NXP Semiconductors Document Number: LPOSRS003SUG

Solution User's Guide Rev. 2, 04/2018

Linux Point of Sale (LPOS) Reader Solution User's Guide

Contents

1. Introduction

The Linux Point of Sale (LPOS) Reader Solution is a collection of hardware, software enablement, middleware and specialized segment-specific software for the point-of-sale market. This document provides detailed information about the hardware, software building blocks and examples provided with the solution as well as a blueprint for developing custom POS applications based on this software.

2. Hardware Overview

The Linux POS Reader Solution is fully assembled and loaded with software at the factory. This section describes each piece of the solution and its configuration settings. The fully assembled hardware is pictured below in Figure 1.

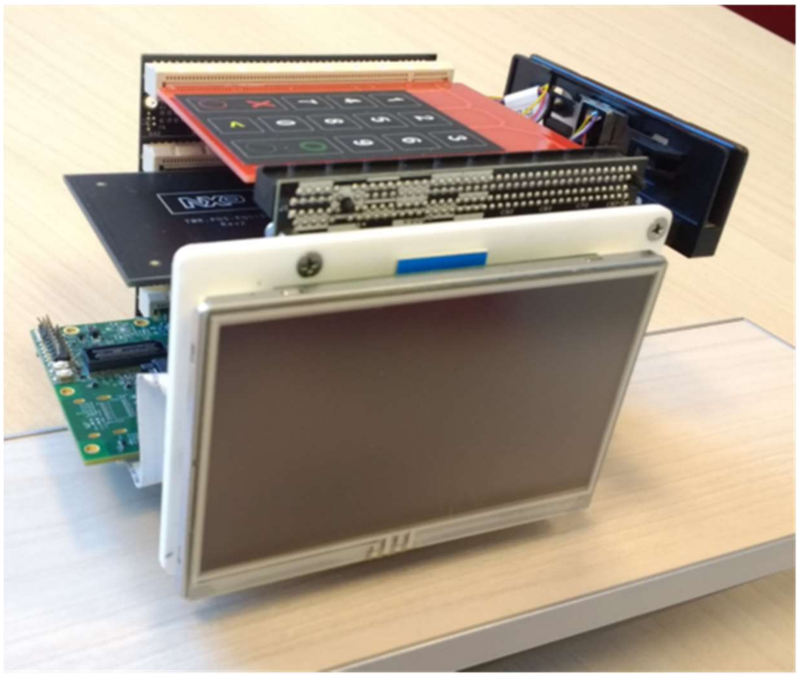


Figure 1. Assembled TWR-LPOS-RDR Solution

TWR-POS-i.MX6UL

The primary board of the Linux POS Reader Solution is the TWR-POS-i.MX6UL3. It’s a Tower module featuring the MCIMX6G3CVM05AB – an ARM® Cortex®-A7 @ 528 MHz core (with TrustZone and NEON MPE), Boot ROM (HAB, 96KB), OCRAM 128KB, Secure RAM 32 KB, 512MB DDR3L, Dual Quad SPI controller, 2x MMC 4.5/SD 3.0/SDIO Port, 2x USB 2.0 OTG, TRNG, Crypto Engine (AES with DPA, TDES/SHA/RSA), Tamper Monitor, Secure Boot, SIMV2/EVMSIM X 2, OTF DRAM Encryption, PCI4.x compliance profile.

