AN12631

Normal and Secure Debug for i.MX8/8X Family of Applications Processors

Rev. 0 — February 2020 Application Note

Contents

1 Introduction

1.1 Purpose

This document describes how to configure and use Normal/Secure JTAG on the i.MX8/8X family applications processors. This document does not apply to i.MX8M family of Application process. See AN4686 for 8M devices.

1.2 Intended audience

This document is intended for users who:

- Need an explanation about the procedure of secure debugging.
- Need an explanation about how to use Lauterbach tools to debug U-boot/Kernel/SCFW.

1.3 Definitions, Acronyms, and Abbreviations

The terms and acronyms used in this document are:

- ADM – Authenticated Debug Module; Module that works with the debug system and fuse configuration to provide security measures
- AHAB – Advanced High Assurance Boot; A software library executed in internal ROM on the NXP processor at boot time which, among other things authenticates software in external memory by verifying digital signatures in accordance with a CSF
- OTP – One-Time Programmable; The OTP hardware includes the masked ROM and electrically programmable fuses (eFuses)
- SCU – System Controller Unit
- SCFW – SCU FirmWare
- SDP – Serial Download Protocol; Also called UART/USB serial download mode. It allows code provisioning through UART or USB during the production and development phases
- SECO – Security Controller
- SJC – System JTAG Controller

2 Overview

2.1 System JTAG Controller

The JTAG port provides debug access to hardware blocks, including the Arm processor and the system bus. This allows program control and manipulation as well as visibility to the chip peripherals and memory.

The JTAG port must be accessible during initial platform development, manufacturing tests, and general troubleshooting. Given its capabilities, JTAG manipulation is a known attack vector for accessing sensitive data and gaining control over software execution. System JTAG Controller (SJC) protects against the whole range of attacks based on unauthorized JTAG manipulation.

In i.MX8/8X family, the System JTAG Controller (SJC) provides a method of regulating the JTAG access.

SJC provides the following security levels:

- JTAG Disabled – JTAG use is permanently blocked
- No-Debug – JTAG security sensitive features are permanently blocked
- Secure JTAG – JTAG use is restricted (as in the No-Debug level) unless a secret-key challenge/response protocol is successfully executed
- JTAG Enabled – JTAG use is unrestricted

Security levels are selected via e-fuse configuration. The fuse burning is an irreversible process, once a fuse is burned it is not possible to change the fuse back to the unburned state.

2.1.1 Boundary Scan

Arm signals DBGEN/NIDEN/SPIDEN/SPNIDEN are not tied with the Boundary Scan functionality, therefore, programming these fuses does not have any impact on Boundary Scan, which is enabled by default.

Boundary Scan functionality can be disabled in either of these scenarios:

- Setting the SJC Disable Fuse, since the JTAG controller is gated off.
- The device is in “No Return” Lifecycle (regardless of the SJC_DISABLE eFUSE).

2.2 The Authenticated Debug Module

The Authenticated Debug Module (ADM) is a module that works with the debug system and fuse configuration to provide security measures. It receives control signals from various sources such as pins, OTP fuses, and the SCU, Debug Access Port, and JTAG at boot and during runtime to determine, restrict, and indicate use of the debugging components. Certain debugging features are allowed or disallowed based on NXP and OEM requirements.

It supports the following functions:

- Debug Security Controls
- SDP Security Controls
- JTAG Security Controls
- Content Key Security Controls
- Chip Lifecycle Security Controls

The chip has multiple debug domains, as shown in the table below. Having access to one debug domain does not grant access to the others.

Table 1. Debug domains on the chip

Life Cycle/ Enabled without authentication	SCU FW	SECO FW	CLOSED OEM (with or without NXP firmware)	PARTIAL FIELD RETURN	FULL FIELD RETURN	NO FIELD RETURN
Debug Domain 0 (SCU)	No	Yes	No	Yes	Yes	No
Debug Domain 3 (APPS)	Yes	Yes	No	Yes	Yes	No
Debug Domain 5 (SECO)	Yes	No	No	No	Yes	No

2.3 Debug Flow when Secure Debug is enabled

In a closed configuration, the debug options are determined at boot time using signed code. However, if the Secure Debug level is chosen in the security fuse configuration then a successful Secure JTAG challenge/response allows access to debug features for the application and CM4 cores.

In other words, secure debug is coupled with secure boot and is enabled only if chip LC is OEM Closed and the JTAG Challenge/Response provides a means to open a Closed device for debug purposes.

Therefore, when the device is in closed state, JTAG use is restricted unless a challenge/response protocol is successfully executed. If the chip is closed, normal JTAG does not work anymore and a signed message must be delivered through ADM Figure 1.

