Application Note

Jacinto7 HS Device Flashing Solution

Neo Wang, Tommy Song and Jadav Brijesh

ABSTRACT

Jacinto7 (J7) series SoC have two types of silicon: The General Purpose (GP) and High-Security (HS). For HS devices, the flashing is more challenging than GP devices as the security features used in HS silicon, because the JTAG is locked in HS device and all the binary should be signed and encrypted. In normal flashing solution, the universal asynchronous receiver/transmitter (UART) interface is necessary, otherwise the JTAG and multiple boot mode is required. This application report introduces the normally flashing solution when the UART interface exists, and also covers the case where there is no UART interface. The document also introduces the HS flashing methods in multiple memory boot mode and even provides a solution to flash the HS silicon in OSPI memory boot mode only.

Project collateral and code mentioned in this document can be downloaded from the following URL: https://www.ti.com/lit/zip/spracz6.

Table of Contents

List of Figures

List of Tables

Trademarks

All trademarks are the property of their respective owners.

1 Introduction

There are multiple boot modes supported by J7 SoC, mainly can be divided into two types: peripheral booting and memory booting. Different boot mode required different hardware design and software configuration, details can be found in boot mode setting.

- For peripheral booting, the ROM code gets the image through selected communication interface to run the silicon. This is normally used in development phase.
- For memory booting, the ROM code gets the image from memory devices, such as flash, SD Card, EMMC, to run the silicon. This is normally used in final product.

There is another special boot mode called “No boot” mode which is used for JTAG connection to directly load the image to DDR through JTAG interface during development phases.

For J7 based design, OSPI boot mode is normally chosen because of boot speed. Then, how to flash the OSPI flash during development and manufacturing is an important part of the design process. Different projects use different hardware design, and bring different interface. Even under the same hardware design, different silicon type will also bring different restrictions. So one flashing solution cannot be used to deal with all cases, this note introduces the OSPI flashing solution in HS silicon with different hardware design.

As shown in Figure 1-1, from security enabled product perspective, three phases included in J7 SOC development. In each phase, different device types have different features and are used for different purposes.

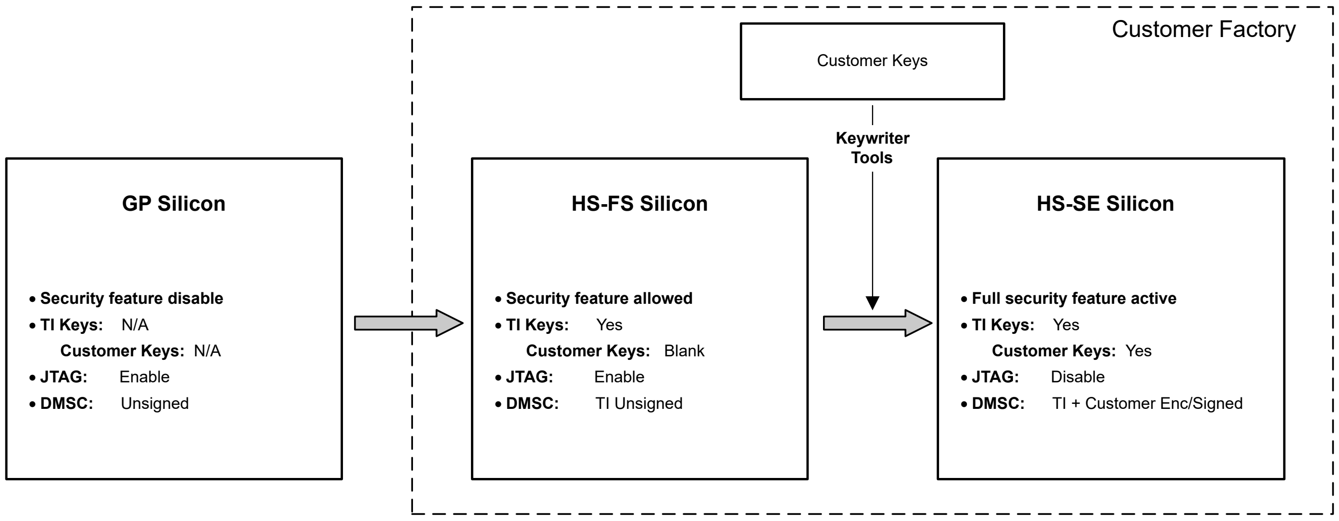

Figure 1-1. J7 Device Types

GP silicon is the basic silicon without security features enabled, on GP silicon, JTAG is enabled and then it can be used for flashing, meanwhile all the system binary does not need to be signed and encrypted, so GP silicon normally used for functionality development.