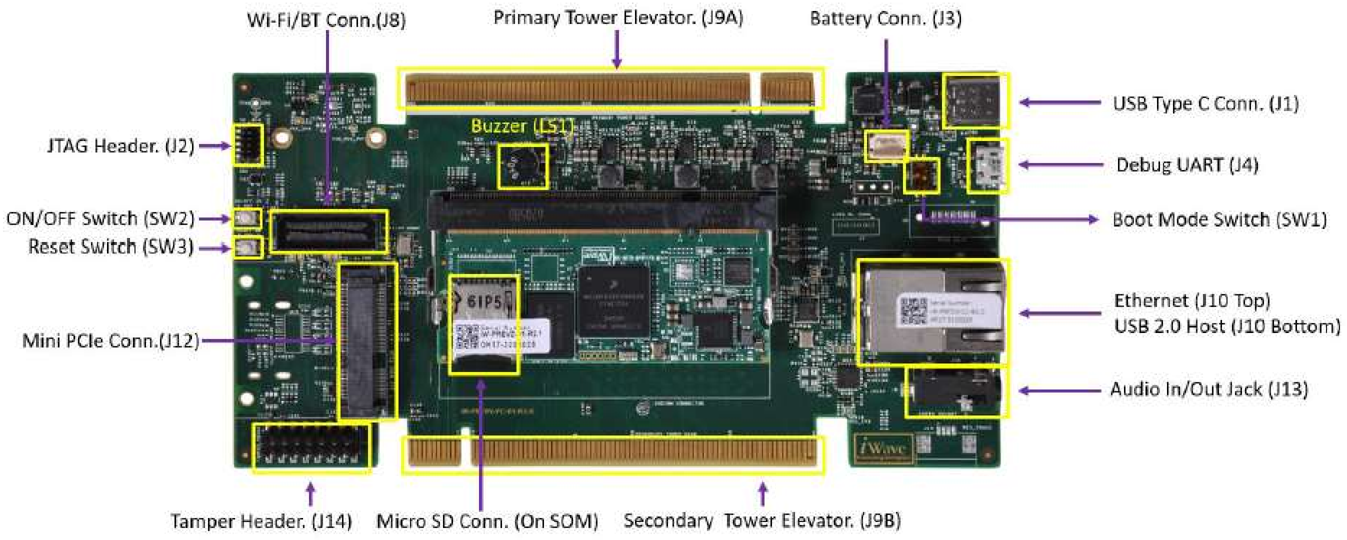


Figure 2. i.MX6UL-TWR-POS Board

Security architecture description for iMX6UL processor

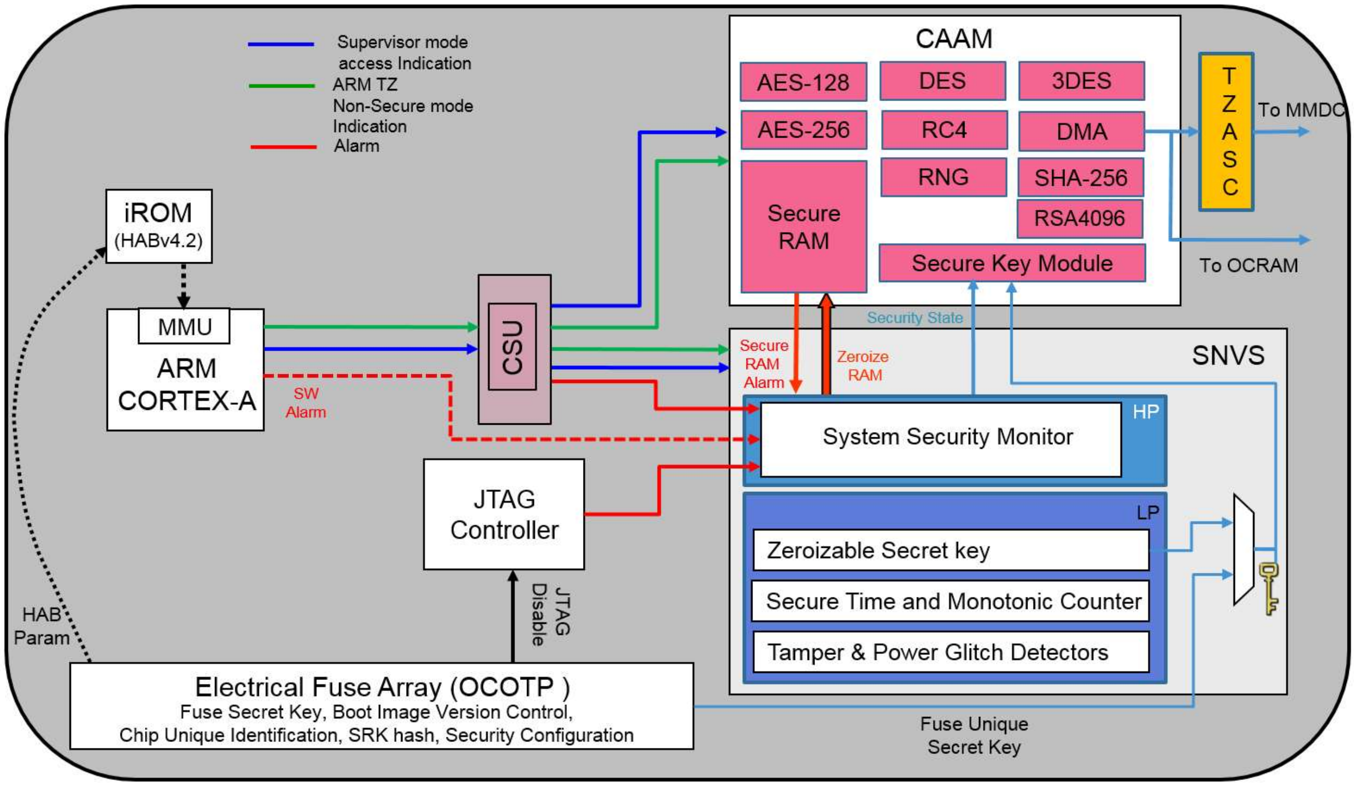
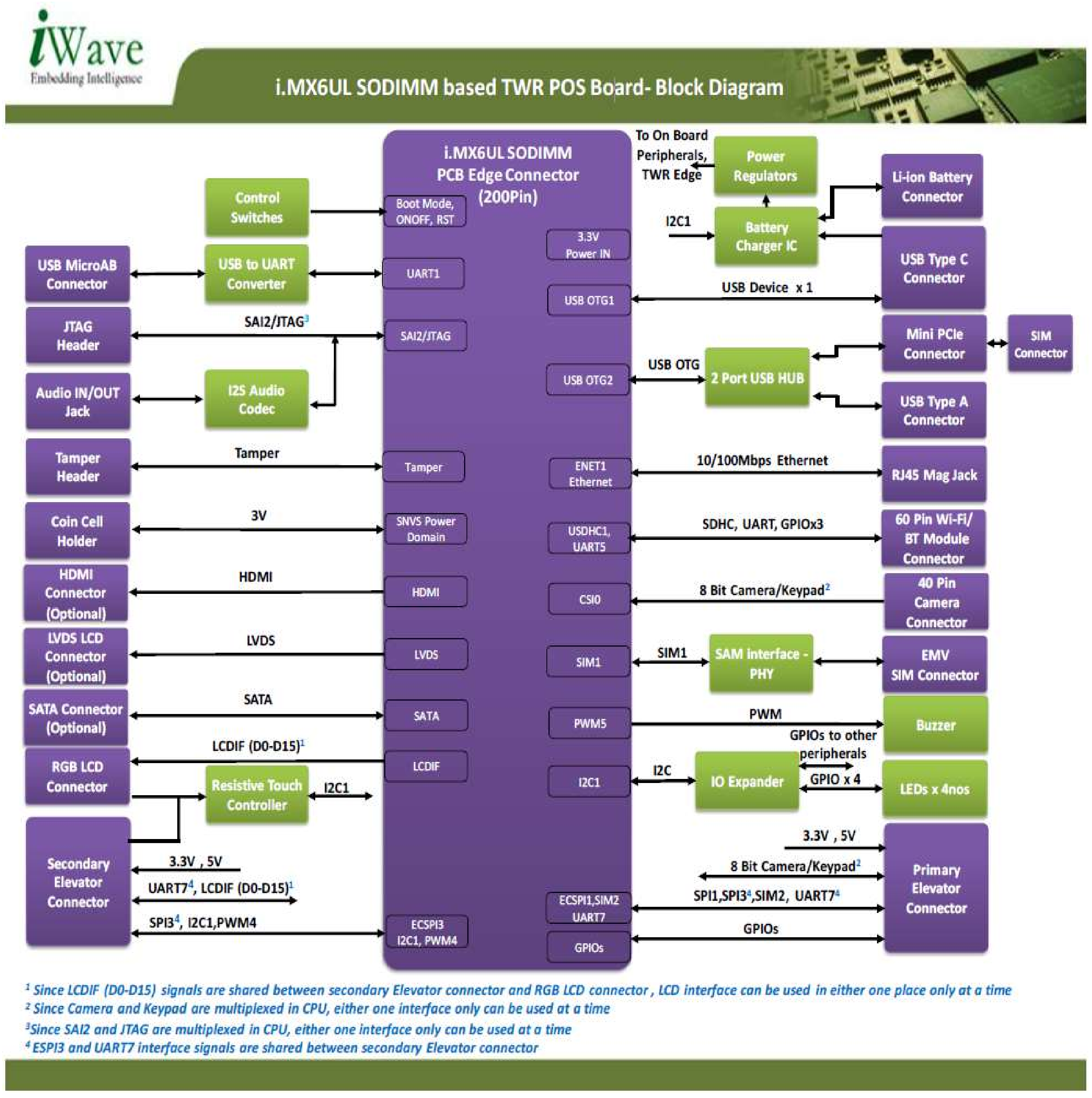