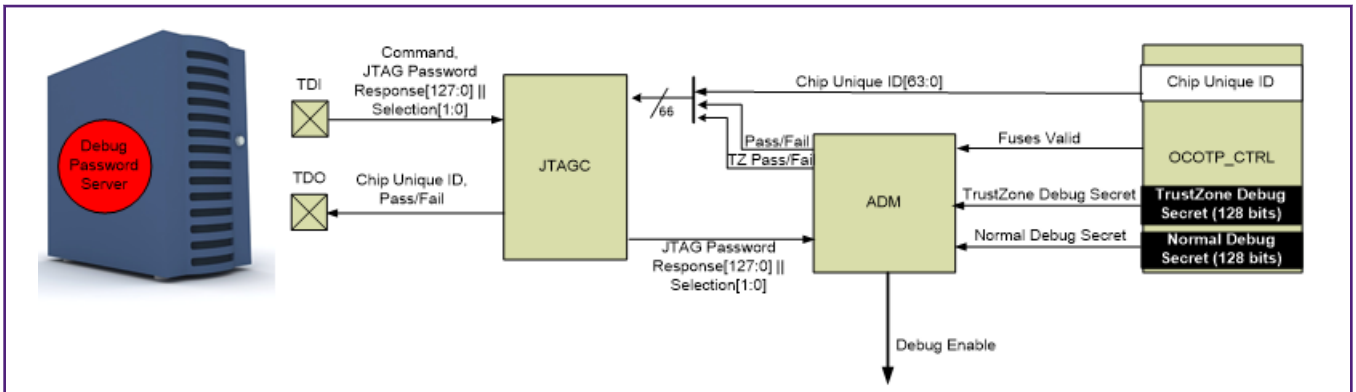

Figure 1. App Cores TrustZone and Normal World Debugging

The debug flow is as follows:

1. User request debug through JTAG interface
2. SOC responds with chip unique ID
3. Server finds corresponding secret (Trust Zone or Normal Debug)
4. User submits secret through JTAG interface
5. Secure JTAG module compares secret to pre-configured secret
6. If a match, ADM asks SECO to clear sensitive data

Arm model is used to restrict debug access based on TZ/normal state. The OEM secret is used to enable debug for non-secure code (EL2 and below). Similarly, the TZ secret enables debug for secure code (EL3). Separating these worlds can cause some difficulties when debugging code that crosses these boundaries (that is, Linux -> ATF), so to get a full view both shall be enabled.

3 Deploy SCFW for OEM Open

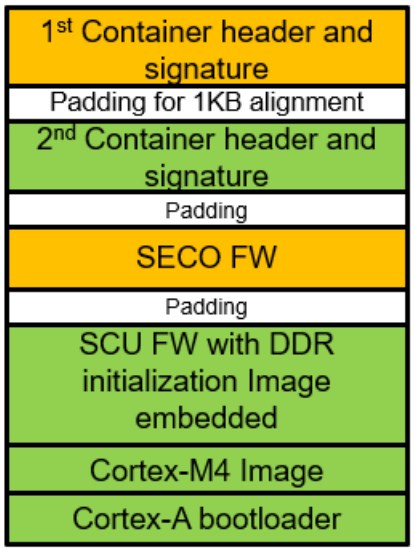

Figure 2. Boot image layout

The boot image has two containers: SECO FW (AHAB) and

SCFW+ATF+U-Boot as shown in Figure 2

1. Headers come first
2. The first container only contains the SECO
3. firmware image
4. Second container header aligned to 1k
5. Flexible image placement for the second container
6. which can contain one or multiple images
7. No CSF

For SECO to be authenticated with success the boot image must be signed using CST.

3.1 Connect Lauterbach to the board

After creating the image, you need to connect Lauterbach hardware to the board by following the steps below:

1. Disconnect the debug cable from the target while the target power is off
2. Connect the host system, the TRACE32 hardware and the debug cable
3. Power ON the TRACE32 hardware
4. Start the TRACE32 software to load the debugger firmware.
5. Connect the debug cable to the target.
6. Switch the target power ON.
7. Configure your debugger, e.g., via a start-up script.