HS-FS is Field Securable silicon with valid secure features, but not enabled until the customer keys are programmed during manufacture. The Device Management and Security Control (DMSC) subsystem is protected by TI security keys. On HS-FS silicon, the JTAG can still be used for flashing.

HS-SE is Security Enforced silicon with customer keys programmed. This is done with the Key writer tools provided by TI at customer site. Therefore, the image running on the silicon need to be signed and encrypted by the keys and JTAG is disabled by default to enforce the security. On HS-SE silicon, separate consideration required for flashing.

According to the system design, different flashing solution can be chosen. The solution can follow the flowchart shown in Figure 1-2.

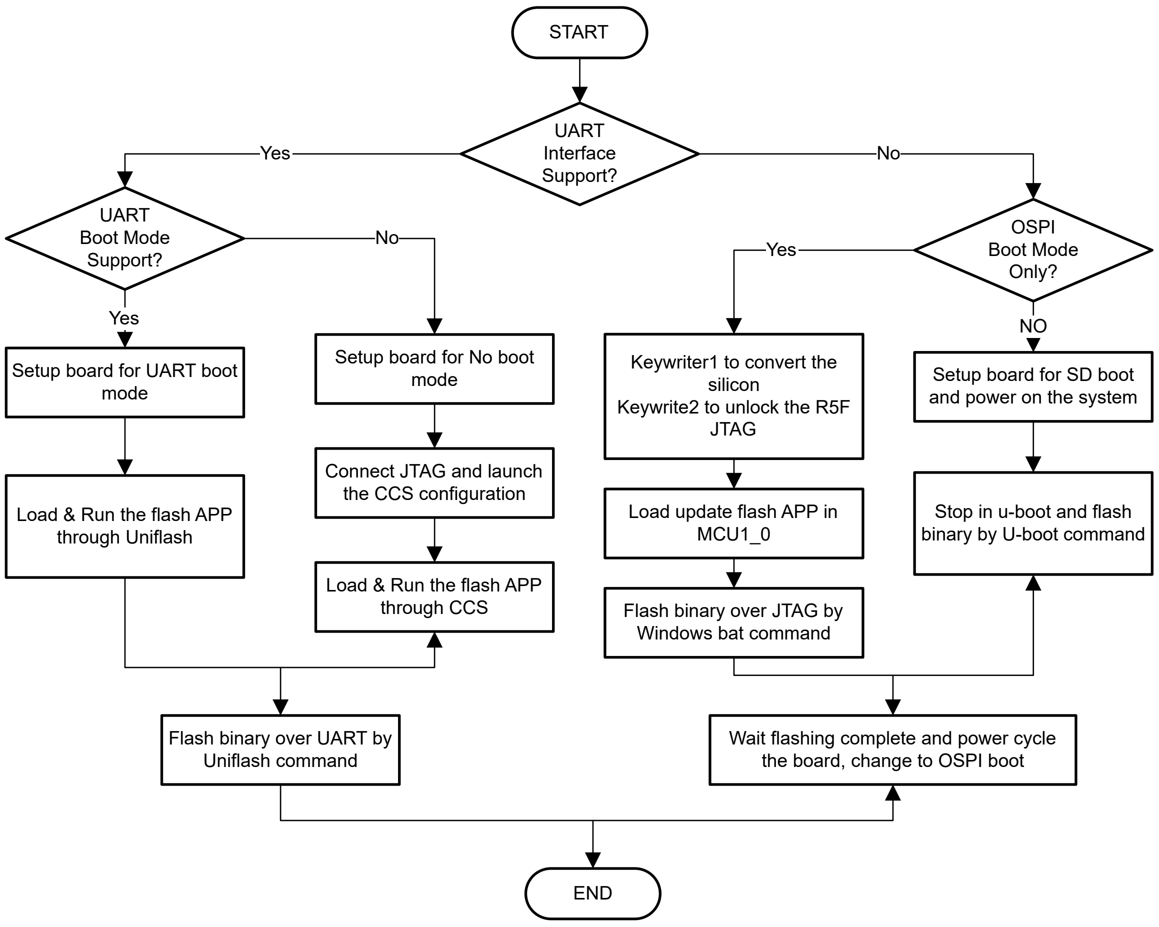
START
UART Interface Support?
Yes
No
UART Boot Mode Support?
Yes
No
OSPI Boot Mode Only?
Yes
NO
Setup board for UART boot mode
Setup board for No boot mode
Keywriter1 to convert the silicon Keywrite2 to unlock the R5F JTAG
Setup board for SD boot and power on the system
Load & Run the flash APP through Uniflash
Connect JTAG and launch the CCS configuration
Load update flash APP in MCU1_0
Stop in u-boot and flash binary by U-boot command
Load & Run the flash APP through CCS
Flash binary over JTAG by Windows bat command
Flash binary over UART by Uniflash command
Wait flashing complete and power cycle the board, change to OSPI boot
END