Figure 3. i.MX6UL-G3 Security Architecture

Application Processors by design are built to be open systems, so to keep the strong security needed, there are many more functions done at the HW level by highly capable peripherals.
The ARM TrustZone® architecture is utilized to add the layer of protection for sensitive SW functions. Beyond memory access, there are peripherals like the CSU (Central Security Unit) that controls access from other bus masters to peripheral devices.
There is a strong interaction between the secure key storage and the Cryptography hardware – to reduce the amount of SW needed to handle the keys in the system.
OCOTP – On-Chip One Time Programmable
CAAM – Cryptographic Accelerator and Assurance Module
SNVS – Secure Non-Volatile Storage

MMDC - Multi-mode DDR Controller

TWR-iMX (SOM Base board for iMX6)



[1] Since LCDIF (D0-D15) signals are shared between secondary Elevator connector and RGB LCD connector , LCD interface can be used in either one place only at a time
[2] Since Camera and Keypad are multiplexed in CPU, either one interface only can be used at a time
[3] Since SAI2 and JTAG are multiplexed in CPU, either one interface only can be used at a time
[4] ESPI3 and UART7 interface signals are shared between secondary Elevator connector

Figure 4. i.MX6UL-TWR-POS Block Diagram

2.3.1. System Power

TWR-POS-i.MX6UL board can be powered through 5V,2A USB Type C Charger or 3.7V,2200mAH Li-Ion Battery. Please follow the below procedure to power ON the TWR POS board.

- Connect the 5V USB Type C charger plug to the USB Type C connector (J1) of the TWR-POS-i.MX6UL board as shown below.

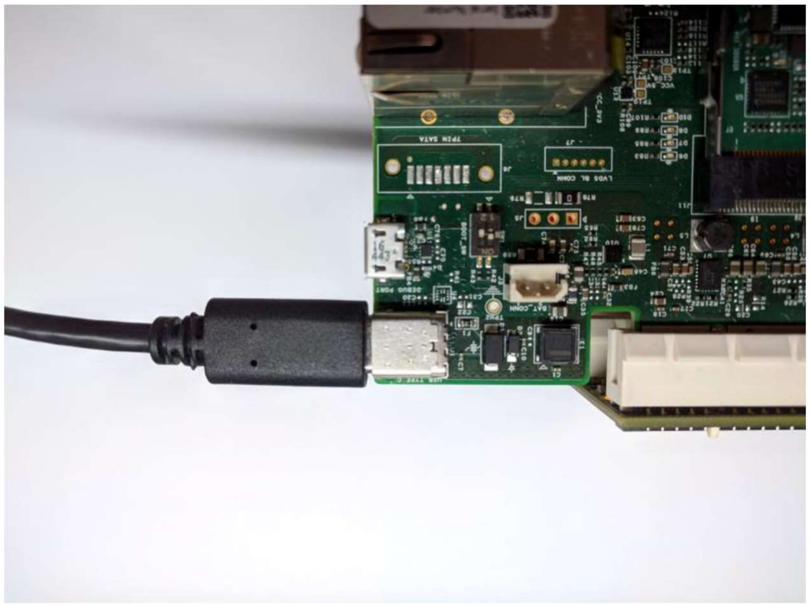


Figure 5. USB Type-C connected to the TWR-POS- i.MX6UL board

- Once power is applied to the TWR POS board, the power status LEDs will glow as shown below.

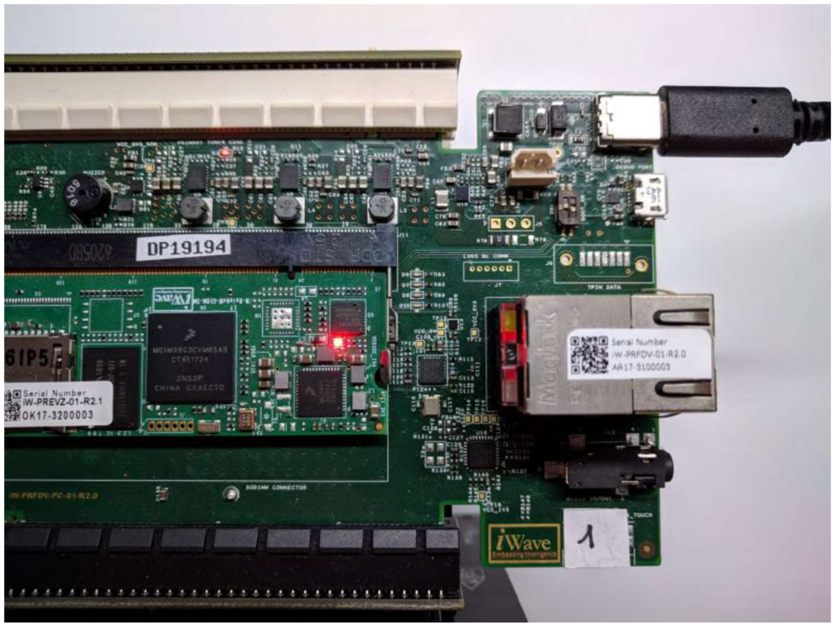


Figure 6. TWR-POS- i.MX6UL board powered on

2.3.2. Secure Real-Time Clock

SNVS/RTC low power domain - The SRTC domain contains only counter, comparator and compared data of the on-chip RTC. This domain should be supplied from an external single cell Li-ion battery and/or an external pre-regulated power supply.

NOTE

For the POS Reader Solution, the battery must be installed for the software to run correctly.

2.3.3. Tamper Header

On the standard version of the TWR-POS-i.MX6UL board the tamper signals are routed to a header (J14). Jumpers installed on the header make connections for the ten active tamper pairs. The jumpers can be removed to test an active tamper event. The jumpers can also be removed so that some or all of the tamper signals can be connected to GND or VDD to test passive tamper detection.

2.3.4. Display

The iMX6UL includes an eLCDIF display controller supporting one parallel 24-bit LCD display with up to WXGA (1366x768) resolution at 60 Hz.
The display included with the Linux Card Reader system is a 16bit color 480x372 resolution display.

The display is connected via a ribbon cable from the display board to the iMX6UL-TWR-POS.

