



mpi3mr Linux Device Driver

Usage Guide and Known Limitations Document

Contents

1. Overview	2
2. Driver Compilation Steps	2
3. OS Support Matrix	2
4. Release Contents	2
5. Errata and Known Limitations	3
6. Usage Guide to Install/Remove/Upgrade Driver Package.....	4
7. Usage Guide to generate the binary RPM from the source RPM.....	5
8. Usage Guide for DKMS source RPM/DEB for Ubuntu.....	7
9. Supported Module Parameters	8
10. Signed Source RPM support	8
11. Signed Binary Support for RHEL/SLES	9
12. RHCOS and SLE Micro Container OS.....	9
13. Notes	11

1. Overview

This README document covers usage details and limitation information about Broadcom's mpi3mr Linux device driver for MPI3 based storage I/O controllers. This document doesn't cover details about OS distributions and for any OS distro-specific limitation and information, please check with OS Vendor support.

2. Driver Compilation Steps

The driver source code is placed inside the released driver package as a compressed tar file with the name – mpi3mr-<driver_version>-src.tar.gz. Below are the steps to compile the driver for the supported kernel versions. If the driver is compiled for kernel versions that are not defined in the os_support.txt then the compilation might fail or there might be some functional issues, if there is an issue seen please contact the Broadcom support team for proceeding further.

- 1) Untar the driver source tarball, make sure the directory path doesn't contain any spaces to avoid compilation errors.
#tar -zxvf mpi3mr-<driver_version>-src.tar.gz
- 2) Goto the driver source directory
#cd mpi3mr
- 3) To compile the driver use the helper script "compile.sh" bundled inside source code
#./compile.sh
- 4) To load and unload the driver for distro use the helper scripts "load.sh" and "unload.sh" bundled inside source code
#./load.sh
#./unload.sh

3. OS Support Matrix

Please refer to the file "os_support.txt" available inside the source tarball.

4. Release Contents

The driver release contains,

- This readme file
- Source tarball
- Generic source RPM

- OS distro-specific source RPMs, binary RPMs, dkms, driver update disk and Broadcom secure signed RPMs.
- Broadcom's public key for signed RPM

The OS distro-specific binary RPMs and source RPMs are provided only for the OS distributions provided in "os-support.txt" file. If you do not find binary level support for your distribution in release contents, please use the source RPM method provided in the sections below or contact the Broadcom support team for further help.

5. Errata and Known Limitations

This section covers the known issues, limitation identified in the driver and the possible workarounds if any had been identified, this list is updated as and when issues are identified and if those are considered as errata or the limitations that exist in the driver. This is not comprehensive list for all known defects in the driver and doesn't cover OS distro-specific issues identified during the validation of this driver.

1. RPM install dependency issue

KMOD RPM uses KMOD packaging dependency data to ensure the dependencies are met before installing the binary RPM. The OSV maintains an allowed-list of kernel symbols which RPM uses to validate against the KMOD binaries. Some symbols may be in the kernel but not in the allowed-list which results in a failed binary RPM install. Users can use the "--nodeps" switch when installing the RPM to skip those symbol checks or any other dependency.

Example #rpm -ivh --nodeps kmod-mpi3mr-vxxxxxx_UEK.xxx.rpm

Please use the "--nodeps" option under your own discretion and if there are any issues please contact the Broadcom support team for further help.

2. Regarding messages that driver displays when controller reset occurs while device addition/removal is pending at SCSI Mid Layer

In the driver log if any of below mentioned messages is observed then that indicates the driver could not properly cleanup pending firmware events after controller reset and the reason for device addition/removal during the controller reset needs to be analyzed to identify the impact of the processing of the stale events.

"Device addition was under process before the reset and completed after reset, verify whether the exposed devices are matched with attached devices for correctness"

(or)

“Device removal was under process before the reset and completed after reset, verify whether the exposed devices are matched with attached devices for correctness”

6. Usage Guide to Install/Remove/Upgrade Driver Package

The driver release may support two types of packaging format for binary RPMs based on supported OS distributions.

1. RPM(RHEL/SLES/Fedora/OEL uses RPM package)
2. DEB(Ubuntu/Debian uses DEB package) when binary support needs to be provided.