Figure 1-2. Flashing Solution Selection Flow

How to select the proper flashing solution depends on the system design.

- Uniflash can be used when there is UART interface available. This is the simpler solution, but the UART interface in final product may introduce additional cost and exposure of system information in the UART log. Also, the device can only be flashing online in factory, lack of flexibility and efficiency, and the UART is slower speed interface and not suitable for flashing large binaries.

As shown in above flow chart, secondary boot mode like SD boot mode is required to do the flashing when UART interface is not available. This requires switching to the secondary memory boot mode that may:

- Need separate interface in the design. The procedure is simple and flexible, but also introduce the additional hardware cost and low efficiency in factory.
- Separate consideration required for OSPI-memory-only boot mode. It is the most cost-effective design as no additional hardware required, and improved factory efficiency because it can be flashing offline. There is a big challenge for this solution: how to support the offline flashing and then how to do online flashing after the board is converted to SE device. A new method is proposed in this application notes to support this scenario.

All the HS flashing solution were tested in SDK7.1 software with J721E-EVM hardware and works fine.

2 HS Device Flashing With UART Interface

This section introduces the OSPI flashing solution with UART interface supported and divided into two parts. For more information on whether to support UART boot mode, see install the Uniflash tools.

2.1 UART Interface With UART Boot Mode Supported

Use the following steps to flash the binary to OSPI through UART in Linux PC:

1. Configure boot mode of board to UART boot and connect MCU UART serial port of board to host PC.
2. Power cycle the board and get the console instance number, like /dev/ttyUSB1.
3. Navigate to the Uniflash installed directory, dslite.sh script should be present here, and use the following command to load the flash programmer in Linux terminal.
4.
```
# sudo ./dslite.sh --mode processors -c <COM Port> -f <Flash Programmer Binary with Full Path>
-i 0
Example:
# sudo ./dslite.sh –mode processors -c /dev/ttyUSB1 -f (Path to Uniflash Install Directory) /
processors/FlashWriter/j721e_evm /uart_j721e_evm_flash_programmer_release.tiimage -i 0
```
5. And in same folder, use the following command to flash the binary to OSPI.
6.
```
# sudo ./dslite.sh --mode processors -c <COM Port> -f <Flash Programmer Binary with Full Path>
-i 0
Example:
# sudo ./dslite.sh –mode processors -c /dev/ttyUSB1 -f (Path to Uniflash Install Directory) /
processors/FlashWriter/j721e_evm /uart_j721e_evm_flash_programmer_release.tiimage -i 0
```
7. Power off the board and change the boot mode to OSPI boot, power on the board and system will start up from OSPI.

2.2 UART Interface Without UART Boot Mode Supported

If the design can support UART interface but cannot support UART boot mode, use the following steps can be used for OSPI flashing.

