Application Report

SPRACD1–December 2018

Early CAN Response on DRA7xx

Venkateswara Rao Mandela

ABSTRACT

Responding to a CAN message in 50 ms from Power ON is a typical automotive usecase. This application report shows the reception and the response time limits that can be met with existing software on DRA7xx.

Contents

1 Problem Statement 2
2 High Level Approach 2
3 Replicating the Results 5
4 Testing Early CAN on EVM 6
5 Patch Description 9
6 Further Optimizations 10
7 Conclusion 10
8 References 10

List of Figures

1 Linux Normal Boot Flow 2
2 Linux Early Boot Flow 3
3 Early CAN Breakdown 3

List of Tables

1 CAN Startup Stages 4
2 Basic Benchmarking and Optimization 9
3 QSPI Support for Early Boot 9
4 MLO Optimization for Early CAN 9
5 Display Boot Time Achieved 10
6 Display Boot Time Achieved 10

Trademarks

1 Problem Statement

Having the infotainment system respond to a CAN message in 50 ms from Power ON is a typical automotive requirement. Normally this requirement is met using a small microcontroller that responds to the CAN messages along side the main application processor. Meeting this requirement using only the TI DRA7xx SOC eliminates the external microcontroller and saves cost for the customer.

This document demonstrates how a CAN message can be received on DRA7xx within 45 ms from Board Power ON and the constraints on CAN software stack to be able to respond within 50 ms/100 ms.

Assuming that:

- CAN stack will be running on one of the M4(IPU) cores on Vayu.
- The customer system will be running Linux as the HLOS on A15.

The procedure described in this document is equally applicable to a customer system running Android.

1.1 Organization of the Document

The approach to the problem is described in Section 2. Section 3 shows how to replicate the results on a DRA7x EVM. In Section 5, the patches used to perform benchmarking and optimization are listed. In Section 6, further steps to reduce the boot time are listed.

2 High Level Approach

In the default SDK boot flow, M4 is loaded by Linux Kernel. As this takes around 2-3 seconds, the default boot flow is not suitable for the early CAN usecase.

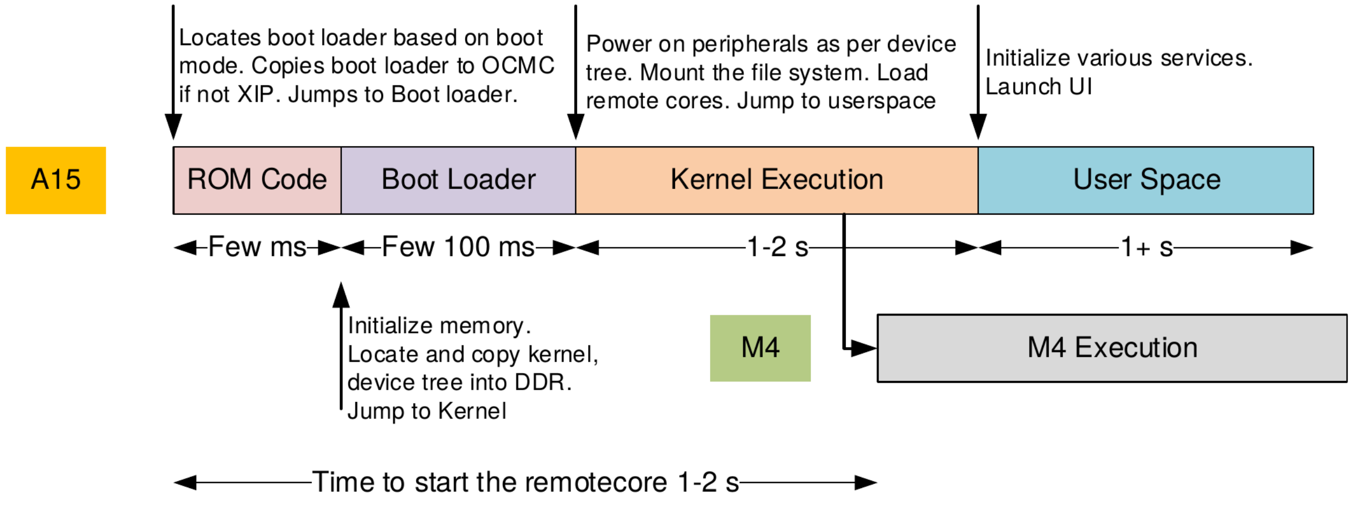
Locates boot loader based on boot mode. Copies boot loader to OCMC if not XIP. Jumps to Boot loader.
Power on peripherals as per device tree. Mount the file system. Load remote cores. Jump to userspace
Initialize various services. Launch UI
A15
ROM Code
Boot Loader
Kernel Execution
User Space
Few ms
Few 100 ms
1-2 s
1+ s
Initialize memory. Locate and copy kernel, device tree into DDR. Jump to Kernel
M4
M4 Execution
Time to start the remotecore 1-2 s