2.3.5. Expansion cards

iMX6UL chip enables up to Four MMC/SD/SDIO card ports all supporting:

- 1-bit or 4-bit transfer mode specifications for SD and SDIO cards up to UHS-ISDR-104 mode (104 MB/s max)
- 1-bit, 4-bit, or 8-bit transfer mode specifications for MMC cards up to 52 MHz in both SDR and DDR modes (104 MB/s max)
- 4-bit or 8-bit transfer mode specification for eMMC chips up to 200 MHz in HS200 mode (200MB/s max).

2.3.6. Buzzer, Pushbuttons, LEDs

The i.MX6UL-TWR-POS features:ssss

- A piezo buzzer connected to PWM5
- IO expander connected to I2C1 will be used to configure GPIO for LED's and other peripherals

2.3.7. USB interface - AICI

Two High Speed (HS) USB 2.0 OTG (Up to 480 Mbps), with integrated HS USB Transceiver PHY.

The TWR-POS-i.MX6UL provides USB Device capability for connection to the IHS tool via the USB Type-C connector.

2.3.8. Re-Work / Modifications

None

3. Software Overview

NXP provides a full-featured POS software solution for customers that utilizes a modular and layered approach. Such an implementation offers the end user flexibility when deciding how to implement their application. This section describes the software architecture in detail and highlights each piece provided in the package.

Software Layers

In the POS market, EMVCo dictates how applications should be architected to meet the strict security and compatibility requirements needed for payment systems. The POS Reader Solution software is designed to comply with this standard and is layered using the same nomenclature used by EMVCo: **Level 1 (L1)**, **Level 2 (L2)** and the payment application (sometimes referred to as Level 3 or L3). The image below shows the layering model used by the software along with the various software building blocks provided by NXP.

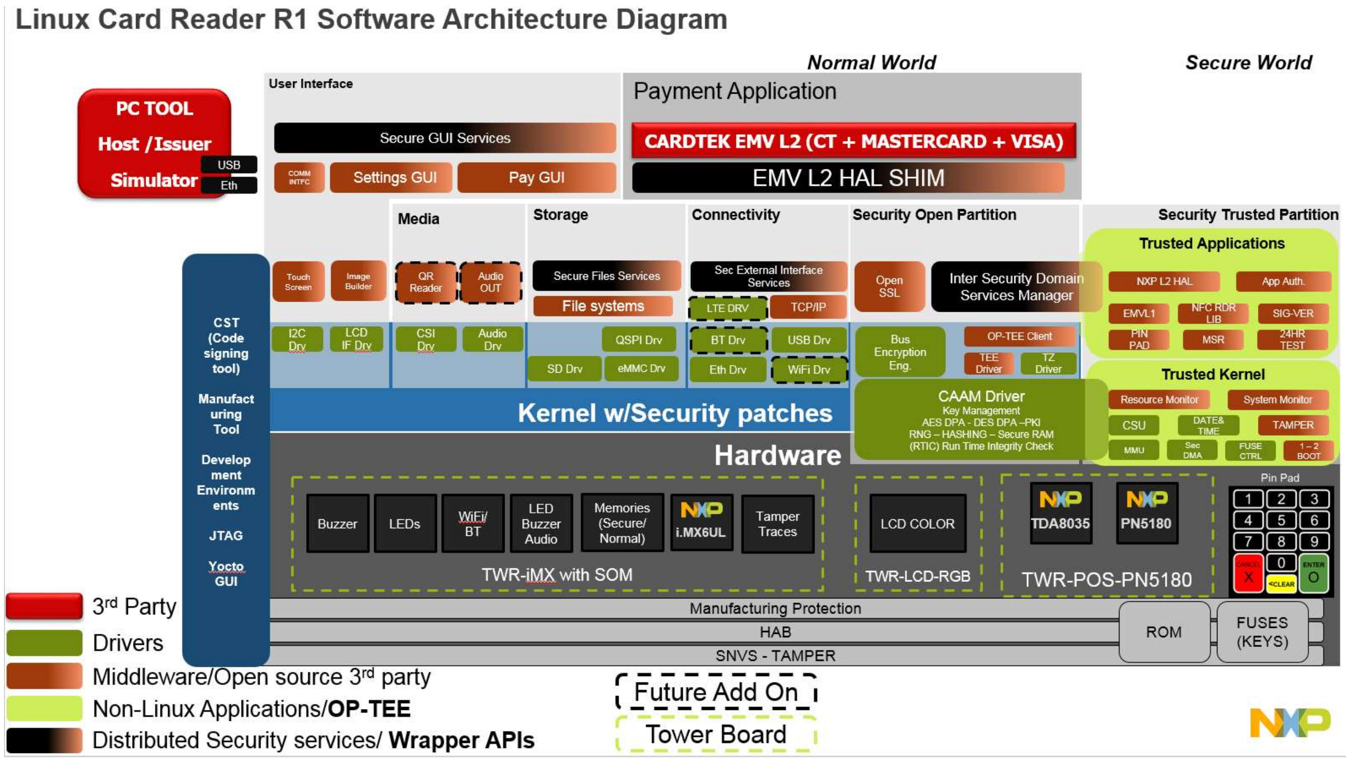


Figure 7. Software Layering Diagram

Starting from the bottom of Figure 7, you see that the Linux POS Reader Solution software is based on the L4.1.15_2.0.0_ga release from NXP and customized by iWave for TWR-POS-i.MX6UL board. The BSP is the standard base enablement provided by NXP for iMX6UL MPU and provides a comprehensive and robust set of peripheral drivers and middleware for the Linux OS.
An important software component for the Linux POS solution is the OPTEE (Open Platform Trusted Execution Environment) OS implementation, running on a trusted partition as a secure service provider for the payment operations. This security mechanism is enforced in hardware by the ARM TrustZone security extension and other NXP hardware peripherals (CSU, SNVS). Most of the secure hardware components (PINPAD, PN5180, Tamper, RTC, SNVS, LCD – shared) used for operations are implemented as device drivers only in the OPTEE secure OS. Besides these, there are specific software components which authenticate the clients making secure requests from normal world and validate them.

Lastly, we have third party code. In the POS Reader Solution, NXP has partnered with Cardtek for the EMVCo L2 implementation. The third party pieces are provided as evaluation software in library/binary format. If you are interested in seeking further information on either of these items, please contact our partners directly for licensing details.