1. Configure boot mode of board to no boot and connect MCU UART serial port of board to host PC, get the console instance number, like /dev/ttyUSB1.
2. Power cycle the board, prepare the CCS target configuration file for the platform under test and JTAG emulator being used.
3. Connect the JTAG port of board to host PC, and run the following command from Uniflash install folder to run the CCS and load the flash programmer.
```
# sudo ./dslite.sh --mode load --config=<CCS Target Config file (ccxml) with Full Path>
-f <Flash Programmer Binary with Full Path> -n <Core ID to be Connected to JTAG on the Target
Platform>
Example:
#sudo ./dslite.sh --mode load --config=/home/ti/CCSTargetConfigurations/j721e.ccxml
-f (Path to Uniflash Install Directory) /processors/FlashWriter/j721e_evm /
uart_j721e_evm_flash_programmer_release.tiimage -n 8
```
4. After successful download of the flash programmer, use the following command from Uniflash install folder to flash the binary to OSPI.
```
# sudo ./dslite.sh --mode processors -c <COM Port> -f <Path to the binary to be flashed> -d
<Flash Device Type> -o <offset>
Example:
#sudo ./dslite.sh –mode processors -c /dev/ttyUSB1 -f (SDK Install Directory)/pdk/packages/ti/
boot/sbl/binary/j721e_evm/cust
/bin/sbl_cust_img_mcu1_0_release.tiimage -d 3 -o 0
```
5. Power off the board and change the boot mode to OSPI boot, power on the board and system will start up from OSPI.

3 HS Device Flashing With Multiple Memory Boot Mode

If the UART interface is not supported and there are other memory boot modes besides OSPI boot mode, like SD boot mode, below steps required for flashing:

1. Install the keywriter addon package in SDK and build keywriter with customer key, find the command and log as shown below.

```
$make keywriter_img
#......
# SBL image ~/ti-processor-sdk-rtos-j721e-evm-07_01_00_11/pdk_jacinto_07_01_00_45/packages/ti/
boot/sbl/example/k3MulticoreApp/binary/keywriter_img_j721e_release.tiimage created.
```

2. In SD boot mode, power on the HS-FS device and run keywriter application, convert the device to HS-SE device, find log as shown below if the key programing is completed.

```
#Keywriter Revision: 01.00.00.00 (Mar  7 2021 - 20:15:01)
# OTP Keywriter ver: 20.8.5-w2020.23-am64x-14-g7409e
#Beginning key programming sequence
#Taking OTP configuration from 0x41c7e000
#Debug response: 0x0
#Key programming is complete
```

3. Sign and encrypt the u-boot binary in customer HSM first, meanwhile also Sign and encrypt all binaries that will flash into OSPI, then copy those binaries to SD card boot partition.
4. Insert the SD card and power cycle the board, stop in u-boot command phase, and flash all the binaries to OSPI from SD card by using the following command.

```
Hit any key to stop autoboot:  0
=>
=> sf probe
SF: Detected s28hs512t with page size 256 Bytes, erase size 256 KiB, total 64 MiB
=> mmc dev 1
switch to partitions #0, OK
mmc1 is current device
=> fatload mmc 1 ${loadaddr} sbl_ospi_img_mcu1_0_release.tiimage
231444 bytes read in 5 ms (44.1 MiB/s)
=> sf update ${loadaddr} 0x0 0x40000
device 0 offset 0x0, size 0x40000
262144 bytes written, 0 bytes skipped in 1.331s, speed 201225 B/s
=>
```

5. Switch to OSPI boot mode and power cycle the board.
6. Repeated the step 3 to approximately 5 to update the system binary.

Same procedure also applies to other memory boot modes, only need to stop in u-boot phase and flash OSPI.

4 HS Device Flashing With OSPI Memory Boot Mode Only

In case, the board design supports only OSPI memory boot mode and there is no support for UART interface, the above methods cannot be used. Also, once the device is converted to HS-SE, as shown in Table 4-1, even the JTAG is locked. This makes it impossible to access flash again after keywriter completed. This requires first JTAG to be unlocked. Here, the secondary boot image is used with JTAG unlock functionality. This boot image replaces the keywriter1 application after the programming key. In this application report, a copy of the keywriter application is used as secondary image to unlock the JTAG, it is named as keywriter2. Then, once the JTAG is unlocked, a JTAG R5F flashing application can be used to flashing the OSPI.