Figure 1. Linux Normal Boot Flow

Instead, the early boot flow is used, where M4 is loaded from the bootloader instead of the kernel. In this approach, the M4 core can be loaded in few tens of milli seconds.

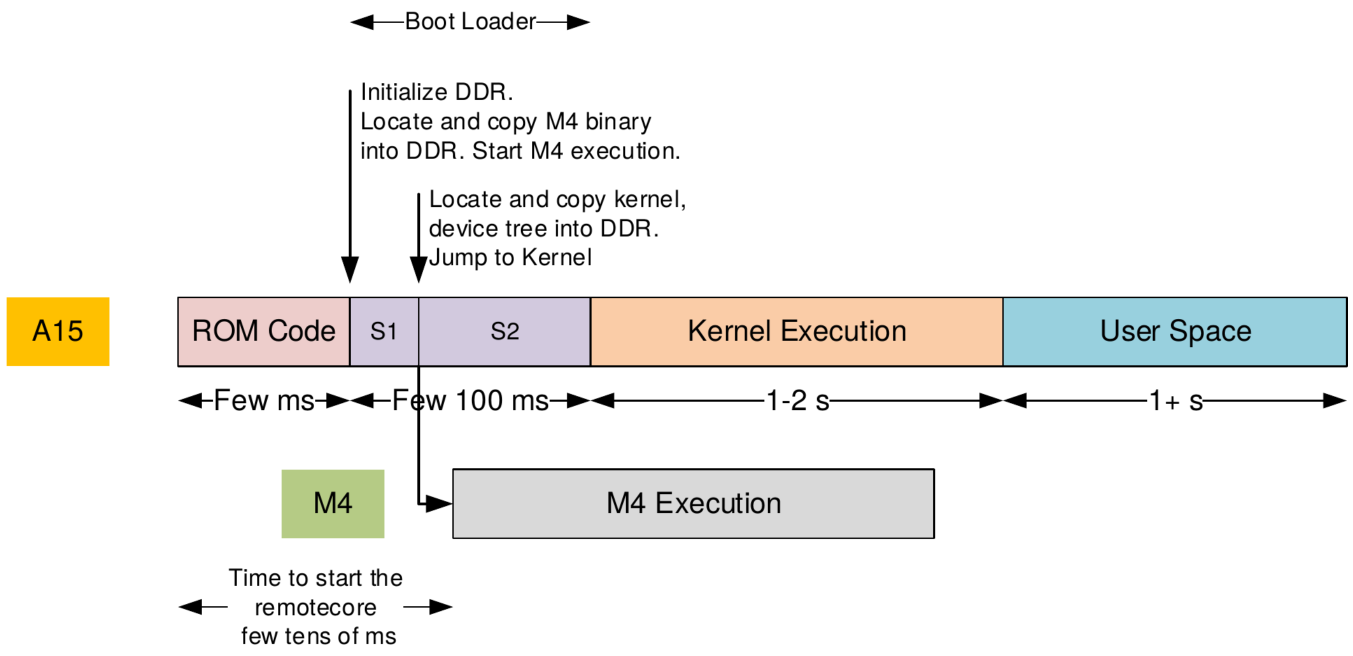

Figure 2. Linux Early Boot Flow

Minimal board initialization is performed in MLO before loading the M4 core. Figure 3 shows the various stages of execution before CAN response.

