Build and Run RSDK Offline Example Applications on S32R45 EVB

This guide details the steps needed to run the RSDK offline example application on the S32R45 EVB with the NXP Automotive Linux BSP. By following this guide the reader will be able to build each necessary software component from source, deploy the Linux BSP to the S32R45 EVB, load the RSDK kernel modules and execute the RSDK SPT example application.

Requirements

Software

- Linux build environment
- NXP Automotive Linux BSP 30.0
- S32 Design Studio 3.4
- Radar SDK for S32R45 0.9.3

Hardware

- S32R45 Evaluation Board (processor board only).
- MicroSD card with minimum 1 GB capacity.

Create a Linux Build Environment

A Linux build environment is required in order to build U-boot, the Linux kernel and the RSDK kernel modules. This can be done using a Linux PC or a Virtual Machine running Ubuntu or Debian. From hereafter, the Linux build environment will be referred to as 'Linux box'.

Install Ubuntu Natively on PC

- Generate a Bootable USB stick to install Ubuntu
- Install Ubuntu Desktop tutorial

Install VMWare Virtual Machine on Windows PC

- Install VM Ware virtual machine

Install the S32DS 3.4 Studio with Radar Support

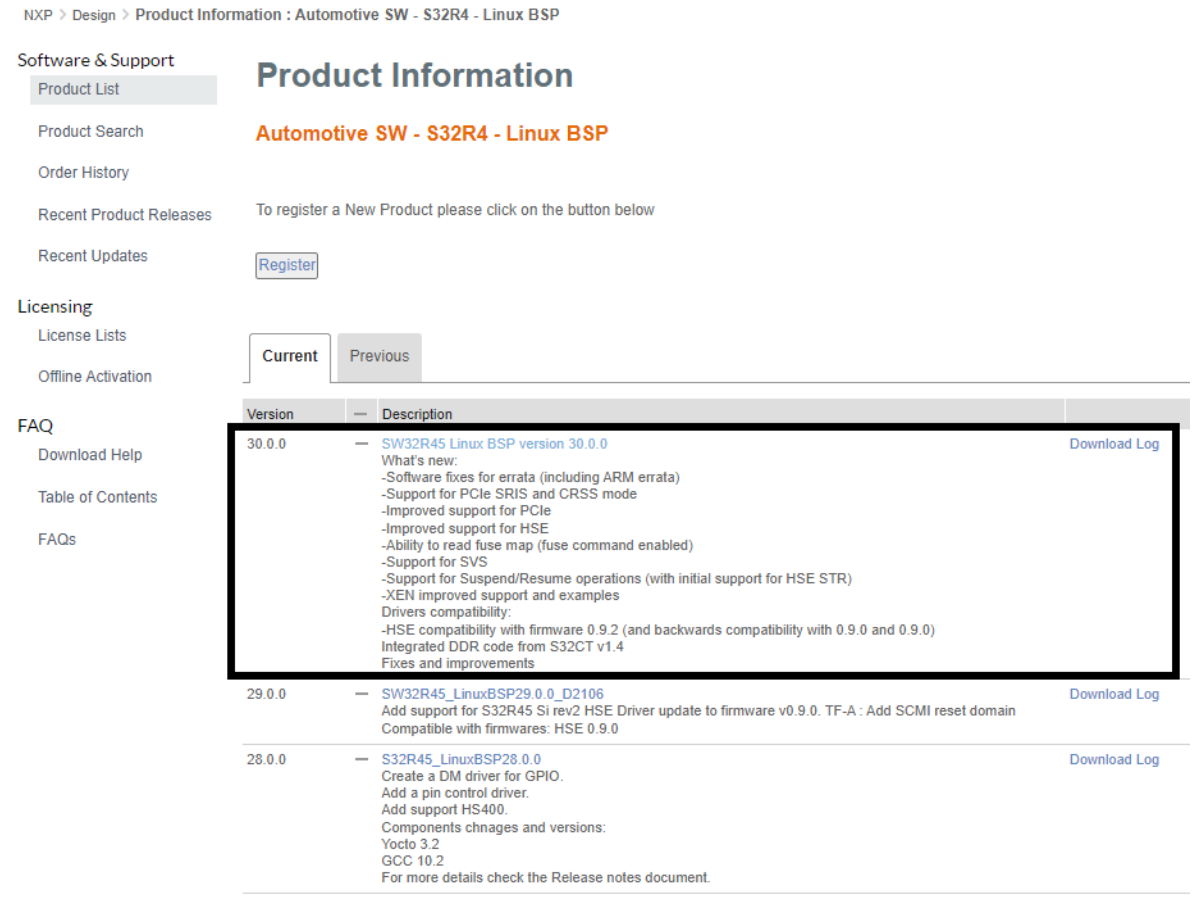
The S32DS studio provide among the IDE several build-tools for the ARM cores and the HW radar accelerators on the S32R45 which required during the building process.

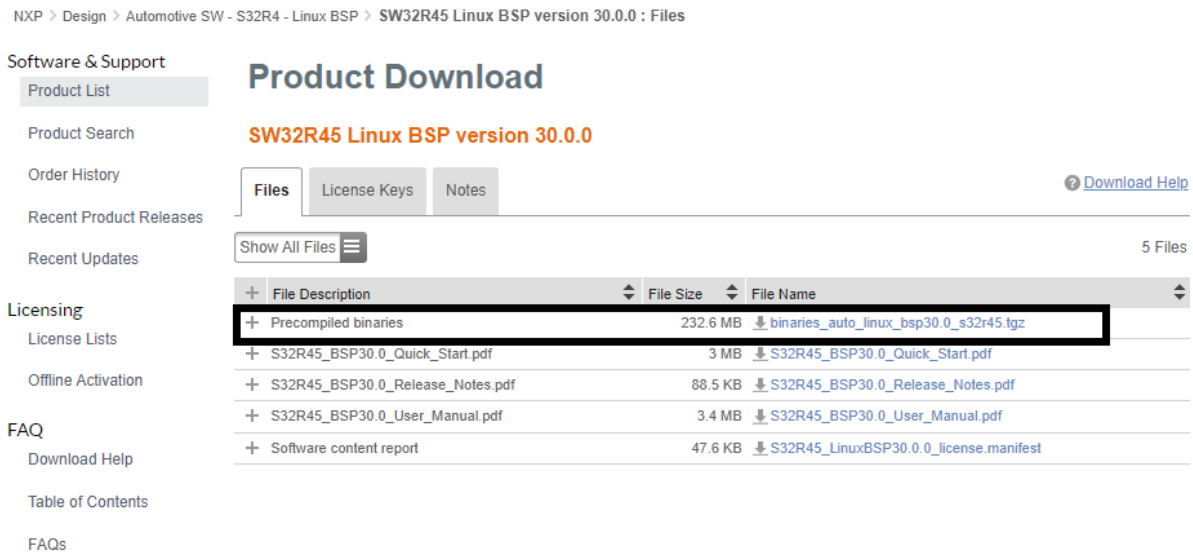
Follow the Install S32 Desing Studio 3.4 with Radar support guide

---

Get the NXP Automotive Linux BSP

Download the pre-compiled BSP 30.0 image tarball from NXP Flexera software distribution. Strictly speaking, only the root file system is needed from the Linux BSP pre-compiled release package as we will be building the U-boot and Linux kernel from source by cloning the git repositories. We will write the full BSP image to SD card and then update the images as required.