Please note that after install/removal of the driver package, the system needs to be rebooted to get the intended driver loaded or the user has to manually remove the driver module and insert the module again (please refer to man pages for “insmod” “rmmod” and “modprobe” for more info)

Whenever the driver package is installed/uninstalled/upgraded, it will be a good practice to check the output of the command “*#modinfo mpi3mr*” to make sure the intended action is properly committed.

To verify the driver is successfully loaded and if the version loaded is as intended, please run the below command and validate the driver version.

```
#cat /sys/modules/mpi3mr/version
```

It is always recommended to verify initramfs image to confirm updated mpi3mr driver is packaged correctly before system reboot. Do not assume that latest driver is loaded after installing rpm generated from source rpm. See OS distribution specific documentation on how to verify initramfs image (Example: [FC21 user guide link for reference](#)).

Steps for Driver install/remove/upgrade for .rpm package

1. To install:

```
#rpm -ivh <DRIVER_PACKAGE>.rpm
```
2. To uninstall:

```
#rpm -qa | grep mpi3mr
```

 (the output of the command will give the package name if installed)

```
#rpm -e <output from the command above>
```
3. To upgrade (installing on top of previously installed RPM without uninstalling):

```
#rpm -Uvh <DRIVER_PACKAGE>.rpm
```

Steps for Driver install/remove/upgrade for .deb package

1. To install:

```
#dpkg -i <DRIVER_PACKAGE>.deb
```
2. To uninstall:

```
#dpkg -r mpi3mr
```
3. To verify the status of the installed deb package

```
#dpkg -s mpi3mr
```

7. Usage Guide to generate the binary RPM from the source RPM

There are three variants of the source RPMS available in this package

1. Source RPM which uses the kmodtool interface (RHEL based)
<http://rpmfusion.org/Packaging/KernelModules/Kmods2>
2. Source RPM which uses the kmp interface (SLES based)
https://en.opensuse.org/Kernel_Module_Packages
3. Source RPM which uses the generic build interface (Creating initramfs internally and without depending upon any external tool)

See the “Notes” section for the sample naming convention used for the three flavors of source RPMs. If the user doesn’t know which source RPM is a better choice for their environment (for any distros other than Redhat/Novell) it is recommended to try the generic source rpm to start with. A quick search of “`rpm -qa |grep kmod`” can provide a hint about if kmod tool support is available in the specific distro or not.

To generate the binary RPM from the source RPM, the build system used should have compilation/build environment to create kernel modules, utilities to build RPM(e.g. rpmbuild..) etc. It is required to install “*kernel-devel*” or “*linux-headers*” packages depending on the distro for setting up the compilation environment.

For example, in a system where the yum repo is configured, use the below steps to get the compilation environment setup.

```
#yum groupinstall “Development Tools”
```

In the Ubuntu distributions the missing compilation environment can be installed using the commands mentioned below.

```
#apt-get install rpm
```

```
#apt-get install make
#apt-get install gcc
#apt-get install alien
```

Exact commands that are required to create the build environment would be different across different distros and this document does not cover those details. Please refer to the OS vendor documentation for comprehensive details on how to setup the build environment.

Installation of the source RPM

Execute the below command to install the source RPM

```
#rpm -ivvh mpi3mr-<driver_version>.src.rpm
```

Installing the above RPM will copy the driver SPEC file to a specific location (configured as part of RPM packaging. That path can be different for each OS distribution.) To locate SPEC file, check the output of the above source RPM installation (see blue marked gives SPEC file location).

(Example)

```
[root@localhost tmp]# rpm -ivvh mpi3mr-8.0.0.60.0-1.src.rpm
D: ===== mpi3mr-8.0.0.60.0-1.src.rpm
..
Updating / installing...
1:mpi3mr-8.0.0.60.0-1
##### [100%]
D: ===== Directories not explicitly included in package:
D:          0 /root/rpmbuild/SOURCES/
D:          1 /root/rpmbuild/SPECS/
D: =====
D:  unknown 100755 1 ( 0, 0) 25
/root/rpmbuild/SOURCES/Module.supported;
D:  unknown 100644 1 ( 0, 0)120552 /root/rpmbuild/SOURCES/mpi3mr-
06.810.00.02.tar.gz;
D: unknown 100644 1 ( 0, 0) 6783
/root/rpmbuild/SPECS/mpi3mr.spec;
GZDIO:          17  reads,          127888 total bytes in 0.000585 secs
D: closed          db index          /var/lib/rpm/Name
D: closed          db index          /var/lib/rpm/Packages
D: closed          db environment /var/lib/rpm
```