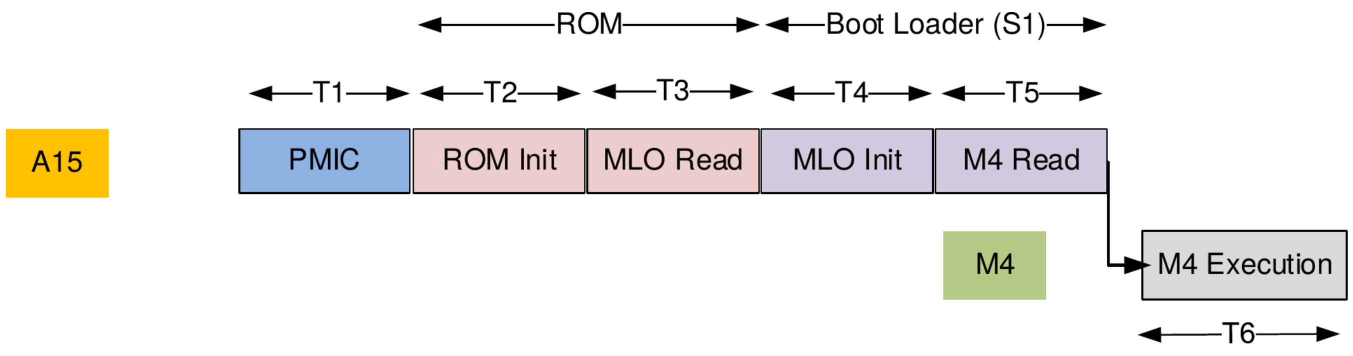

Figure 3. Early CAN Breakdown

Table 1. CAN Startup Stages

Stage	Stage Name	Processor	Binary	Explanation
T1	PMIC	NA	NA	Time taken from the time the ECU is powered on to the time PMIC has ramped up all the power rails and takes the SOC out of reset. This is board dependent.
T2	ROM Init	A15	ROM	Time taken by the ROM to do basic setup and initialize the boot media
T3	MLO Read	A15	ROM	Time to copy MLO from QSPI into OCMC memory
T4	MLO Init	A15	MLO	Time taken by MLO to initialize DDR, setup core voltages
T5	M4 Copy	A15	MLO	Time taken by MLO to read M4 binary from QSPI and take M4 core out of reset.
T6	M4 Exec	M4	CAN Stack	Time taken by M4 image to initialize and be ready to receive a CAN message

This document focuses on minimizing the generic components of this initialization sequence, for example, `T2`, `T3`, `T4` and `T5`. The remaining two components `T1` (PMIC) and `T6` (M4 Exec) are heavily dependent on the final board design and the CAN stack, respectively.

The next few sections go into high level details on how each of these components were optimized.

2.1 ROM Init Time

DRA7xx boot ROM is described in detail in the *Initialization* chapter in the *DRA75xP, DRA74xP, DRA77xP, DRA76xP SoC for Automotive Infotainment Silicon Revision 1.0 Technical Reference Manual*. Out of the boot modes supported by the DRA7xx SOC, the DRA7xx EVM by default supports booting out of the following persistent media:

- MMC/SD
- eMMC
- QSPI

Of these three boot media, MMC/SD and eMMC require a handshake as part of the initialization sequence. This handshake also involves a compulsory timeout in order to handle parts from different manufacturers. As a result, time taken by ROM to be ready to read from MMC/SD and eMMC is more than 100 ms. This makes QSPI the only boot media that can be used for early CAN response on the DRA7xx EVM.

A number for `T2` is not provided as there is not a way to measure `T2` standalone on the EVM. Based on estimations, `T2` should be around 15 ms when booting from QSPI.

2.2 MLO Read

On DRA7xx, Boot ROM copies the first stage bootloader (MLO/SPL) from QSPI at a conservative speed of 11 MBps. A 120 KB MLO binary takes around 11 ms to be copied from QSPI into OCMC. Usually this is not an issue. However, in case of early CAN response, the 11 ms spent in copying MLO forms a significant portion of the usecase time.

To reduce this copy time, a micro bootloader `umlo` is used (see Section 3.2.3). `umlo` is flashed at offset 0 in QSPI and the actual MLO at a 64 KB offset. ROM reads and executes `umlo`. As `umlo` is < 1 KB in size, the data copy time for `umlo` is negligible. `umlo` first sets up the QSPI interface to the maximum speed possible on DRA7xx (76.8 MHz interface clock, Quad Mode and Mode 0 operation). Then, `umlo` copies the `MLO` to the execution address in OCMC and jumps to it.

With this change, the time taken from A15 to start executing a 120 KB MLO from PORz reduces from 24.5 ms to 19 ms, a saving of 5.5 ms. The 19 ms measured in this case is `T2 + T3` as per the diagram shown in Figure 3.

2.3 MLO Init

The first stage bootloader (`MLO`) delivered as part of the SDK performs initialization necessary for a wide variety of usecases. As a result, the initialization time before `MLO` is ready to load the M4 core is high. To reduce this initialization time, the following was done:

- Split the initialization into two portions:
  - Initialization required for loading the M4 core.
  - Initialization not required for loading the M4 core but required for the full boot.
- Optimized the initialization time required for loading the M4 core.