Next, add cores and start the debugger:

1. Add cores on which to run the debugger
2. Change the Temp file location (Advanced Settings->Paths)

3. Select the Arm core, and press start Figure 3

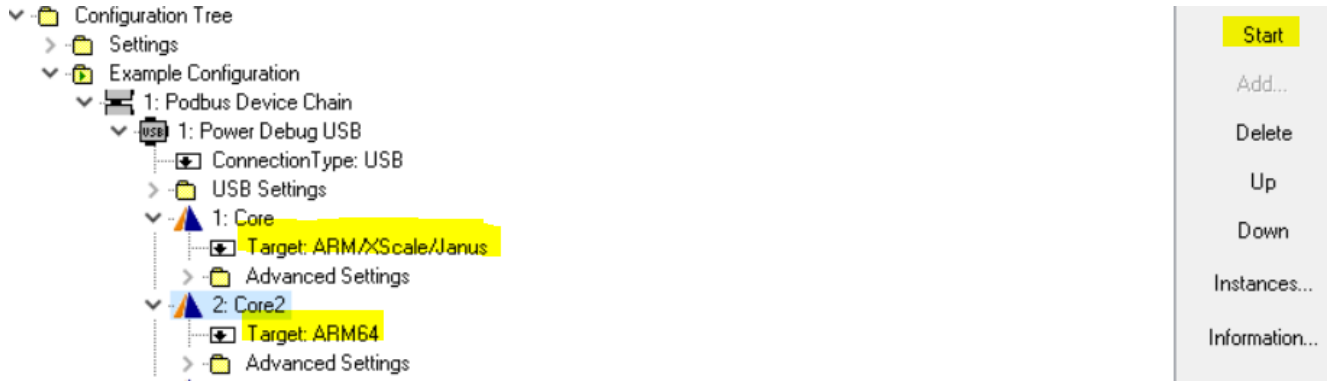

Figure 3. Add Cores and start the debugger

3.2 SCFW Debugging

Lauterbach scripts can be found either in the Lauterbach release or by contacting NXP support. For SCFW Debugging the *MX8_misc/coresight-jtag-provision.cmm* script can be used. The *&binFile* and *&scfwFile* script variables should be set with the paths to *flash.signed.bin* and *scfw_tcm.elf*. In order to link the disassembled code to the C code, you need to set &scfwPath variable with the path to SCFW source code.

Run SCFW Debugging script:

1. Put the board in SDP Mode
2. Power on the board, start the ARM-core and run the *MX8_misc/coresight-jtag-provision.cmm* script Figure 4.
3. If the script succeeds, “SECO AUTH SUCCESS” is reported in the AREA window Figure 5.

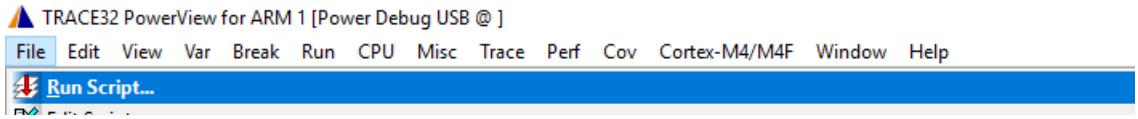

Figure 4. Run script

```
file 'C:\Users\nxf52873\Documents\T32\scripts_Lauterbach\flash.signed.bin' (Binary) loaded. 256 byte
file 'C:\Users\nxf52873\Documents\T32\scripts_Lauterbach\flash.signed.bin' (Binary) loaded. 23808 by
SECO AUTH SUCCESS
file 'C:\Users\nxf52873\Documents\T32\scripts_Lauterbach\flash.signed.bin' (Binary) loaded. 256 byte
loading ELF/DWARF
file 'C:\Users\nxf52873\Documents\T32\scripts_Lauterbach\scfw_tcm.elf' (ELF/DWARF2) loaded.
(*(((sc_boot_data2_t *) 0x2001fd00))).bt_dev_info.dev_type = 14 = SC_BT_DEV_TYPE_USB
```

Figure 5. Script log

3.3 Run SCFW

In the B:: console use “list board_init”. If you are seeing only disassembled code the path to the current source file (*board.c*) must be added from View->Symbols. Add a breakpoint at *board_init* from Break -> Set or press right click on the line you want to set it and select breakpoint. To start debugging press the go button Figure 6.

Figure 6. Menu bar

The debugger should be at the beginning of the board_init function. Figure 7. You can advance with the single-step button.

If you have built the SCFW with the M=1 option, SCU Console was activated (see iMX8 User Guide Manual). Logs can be seen in the SCU console while advancing with the go/step button. If the break button is pressed Figure 8, the execution is halted and the terminals should not accept any input.