Building the binary RPM

Go to the directory where the driver SPEC file is copied as part of the above step. There must be SPEC file in one of the names with mpi3mr in then name (the driver SPEC file names could be

different for different distros). Check spec file name seen in your source RPM install as similar to highlighted in the above step. It provides the location and spec filename. Execute the below command to build the binary rpm

```
#rpmbuild -ba <DRIVER_SPEC_FILE>
```

Note: for Fedora execute the command:

```
#rpmbuild -ba --define "debug_package %{nil}" <DRIVER_SPEC_FILE>
```

(where “debug_package”(debuginfo) is mandatory on Fedora23 and later versions)

Binary RPMs will be available if the rpmbuild command executes without any error. Go to the directory where a new binary RPM is generated.

Example

Snippet of working case –

```
--
Wrote: /root/rpmbuild/SRPMS/mpi3mr-8.0.0.60.0-1.src.rpm
Wrote: /root/rpmbuild/RPMS/x86_64/mpi3mr-8.0.0.60.0-1.x86_64.rpm
Wrote: /root/rpmbuild/RPMS/x86_64/mpi3mr-debuginfo-8.0.0.60.0-1.x86_64.rpm
Executing(%clean): /bin/sh -e /var/tmp/rpm-tmp.ZbHbmH
+ umask 022
+ cd /root/rpmbuild/BUILD
+ cd mpi3mr-8.0.0.60.0-1
+ /usr/bin/rm -rf /root/rpmbuild/BUILDROOT/mpi3mr-8.0.0.60.0-1.x86_64
+ exit 0
--
```

Step #5 is applicable only for deb package-based OS distribution (Ubuntu/Debian).

For deb package based OS distribution (Ubuntu/Debian), create driver .deb package from binary RPM using the below command

```
#alien -k --to-deb --scripts <GENERATED_DRIVER_RPM>
```

8. Usage Guide for DKMS source RPM/DEB for Ubuntu

Canonical recommends that driver support should be provided through DKMS based driver source deb package for Ubuntu distributions. A single driver source Deb package which should work for all Ubuntu versions supported by Broadcom is provided as part of the release package. This section provides some information that will help to use DKMS based source RPM/DEB packages.

To Install dkms: `#apt-get install dkms`

To install driver DEB package: `#dpkg -i dkms-<version>-<release>.noarch.deb`

To install driver RPM package: `#rpm -ivh dkms-<version>-<release>.noarch.rpm`

The above steps are prerequisites for installing DKMS-enabled module RPMs. DKMS enabled module DEB/RPM can be installed like any other Deb/RPM after executing the above commands

With a DKMS enabled module DEB/RPM, most of the installation work done by the RPM is actually handed off to DKMS within the RPM/DEB. Generally the DKMS does the following:

1. Installs module source into /usr/src/<module>-<moduleversion>/
2. Places a dkms.conf file into /usr/src/<module>-<moduleversion>/
3. Executes: # dkms add -m <module> -v <version>
4. Executes: # dkms build -m <module> -v <version>
5. Executes: # dkms install -m <module> -v <version>

Once the RPM/DEB installation is complete, the DKMS command can be used to understand which module and which module version is installed on which kernels. That can be accomplished via the command:

```
# dkms status
```

From this point various DKMS commands (eg. add, build, install, uninstall) can be used to load the driver module onto other kernels. For more details, please refer to

<https://github.com/dell/dkms>

9. Supported Module Parameters

Please refer to the file “*moduleparams*” available inside the source tarball.

10. Signed Source RPM support

The source RPMs in this release package are signed/encrypted with GnuPG public-private encryption scheme to ensure the authenticity of the source RPMs. GPG public-private key pair is 4k bit long with RSA encryption algorithm and all Source RPMS are signed with the same. The private key is held by Broadcom and the associated public key is kept on the Broadcom website.