With these changes, the time spent in MLO initialization before loading the M4 core (`T4`) comes down to 5.6 ms.

2.4 M4 Read

Reading the M4 binary from QSPI is another major contributor to the boot time. compressing the binary when flashing it to QSPI and uncompressing it after reading it into DDR reduced the read time. The choice of the compression format depends on:

- The compression offered by the compressor on the host PC - This impacts the time to read the binary from QSPI.
- The decompression speed on the target at the chosen CPU speed - This is an additional line item compared to the loading the uncompressed binary.

After experiments with different compression formats, it was decided to use the `lzo` compression format for M4 binary. With this change, the time to read the binary (`T5`) reduces to 4.05 ms.

In the next section, the steps to replicate the results on a TI EVM are described.

3 Replicating the Results

3.1 Hardware Requirements

- Linux PC running Ubuntu 14.04 LTS or any other version supported by Processor SDK Linux Automotive.
- Micro-USB to USB cable for connecting EVM to the PC. This is used for transferring the binaries from the PC to the board.
- Mini-USB to USB cable to displaying UART logs on the PC.
- An EVM of one of the Jacinto 6 family devices.
  - These patches were tested on a Rev B J6 Eco EVM. The patches should work without modification on all other J6 variant SOC's as well.

3.2 Software Requirements

This application report is intended for use with Processor SDK Linux Automotive 3.04 or later. The patches described below apply on top of the U-Boot commit used in the SDK. To obtain the latest Processor SDK Linux Automotive, go to:

http://processors.wiki.ti.com/index.php/Category:Processor_SDK_Linux_Automotive

Ensure that you:

- Have a working Processor SDK Linux Automotive 3.04 or later installed.
- Are able to build U-boot and kernel
- Are able to bring up the EVM with the U-Boot and Kernel images you have built.

3.2.1 u-boot-tools

The `mkimage` binary was used from the `u-boot-tools` package for wrapping binaries in the uImage header. This package can be installed by running the following command:

```
host $ sudo apt-get install u-boot-tools
```

3.2.2 lzop

The `lzop` utility was used to compress the binaries with the LZO compression. This utility can be installed by using the command.

```
host $ sudo apt-get install lzop
```

3.2.3 umlo

A micro MLO (`umlo`) was used to speed up the copying of MLO from QSPI into DRA7xx on-chip memory (OCMC). You can obtain the source and prebuilt binary of `umlo` by cloning the following git repository:

```
$ git clone git://git.ti.com/glsdk/dra7xx-umlo.git
```

For more information on `umlo`, see the included README.md file.

4 Testing Early CAN on EVM

4.1 Patching and Building U-Boot

You can fetch the u-boot changes required for this document and build it using the following command. Review 38746 patch revision 2 is the latest version of the patches as of this application report.

```
host $ cd u-boot/
host $ git fetch http://review.omapzoom.org/repo/u-boot refs/changes/46/38746/2
host $ git checkout FETCH_HEAD
host $ make dra7xx_evm_defconfig
host $ make
```

Copy `MLO` and `u-boot.img` from the build output into the FAT partition of the SD card.

4.2 Preparing the IPU Binary

The `messageq_single.xem4` binary is used from the PSDKLA release for the testing. The binary can be found in the `lib/firmware` folder of the target filesystem. The binary is about 120 KB in size, which is in the ballpark of a minimal CAN stack size.

1. Strip the symbols from the binary to reduce the size. You might want to make a copy of the binary as the original is overwritten.

   ```
   host $ du -k messageq_single.xem4
   4248    messageq_single.xem4
   host $ /opt/ti-devkit/ti-cgt-arm_5.2.7/bin/armstrip -p messageq_single.xem4
   host $ du -k messageq_single.xem4
   120     messageq_single.xem4
   ```

2. Compress the binary using LZO compression.

   ```
   host $ lzop -0 -c messageq_single.xem4 > messageq_single.xem4.lzo
   host $ du -k messageq_single.xem4.lzo
   68      messageq_single.xem4.lzo
   ```

3. Wrap the compressed binary in a uImage header. This is required for MLO to determine the amount of data to be read.

   ```
   host $ mkimage -d messageq_single.xem4.lzo messageq_single.xem4.lzo.uImage
   ```

4.3 Hardware Setup Instructions

1. Modify Switch settings on the EVM to the following - which places the EVM is SD boot mode.

   ```
   SW2[7:0] 0000 0111
   SW3[7:0] 0000 0001

   SW5[9:0] 00 0001 0100

   SW8[1:0] 11
   ```

2. Connect a USB cable from P2/USB1 to host PC. This is used for flashing the EVM using `fastboot`.
3. Connect a USB cable from the USB-UART adapter on the EVM to the host PC.

