ENT-AN1163-4.1 Application Note
Linux Customizations

Microsemi
Power Matters.™

Microsemi Corporate Headquarters
One Enterprise, Aliso Viejo,
CA 92656 USA
Within the USA: +1 (800) 713-4113
Outside the USA: +1 (949) 380-6100
Fax: +1 (949) 215-4996
Email: sales.support@microsemi.com
www.microsemi.com

About Microsemi

Microsemi Corporation (Nasdaq: MSCC) offers a comprehensive portfolio of semiconductor and system solutions for aerospace & defense, communications, data center and industrial markets. Products include high-performance and radiation-hardened analog mixed-signal integrated circuits, FPGAs, SoCs and ASICs; power management products; timing and synchronization devices and precise time solutions, setting the world's standard for time; voice processing devices; RF solutions; discrete components; enterprise storage and communication solutions, security technologies and scalable anti-tamper products; Ethernet solutions; Power-over-Ethernet ICs and midspans; as well as custom design capabilities and services. Microsemi is headquartered in Aliso Viejo, California, and has approximately 4,800 employees globally. Learn more at www.microsemi.com.

VPPD-04449. 1.0 6/17

Contents

Figures

Tables

1 Revision History

The revision history describes the changes that were implemented in the document. The changes are listed by revision, starting with the most current publication.

1.1 Revision 1.0

Revision 1.0 was published in June 2017. It was the first publication of this document.

2 Linux Customizations

This document provides a collection of options to customize the firmware images managed by the WebStaX software application used in the Microsemi switch products. All facilities presented here do not require source code access for the software, but do require a board support package (BSP), and can be used to customize without recompiling the WebStaX image. This document mostly focuses on simple customization facilities. For more advanced customization, knowledge of embedded Linux and C programming is required.

The document presents a set of customization facilities that can be used and combined and a few use cases with examples of how the different facilities can be used.

2.1 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.

2.1.1 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: stage1 and stage2. Stage1 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 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-squash  Inspect or alter the rootfs tlv(s) in the firmware image (assuming squashfs).
```

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.

2.1.1.1 Firmware File Format

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.

2.1.1.1.1 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.

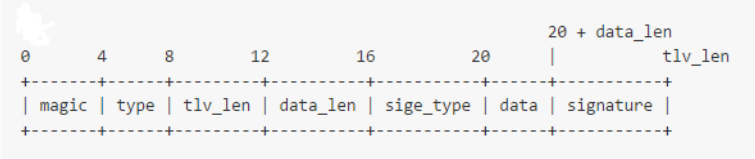
2.1.1.1.2 Stage 2

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, stage2 TLVs embed the signature directly into each TLV. This means that the stage2 TLVs is signed individually. The binary layout is as follows:

Figure 1 • **Stage2 TLVs Binary Layout**



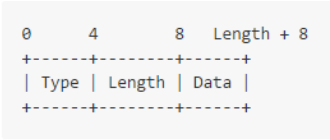
The supported stage2 TLV types are documented by the following enumeration (currently only ROOTFS is supported):

```
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:

Figure 2 • **ROOTFS TLV Header**



The following sub-tlv types are supported by the root file system elements.

- Name—an ASCII encoded string with the name of this element.
- Version—an ASCII encoded string with the version information of this element.
- License terms—an ASCII encoded string with the license terms of this element.
- PreExec—an executable that is being invoked before the tar archive is extracted into the root file system. Not implemented in current release.
- Content—an xz compressed tar archive or squashfs filesystem.
- 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.

2.1.1.2 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, and so should work across platforms (WIN, OSX, and LINUX) as long as Ruby is supported and properly installed.

The 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-squash  Inspect or alter the rootfs tlv(s) in the firmware image (assuming squashfs).
```

Note: The MFI tool currently supports submodules, such as stage1, bootloader, and rootfs. More submodules might be added in the future.

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
```

Stage1 is composed as follows:

```
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. Because 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 are 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-squash\
       --action append \
       --name "rootfs" \
       --version "SDK_VERSION" \
       --file <path_to_rootfs_file>.squashfs
```

2.1.2 ServiceD

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

2.1.2.1 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
env = FOO=bar
env = Hello=world
# depend =
cmd = /usr/bin/switch_app
ready_file = /tmp/switch_app.ready
```

2.1.2.2 Appearance and Requirements

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

Table 1 • **Fields in ServiceD Configuration File**

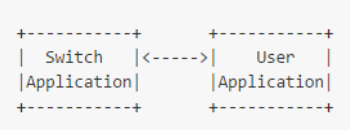
Fields	Appearance and Requirements	Description
name	Must appear once	Name of service (only used for debugging and logging).
type	Allow once	Type of service. Allowed values are: Service—service is a long running process that will be auto-restarted by ServiceD, if it exits. Its ready file will be also deleted until it is ready again. Oneshot—If the oneshot process executes and exits normally, it will be seen and its ready file will be created accordingly, on restart. Otherwise, it is seen as non-ready and will not be restarted.
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 that 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.
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.

2.1.3 JSON-IPC

The WebStaX switch application includes a JSON Inter-Process Communication (IPC) module that provides 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.

Figure 3 • **JSON-IPC Architecture**

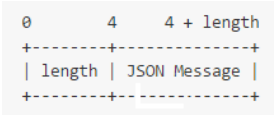


2.1.3.1 IPC Message Format

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 Add registration (`jsonIpc.config.notification.add`)method and Delete registration (`jsonIpc.config.notification.del`) method.

Figure 4 • **IPC Message Format**



2.1.3.2 JSON Message Examples

The following examples only show the JSON message part of the JSON IPC message (the length of 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 de-registration (port status update) are 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"}
```

2.1.4 Boot-time Configuration

The switch application includes a number of features that 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 through console port
- CLI through Telnet
- CLI through 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
  }
}
```

2.2 Use Cases

The following sections discusses various use cases with examples of how the different facilities can be used.

2.2.1 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 through JSON interface will be demonstrated.

2.2.1.1 Create Custom Web File

Execute the following script (`custom_web.sh`)to create a custom web file.