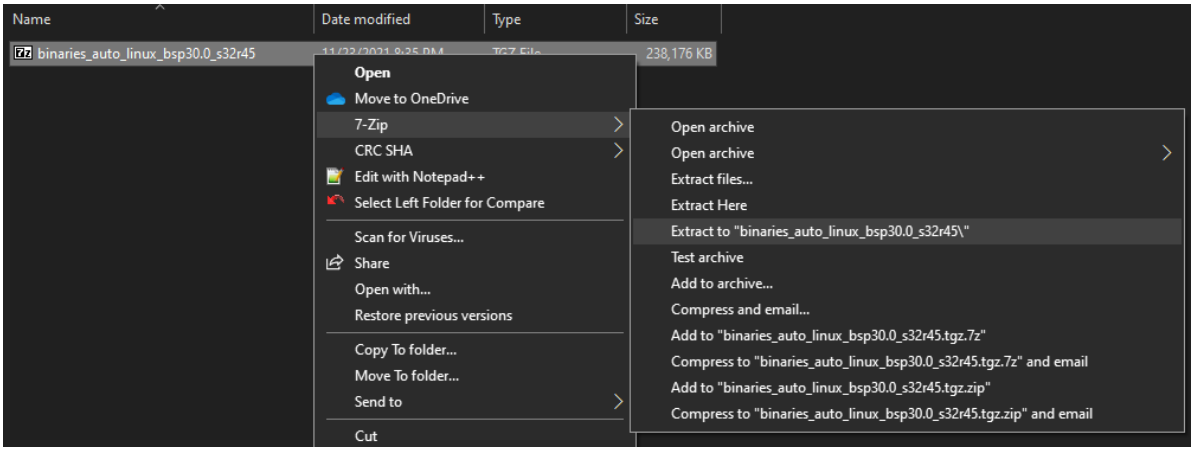
Once the tarball (.tgz file) is downloaded, extract the contents as follows.

On Linux, extract as below:

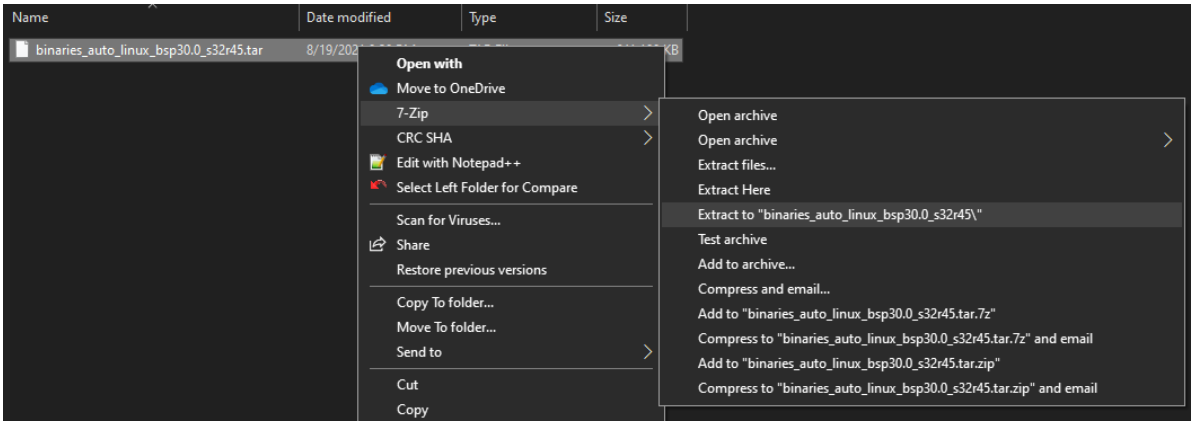
For bsp 30:

```
tar -xvf binaries_auto_linux_bsp30.0_s32r45.tgz
```

On Windows, use [7-zip] to extract the contents of the downloaded tarball file (.tar.gz):



Right click on the *binaries_auto_linux_bsp30.0_s32r45.tgz* file and select where to extract the content. **Two extractions are needed: first to uncompressed the file (remove .gz extension and get a .tar file) and the second to de-archive the .tar file get the binary image files.**



Following extraction the files are on disk:

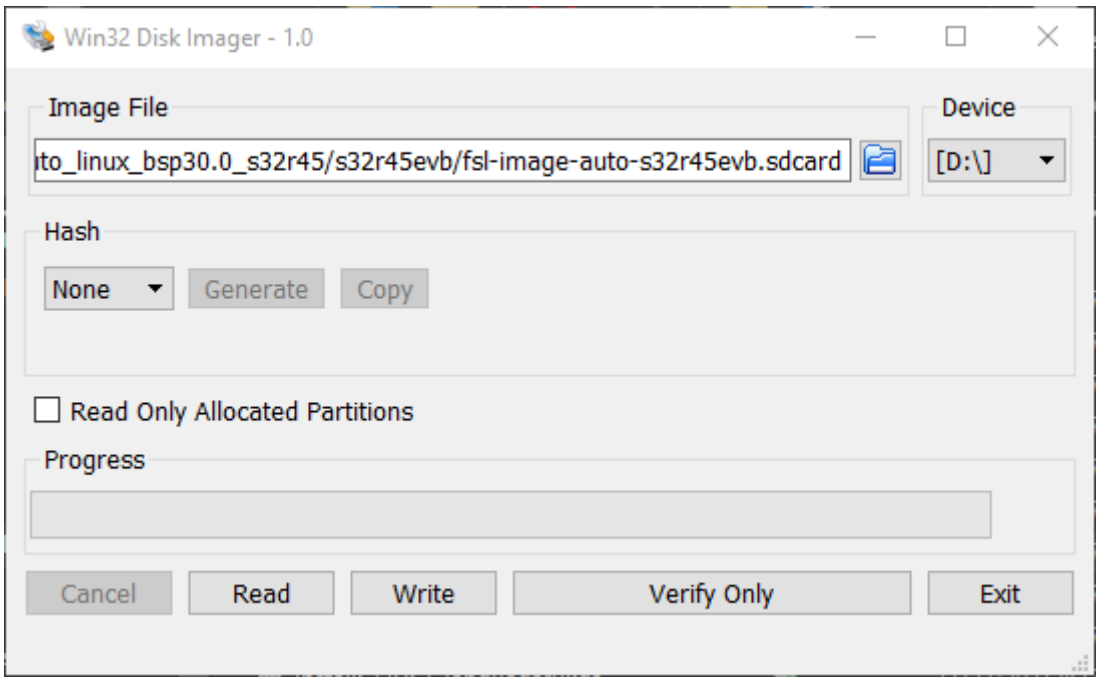
- **Image:** the Linux Kernel image binary.
- **u-boot-s32r45evb.s32:** the u-boot binary image.
- **fsl-image-auto-s32r45evb.sdcard:** The complete BSP image ready to be written to SD card.
- **fsl-image-auto-s32r45evb.tar.gz:** The filesystem (in compressed format).
- **fsl-s32r45-evb.dtb:** The device tree blob binary.

NOTE : Instead of using the pre-built binaries you can follow the [Generate S32R45 BSP Image using Yocto] guide to build the BSP image on your own.

Write the BSP Image to an SD Card

Follow the below steps to create a bootable SD Card for the S32R45 EVB by writing the **fsl-image-auto-s32r45evb.sdcard** image file.

Use [Win32DiskImager]. Select the .sdcard image file, select the the target device (SD card) and press the Write button.