OPTEE Architecture

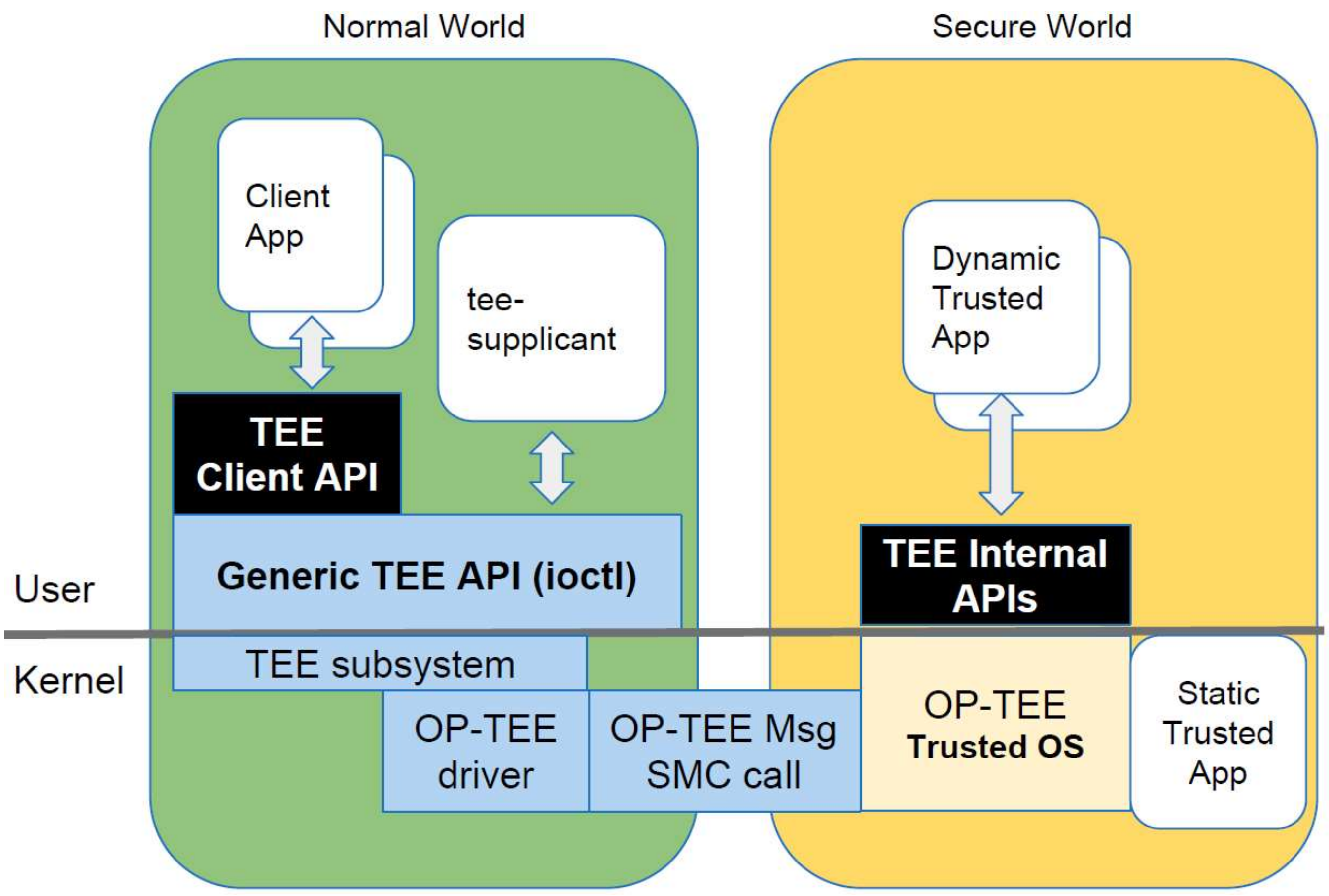


Figure 8. OPTEE architecture

OPTEE consists of three components:

- OP-TEE Client, which is the client API running in normal world user space.
- OP-TEE Linux Kernel driver, which is the driver that handles the communication between normal world user space and secure world.
- OP-TEE Trusted OS, which is the Trusted OS running in secure world.

3.2.1. Pseudo Trusted Applications

The Pseudo Trusted Applications included in OP-TEE operate at the OP-TEE secure privileged level. They provide services hidden behind a "GlobalPlatform TA Client" API. These pseudo-TAs are used for various purposes such as specific secure services or embedded tests services. As an example, many secure peripheral drivers have been integrated as PTAs in the Linux Card Reader.

Pseudo TAs do not benefit from the GlobalPlatform Core Internal API support specified by the GlobalPlatform TEE specs. These APIs are provided to TAs as a static library. Each TA shall link against (the "libutee") and that calls OP-TEE core service through system calls. As the OP-TEE core does not link with libutee, Pseudo TAs can only use the OP-TEE core internal APIs and routines.

Pseudo Trusted Applications (PTAs) are implemented directly in the OP-TEE core and are located in , `core/arch/arm/pta`. They are built statically into the OP-TEE core blob.

As pseudo TAs have the same privileged execution level as the OP-TEE core code itself, such situation may not be desirable for complex TAs.

In most cases an unprivileged (Secure User Mode) TA is the best choice instead of adding your code directly to the OP-TEE core. However, if you decide your application is best handled directly in OP-TEE core, you can look at `core/arch/arm/pta/stats.c` as a template. Just add your pseudo TA based on that to the `sub.mk` in the same directory.

3.2.2. User Mode Trusted Applications

Secure User Mode Trusted Applications are loaded (mapped into memory) by OP-TEE core in the Secure World when components in the REE requires a service from a particular trusted application designated by a UUID. They run at a lower CPU privilege level than OP-TEE core components. In that respect, they are quite like regular applications running in the Rich Execution Environment (REE), except that they execute in Secure World.

Trusted Applications (TAs) benefit from the GlobalPlatform Core Internal API as specified by the GlobalPlatform TEE specifications.

NXP Modules for LPOS Applications

NXP has created a set of modular software blocks that can be used together or separately in custom POS systems. Users have the choice of whether they want to use all or none of these modules depending on their specific needs.

The yellow-green blocks in Figure 7 show the pieces created by NXP specifically for the POS Reader Solution. This section describes each of these software modules and how they are used in typical POS applications. The full list of modules, located in the /pos/modules folder, includes:

- Host Communication Interface (COMMIF)

- Filesystem (FILE)
- Personal Identification Number (PIN)
- User Interface (UI)

For detailed API documentation for LPOS Reader Solution software modules (provided by NXP, not third party), please refer to the *Point of Sale (LPOS) Reader Solution API Reference Manual.*

3.3.1. Host Communication Interface (COMMIF) Module

All payment devices need a mechanism to pass data to the payment network, whether the device uses online or offline transactions. In the LPOS Reader Solution, the COMMIF module is responsible for this activity and passes the information to the Issuer Host Simulator (IHS) tool.