Table 4-1. The JTAG State in GP/HS Silicon

Device Type	Variant	M3 JTAG States	R5F JTAG States
General Purpose (GP)	NA	Open	Open
High Security	Field Securable (FS)	Closed	Open
High Security	Security Enforced (SE)	Closed	Default Closed

4.1 Backup-Read Based Keywriter

TDA4 ROM supports Backup OSPI offset. It jumps to this backup OSPI offset, if it does not find valid image at primary OSPI offset, that is offset0. For more information on backup OSPI offset, see the device-specific TRM or data sheet.

This mechanism is used in this application note to unlock the JTAG. Firstly, on HS-FS device, since JTAG connection is available for MCU R5F, keywritter1 can be flashed using the CCS/JTAG-based flash writer tool, this application programs the customer key and convert the device to HS-SE. Along with the keywritter1, the JTAG-based flash writer tool also flashes the second kerywriter2 application at the backup OSPI offset (that is, at offset 4MB). When the board is rebooted, the keywriter1 application flashes customer keys and converts the HS-FS device into HS-SE. On the second reboot, since keywriter1 application is not signed with the customer key, it is not valid and ROM jumps to the backup OSPI offset, where the customer key signed keywriter2 application is flashed. This application unlocks JTAG for MCU R5F, then CCS/JTAG-based flashing tools can be used to flash rest of the binaries. Overall process as shown in Figure 4-1.

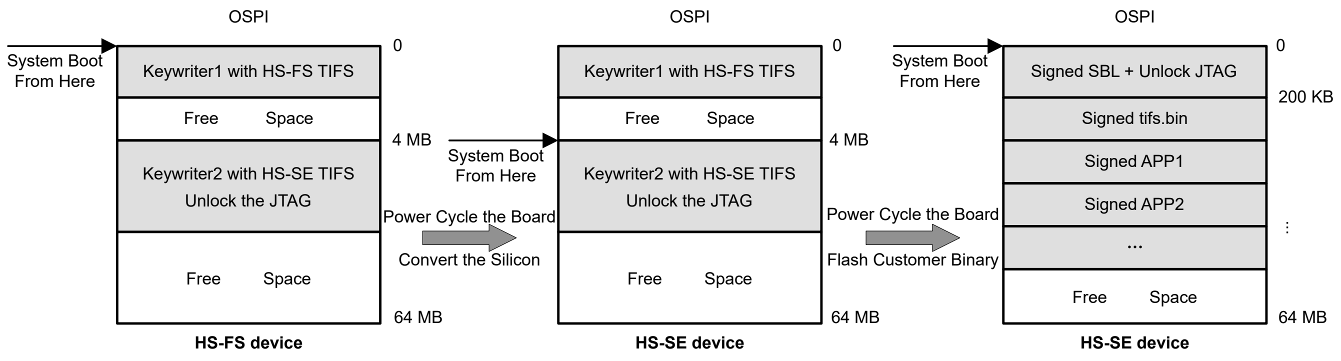

Figure 4-1. Backup-Read Based Keywriter Architecture

So, two separate images need to be built. The first one is the default keywriter, named as Keywriter1, which can be compiled directly with same steps mentioned in Chapter 3. The second image adds the JTAG unlock functionality to the keywriter application and integrates the HS-SE TIFS. This is named as keywriter2 and needs the following additional steps:

1. Comment the actual key programming API at $SDK_PATH/ti-processor-sdk-rtos-j721e-evm-07_01_00_11/pdk_jacinto_07_01_00_45/packages/ti/boot/sbl/example/k3MulticoreApp/keywriter/main.c, and add the log to distinguish between keywriter1 and keywriter2 after flashing them together.

```
UART_printf("Taking OTP configuration from 0x%x\n", (uint32_t *)keywriter_cert);
UART_printf("this is keywriter2 in offset 4MB!\r\n");
// status = Sciclient_otpProcessKeyCfg((uint32_t *)keywriter_cert,
SCICLIENT_SERVICE_WAIT_FOREVER, &debug_response);
if (status != CSL_PASS){
    UART_printf("Something wrong happened!!\n");
}
```