Boot the BSP On S32R45 EVB

EVB Hardware Configuration

Remove the SD card from PC and insert into the S32R45 EVB SD slot (J34). Configure the EVB to boot from SD using the following jumper and switch settings:

BootMODE = Boot from RCON

- J136 = position 4-6
- J137 = position 1-3

Boot Config = SD boot

- BOOT_CFG1[7:5] = 0b010

The BOOT_CFG setting is applied using the dip switches:

DIP switch SW6	DIP switch SW7	DIP switch SW8	DIP switch SW9
1: OFF 2: OFF 3: OFF 4: OFF 5: OFF 6: OFF **7: ON** 8: OFF	1: OFF 2: OFF 3: OFF 4: OFF 5: OFF 6: OFF 7: OFF 8: OFF	1: OFF 2: OFF 3: OFF 4: OFF 5: OFF 6: OFF 7: OFF 8: OFF	1: OFF 2: OFF 3: OFF 4: OFF 5: OFF 6: OFF 7: OFF 8: OFF

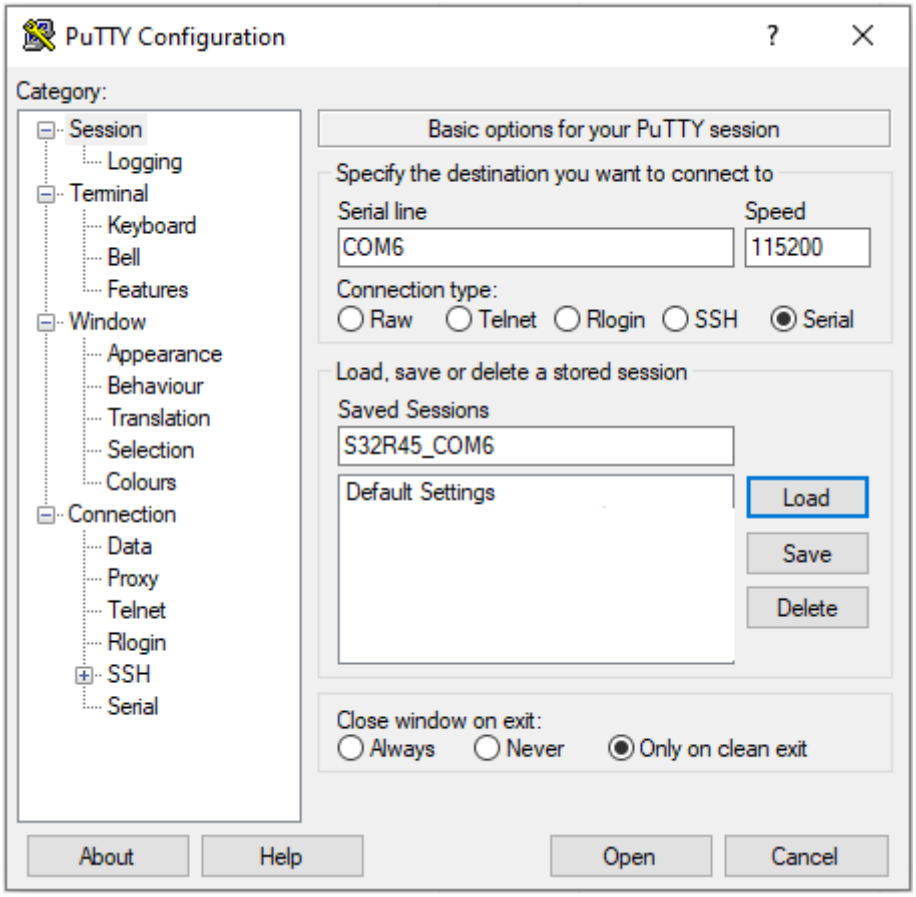
Select uSDHC signal routing

- J50 = position 1-2

Attach Serial Terminal

Connect a USB cable between the micro USB connector on the S32R45 EVB and PC. A terminal emulator program is needed to interact with S32R45 over USB-UART serial connection. Install PuTTy and configure a serial connection with the settings shown.

NOTE: You need to identify the correct COM port number for your S32R45 EVB. On Windows, open Device Manager and expand Ports (COM & LPT) to identify the number.



Boot Linux BSP

Power-on the EVB and observe the terminal window for output. The first thing that starts is U-Boot:

```
U-Boot 2020.04+gacc70867f0 (Sep 11 2020 - 02:17:48 +0000)

CPU:    NXP S32R45
Reset cause: Power-On Reset
Model: NXP S32R45
Board:  NXP S32R45-EVB
DRAM:  4 GiB
CA53 core 1 running.
CA53 core 2 running.
CA53 core 3 running.
All (4) cores are up.
MMC:   FSL_SDHC: 0
Loading Environment from MMC... OK
Using external clock for PCIe0
Configuring PCIe0 as RootComplex(x2)
PCIe0: Failed to get link up
Pcie0: LINK_DBG_1: 0x00000000, LINK_DBG_2: 0x00000800 (expected 0x000000d1)
DEBUG_R0: 0x00a2fd00, DEBUG_R1: 0x08200000
PCI: Failed autoconfig bar 1c
In:    serial
Out:   serial
Err:   serial
Net:   EQOS phy: rgmii @ 1

Warning: eth_eqos (eth0) using random MAC address - ee:c1:be:9a:c0:87
eth0: eth_eqos
Hit any key to stop autoboot:  0
```

Following this, U-Boot will load and start the Linux kernel. Many status messages are shown, wait until the kernel has started and presented the login prompt then enter the username 'root':

```
Auto Linux BSP 30.0 s32r45evb ttyLF0

s32r45evb login: root
root@s32r45evb:~#
```

You are now booted and logged in as root user.

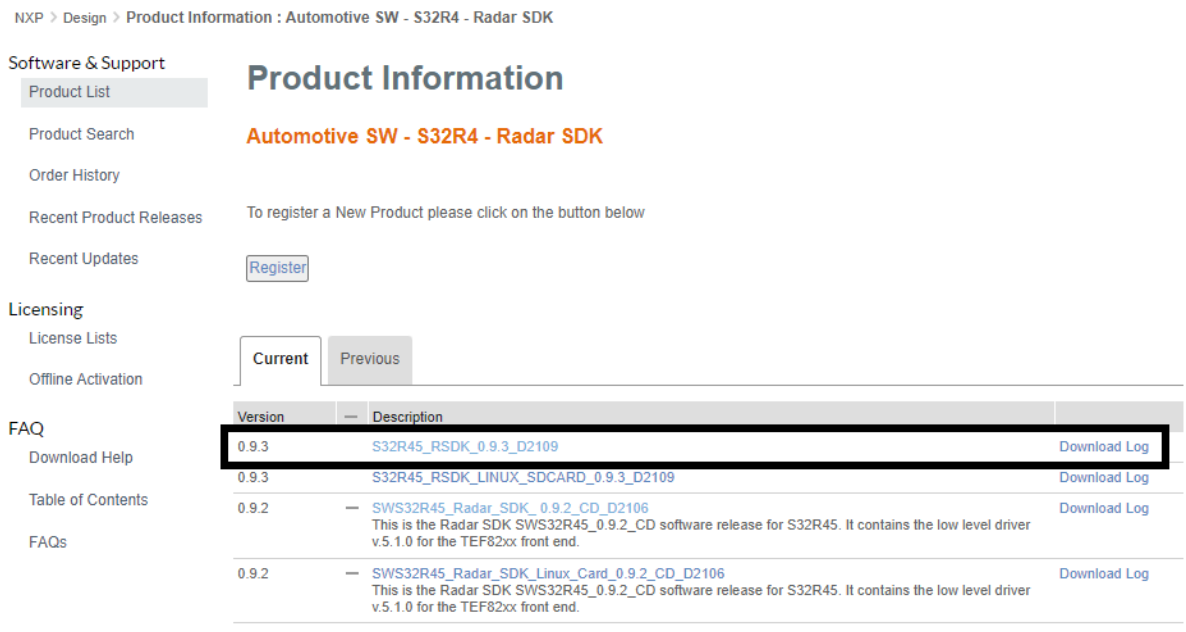
NOTE: If the Linux kernel does not automatically boot it may be necessary to use the command 'run mmcboot' in U-boot to do it manually:

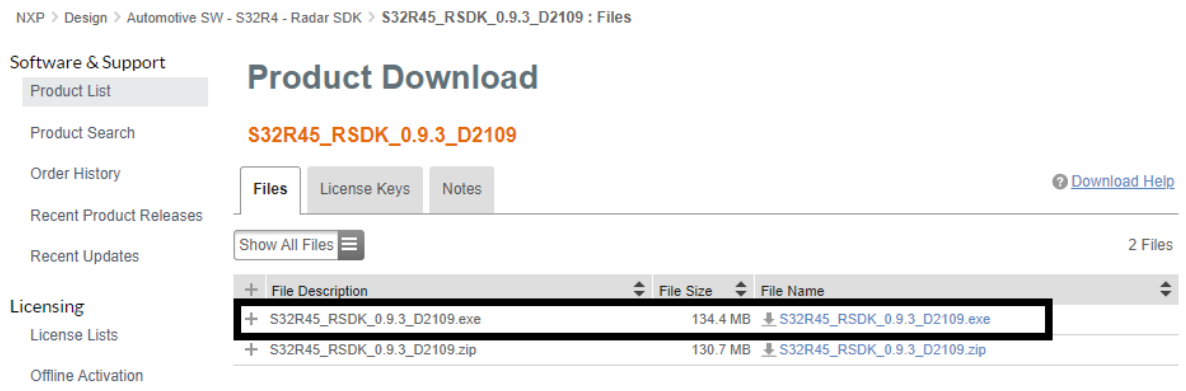
```
=> run mmcboot
Booting from mmc ...
7505928 bytes read in 260 ms (27.5 MiB/s)
26336 bytes read in 15 ms (1.7 MiB/s)
## Flattened Device Tree blob at 83e00000
   Booting using the fdt blob at 0x83e00000
   reserving fdt memory region: addr=8000fff8 size=10000
   Loading Device Tree to 000000009fd3d000, end 000000009fd466df ... OK

Starting kernel ...
```

Install the Radar SDK

Get the Radar SDK

On Windows, download [S32R45_RSDK_0.9.3_D2109.exe] from the nxp.com site.





This package contains pre-built binaries and the source code for the following components:

- Operating System Abstraction Layer (OAL).
- SPT Driver for both kernel and user-space modules.
- SPT kernels
- MIPI-CSI driver
- LAX module
- DSP Dispatcher and DSP radar baseband algorithms
- RSDK example application.

Double click on the executable and install the RSDK content, typically on C:\NXP\S32R45_RSDK__0.9.3

NOTE: if you have previously installed the RSDK, the installer will complain and not allow you to proceed, as a workaround, just rename the S32R45_RSDK__0_9_3 previously installed and rename it back after the installation finishes.

The reader can choose to use the pre-built binaries but can also build each component from source if desired.

Patch the RSDK

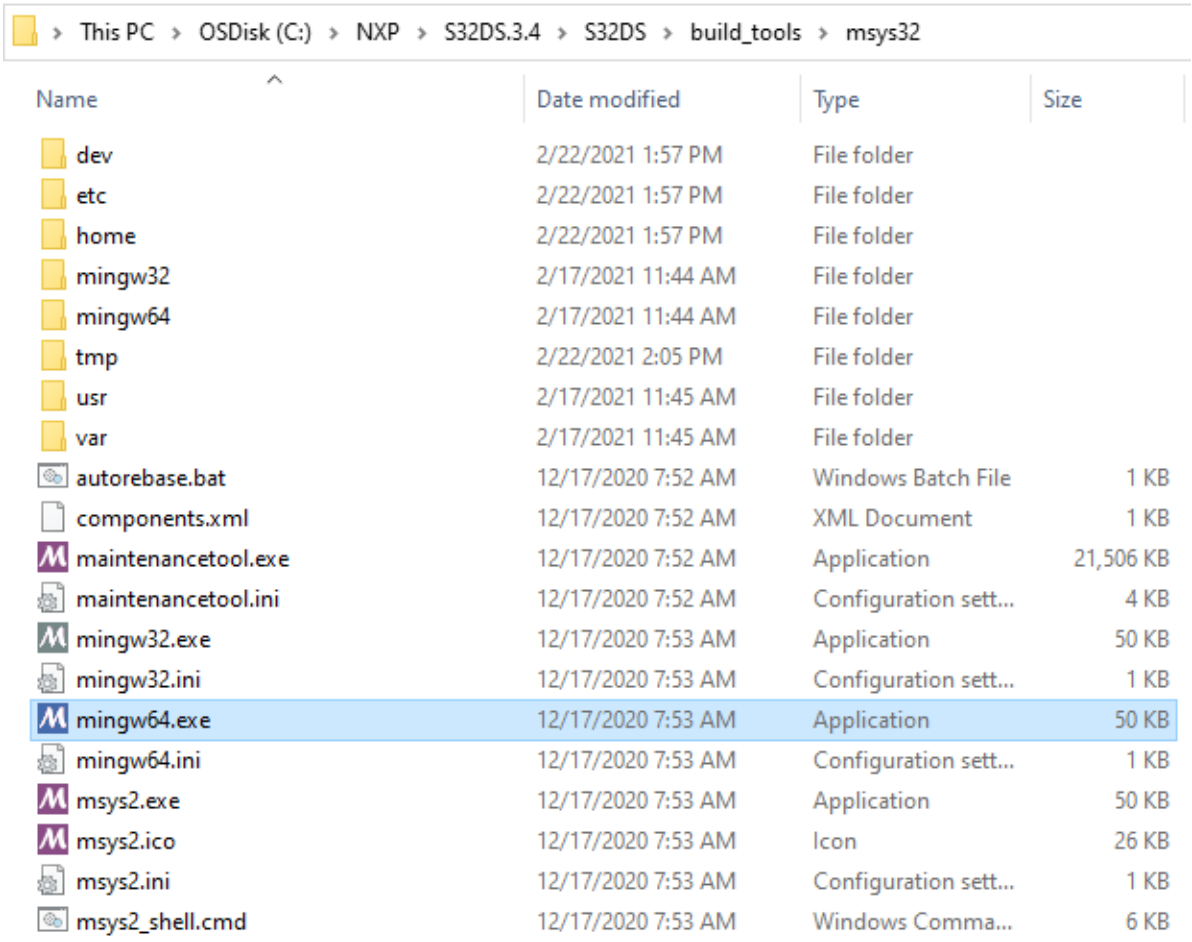
Before building the modules, there is this definition on the oal_mem_constants.h file that needs to de changed to proper operation of OAL and LAX drivers. The required changes are show on the following diff format:

```
diff --git a/oal/libs/kernel/common/include/oal_mem_constants.h
b/oal/libs/kernel/common/include/oal_mem_constants.h
index 1d72e85..97f9000 100755
--- a/oal/libs/kernel/common/include/oal_mem_constants.h
+++ b/oal/libs/kernel/common/include/oal_mem_constants.h
@@ -40,7 +40,7 @@

 /* Size of memory reservations in bytes */
 #ifndef OAL_MAX_REGION_SIZE
-#define OAL_MAX_REGION_SIZE 0x7000000U
+#define OAL_MAX_REGION_SIZE 0x1f400000U
 #endif

 /* Allowed allocations from a Virtual Address Space */
@@ -75,7 +75,7 @@
```

```
 /* Number of events per RPC connection */
 #ifndef OAL_MAX_EVENTS_PER_SERVICE
-#define OAL_MAX_EVENTS_PER_SERVICE 32U
+#define OAL_MAX_EVENTS_PER_SERVICE 64U
 #endif

 /* QNX: Number of allocated QNX services per process, user space */
@@ -100,7 +100,7 @@

 /* Linux & QNX: Memory allocated for communication serialization (bytes) */
 #ifndef OAL_MAX_PROCESS_COMM_SHARED_BUFFER
-#define OAL_MAX_PROCESS_COMM_SHARED_BUFFER (4U * 1024U)
+#define OAL_MAX_PROCESS_COMM_SHARED_BUFFER (15000U)
 #endif

 #endif /* OAL_MEM_CONSTANTS_H
```