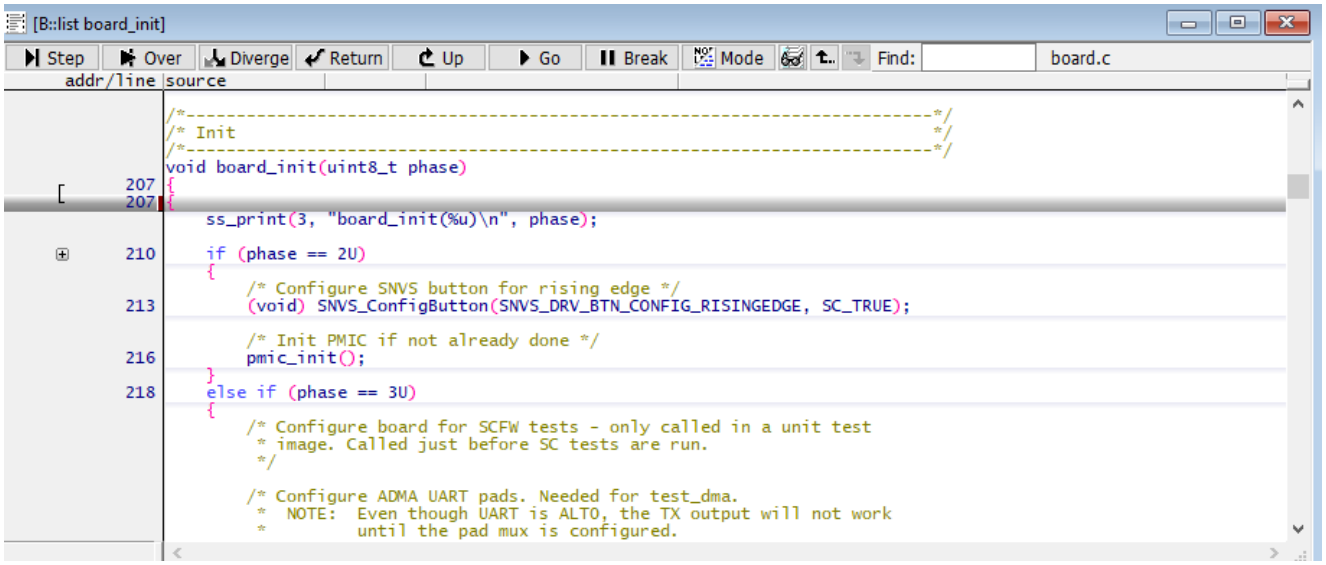

Figure 7. board_init function

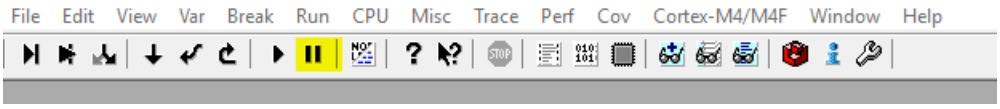

Figure 8. Break button

3.4 Attach to SCFW

For attaching to SCFW you need to write the previous image on the SD card. For i.MX8QXP devices, this can be done using the following command:

```
dd if=flash.signed.bin of=/dev/sde bs=1K seek=32;
```

Set the switches for SD1 boot (1100) and insert the SD-card. The U-boot has to be stopped from console terminal before booting. On the Arm-core, the *coresight-scu.cmm* script is used to attach to SCFW.

4 Debug U-boot

4.1 Attach to U-Boot

Adding a branch to self at the start label in U-Boot sources and recompiling the imx-boot is needed as in section 3.

For 2018.03r0 U-Boot version it can be found in *$YOCTO_BUILD/tmp/work/imx8-poky-linux/u-boot-imx/2018.03-r0/git/arch/arm/cpu/armv8/start.V* Figure 9.

```
*
* Startup Code (reset vector)
*
*************************************************

.globl  _start
_start:
    b start
```

Figure 9. u-Boot start.v

Write the U-Boot on the SD-card, set the switches for SD1 boot (1100) and insert the SD-card. In Lauterbach start the ARM64 core Figure 10.

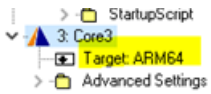

Figure 10. Start core

Next, run the *coresight-ca35.cmm* script to attach to U-Boot.

To load the elf file in Lauterbach, you need to run:

```
data.load.elf $PATH_TO_u_boot /NoCODE
```

To add sources to the disassembled code, you need the “U-Boot” file.

For iMX8QX, you can find it in *$YOCTO_BUILD/tmp/work/imx8qxpmek-poky-linux/u-boot-imx/2018.03r0/build/imx8qxp_mek_config/.*

For attaching the source code, the following command can be used:

```
y.SourcePATH $PATH_TO_u-boot-imx
```

The elf can be loaded with the sources simultaneously with:

```
data.load.elf $PATH_TO_u_boot /Strippart 4 /NoCode /PATH $PATH_TO_u-boot-imx
```

To see the code, go to View->List Source and move the PC to the next line by pressing right click “*b reset*” and Set PC Here to jump over the loop Figure 11.