4.4 Flashing Instructions

1. Insert the SD card with MLO and U-Boot into the EVM. Place the EVM in SD boot mode. Reboot and stop at the U-Boot prompt.

   ```
   SW2[7:0] 0000 0111
   SW3[7:0] 1000 0001
   ```

2. Run the following commands to clear any old env settings and reboot.

   ```
   => env default -f -a
   => env save
   ```

3. Stop again at the U-Boot prompt and enter the fastboot state using the following command:

   ```
   => fastboot 0
   ```

4. Run the following commands on the host PC. These commands flash the QSPI with umlo, MLO, and the remotecore binary. The rest of the binaries are not flashed as our intent is to demonstrate the reduction in IPU1 load time.

```
host $ fastboot oem spi
host $ fastboot flash umlo umlo
host $ fastboot flash xloader MLO
host $ fastboot flash ipu1 messageq_single.xem4.lzo.uImage
```

5. Once the above commands are complete, change the SW2 setting as shown below. This places the EVM in production QSPI4 boot mode. Reboot the EVM.

```
SW2[7:0] 0011 0111
```

Section 4.5 describes how to interpret the boot logs to obtain the IPU boot time.

4.5 Interpreting the Output

Below is a line by line break down of the logs printed from MLO. Line numbers are added for illustration in the document.

Lines 1-2 are the standard SPL prints after the console has been initialized. As console initialization is deferred until IPU1 is loaded, IPU1 is already running at this point.

```
01: U-Boot SPL 2016.05-00027-g12e5f53400 (Dec 06 2017 - 12:19:54)
02: DRA722-GP ES1.0
```

Lines 3-8 are debug prints from IPU1. The debug prints are usually stored in a trace buffer(DDR) and are displayed by the kernel via the `debugfs` interface. The trace buffer is read and printed the first 1024 characters to the screen as an indication that IPU1 is executing.

```
03: [0][      0.000] Watchdog enabled: TimerBase = 0x68824000 ...
04: [0][      0.000] Watchdog enabled: TimerBase = 0x68826000 ...
05: [0][      0.000] Watchdog_restore registered as a resume callback
06: [0][      0.000] 18 Resource entries at 0x3000
07: [0][      0.000] messageq_single.c:main: MultiProc id = 2
08:
```

Lines 9-13 provide timestamps at various phases of MLO execution in terms of the 32 KHz timer on the SOC. An explanation of each of these points is provided in subsequent chapters.

```
09: rom_handoff_time         is    619 ticks
10: board_init_entry_time    is    761 ticks
11: mlo_init_done_time       is    804 ticks
12: ipu1_start_time          is    937 ticks
13: Entering Infinite loop
```

For now, the interest is only about `ipu1_start_time`, which indicates the time at which IPU1 has started execution. This is 937 ticks of the 32 KHz timer or **28.6 ms** (937/32.768 = 28.6). This 28.6 ms measurement corresponds to `T2 + T3 + T4 + T5` in Figure 3.

Section 5 describes the u-boot patches used for this purpose.

5 Patch Description

The patches are in four sets.

- **Basic benchmarking and optimization** - These patches are reused from SPRAC82 and provide a framework for benchmarking MLO execution and for flashing the binaries.

Table 2. Basic Benchmarking and Optimization

S.No	URL	Headline
1	http://review.omapzoom.org/38255	fastboot: update linux partition table
2	http://review.omapzoom.org/38729	fastboot: flash: add buffer overflow check for cmd
3	http://review.omapzoom.org/38730	fastboot: erase QSPI boot areas only when necessary
4	http://review.omapzoom.org/38258	fastboot: add more partitions to QSPI
5	http://review.omapzoom.org/38259	dra7xx: add functions for timestamping
6	http://review.omapzoom.org/38260	spl: dra7xx: timestamp various points in execution
7	http://review.omapzoom.org/38261	spl: dra7xx: propagate boot time measurements via dtb in single stage boot
8	http://review.omapzoom.org/38262	dra7xx_evm: minor change to config option
9	http://review.omapzoom.org/38263	dra7xx_evm: spl: disable env support

- **QSPI Support for Early Boot -** These patches add support for reading the remotecore binaries from a specified offset in QSPI memory. They also update U-Boot for flashing the IPU binary into QSPI.