```
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
- Microsemi switch application

```
mkdir -p etc
cat > etc/switch.conf <<\EOF
{
    "web":{
        "enable":true,
        "handlers":false
    }
}
EOF
mksquashfs ./custom-web custom_web.squashfs
```

Now a new file, `custom_web.squashfs`, is available for replacement.

2.2.1.2 Replace

Before replacing the file, `custom_web.squashfs`, we 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.squashfs
    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.squashfs
    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.squashfs
    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.squashfs`.

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

At this point, a image named `<targeted switch>_custom_web.mfi` is generated. Execute the commands below to inspect it again.

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

The example output is shown as below (note that the Index 2 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.squashfs
    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.squashfs
    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.squashfs
  Content file length : 488900
```

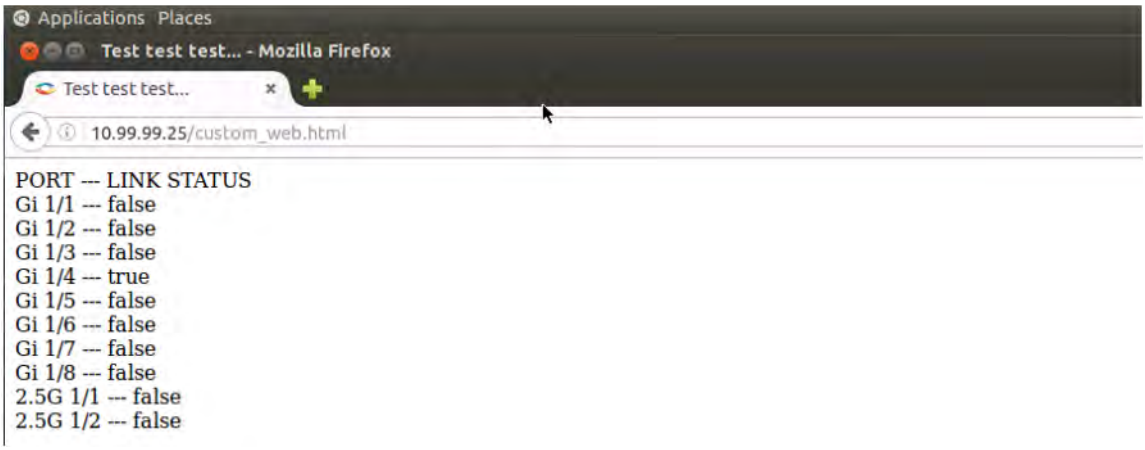
Load the new image onto the device and the desired HTML files should be accessible now.

Verify it through the following commands.

```
# 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 on to the device and open the customized web page to see the port status.

Figure 5 • **Custom Web: Port Status**



2.2.2 Quagga Integration

Quagga is a routing software suite supporting most of the main routing protocols, such as RIP and OSPF. 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 through ServiceD.

First a `quagga.service` configuration file, informing 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.

2.2.2.1 Service Configuration File

A simple quagga service configuration file (`quagga.service`)can be:

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

You 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 automatically.

2.2.2.2 Configure, Compile, and Install

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

As the very first step, set the environment variables, such as `PATH`, `GCC`, and `LD` in `build_quagga.sh` file correctly. A Microsemi SDK, supporting ServiceD, has to be correctly set in `PATH` as well.

```
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 through checking gcc version
$GCC --version
if [ "$?" -ne 0 ]
then
    echo "PATH is wrong!"
    exit 1
fi
```

Next, follow the usual procedure to compile as any other Linux software that comes with source code.

```
./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
```

The following `zebra.conf` file is needed to start zebra and it should be placed under the `/tmp/quagga_install/etc/quagga` folder. (For more information, see the `Quagga User manual`.)

```
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 appending it later by `mfi.rb` script.

```
## 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
mksquashfs /tmp/quagga_install/*
quagga.squashfs -comp xz -all-root
```

At this point, a file named, `quagga.squashfs`, should be generated after executing the shell script `build_quagga.sh`.

2.2.2.3 Append

This operation can be automated by utilizing the 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.

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

The final MFI image named `<targeted switch>_quagga.mfi` is now available for use. Load it on the target device and issue the commands below. As per the log, ServiceD has spawned zebra daemon successfully.

```
# 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>
```

2.2.3 JSON CLI-Client

As a follow-up to JSON Message Examples, page 7, we will create the following user application, `json_ipc.c`, to get the device port status through JSON-IPC (see JSON-IPC, page 6).

```
#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 the Microsemi SDK and then append its executable file upon a MFI image.

Run the following script (`json_ipc.sh`) to generate a MFI image containing a user application: `json_ipc.out`.

```
/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/.
mksquashfs json_ipc/. json_ipc.squashfs

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

## 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 the following commands 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","SFPT
ype":"none","SFPVendorName":"","SFPVendorPN":"","SFPVendorRev":"","LossOfSig
nal":false,"TxFault":false,"Present":false,"SFPVendorSN":""}},]}
```

Note: Response is partially shown due to space constraints.

2.2.4 Custom Default Configuration

The default configuration of Microsemi 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) instead of the Microsemi factory default (192.0.2.1).

Execute the following script (`custom_default-config.sh`) to customize the default IP address.

```
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
mksquashfs ./default_config default_config.squashfs

/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-squash \
    --action append \
    --name "default_config" \
    --file default_config.squashfs

## 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 the following commands 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 performs ICLI command `reload default`.