The COMMIF module delivered with the solution uses the on-chip USB interface to pass data to a USB host. The CDC class (serial) was chosen since it’s simple to use and configure and doesn’t require a custom driver.

Abstracting the host interface from the application is the primary goal of each of NXP’s software modules, so the API provided to the application is very simple. The COMMIF module will be extended in the future to support additional interfaces and can be easily adapted by end users should they need a custom protocol.

The USB CDC implementation requires a task to process USB background events, so the COMMIF module does have its own task.

4. Build system setup

The Linux POS Reader project is based on the Linux 4.1.15 BSP (Board Support Package) provided by NXP. The Linux distribution is based on the Yocto build system which downloads the source code from the Internet and compiles the Linux distribution customized for the specific board hardware (e.g. CPU model).

For the Linux POS Reader project, the board (SoC) is based on the ARM v7 processor, more specifically the NXP IMX6UL processor. The build downloads approximately 2.5GB of sources and requires about 40GB of space for compilation. Numerous Yocto Recipes are grouped in the Linux Card Reader package.

Prerequisites

The recommended and tested Linux distributions for compiling the project are Ubuntu 14.64 or Ubuntu 16.04, either the 32-bit or the 64-bit versions.

4.1.1. The Git source code versioning tool

To clone the source code from the Internet and eventually make changes to the code, we need to install the "git" tool.

```
host@host~$ sudo apt install git
```

Now we need to configure "git" so that the next time you commit a change the Author will be set from the below settings.

```
host@host~$ git config --global user.name "John Doe"

host@host~$ git config --global user.email johndoe@example.com
```

4.1.2. Yocto build requirements

Next, we need to install the tools required by the Yocto build

```
host@host~$ sudo apt install texinfo gawk chrpath g++
```

Building the project

4.2.1. Extract the build installer tarball

The Linux POS software kit is packed in a tarball provided by NXP. Extract the tarball and change the current directory to the extracted folder

```
tar xzf LPOS_R1.0_b300.tar.gz

cd lpos-r1.0-b303
```

4.2.2. Initialize the source code repositories

To initialize the source code repositories run the following command

```
host@host~$ bash scripts/build-pos-reader.sh setup
```

This will create the Git repositories based on the public repositories and the Git bundles provided in the tarball. The Git bundles provide the NXP Git commits which are only available on the NXP private repositories.

The command fetches the public repositories from the Internet and applies the Git bundles on top, recreating the same Git history as on the NXP private repositories.

4.2.3. Build the Linux POS Reader

The LPOS project is built part of the Linux Base Support Package (BSP), hence the name of the argument "build-bsp".

```
host@host~$ bash scripts/build-pos-reader.sh build-bsp
```

This invokes the "bitbake" Yocto command with the "linux-pos-image" argument. It will first compile all the native tools (e.g. the cross compiler, python, etc) and then it will compile all the Yocto recipes, including the recipes which contain the LPOS packages.

4.2.4. Write the image to an SDCard

For Linux machines, after inserting the SDCard in the PC we must find which device was assigned to the SDCard.

```
host@host~$ sudo fdisk -l
```

This should provide a list of the devices and their corresponding size. For example, we are using an 8GB SDCard and part of the output is

```
Disk /dev/sdd: 7,4 GiB, 7948206080 bytes, 15523840 sectors

Units: sectors of 1 * 512 = 512 bytes

Sector size (logical/physical): 512 bytes / 512 bytes

I/O size (minimum/optimal): 512 bytes / 512 bytes

Disklabel type: dos

Disk identifier: 0xdc2bd234
```

The SDCard was assigned device node /dev/sdd. Next, we need to make sure the SDCard is not automatically mounted by Linux:

```
host@host~$ sudo umount /dev/sdd*
```

Write the LPOS image to the SDCard

```
host@host~$ sudo dd if=build/build-imx6ul-iwg18m-twr-pos/tmp/deploy/images/imx6ul-iwg18m-
    twr-pos/linux-pos-image-imx6ul-iwg18m-twr-pos.sdcard of=/dev/sdd bs=1M && sync
```

For Windows machines the "Win32DiskImager" tool can be downloaded from the Internet to write the image to an SDCard.

Code repositories and their Yocto recipes

4.3.1. linux-pos-reader

This is the main Linux POS Reader project repository. It provides the build-pos-reader.sh script to initialize the Yocto repo and to build the SDCard image.

The repository contains the code for the Payment Demo application (apps/payment-demo folder), the L2 HAL (pos/l2-hal and pos/modules folders) and the test suite (tests folder). There are multiple Yocto recipes that use this repository and at least the Payment Demo recipe and the L2 HAL recipe use this.

4.3.2. POS Payment Demo Application

The Payment Demo app links statically with the NXP L2 HAL library. The application is built from the following recipe:

build/sources/meta-fsl-miss/recipes-pos/pos-payment-demo/pos-payment-demo_1.0.bb

The location where Yocto downloads the source code

build/build-imx6ul-iwg18m-twr-pos/tmp/work/imx6ul_iwg18m_twr_pos-poky-linux-gnueabi/pos-l2-hal/1.0+gitAUTOINC+COMMIT/git

4.3.3. NXP L2 HAL

The NXP L2 HAL is split into two parts. One part consists of the L2 HAL and one the GUI application written in QT.

The L2 HAL recipe is located here:

build/sources/meta-fsl-miss/recipes-pos/pos-l2-hal/pos-l2-hal_1.0.bb

The location where Yocto downloads the source code

build-imx6ul-iwg18m-twr-pos/tmp/work/imx6ul_iwg18m_twr_pos-poky-linux-gnueabi/pos-l2-hal/1.0+gitAUTOINC+COMMIT/git

The QT application recipe is located here:

build/sources/meta-fsl-miss/recipes-qt5/pos-gui/pos-gui_1.0.bb

The location where Yocto downloads the source code

build-imx6ul-iwg18m-twr-pos/tmp/work/cortexa7hf-vfp-neon-poky-linux-gnueabi/pos-gui/1.0+gitAUTOINC+COMMIT/gui

4.3.4. The meta-fsl-miss Yocto layer

This is the main Yocto layer for the Linux POS Reader project. The MISS acronym stands for MICR IoT Security Solutions.