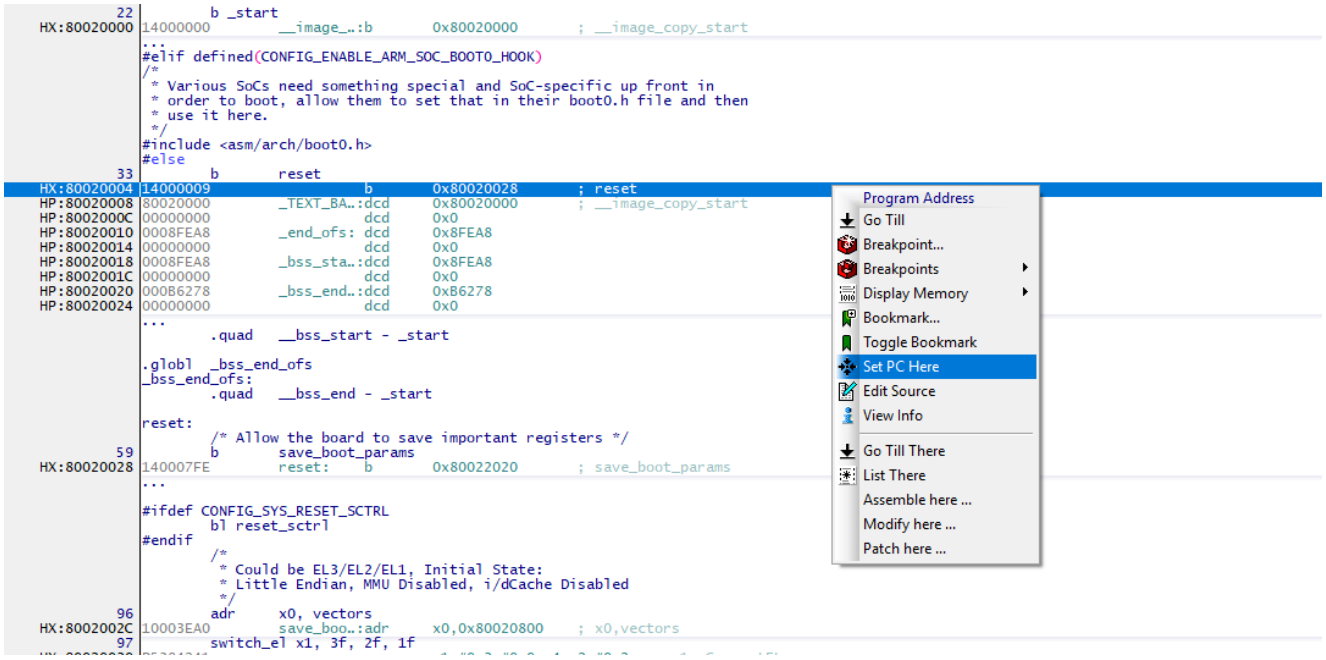

Figure 11. Set PC

The U-boot needs to relocate at a specific address in RAM. For that it uses *relocate_code* and *relocate_done* functions. Set two breakpoints at *relocate_code* and *relocate_done* Figure 12) and press GO until you reach *relocate_done* Figure 13.