Below is the verification guide link which contains the detailed steps to verify the source RPM and the link to download the public key file.

Verification guide link: [VERIFICATION-GUIDE](#)

Public key link: [PUBLIC-KEY](#)

Note: The verification guide also contains the public key link.

11. Signed Binary Support for RHEL/SLES

This release package contains unsigned binaries for all the supported OS's. In addition to the unsigned binaries, this package also contains signed binaries for SLES and RHEL. The directories prefixed with the string "signed" contains signed driver binaries.

Example:

signed_sles15 => signed sles15 binaries;

sles15 => unsigned sles15 binaries;

Broadcom signs SLES and RHEL driver binaries with Broadcom owned public-private key pair. The private key is held by Broadcom and associated public key is provided as part of this release package with the name "DCSG00411462_selfcert.der".

To use signed the driver, Please refer the OS vendor documentation.

SUSE: https://drivers.suse.com/doc/Usage/Secure_Boot_Certificate.html#secure-boot-certificate

12. RHCOS and SLE Micro Container OS

RHCOS Description:

RHCOS is a lightweight, container operating system based on Red Hat Enterprise Linux. It is Immutable OS, meaning most of the file system is read-only and cannot be modified directly. As a result, traditional tools like rpm cannot be used to install packages. Instead, RHCOS uses a specialized tool called rpm-ostree to install and manage RPM packages.

RHCOS Installation:

Detailed installation steps are available on the Red Hat website. - Key steps to note during RHCOS installation: Use the coreos-installer tool to install RHCOS to disk. During installation, supply an Ignition file containing the SSH public key from a remote Linux machine. After installation, direct login is not allowed, access the system via password-less SSH from the machine whose public key was provided to RHCOS through ignition file.

Driver RPM Installation:

1. Enable initramfs generation before installing the driver RPM. Otherwise, the driver will not be included in the initramfs:
`# sudo rpm-ostree initramfs --enable`
2. Enable syncing of /etc files into initramfs. This ensures files like /etc/depmod.d/mpi3mr.conf and /etc/dracut.conf.d/ mpi3mr.conf are included in the initramfs during future kernel upgrades:
`# sudo rpm-ostree initramfs-etc --force-sync`
3. Copy the `kmod-mpi3mr -<version>.rpm` to the RHCOS system.
4. Install the driver:
`# sudo rpm-ostree install kmod-mpi3mr-<version>.rpm`
5. Reboot the system:
`# sudo systemctl reboot`

Driver RPM Un-Installation:

1. Uninstall the driver:
`# sudo rpm-ostree uninstall kmod-mpi3mr-<version>`
2. Reboot the system:
`# sudo systemctl reboot`
3. Disable initramfs:
`# sudo rpm-ostree initramfs --disable`
`# sudo rpm-ostree initramfs-etc --untrack-all`

SLE Micro OS Description:

SLE Micro is a lightweight, container operating system based on SUSE Enterprise Linux. It follows the Immutable Infrastructure concept, where System updates are typically applied using "transactional-update" tool.

Driver RPM Installation:

1. Copy the driver RPM to the SLE Micro system.
2. Install the driver:
`# transactional-update pkg install mpi3mr-kmp-default-<version>.rpm`
3. sudo reboot

Driver RPM Un-Installation:

1. sudo rpm -e mpi3mr-kmp-default-<version>
2. sudo reboot

13. Notes

Initramfs creation failure during RPM installation:

If initramfs creation is failed during the RPM installation, try generating initrd manually. Make sure “*#modinfo mpi3mr*” provides the expected driver version prior to creating initrd .

Compatibility issues between prebuilt binary RPM and RPM built from source RPM:

Do not try to install/upgrade from the binary RPM generated from the source RPM along with the pre-compiled binary RPMs provided by Broadcom. Naming convention of the pre-compiled binary RPM and sourceRPM-based binary RPM differs and that will create issues and it is recommended to stick with one method to avoid compatibility issues.

Source RPM method is the quickest method to deploy the driver on many supported kernel versions, whereas precompiled binary is specific to a kernel version or set of kernel versions.