Build the User-Space Components

With MSYS32 (included on Design Studio installation) all the user-space component and SPT Kernels can be built from command line. To open a MSYS32 terminal, go to the msys32 folder on your S32DS installation on Windows Explorer and double click on the msys2.exe or mingw64.exe



On the msys32 terminal, export the following environment variables

```
export ARCH="aarch64"
```

```
export COMPILER="gcc"
export OS="linux"
export OSENV="linux"
export TARGET="a53"
export PLATFORM="S32R45"

export DS_BINPATH="/c/NXP/S32DS.3.4/S32DS/build_tools"
export DS_BUILDTOOLS_PATH="/c/NXP/S32DS.3.4/S32DS/build_tools"

export CPU_TOOLCHAIN_BINPATH="/c/NXP/S32DS.3.4/S32DS/build_tools/gcc_v10.2/gcc-10.2-arm64-linux/bin/aarch64-linux-gnu-"
export CROSS_COMPILE="/c/NXP/S32DS.3.4/S32DS/build_tools/gcc_v10.2/gcc-10.2-arm64-linux/bin/aarch64-linux-gnu-"

export LAX_TOOLCHAIN_PATH="/c/NXP/S32DS.3.4/S32DS/build_tools/LAX/"
export SPT_TOOLCHAIN_BINPATH="/c/NXP/S32DS.3.4/S32DS/build_tools/SPT3/bin/"

export XTENSAD_LICENSE_FILE="<PUT HERE THE PATH TO YOUR LICENSE FILE>"
export XTENSA_CORE="spt_v3_130717"
export XTENSA_SYSTEM="/c/NXP/S32DS.3.4/S32DS/build_tools/BBE32/builds/RI-2018.0-win32/spt_v3_130717/config"

export SYSROOT=""
export KERNEL_DIR=""
export TRACE_ENABLE="1"

export EXE_PATH=""
export TOOL_PATH=""
```

Run the make all command as show below, to start the building

```
cd /c/NXP/S32R45_RSDK__0.9.3/
make all
```

The full build can take some time (around 20 - 30 minutes) mainly because now the BBE32DSP code is also compiled from command line and the cross-compilation of the DSP is most time consuming.

Make RSDK installation accessible to Linux box

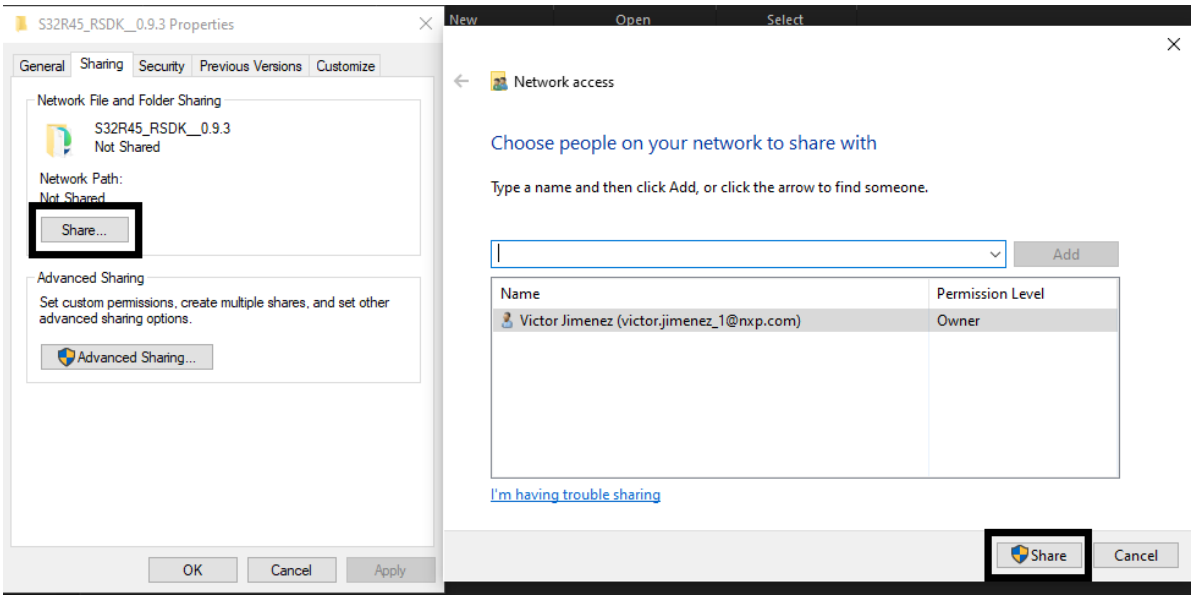
NOTE: The steps of this section only apply if you are running Linux on a VM. If Linux is running natively, just transfer the compiled RSDK as you find it convenient.

Since the kernel modules on the RSDK, need to be build from a Linux box (or VM with Linux), as well as the Linux kernel and u-boot we need to make the RSDK installation (usually on Windows side) accessible from the Linux box.

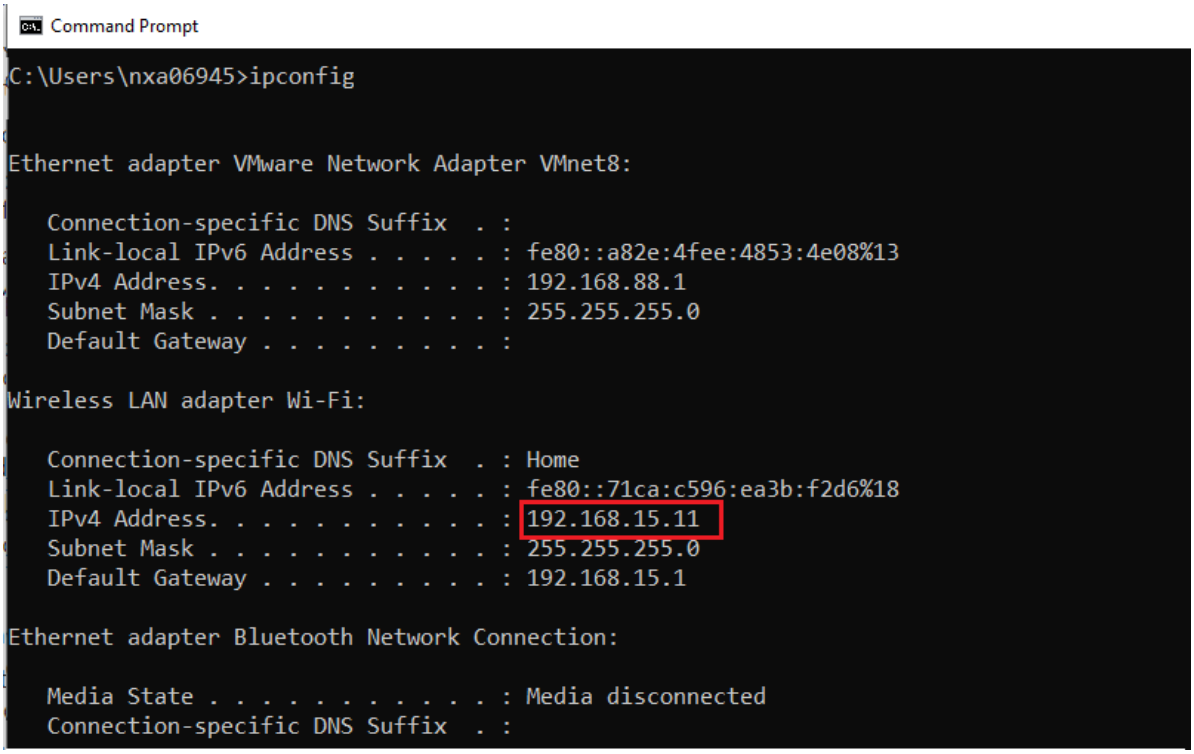
Follow the next steps to share from Windows and mount on Linux the RSDK installation.