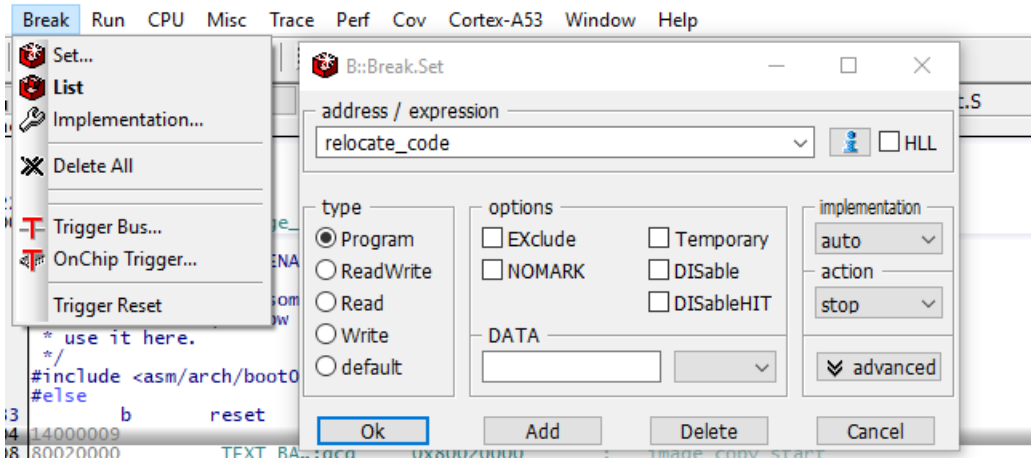

Figure 12. Set breakpoints

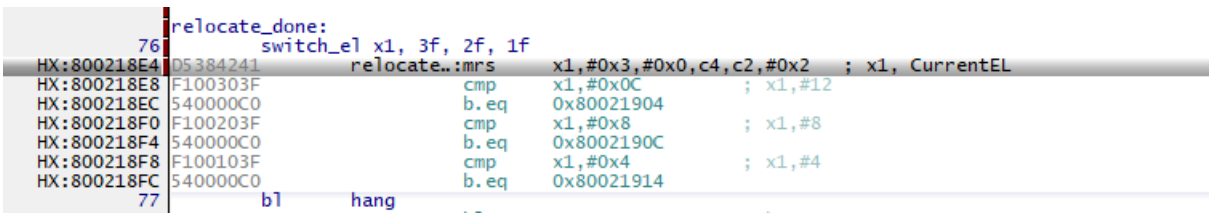

Figure 13. Old rellocate_done adress

In Figure 14 you can see the relocate_done address highlighted. Run some steps until you reach the 2nd stage and there is no source file mapping synchronization.

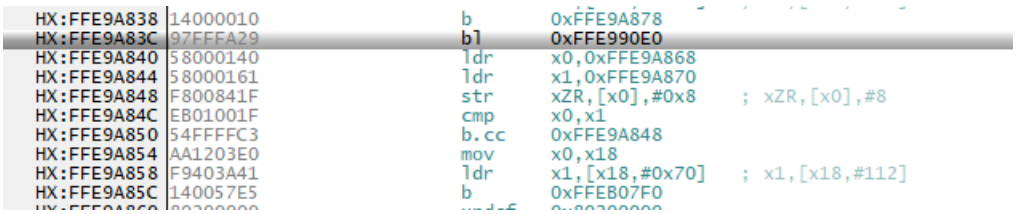

Figure 14. New relocate_done address

The offset is the difference between the new *relocate_done* address and the old one. In this case

Offset = 0xFFE9A838 – 0x800218E4 = 0x7FE78F54.

Now you need to load the elf file with the new offset. You can do that using:

```
B::data.load.elf C:\Users\nxf52873\Documents\T32\iMX8QXP\u-boot-loop\u-boot 0x7FE78F54 /NoCODE
```

Figure 15. Offset address

5 Kernel Debug

For Kernel Debugging the linux image must be compiled with debug symbols. Enter in kernel menuconfig Figure 16 and:

- disable General Setup -> unset “Compile drivers which will not load” (without this step the option below donot appear in menuconfig)
- enable In “Kernel Hacking -> Compile-time checks and compiler options” -> “Compile the kernel with debug info” -> Save to default -> Exit