Example: Source rpm based binary will be generated in the below format -

kmp_rpm: broadcom-mpi3mr-8.0.0.60.0-1.src.rpm

kmod_rpm: kmod-mpi3mr-8.0.0.60.0-1.src.rpm

generic_rpm: mpi3mr-8.0.0.60.0-1.src.rpm

Kernel updates require prebuilt binary RPM update:

In a system where a prebuilt binary RPM is installed for kernel X and OS distro Y, if the kernel is updated to X+n or the OS release is updated to Y+m (which has the kernel X+n) then the existing driver installed through prebuilt binary RPM for old kernel X cannot be guaranteed to work after the update if there is a KABI change between the existing kernel and updated kernel. This requires the user to install the driver again for the updated kernel/OS release from the binary RPM matching the updated kernel version or from the source.

Detailed Notes about controller fault with code 0xF000:

A controller fault with 0xF000 fault code is not due to an exception detected by the firmware. It is a reaction of the firmware for the driver-issued diagnostic fault reset and it shouldn't be treated as a critical error always by default without identifying the original reason for diagnostic fault reset.

Due to various reasons, the driver could issue a diagnostic fault reset and the driver clearly prints the reason for diagnostic fault reset as a readable string in the format of ***"diag fault reset due to XYZ"***. That string has to be inferred to understand the real reason for the 0xF000 fault and based on that the 0xF000 fault can be treated as a critical issue or not.

Some of the reasons for which the driver issues diagnostic fault reset which leads to 0xF000 fault are captured below:

- *Any admin command times out (IOC init, port enable, diag buffer post, timestamp update etc.,)*
- *Any IOCTL times out (if the IOCTL is neither a SCSI IO nor an NVMe Encapsulated command)*
- *Any task management command timeout*
- *An application issued diag fault reset through IOCTL interface*
- *OS SCSI Mid Layer error handler invoked host reset*
- *Failures (e.g. memory allocation failure, interrupt setup failure, controller does not transition to READY state etc..) during driver load/controller initialization sequence.*

In the above list, all except the last two (anything with a reason string in the driver log for diag fault reset with text containing timeout or failure) may be treated as an error directly.

Example strings that can be triaged as an error are:

"mpi3mr0: diag fault reset due to IOCTL timeout"

"mpi3mr0: diag fault reset due to task management timeout"

The diag fault reset issued from applications (with the reason string containing *"application invocation"* or *"sysfs invocation"*) shouldn't be treated as a critical error as the reset is initiated by the user.

The diag fault reset issued from SCSI error handler host reset call back (with the reason string as *"diag fault reset due to host reset from the OS"*) should be analyzed in more detail on case by case basis and to be decided either as an expected behavior from the driver/kernel design or not. Below are a few cases due to which the diagnostic fault reset could be issued by the driver as a response to OS-issued host reset.

- If a privileged user issues `sg_reset -h /dev/sgX` for a JBOD then the driver will issue the diag fault and the firmware will fault with fault code 0xF000. This is expected behavior and should not be treated as an error.
- If the user issued (using `sg_reset -d /dev/sgX` or `sg_reset -t /dev/sgX`) or kernel error handler issued LUN reset or target reset is failed due to device getting removed then the failure will escalate to the 0xF000 fault and if the device removal is expected then this should not be treated as error otherwise the reason for the device removal needs to be analyzed.
- If the diag fault reset is invoked by the host reset handler due to normal error recovery escalation then that could indicate an issue. The OS error recovery escalation could happen due to Test Unit Ready (TUR) command failure observed during error handling. The OS error handler issues a TUR to check the readiness of a drive post LUN/Target reset and if that TUR fails due to any reason then the error handler escalates to the next level reset, eventually leads to host reset and 0xF000 fault.

The diag fault reset (and hence 0xF000 fault) with *"host reset"* reason shouldn't be treated as an error blindly as the driver cannot identify whether the host reset is internally invoked by the error handler or through `sg_reset` and whether the error handler escalation is expected or not. The driver and kernel log needs to be analyzed in detail to identify the exact reason for diag fault reset when the reason code indicates the diag fault reset is due to *"host reset"*.