2. Enable the R5F JTAG open/close flag at $SDK_PATH/ti-processor-sdk-rtos-j721e-evm-07_01_00_11/pdk_jacinto_07_01_00_45/ti/build/makerules/common.mk, need add below compile flag.

```
$(SBL_CERT_GEN) -b $(SBL_BIN_PATH) -o $(SBL_TIIMAGE_PATH) -c R5 -l $(SBL_RUN_ADDRESS) -k $($
(APP_NAME)_SBL_CERT_KEY) -d DEBUG -j DBG_FULL_ENABLE -m $(SBL_MCU_STARTUP_MODE)
```

3. Adopt the customer keys in SDK and signed the keywriter with these keys. Make sure the customer keys were updated in the $SDK_PATH/ti-processor-sdk-rtos-j721e-evm-07_01_00_11/pdk_jacinto_07_01_00_45/packages/ti/boot/sbl/example/k3MulticoreApp/keywriter/scripts/keys folder. And these keys need exactly same as the keys flashed into the TDA4 silicon while compile the keywriter1.
4. Update the TIFS for SE device. The TIFS would load the keywriter in the form of array. After manual update, use the following command; the array would update after compiling.

```
$cp <pdk>/drv/sciclient/soc/V1/sciclient_firmware_V1-hs-enc.h <pdk>/<keywriter>/ti-fs-keywriter.h
$cp <pdk>/drv/sciclient/soc/V1/tifs-hs-enc <pdk>/<keywriter>/tifs_bin/ti-fs-keywriter.bin
```

5. Compile the related lib, keywriter2 application and SBL. By using the following command, you can get the keywriter2 in folder $SDK_PATH/pdk_jacinto_07_01_00_45/packages/ti/boot/sbl/example/k3MulticoreApp/binary. And update SBL in $SDK_PATH/pdk_jacinto_07_01_00_45/packages/ti/binary/sbl_cust_img_hs.

```
#make -s BOARD=j721e_evm BUILD_PROFILE=release sciclient_boardcfg BUILD_HS=yes
#make sciclient_direct_hs
#make keywriter_img -j8
#make sbl_cust_img_hs -j8
```

After flash these keywriter1 and keywriter2 binaries to OSPI via JTAG on HS-FS device. On first power cycle, keywriter1 application would run and convert HS-FS device to HS-SE by programming keys. On the second power cycle, since keywriter1 application can no longer be recognized, system would jump to second keywriter application and would program TIFS and unlock JTAG. In order to run the complete system function, the updated SBL should be flashed to OSPI address 0 and care should be taken to ensure that keywriter2 is not overwritten by SBL or other images.

4.2 JTAG Flashing Application on MCU1_0

After converting silicon to HS-SE and unlocking the JTAG, there is still need to design a JTAG flashing application on MCU1_0, which can load the binary from customer HSM and program to OSPI. In normal boot process, the system firmware should be loaded in DMSC first, then boot the MCU and other cores. But the DMSC ROM sets up a 3-minutes watchdog timer, the MCU boot needs to complete within this period, otherwise the WDT reset will occur and reboot the whole system. In the third bullet in this section, the keywriter1 and kerywriter2 can successfully be flashed because the binary size is small. Flashing can be finished in 3-minutes, but due to a bigger image size, you cannot flash all firmware binaries in 3-minutes. So, in order to avoid 3-minute watchdog timeout, you need to port the flashing application to MCU domain MCU1_0, such that it does not load TIFS to DMSC and also does not have dependency on SciClient services. The flashing flow shown in Figure 4-2 and all of the following source code can be downloaded from the following URL: https://www.ti.com/lit/zip/spracz6.

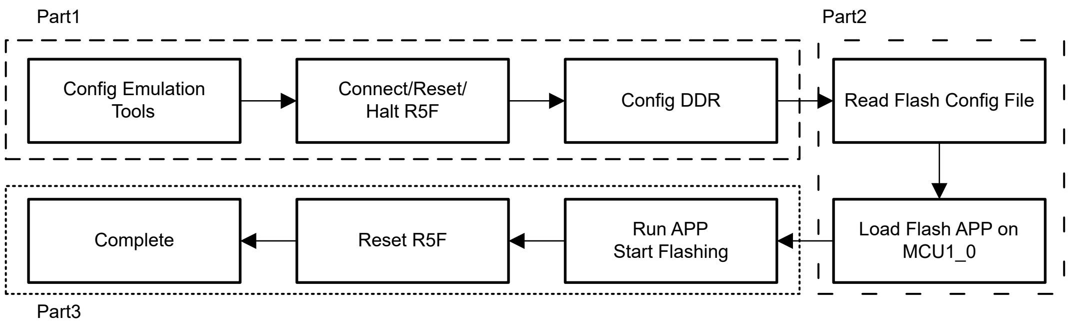

Figure 4-2. JTAG Flashing Application Process

The flash tools based on Windows system and including three parts:

- The first part, you need to use emulation tools to connect the TDA4 via JTAG, then using CCS .js and .gel file to connect/reset/halt the MCU1_0, initialize the DDR and configure everything from MCU1_0, you can design the part1 process by reference from the file in default SDK, which located in ${PSDKRA_PATH}/pdk/packages/ti/drv/sciclient/tools/ccsLoadDmsc/j721e/launch.js Also, you can refer to the below code to achieve the function:

  This part is a little difference for HS-FS and HS-SE device, in case of HS-FS, the DDR is initialized by .js file in no boot mode, but for HS-SE device, the system is working in the OSPI boot mode, and the DDR is initialized by system image.

```
function connectTargets()
{
    script.setScriptTimeout(200000);
    updateScriptVars();
    print("Connecting to MCU Cortex_R5_0!");
    // Connect the MCU R5F
    dsMCU1_0.target.connect();
     // Reset the R5F to be in clean state.
    dsMCU1_0.target.reset();
    dsMCU1_0.target.halt();
    dsMCU1_0.expression.evaluate("J7ES_LPDDR4_4266MTs_Config()");
}
function flashOspi()
{
    var configFileObj = new File("Configuration.txt");
    print("Starting Flashing !");
    if (configFileObj.exists())
{
var scanner = new Scanner(configFileObj);
var ospiOffset;
var ospiFile;
while (scanner.hasNextLine())
{
         var data = scanner.nextLine();
         var parts = data.split(" ");
         ospiOffset = parseInt(parts[0], 16)
         ospiFile = parts[1];
         var file = new File(ospiFile);
         fileSize = file.length();
        dsMCU1_0.memory.writeWord(0x0, 0x80000000, ospiOffset);
        dsMCU1_0.memory.writeWord(0x0, 0x80000004, fileSize);
        dsMCU1_0.memory.loadRaw(0, 0x90000000, ospiFile, 32, false);
        print("Flashing "+ospiFile+" of size " + fileSize + " at offset " + ospiOffset);
        // Connect the MCU R5F
        dsMCU1_0.target.connect();
        print("Running the OSPI load program from R5!");
        // Load the board configuration init file.
        dsMCU1_0.memory.loadProgram("bin/uart_j721e_evm_flash_programmer_release.xer5f");
       // Halt the R5F and re-run.
       dsMCU1_0.target.halt();
      dsMCU1_0.target.reset();
      dsMCU1_0.target.restart();
      dsMCU1_0.target.run();
      // Reset the R5F to be in clean state.
      dsMCU1_0.target.reset();
 }
    }
        scanner.close();
    }
    else
    {
        print("File Does not exist!");
    }
}
function disconnectTargets()
{
    updateScriptVars();
    // Reset the R5F to be in clean state.
dsMCU1_0.target.reset();
}
function doEverything()
{
    printVars();
    connectTargets();
    disconnectTargets();
    flashOspi();
    print("Burning is complete, happy to go !!!");
}
```

- The second part is mainly used to specify the file path and flashing address. The flashing tools in the third bullet in this section will read this configuration file and scan line by line, used to flash binary to the specified address, using the file format shown below:

```
0 image/se/tiboot3.bin
80000 image/se/tifs.bin
E1000 image/se/app
1E1000 image/se/app3_0
F5E000 image/se/lateapp1
175E000 image/se/lateapp2
```

- The third part is actual flashing application running in MCU1_0, this application will be running in OCMC, initialize the OSPI interface load the binary from PC to TDA4 DDR and flashing it to OSPI. You can find the complete patch as attached and use the following steps to compile.

```
$ cp ccs_main.c ${PSDKRA_PATH}/pdk_jacinto_07_00_00/packages/ti/board/utils/uniflash/target/src
$ cp uart_make.mk ${PSDKRA_PATH}/pdk_jacinto_07_00_00/packages/ti/board/utils/uniflash/target/
build
$ cd  ${PSDKRA_PATH}/pdk_jacinto_07_00_00/packages/ti/build
$ make -s   PLATFORM=j721e_evm BUILD_PROFILE=release board_utils_uart_flash_programmer
```

Based on the current backup-read keywriter mechanism and MCU1_0 flashing application, see the steps in Figure 4-3 to complete the flashing process.