It contains the following recipes:

- POS Payment Demo
- POS NXP L2 HAL
- POS NXP L2 HAL GUI application
- OPTEE-OS
- OPTEE-Client
- OPTEE-POS-TA

It also contains the Linux POS Image (SDCard) recipe. The location of the recipe is:

build/sources/meta-fsl-miss/recipes-core/images/linux-pos-image.bb

The default image comes with 20MB extra disk space, while the development image comes with 100MB extra disk space.

Local Yocto output directory:

build/build-imx6ul-iwg18m-twr-pos/tmp/work/imx6ul_iwg18m_twr_pos-poky-linux-gnueabi/linux-pos-image/1.0-r0/rootfs/

It also contains the Yocto image recipe (classes/linux-pos-image-base.bbclass) which contains the "IMAGE_INSTALL_append" variable. This variable can be used to install additional tools/packages on the SDCard image.

4.3.5. OPTEE-OS

Recipe location:

build/sources/meta-fsl-arm-extra/recipes-bsp/u-boot/u-boot-iwg18_2016.03.bb

The location where Yocto downloads the source code

build-imx6ul-iwg18m-twr-pos/tmp/work/imx6ul_iwg18m_twr_pos-poky-linux-gnueabi/optee-os/1.0+gitAUTOINC+COMMIT/git

4.3.6. OPTEE POS Trusted Applications

Recipe location:

build/sources/meta-fsl-miss/recipes-optee/optee-pos-ta/optee-pos-ta_1.0.bb

Local Yocto source code copy

build/build-imx6ul-iwg18m-twr-pos/tmp/work/imx6ul_iwg18m_twr_pos-poky-linux-gnueabi/optee-pos-ta/1.0+gitAUTOINC+COMMIT/git

4.3.7. The fsl-arm-yocto-bsp Yocto repository

Main Yocto BSP repository. The "repo init" command downloads all the other Yocto layers for this project based on the entries in default.xml. We are using the "imx-4.1.15-1.0.0_ga" stable branch.

4.3.8. The meta-fsl-arm Yocto layer

We had to clone this layer (from github) in order to make change for OPTEE-OS to copy the OPTEE-OS binary (uTee) to the first partition of the SDCard (FAT32 partition). We are using the jethro-4.1.15-ga branch.

4.3.9. The meta-fsl-arm-extra Yocto layer

This Yocto layer contains the recipes for Linux kernel and U-boot and on top of these we have applied the changes from iWave. We are using the jethro-4.1.15-ga branch.

4.3.10. Linux kernel

The iWave build was provided on top of the “imx_4.1.15_2.0.0_ga-pos” branch at revision b63f3f52cb393e3287352cf63f0caef31a33ab63. We have forked this branch revision to “imx_4.1.15_2.0.0_ga-pos” branch.

Recipe location:

build/sources/meta-fsl-arm-extra/recipes-kernel/linux/linux-iwg18_4.1.15.bb

The location where Yocto downloads the source code

build/build-imx6ul-iwg18m-twr-pos/tmp/work/imx6ul_iwg18m_twr_pos-poky-linux-gnueabi/linux-iwg18/4.1.15-r0/git

4.3.11. U-Boot

Recipe location:

build/sources/meta-fsl-arm-extra/recipes-bsp/u-boot/u-boot-iwg18_2016.03.bb

The location where Yocto downloads the source code

build/build-imx6ul-iwg18m-twr-pos/tmp/work/imx6ul_iwg18m_twr_pos-poky-linux-gnueabi/u-boot-iwg18/2016.03-r0/git

Working with Yocto

4.4.1. The recipes

When the project is first set up by the “build-pos-reader.sh” script with the “setup” argument, the repositories are fetched in the “repos” folder. The following repositories consist of Yocto recipes:

- meta-fsl-arm
- meta-fsl-arm-extra
- meta-fsl-bsp-release
- meta-fsl-miss

When running the build with the “build-bsp” argument, the recipes are copied in the “build/sources” folder. If changes in the recipes are required then the developer must change the repository in “build/sources” (e.g. in build/sources/meta-fsl-miss/recipes-pos/pos-payment-demo/pos-payment-demo_1.0.bb).

4.4.2. Rebuilding a single Yocto recipe

Once the build process ends (after running the “build-bsp” stage of build-pos-reader.sh script) the build will be placed in the “build/build-imx6ul-iwg18m-twr-pos” folder.

If the bash shell used for compiling was closed then the “setup-environment” command must be used to make the “bitbake” command available and to set the Yocto environment variables.

```
cd build

source setup-environment build-imx6ul-iwg18m-twr-pos
```

This will make the “bitbake” command available in the terminal. Next, we can recompile the recipes.

```
bitbake -f -c compile pos-payment-demo && bitbake linux-pos-image
```

In this example, we have recompiled the “pos-payment-demo” recipe but other recipes like “pos-l2-hal”, “optee-os”, “linux-iwg18” can be used too.

4.4.3. Building the toolchain

First, we need to setup the Yocto environment script and then we can compile the toolchain

```
cd build

source setup-environment build-imx6ul-iwg18m-twr-pos

bitbake meta-toolchain
```

The installer will be available at

```
build/build-imx6ul-iwg18m-twr-pos/tmp/deploy/sdk/fsl-imx-fb-glibc-x86_64-meta-toolchain-
    cortexa7hf-vfp-neon-toolchain-4.1.15-1.2.0.sh
```

4.4.4. Using a different branch for a repository

When setting up the source code repositories with the “setup” command of the “build-pos-reader.sh” script, the structure of each repository is recreated from the public repositories and the Git bundles (commits) provided by NXP. The history of the repository is reconstructed and a default branch named “lpos” is created for each repository.

If later work is required on a different branch someone can add their private repository and switch to that branch. In this example, we have the private NXP repository hosted on bitbucket and we check out the “card-reader” branch

```
cd build-imx6ul-iwg18m-twr-pos/tmp/work/imx6ul_iwg18m_twr_pos-poky-linux-gnueabi/pos-
    payment-demo/1.0+gitAUTOINC+72381a4bc7-r0/git

git remote add nxf ssh://git@bitbucket.sw.nxp.com/miss/linux-pos-reader.git

git fetch nxf
git checkout nxf/card-reader
```

4.4.5. Yocto recipe build process

Before building the recipes for the target board (the ARMv7 board), the first part of the Yocto build is to compile the native (x86_64) tools that need to run on the developer's machine (e.g. the ARM Cross Compiler, Python, Perl, etc).