Table 3. QSPI Support for Early Boot

S.No	URL	Headline
10	http://review.omapzoom.org/38731	spl: dra7xx: early boot: refactor spl_load_cores
11	http://review.omapzoom.org/38732	spl: dra7xx: early boot: read remotecore binary from QSPI
12	http://review.omapzoom.org/38733	dra7xx: update spi partition table for ipu1 early boot
13	http://review.omapzoom.org/38734	config: dra7xx: enable late attach

- **MLO Optimization for early CAN -** These patches perform various optimizations to
  – Reduce the time spent in MLO before loading the remotecores:
    - Defering initialization not required for starting the remotecores
    - Optimize operations that cannot be deferred
- Reduce the time spent in reading the remotecores from QSPI by using compressed binaries.

Table 4. MLO Optimization for Early CAN

S.No	URL	Headline
14	http://review.omapzoom.org/38735	spl: dra7xx: disable board detection
15	http://review.omapzoom.org/38736	config: dra7xx: do not clear malloc memory
16	http://review.omapzoom.org/38737	dra7xx: i2c: reduce sleeps in i2c initialization
17	http://review.omapzoom.org/38738	spl: dra7xx: defer console init to reduce ipu boot time
18	http://review.omapzoom.org/38739	spl: dra7xx: defer scaling unnecessary core voltages
19	http://review.omapzoom.org/38740	spl: dra7xx: early boot: do not clear unsed memory
20	http://review.omapzoom.org/38741	spl: dra7xx: support lzo compression for remotecore binaries
21	http://review.omapzoom.org/38742	config: dra7xx: enable compression for remotecore binaries
22	http://review.omapzoom.org/38743	spl: dra7xx: use dma to clear bss section

These patches have been tested on DRA74x and DRA72x platforms. When testing on other DRA7xx platforms, revert the patch No. 14 in case of issues.

- **Boot time display** - These patches print the benchmarking information from various stages of the boot to the serial terminal. This allows us immediate feedback instead booting the kernel to obtain the boot numbers.

Table 5. Display Boot Time Achieved

S.No	URL	Headline
23	http://review.omapzoom.org/38744	spl: dra7xx: add code to print boot measurements
24	http://review.omapzoom.org/38745	spl: dra7xx: early boot: add function to print trace buffer
25	http://review.omapzoom.org/38670	spl: dra7xx: add an infinite loop function for debug
26	http://review.omapzoom.org/38746	spl: dra7xx: print boot times,traces and loop forever

5.1 Other tools

While developing the above patches, a peripheral boot was used to reduce the testing and benchmarking time of MLO. The procedure is documented in *Using Peripheral Boot and DFU for Rapid Development on Jacinto 6 Devices*

The following patch was also used for benchmarking code at a finer level to determine the time consumed at a function call level.

Table 6. Display Boot Time Achieved

S.No	URL	Headline
1	http://review.omapzoom.org/38846	spl: dra7xx: finer benchmarking changes

6 Further Optimizations

Below is a list of more optimizations possible to reduce the load time even further:

- CAN message reception - The CAN peripheral can be configured to receive messages from the CAN bus as soon as the IO delay configuration is complete. This operation can be done in MLO. CAN peripheral has internal message memory where messages meeting the acceptance criteria can be stored. Once the IPU is booted, it can read the messages from the CAN peripheral internal memory and respond to them.
- ROM Execution time - Alternatively one can use GPMC NOR to start MLO in XIP fashion and save the MLO Read time. However, the I/O delay recalibration code needs to run in isolation mode. As a result, one can only run MLO in XIP mode only partially. MLO needs to be copied into OCMC before running IO delay recalibration. However, GPMC NOR offers ~40 MBps throughput would reduce the MLO copy time by at least 2 ms.

7 Conclusion

This document discusses how the M4 core on DRA7xx can be brought up within 29 ms from PORz. This allows the M4 core to be ready to respond to a CAN message within 50 ms after allowing 20 ms PMIC rampup and M4 execution time.

8 References

- *DRA75xP, DRA74xP, DRA77xP, DRA76xP SoC for Automotive Infotainment Silicon Revision 1.0 Technical Reference Manual*
- *Using Peripheral Boot and DFU for Rapid Development on Jacinto 6 Devices*