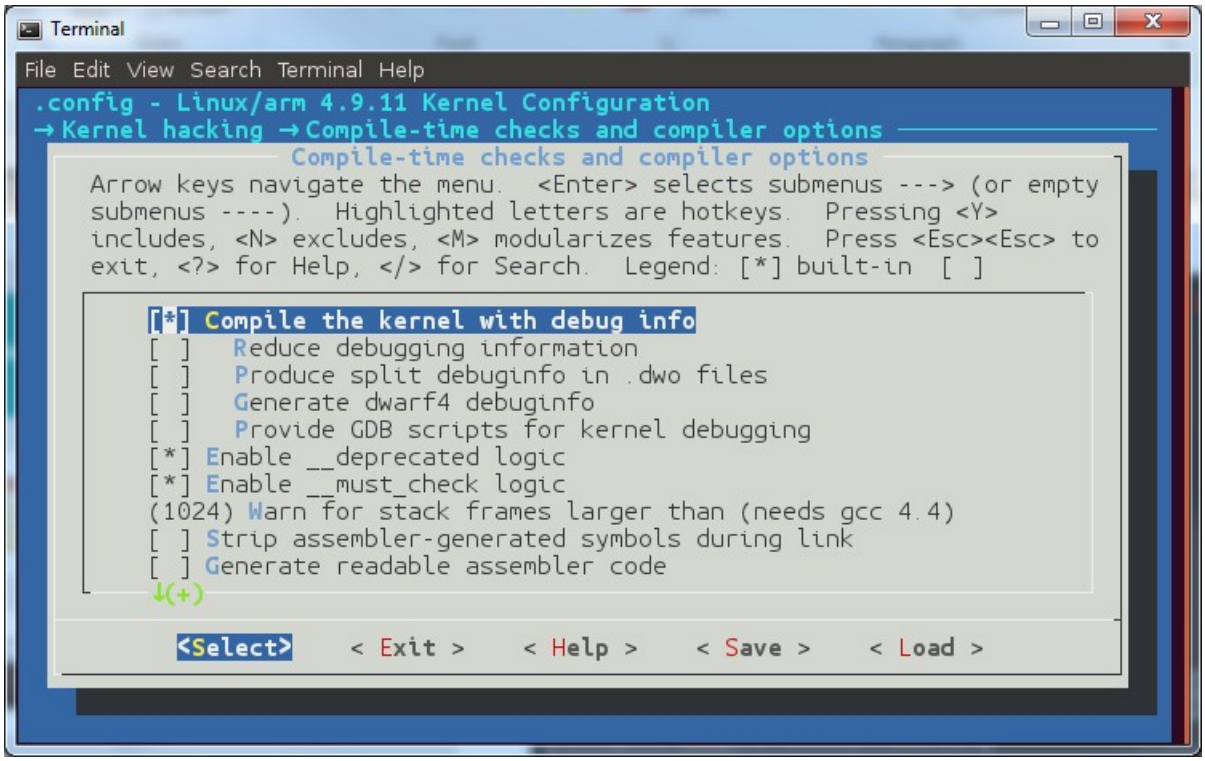

Figure 16. Enable debug info

The U-Boot must be suspended in serial console to prevent kernel booting up. The address should be around *relocate_done*. Attach with Lauterbach using *coresight-ca35.cmm* and follow these steps:

1. Set a hardware breakpoint to kernel load address Figure 17

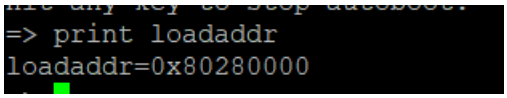

Figure 17. Print loadaddr

2. Set a hardware breakpoint at *start_kernel* and resume execution.
3. Load the elf using data.load.elf $PATH/vmlinux /Locateat $loadaddress /NoCode
4. Set hardware breakpoint at *__enable_mmu* Figure 18
5. Resume the execution
6. In U-Boot console type boot for loading the kernel. The first breakpoint should be hit Figure 19

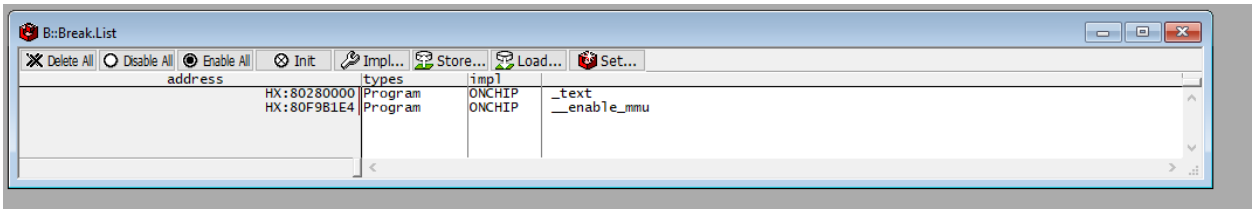

Figure 18. Breakpoint list

```
 */
ENTRY(__enable_mmu)
        mrs     x1, ID_AA64MMFR0_EL1
        ubfx    x2, x1, #ID_AA64MMFR0_TGRAN_SHIFT, 4
        cmp     x2, #ID_AA64MMFR0_TGRAN_SUPPORTED
        b.ne    __no_granule_support
        update_early_cpu_boot_status 0, x1, x2
        adrp    x1, idmap_pg_dir
        adrp    x2, swapper_pg_dir
        msr     ttbr0_el1, x1                   // load TTBR0
        msr     ttbr1_el1, x2                   // load TTBR1
        isb
        msr     sctlr_el1, x0
        isb
        /*
         * Invalidate the local I-cache so that any instructions fetched
         * speculatively from the PoC are discarded, since they may have
         * been dynamically patched at the PoU.
         */
        ic      iallu
        dsb     nsh
        isb
        ret
```

Figure 19. First breakpoint

Resume the execution. The breakpoint from *enable_mmu* should be hit (Figure 20). The following fragment of code is executed with the MMU on in MMU mode. You can attach sources to disassembled code by using:

```
data.load.elf $PATH_to_vmlinux /Locateat $loadaddress /NoCode /Strippart 4/PATH $PATH_TO_linux_imx
```

The kernel entry point is at *start_kernel*. The elf file needs to be reloaded using:

```
data.load.elf $PATH_to_vmlinux /NoCode /Strippart 4 /PATH $PATH_TO_linux_imx
```

```
         */
        add     x13, x18, #0x16
        b       stext
#else
        b       stext                           // branch to kernel start, magic
        .long   0                               // reserved
#endif
        le64sym _kernel_offset_le               // Image load offset from start of RAM, little-endian
        le64sym _kernel_size_le                 // Effective size of kernel image, little-endian
        le64sym _kernel_flags_le                // Informative flags, little-endian
        .quad   0                               // reserved
        .quad   0                               // reserved
        .quad   0                               // reserved
        .ascii  "ARM\x64"                       // Magic number
#ifdef CONFIG_EFI
        .long   pe_header - _head               // Offset to the PE header.

pe_header:
        __EFI_PE_HEADER
#else
        .long   0                               // reserved
#endif
```

Figure 20. Code fragment

If there are any problems in loading the sources, the /Strippart parameter should be adjusted.