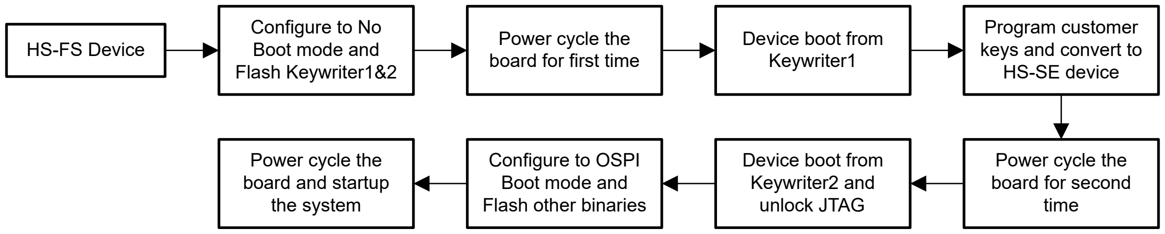
HS-FS Device
Configure to No Boot mode and Flash Keywriter1&2
Power cycle the board for first time
Device boot from Keywriter1
Program customer keys and convert to HS-SE device
Power cycle the board for second time
Device boot from Keywriter2 and unlock JTAG
Configure to OSPI Boot mode and Flash other binaries
Power cycle the board and startup the system

Figure 4-3. Whole Flashing Process

5 Flashing Demo

In case of only OSPI memory boot mode, use the complete file to flash the binary in windows PC. There are some slight differences of flashing tools between HS-FS device and HS-SE device, as shown in Table 5-1. The difference is FS device flashing based on no-boot mode and corresponding file will initialize the DDR, but SE device will not. You can find the complete sample flashing tools as attachment.

Table 5-1. Flashing Tools for OSPI Boot Mode Only

HS-FS Device	HS-SE Device
Binary/ Configuration.txt **Launch_fs.js** **flash_blackhawk560_NOboot.ccxml** uart_j721e_evm_flash_programmer_release.xer5f	Binary/ Configuration.txt **Launch_se.js** **flash_blackhawk560_OSPIboot.ccxml** uart_j721e_evm_flash_programmer_release.xer5f

The flashing machine should have already installed the CCS environment, the launch.js file should point to the DDR configuration folder, and the CCS ccxml file should point to CCS target configuration file with correct chip and emulation model. You can find more details here.

In windows PC, using win+R and input cmd to open the command windows, then enter the following command to start the flashing:

${ccs_install_path}\ccs_base\scripting\bin\dss.bat Launch_fs/se.js

The log as below when flash is complete:

```
D:\user\Desktop \Flash_Burn>C:\ti\ccs1000\ccs\\ccs_base\scripting\bin\dss.bat  Launch_se.js
Loaded FPGA Image: C:\ti\ccs1000\ccs\ccs_base\common\uscif\dtc_top.jbc
Connecting to MCU Cortex_R5_0!
Starting Flashing !
Flashing image/se/app of size 738712 at offset 921600
Running the OSPI load program from R5!
Flashing image/se/app3_0 of size 1064196 at offset 1970176
Running the OSPI load program from R5!
Flashing image/se/lateapp1 of size 7772252 at offset 16113664
Running the OSPI load program from R5!
Flashing image/se/lateapp2 of size 7336400 at offset 24502272
Running the OSPI load program from R5!
Burning is complete, happy to go !!!
Press Key to Continue … …
```

6 Summary

This application report summarizes the J7 HS device OSPI flashing solution in different scenarios. The same method also applies to J7 GP device. And, the proposed solution to flash the HS silicon with OSPI memory boot mode only can reduce the design complexity, hardware cost and improve the factory efficiency.