Share the RSDK directory on Windows:

- Use Windows explorer and locate your RSDK installation, usually at C:\NXP\S32R45_RSDK__0.9.3
- Right-click the S32R45_RSDK__0.9.3 folder, and select the **Properties** option
- Click on the **Sharing** tab
- Click the **Share** button
    - Optionally use the drop-down menu to select other the user or group to share a file or folder besides yourself
    - Click the **Add** button.
- Click on the Share button and grant access



Identify your IP address. You can use the Windows comand prompt and use the ipconfig command to look for your IP address



Mount the RSDK directory on Linux box

NOTE: The steps of this section only apply if you are running Linux on a VM. If Linux is running natively, just transfer the compiled RSDK as you find it convenient.

To be able to mount a shared directory from Windows on Linux (Ubuntu),first you need to install the CIFS utilities package.

Install the CIFS utilities by

```
sudo apt install cifs-utils
```

Create a new directory, where the remote RSDK will be mounted

```
mkdir ~/S32R4/RSDK/S32R45_RSDK__0.9.3
```

Mount the remote RSDK directory as show below.

```
sudo mount.cifs //192.168.15.11/S32R45_RSDK__0.9.3
~/S32R4/RSDK/S32R45_RSDK__0.9.3/ -o
user=victor.jimenez_1@nxp.com,gid=1000,uid=1000
```

Please, use your own Windows user or email on the user parameter and the IP address of your Windows PC, instead of the ones show on the above command.

Install and Build U-Boot and Linux for S32R45 (on Linux box)

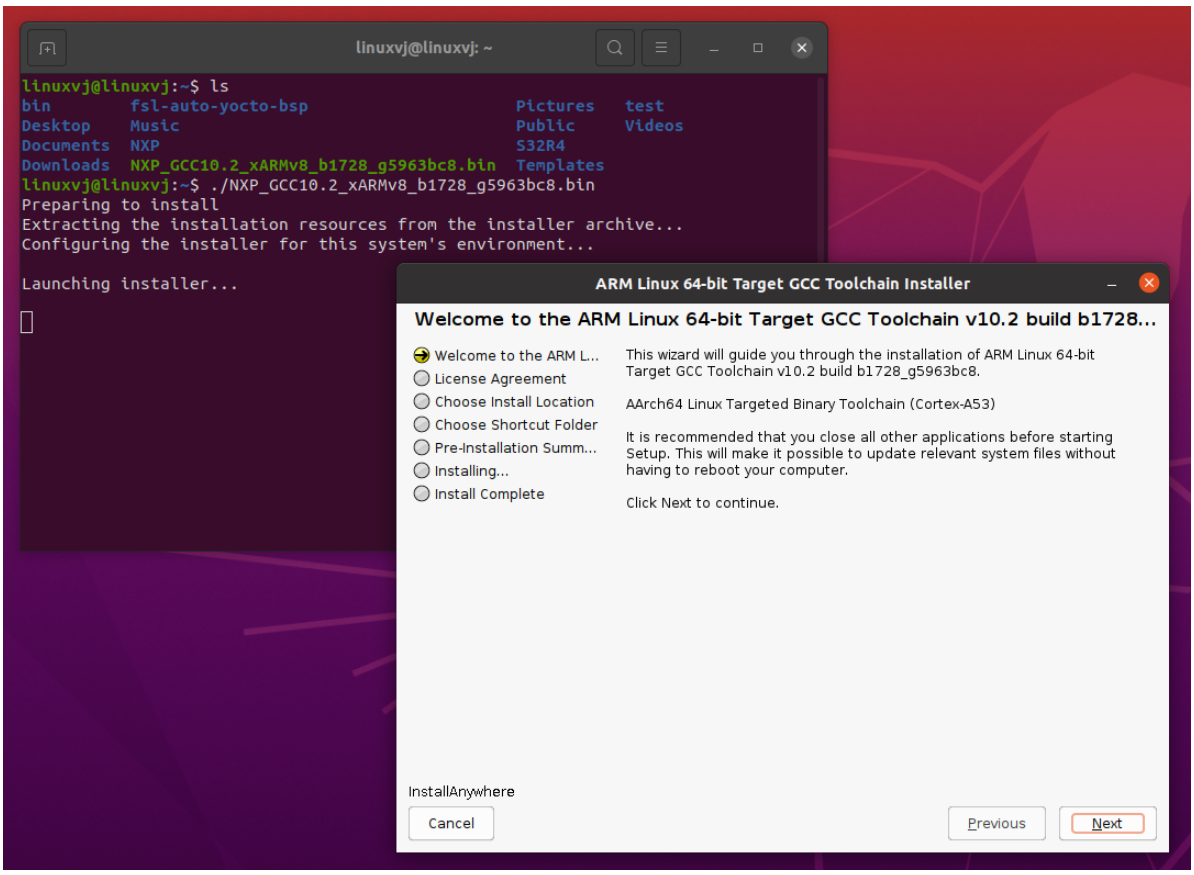
Install the GCC toolchain for ARM64

To be able to compile the U-Boot, Linux and RSDK kernel modules you need to install the NXP GCC ARM64 toolchain on your Linux box. From your Linux box, go to the NXP Linux GCC 10.2 Compiler Tools for ARM v8 64-bit, build 1728 - Linux and download the NXP_GCC10.2_xARMv8_b1728_g5963bc8.bin.

Change the permissions of the the .bin file so that it can be executed.

```
chmod a+x NXP_GCC10.2_xARMv8_b1728_g5963bc8.bin
```

Execute the .bin file in your Linux machine. A prompt will appear, follow all the steps to go through the installation process.



NOTE: for this step you need to have JDK version 8.0 installed on your Linux machine. If you don't have it, install it with the following command:

```
sudo apt-get install openjdk-8-jre
```

Since the toolchain binaries installed like aarch64-linux-gnu-gcc are 32-bit elf executable files and Ubuntu is running on a x64 host, the 32bit multi-lib support needs to be installed as follows:

```
sudo apt-get install gcc-multilib
```

Also in order to be able to successfully compile the kernel and u-boot the next packages needs to be installed (if not already)

```
sudo apt install make
sudo apt install bison
sudo apt install flex
```

Get the U-Boot Source

In order to generate a Linux kernel bootable image we need the mkImage tool from U-Boot. Clone the U-Boot git repository:

```
mkdir ~/S32R4/
cd ~/S32R4/
git clone https://source.codeaurora.org/external/autobsps32/u-boot
```

```
cd u-boot
git checkout bsp30.0-2020.04 -b bsp30.0
```

Build the U-Boot

To build u-boot, run make to set the defconfig first and then to build, as show next:

```
cd ~/S32R4/u-boot
make CROSS_COMPILE=~/NXP/gcc-10.2-arm64-linux/bin/aarch64-linux-gnu-
s32r45evb_defconfig
make -j8 CROSS_COMPILE=~/NXP/gcc-10.2-arm64-linux/bin/aarch64-linux-gnu-
```