6 Secure Debug

6.1 Introduction

The Challenge eFuse Table 2 has double roles: Secure JTAG Challenge and Unique ID (64 bits). It is used by both Normal/ TrustZone environments.

Table 2. Challenge eFuse

INDEX
(SJC_CHALL[7:0] / UNIQUE_ID[7:0])
(SJC_CHALL[15:8] / UNIQUE_ID[15:8])
(SJC_CHALL[23:16] / UNIQUE_ID[23:16])
(SJC_CHALL[31:24] / UNIQUE_ID[31:24])
(SJC_CHALL[39:32] / UNIQUE_ID[39:32])
(SJC_CHALL/UNIQUE_ID[46:40])
(SJC_CHALL[47:43] / UNIQUE_ID[47:43])
(SJC_CHALL[55:48] / UNIQUE_ID[55:48])
(SJC_CHALL[63:56] / UNIQUE_ID[63:56])

The Challenge eFuse can be read using the **fuse read** command as in Figure 21:

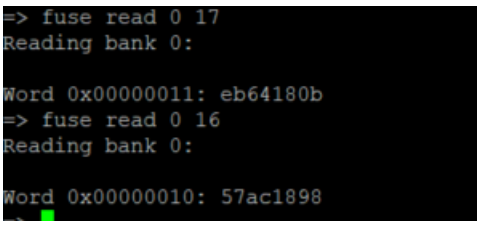

Figure 21. Read eFuse

The corresponding Challenge key is obtained by concatenating the fuses in: *0xeb64180b57ac1898*.

There are two response keys:

- OEM key for Normal world
- TrustZone key for Secure world

If the device is NXP closed (OEM Open), then you can use normal debug for debugging the A cores (U-Boot, Linux, SCFW). Once it becomes OEM closed then you need the challenge/response fuses programmed and the challenge/response function becomes active.

NOTE

The response eFuses are writable only from LC OEM Open. After switching to LC OEM Closed, these eFuses are locked out.

The OEM and TrustZone keys are formed as the Challenge Key from fuses and by default are 0x0. To modify the keys, you need to use: **fuse prog [-y] <bank> <word> <hexval>** command in the U-Boot terminal or **fuse.w <word><hexval>** from SCU firmware. Below is an example of writing the OEM and TrustZone keys from U-Boot.

```
TrustZone KEY [127:0]
  fuse prog 0 704 0xcafecafe
  fuse prog 0 705 0xbabeface
  fuse prog 0 706 0xc001d00d
```

```
  fuse prog 0 707 0x12345678
//secure secret as you can see it in Lauterbach scripts
  &response_secret1= 0xBABEFACECAFECAFE
  &response_secret2= 0x12345678C001D00D
OEM Key [127:0]
  fuse prog 0 722 0xca11b005
  fuse prog 0 723 0xd00dca11
  fuse prog 0 724 0xdeadbeef
  fuse prog 0 725 0xabcdefda
   // normal secret 128 bit as you can see it in Lauterbach scripts
  &response_normal1= 0xD00DCA11CA11B005
  &response_normal2= 0xABCDEFDADEADBEEF
```

6.2 Steps to connect Lauterbach debug tool via Secure Debug

The following steps connect the Lauterbach debug tool to the i.MX8/8X family when using Secure JTAG:

Table 3. Fuse map for OEM key and TrustZone key

Fuse Row Index	7	6	5	4	3	2	1	0
722-725	OEM Key [127:0]							
704-707	TrustZone KEY [127:0]							

1. Read the OEM Key and the TrustZone Key from fuses using the "*fuse read*" command in u-boot terminal.
2. In the iMX8_misc folder you can find the *jtag_attach.cmm* script. This script uses *jtagc_calculateresponse.cmm*, where you need to set the normal_response and trustzone_response variables.
3. Run *jtag_attach.cmm* from Lauterbach. If the keys correspond to the expected values, you should see the challenge.

```
B::
Challenge status : 0x2 (=0 : fail, =1: Normal challenge ok, =2 :TZ challenge ok)
```

Figure 22. Challenge status

4. To attach to the ARM64 core, coresight_ca35.cm can be used. In OEM Closed state the line from Figure 23 has to be commented.

```
; DO ~~~~/csgpr CPU=imx8-a35
DO ~~~~/coresight CPU=imx8-a35
```

Figure 23. Attach to Arm64 core

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

For more information, please visit: http://www.nxp.com

For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: February 2020

Document identifier: AN12631

arm