some dir>/quagga.tar.xz
```

Now the final MFI image named `<targeted switch>_quagga.mfi` in this case is available for use.

Load it on the target device and issue the commands below. As the log shown, `ServiceD` successfully spawned `zebra` daemon.

```
# platform debug allow
# debug system shell
/ # ps
PID   USER     COMMAND
    1 root     /usr/bin/stage1-loader

   94 root     /bin/sh -c /usr/sbin/zebra -f /etc/quagga/zebra.conf -i /tmp/q.p
   95 root     /usr/sbin/zebra -f /etc/quagga/zebra.conf -i /tmp/q.pid
```

It is also possible to telnet to zebra port-2601 of the device from your development PC now, see log below. Hostname and password are set in the `zebra.conf` file we have defined previously.

```
$ telnet 10.99.99.25 2601
Trying 10.99.99.25...
Connected to 10.99.99.25.
Escape character is '^]'.

Hello, this is Quagga (version 0.99.24.1).
Copyright 1996-2005 Kunihiro Ishiguro, et al.


User Access Verification

Password:
Router>
```

2.3. JSON CLI-Client

As a follow-up of JSON Message Examples, we will try to create a user application to get the device port status via JSON-IPC.

json_ipc.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
    const char *msg = "{\"method\":\"port.status.get\", \"params\":[], \"id\":1}";
    struct sockaddr_un remote = {
        .sun_family = AF_UNIX,                  // Socket type
        .sun_path = "/var/run/json_ipc.socket"  // Path of the JSON_IPC pipe
    };

    int s = socket(AF_UNIX, SOCK_STREAM, 0); // create socket
    connect(s, (struct sockaddr *)&remote, sizeof(remote)); // connect it

    int i = strlen(msg);
    write(s, &i, sizeof(i));        // Write size of message
    write(s, msg, i);               // Write message

    read(s, &i, sizeof(i));         // Read sizeof response
    char *res = calloc(i + 1, 1);   // Allocate memory for the response and null
terminate
    read(s, res, i);                // Read response

    printf("Response: %s\n", res);

    free(res);
    close(s);

    return 0;
}
```

Save the C source file above and cross compile it by utilizing MSCC SDK, and then append its executable file upon a MFI image.

Run the script below to generate a MFI image containing an usr application. `json_ipc.out`.

json_ipc.sh

```
/opt/mscc/mscc-brsdk-mips-vXX.XX/stage2/smb/x86_64-linux/usr/bin/
mipsel-buildroot-linux-uclibc-gcc \
    -Wall -o json_ipc.out json_ipc.c

mkdir -p json_ipc/usr/bin/
cp json_ipc.out json_ipc/usr/bin/.
tar -C json_ipc -cvf json_ipc.tar usr/
xz --check=none json_ipc.tar

/opt/mscc/mscc-brsdk-mips-vXX.XX/stage1/x86_64-linux/usr/bin/mfi.rb \
    -i <targeted switch>.mfi \
    -o json_ipc.mfi rootfs \
    --action append \
    --name "json_ipc" \
    --file json_ipc.tar.xz

## Optional, tlv check
/opt/mscc/mscc-brsdk-mips-vXX.XX/stage1/x86_64-linux/usr/bin/mfi.rb \
    -i json_ipc.mfi \
    -d
```

Load it on the device and issue commands below to get the port status:

```
# debug system shell
/ # /usr/bin/json_ipc.out
Response: {"id":1,"error":null,"result":[{"key":"Gi
1/
1","val":{"Link":false,"Fdx":false,"Fiber":false,"Speed":"undefined","SFPType":"none"
,"SFPVendorName":"","SFPVendorPN":"","SFPVendorRev":"","LossOfSignal":false,"TxFault"
:false,"Present":false,"SFPVendorSN":""}},]}
```

NOTE: Response is partially shown for space concern.

2.4. Custom Default Configuration

The default configuration of ENT switch products can be customized as well. In this use case, we are going to assign a different default IP address 192.168.0.1 other than MSCC factory default (DHCP).

Execute the script below to customize the default IP address.

custom_default-config.sh

```
mkdir -p default_config/etc/mscc/icfg/
cat > default_config/etc/mscc/icfg/default-config <<\EOF
! Default configuration file
! -------------------------
!
! This file is read and applied immediately after the system configuration is
! reset to default. The file is read-only and cannot be modified.

vlan 1
 name default

interface vlan 1
 ip address 192.168.0.1 255.255.255.0

end
EOF
tar -C default_config -cf default_config.tar etc/
xz --check=none default_config.tar

/opt/mscc/mscc-brsdk-mips-vXX.XX/stage1/x86_64-linux/usr/bin/mfi.rb \
    -i <some dir>/<targeted switch>.mfi \
    -o <targeted switch>_default_config.mfi rootfs \
    --action append \
    --name "default_config" \
    --file default_config.tar.xz

## Optional, tlc check
/opt/mscc/mscc-brsdk-mips-vXX.XX/stage1/x86_64-linux/usr/bin/mfi.rb \
    -i <targeted switch>_default_config.mfi \
    -d
```

After executing the script, a new MFI image `<targeted switch>_default_config.mfi` is generated. Load it on the device and use two commands below to check the newly installed customized default configuration.

- `show running-config`
- `reload default`

```
# show running-config
## ... Non-related log omitted
interface vlan 1
 ip address 10.99.99.25 255.255.255.0
## ...
# reload default
% Reloading defaults. Please stand by.
# show running-config
## ...
interface vlan 1
 ip address 192.168.0.1 255.255.255.0
## ...
# platform debug allow
# debug system shell
/ # ls -la /switch/icfg/default-config
lrwxrwxrwx    1 root     root            29 Jan  1 00:00
#/switch/icfg/default-config -> /etc/mscc/icfg/default-config
```

As we can see from the log, 192.168.0.1 becomes the default IP address every time the user `reload default`.