Get the Linux Kernel Source

Clone the automotive Linux kernel git repository and check-out the v5.4.24_bsp26.0 branch.

```
cd ~/S32R4/
git clone https://source.codeaurora.org/external/autobsps32/linux
```

```
cd linux
git checkout -b bsp26.0 v5.4.24_bsp26.0
git checkout bsp30.0-5.10.41-rt -b bsp30.0
```

Patch Linux kernel with the patches and device tree from RSDK

The BSP does not natively support the SPT and OAL devices. We need to modify the S32R45 EVB device tree file to add support for these devices, then we can insert the kernel modules to provide drivers. Go to the following path in your Linux computer:

```
cd S32R4/RSDK/S32R45_RSDK__0.9.3/platform_setup/config/S32R45/linux/
```

Copy the files from the RSDK to the Linux kernel.

```
~/S32R4/S32R45_RSDK__0.9.3/platform_setup/config/S32R45/linux$ cp fsl-s32r45*
/home/linuxvj/S32R4/linux/arch/arm64/boot/dts/freescale/
```

There is an extra patch required for the proper operation of the RSDK that needs to be applied on top of bsp30.0.

Go to the *oal* folder of the RSDK and copy the *file oal_patch_linux_5.10.diff* into this folder:

```
linuxvj@linuxvj:~/S32R4/RSDK/S32R45_RSDK__0.9.3/oal$ cp
../platform_setup/config/S32R45/linux/oal_patch_linux_5.10.diff .
```

After copying the file apply the patch with the following command:

```
linuxvj@linuxvj:~/S32R4/RSDK/S32R45_RSDK__0.9.3/oal$ git apply
oal_patch_linux_5.10.diff
```

Build the Linux Kernel

After the Linux kernel souce has been successfully patched, now we can build it

```
cd ~/S32R4/linux/
make ARCH=arm64 CROSS_COMPILE=~/NXP/gcc-10.2-arm64-linux/bin/aarch64-linux-gnu-
s32gen1_defconfig
make -j8 ARCH=arm64 CROSS_COMPILE=~/NXP/gcc-10.2-arm64-linux/bin/aarch64-linux-
gnu-
```

Build the RSDK kernel modules

The kernel objects from the RSDK like oal, spt, mipi-csi and lax drivers needs also be built from a Linux box from the remote directory mounted. Below are the detailed instructions to build each of the kernel modules present on the RSDK.

Build the OAL Memory Driver

Pre-built binary location:
S32R45_RSDK__0.9.3/oal/libs/kernel/driver/build-linux-kernel/oal_driver.ko

```
cd ~/S32R4/RSDK/S32R45_RSDK__0.9.3
cd oal/libs/kernel/driver/build-linux-kernel

export CROSS_COMPILE=~/NXP/gcc-10.2-arm64-linux/bin/aarch64-linux-gnu-
export KERNEL_DIR=~/S32R4/linux

make clean PLATFORM=S32R45 TARGET=a53 COMPILER=gcc OSENV=linux
    make PLATFORM=S32R45 TARGET=a53 COMPILER=gcc OSENV=linux
```

Check the kernel object (.ko) file to see the version magic number:

```
modinfo oal_driver.ko | grep vermagic
```

Build the SPT Linux Kernel Driver

Pre-built binary location:
S32R45_RSDK__0.9.3/SPT/SPT_driver/build/linux_kernel/spt_driver.ko

```
cd ~/S32R4/RSDK/S32R45_RSDK__0.9.3
cd SPT/SPT_driver/build/linux_kernel/

export CROSS_COMPILE=~/NXP/gcc-10.2-arm64-linux/bin/aarch64-linux-gnu-
export KERNEL_DIR=~/S32R4/linux

make cleanall PLATFORM=S32R45 TARGET=a53 COMPILER=gcc OSENV=linux
make module PLATFORM=S32R45 TARGET=a53 COMPILER=gcc OSENV=linux
```

Check the .ko file to see the version magic number:

```
modinfo rsdk_spt_driver.ko | grep vermagic
```

Build the LAX linux-kernel driver

Pre-built binary location:
S32R45_RSDK__0.9.3/LAX/LAX_bin/rsdk_lax_driver.ko

```
cd ~/S32R4/RSDK/S32R45_RSDK__0.9.3
cd LAX/LAX_host/driver/lax/build-linux-kernel

export CROSS_COMPILE=~/NXP/gcc-10.2-arm64-linux/bin/aarch64-linux-gnu-
export KERNEL_DIR=~/S32R4/linux

make clean PLATFORM=S32R45 TARGET=a53 COMPILER=gcc OSENV=linux
make PLATFORM=S32R45 TARGET=a53 COMPILER=gcc OSENV=linux OS=linux OSENV=linux
LAX_PLATFORM=RRM
```

Check the .ko file to see the version magic number:

```
modinfo rsdk_lax_driver.ko | grep vermagic
```

Build the MIPI CSI2 linux-kernel driver

Pre-built binary location:
S32R45_RSDK0.9.3/CSI2/CSI2_driver/bin/rsdk_csi2_driver.ko

```
cd ~/S32R4/RSDK/S32R45_RSDK__0.9.3
cd CSI2/CSI2_driver

export CROSS_COMPILE=~/NXP/gcc-10.2-arm64-linux/bin/aarch64-linux-gnu-
export KERNEL_DIR=~/S32R4/linux

make cleanall PLATFORM=S32R45 TARGET=a53 COMPILER=gcc OSENV=linux
make module PLATFORM=S32R45 TARGET=a53 COMPILER=gcc OSENV=linux
```

Check the .ko file to see the version magic number:

```
modinfo bin/rsdk_csi2_driver.ko | grep vermagic
```

Build the CTE linux-kernel driver

Pre-built binary location:
S32R45_RSDK__0.9.3/CTE/CTE_driver/bin/rsdk_cte_driver.ko

```
cd ~/S32R4/RSDK/S32R45_RSDK__0.9.3
cd CTE/CTE_driver/

export CROSS_COMPILE=~/NXP/gcc-10.2-arm64-linux/bin/aarch64-linux-gnu-
export KERNEL_DIR=~/S32R4/linux

make cleanall PLATFORM=S32R45 TARGET=a53 COMPILER=gcc OSENV=linux
make module PLATFORM=S32R45 TARGET=a53 COMPILER=gcc OSENV=linux
```

Check the .ko file to see the version magic number:

```
modinfo bin/rsdk_cte_driver.ko | grep vermagic
```

Build the trace linux-kernel module

This module has no pre-built binary.

```
cd ~/S32R4/RSDK/S32R45_RSDK__0.9.3
cd Tools/Trace/build/linux_kernel

make clean_module PLATFORM=S32R45 TARGET=a53 COMPILER=gcc OSENV=linux
make module PLATFORM=S32R45 TARGET=a53 COMPILER=gcc OSENV=linux
```

Check the .ko file to see the version magic number:

```
modinfo trace_module.ko | grep vermagic
```

Version Magic Number Mismatch

The version magic number (vermagic) of the kernel modules must match the version of the Linux kernel on which they will be executed. If there is a mismatch on the version the module will fail to be inserted. Since we built the Linux kernel from source and referenced this during the build of the RSDK kernel modules, the magic numbers should match already.