Please contact Broadcom support, if the 0xF000 fault is seen without a user invocation to analyze the reason for the diagnostic fault reset.

Dynamic Debug Logging Level configuration

The driver has only mandatorily required information logging by default to avoid cluttering the kernel log. The additional debug logging prints can be dynamically enabled by providing the logging level through the module parameter or dynamically changing the logging level value through `sysfs` on per controller basis.

For capturing the controller initialization logs the logging level has to be set through module parameter *“logging_level”* during the driver load either as an argument to *“modprobe”* or through the kernel commandline entry *“mpi3mr.logging_level=<value>”*. The logging level set through module parameter will be applicable to all the controllers managed by the driver.

For post initialization log capture, the logging level can be set on a per controller basis through sysfs using a command similar to the one below

```
#echo 0xFFFF > /sys/class/scsi_host/hostX/logging_level
```

To turn off the additional logging, the logging level has to be set to 0.

The logging level values can be found in the *“mpi3mr_debug.h”* file in the source tarball.

If any issues are observed and to be reported to Broadcom, It is recommended to set the logging level to 0xFFFF, reproduce the failure again, collect the logs and share with Broadcom along with the original first time failure logs.

Back to Back VD creation/deletion or drive attach/detach

If the user connects or removes drives, creates or delete RAID Virtual Drives, the user needs to wait for the drive/RAID VD to be completely exposed at the OS level or deleted from the OS level. Back-to-back VD deletions or creations, drive additions/removals while the driver is in the process of adding or deleting the VD/drives at the OS level, will result in unknown behavior. Hence if a user connects drives or creates VD then the user has to wait until all the connected drives/created VD are seen in the OS properly and then only he has to progress to the next step. If all of the connected drives/created VD are not seen at both in the OS level and at the FW level after a specified time T (T is not static and increases with the number of drives connected/VD created) then the user has to pause at that step without progressing further. The same is true for drive removals/VD deletion too and the user has to wait for the drives/VD to be properly removed from both the OS and the firmware.

Topology changes during the controller reset

The controller reset leads to complete topology discovery at the firmware level and the firmware will report the discovered devices to the driver through MPI events after port enable is issued by the driver. The driver matches the newly discovered devices with the existing devices at the driver level and either adds the newly discovered devices or removes the devices that are not discovered. The driver waits for a settling time after port enable completion before

adding or removing devices. The device removal and addition is executed through worker thread and the driver prints below messages for indicating start and end of post reset device add/removal processing

"scan for non responding and newly added devices after soft reset started"

"scan for non responding and newly added devices after soft reset completed"

If an user invoked a controller reset or intended to issue controller resets in a loop it is recommended to wait until the device removal/additions are finished prior to continue with further steps (like issuing another reset)

Memory allocation failures in crashdump(kdump) kernel

If memory allocation requests made by driver fail in kdump mode, it's recommended to increase the memory reserved for kdump kernel accordingly. Please refer to the OS vendor documentation for steps to increase memory for kdump kernel. The recommendation would reduce occurrences of memory allocation failures but cannot guarantee to avoid them completely.

Controller Time changes during timestamp update by the driver

In some systems, a time difference is observed in the controller timestamp recorded in firmware logs / Snapdumps between system boot and approximately 15 minutes later. This drift may affect the execution of time-sensitive background operations scheduled in the controller.

- The driver updates the controller time during initial controller initialization and periodically thereafter for every N minutes (default: every 15 minutes; interval configurable via controller NVDATA).
- It retrieves the current GMT time using the kernel function `ktime_get_real()`.
- When the driver loads from the initial RAM disk (`initrd`) — as is the case for boot controllers — it uses the Real-Time Clock (RTC) value available at that early stage.
- Subsequently, the system may synchronize its clock via external services (e.g., `ntpd` or `systemd-timesyncd`), which adjusts the RTC.
- At the next scheduled controller time update (after N minutes), the driver fetches the updated time using `ktime_get_real()`, now reflecting the synchronized system time.
- This sudden time shift on the controller can interfere with previously scheduled background operations, causing them to miss their intended execution windows.

Ensure the RTC is configured with an accurate time during system boot and avoid large system time shifts after the initial driver load to avoid any such issues.

Special Notes for this Build

- None