However if a **pre-built Linux kernel** is used instead (such as that delivered with the BSP release), then there is a method to overcome this problem by forcing the RSDK kernel module vermagic to the value expected by the Linux-kernel where the module will be installed. First check the version of the currently running kernel on S32R45:

```
root@s32r45xevb:~# cat /proc/version
Linux version 4.19.59-rt24+gae9ef3c (oe-user@oe-host) (gcc version 6.3.1
20170509 (Linaro GCC 6.3-2017.06~dev)) #1 SMP PREEMPT Tue Feb 4 10:51:22 UTC
2020
```

In this case the Linux version is reported as: **4.19.59-rt24+gae9ef3c**

We can specify the version to be used for compiled kernel modules (and compiled Linux kernel) by modifying the *utsrelease.h* file to match the version of the kernel from the BSP pre-built binaries:

```
cd ~/S32R4/linux
vim include/generated/utsrelease.h
```

Modify to match the kernel version reported by S32R45 BSP:

```
#define UTS_RELEASE "4.19.59-rt24+gae9ef3c"
```

Now rebuild the SPT, OAL, MIPI-CSI and LAX kernel modules again and check the version magic number. This time the version magic number should match that of the running Linux kernel.

NOTE: This hack is not necessary if you replace the Linux kernel image on the first (boot) partition of the SD Card with the re-compiled kernel output found at arch/arm64/boot/Image. We will do this later in the guide.

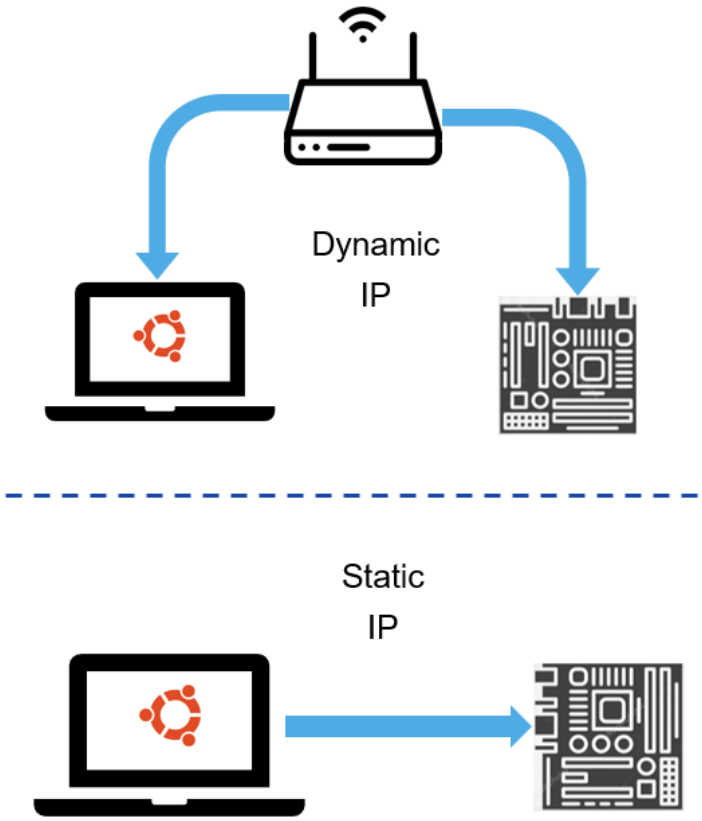
Deploy the binaries to the EVB filesystem

We can use **Secure Copy (scp)** to copy the files we compile and generate to the EVB over the network. First we need to enable the Ethernet and grant an ip address on the EVB. On the EVB, connect the Ethernet cable to J104 and create a shell script to set desired MAC and IP addresses to the S32R45 Ethernet interface:

```
root@s32r45evb:~# vi enableEth0.sh
```

```
#!/bin/sh

# Down the eth0 interface
ifconfig eth0 down

# Fix mac address
ifconfig eth0 hw ether 62:e7:b2:9c:e7:64

# Set static IP address  # Uncomment this on case of static ip
#ifconfig eth0 192.168.1.104

# Enable Eth0 interface
ifconfig eth0 up

# Get dynamic ip # Comment out this in case of static ip
udhcpc -i eth0
```

Depending if you are using a router (dynamic ip) or connecting directly to a PC (static ip) on your ethernet setup, you can comment out or uncomment the proper lines on the above script.



Give executable permissions to the script:

```
chmod +x ./enableEth0.sh
```

Connect the Ethernet cable and run the script:

```
./enableEth0.sh
```

Check for the ip assigned to eth0 on the EVB:

```
root@s32r45evb:~# ifconfig eth
eth0      Link encap:Ethernet  HWaddr 62:e7:b2:9c:e7:61
          inet addr:192.168.1.104  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::60e7:b2ff:fe9c:e761/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:6 errors:0 dropped:0 overruns:0 frame:0
          TX packets:14 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:926 (926.0 B)  TX bytes:1556 (1.5 KiB)
          Interrupt:71 Base address:0xc000
```

Now from your Linux box copy the files generated from RSDK builds:

```
cd ~/S32R4/RSDK/S32R45_RSDK__0.9.3

scp ./SPT/SPT_driver/build/linux_kernel/rsdk_spt_driver.ko
root@192.168.1.104:/home/root/
scp ./oal/libs/kernel/driver/build-linux-kernel/oal_driver.ko
root@192.168.1.104:/home/root/
scp ./LAX/LAX_bin/rsdk_lax_driver.ko root@192.168.1.104:/home/root/
scp ./CSI2/CSI2_driver/project/linux/rsdk_csi2_driver.ko
root@192.168.1.104:/home/root/

scp
./Apps/RSDK_offline_example/bin/rsdk_offline_example_linux_gcc_S32R45_a53_debug.
elf root@192.168.1.104:/home/root/
```

Copy the **data** directory from RSDK which is used by the SPT example app to read and write data and the **.eld** files used by LAX:

```
scp -r ./Apps/RSDK_offline_example/data root@192.168.1.104:/home/root/
scp ./Apps/RSDK_offline_example/bin/offline_example_lax_0_trace.eld
root@192.168.1.104:/home/root/
scp ./Apps/RSDK_offline_example/bin/offline_example_lax_1_trace.eld
root@192.168.1.104:/home/root/
```

Copy the device tree blob (.dtb):

```
cd ~/S32R4/linux
scp arch/arm64/boot/dts/freescale/fsl-s32r45-evb.dtb
root@192.168.1.104:/home/root/
```

Copy the Linux kernel image:

```
cd ~/S32R4/linux
scp arch/arm64/boot/Image root@192.168.1.104:/home/root/
```

All the required files are now copied into the EVB filesystem (2nd SD card partition), however some of them need to be placed on the boot partition of the SD Card.

Replace kernel and DTB on EVB

On the EVB, mount the SD card partition 1 so you can replace the .dtb file:

```
mkdir part1
mount /dev/mmcblk0p1 part1/
ls part1/
mv part1/fsl-s32r45-evb.dtb ~/fsl-s32r45-evb.dtb.Orig
cp fsl-s32r45-evb.dtb part1/
sync
```

Also replace the Linux kernel image:

```
mv part1/Image ~/Image.Orig
cp Image part1/
sync
```

Within the data directory an out/S32R45 directory is required to collect the output files when running the RSDK offline sample. Create this directory by following command (if not already there):

```
mkdir -p ~/data/out/S32R45/
```

Now power cycle the board and reach the Linux prompt again.

Run the RSDK Offline Example

To run the SPT example you first need to install the Linux kernel modules:

```
insmod oal_driver.ko
insmod rsdk_spt_driver.ko
insmod rsdk_lax_driver.ko
```

Now execute the SPT example application:

```
./rsdk_offline_example_linux_gcc_S32R45_a53_debug.elf
```

The output log should look something like RSDKOfflineExampleApp.log.