All these native tools are built in the x86_64-linux sysroot

```
build/build-imx6ul-iwg18m-twr-pos/tmp/sysroots/x86_64-linux
```

The Cross Compiler (CC) will be built at the following location

```
build/build-imx6ul-iwg18m-twr-pos/tmp/sysroots/x86_64-linux/usr/bin/arm-poky-linux-
    gnueabi/arm-poky-linux-gnueabi-gcc
```

Once the CC is built then Yocto invokes the “run.do_compile” script for each package it needs to compile. For example, for the Payment Demo application, Yocto generates the “run.do_compile” script based on the Payment Demo recipe and generates the following file

```
build/build-imx6ul-iwg18m-twr-pos/tmp/work/imx6ul_iwg18m_twr_pos-poky-linux-gnueabi/pos-
    payment-demo/1.0+gitAUTOINC+16a2a97959-r0/temp/run.do_compile
```

Note: the “AUTOINC+” number generated above is based on the commit SHA of the latest commit on that repository (the SHA of that comit being 16a2a97959).

In the “run.do_compile” file we can notice multiple shell variables, including the $CC variable which is used by the Makefile of a package to cross compile an application

```
export CC="arm-poky-linux-gnueabi-gcc   -march=armv7-a  -mfloat-abi=hard  -mfpu=neon  -
    mtune=cortex-a7  --sysroot=/mnt/devel/linux-pos-reader-weiping/linux-pos-reader/build/build-
    imx6ul-iwg18m-twr-pos/tmp/sysroots/imx6ul-iwg18m-twr-pos"
```

Also, a quick way of recompiling a package without invoking the “bitbake” command which takes more time is to invoke the script directly. First, we CD to the source code, make modifications to the code and then recompile

```
cd build/build-imx6ul-iwg18m-twr-pos/tmp/work/imx6ul_iwg18m_twr_pos-poky-linux-gnueabi/pos-
    payment-demo/1.0+gitAUTOINC+16a2a97959-r0/git

bash ../temp/run.do_compile
```

The output of the build will be found in the “build” directory

```
build/build-imx6ul-iwg18m-twr-pos/tmp/work/imx6ul_iwg18m_twr_pos-poky-linux-gnueabi/pos-
    payment-demo/1.0+gitAUTOINC+16a2a97959-r0/build
```

4.4.6. Submitting new code

When making code changes the recommended way is to edit the code source directly from the Yocto build tree. In this example, we are using the “pos-payment-demo” package.

```
cd build/build-imx6ul-iwg18m-twr-pos/tmp/work/imx6ul_iwg18m_twr_pos-poky-linux-gnueabi/pos-
    payment-demo/1.0+gitAUTOINC+72381a4bc7-r0/git
```

Make the code changes

```
vim apps/payment-demo/pos/pos_payment.c
```

Commit the code locally

```
git add apps/payment-demo/pos/pos_payment.c

git commit
```

The common Git commands can be used to view the changes

```
git show

git log

git diff
```

When using a Git server for hosting the code the Git remotes need to be updated before pushing the code to the server. In this example, we are using the NXP private repository hosted on Bitbucket

```
git remote add miss ssh://git@bitbucket.sw.nxp.com/miss/linux-pos-reader.git

git remote -v
```

This will add the remote named “miss” to the list of remotes. Next, we need to get the latest sources from the Git server and apply our new commit on top. Assuming the branch is called “master” then Git will do this automatically with the following command

```
git pull –rebase miss master
```

In case of rebase conflicts, when other developers pushed code to the Git server on the same files and code lines as our commit, we need to resolve the conflict

```
vim apps/payment-demo/pos/pos_payment.c

git add apps/payment-demo/pos/pos_payment.c

git rebase –continue
```

Next, we can push our new commit

```
git push miss master
```

5. Secure Boot Flow

Introduction

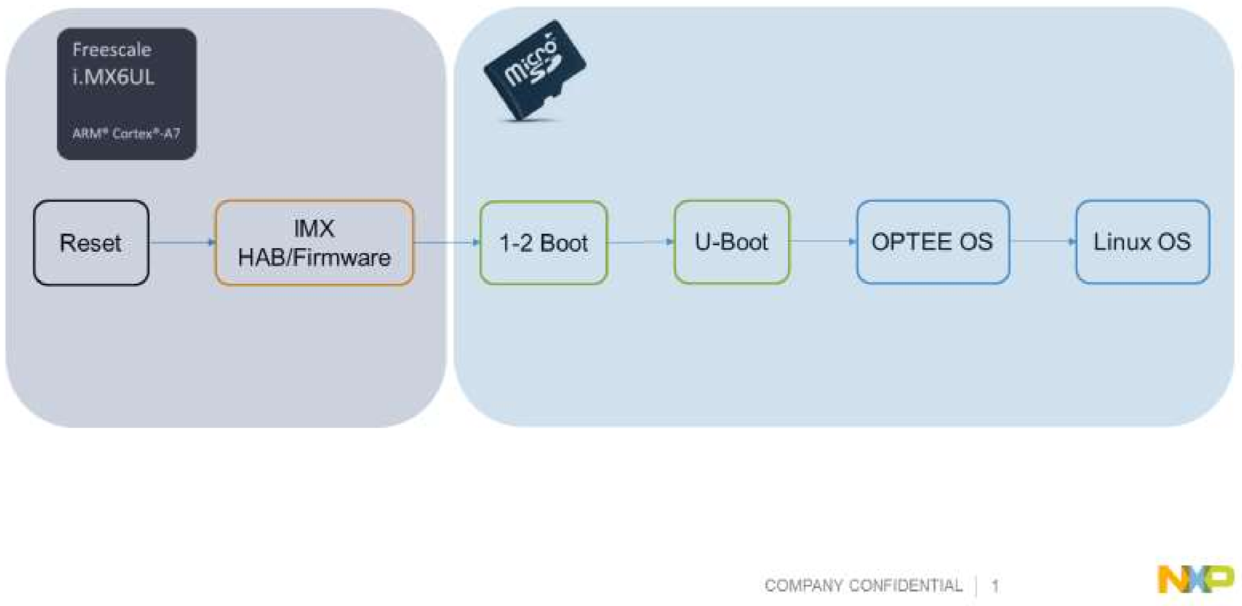
Secure Boot Flow is an important part in providing secure services in the Linux POS Reader. It ensures that the code that starts when the processor starts is the one which was first shipped with the appliance.

Implementation

The main components used to implement the Secure Boot Flow are:

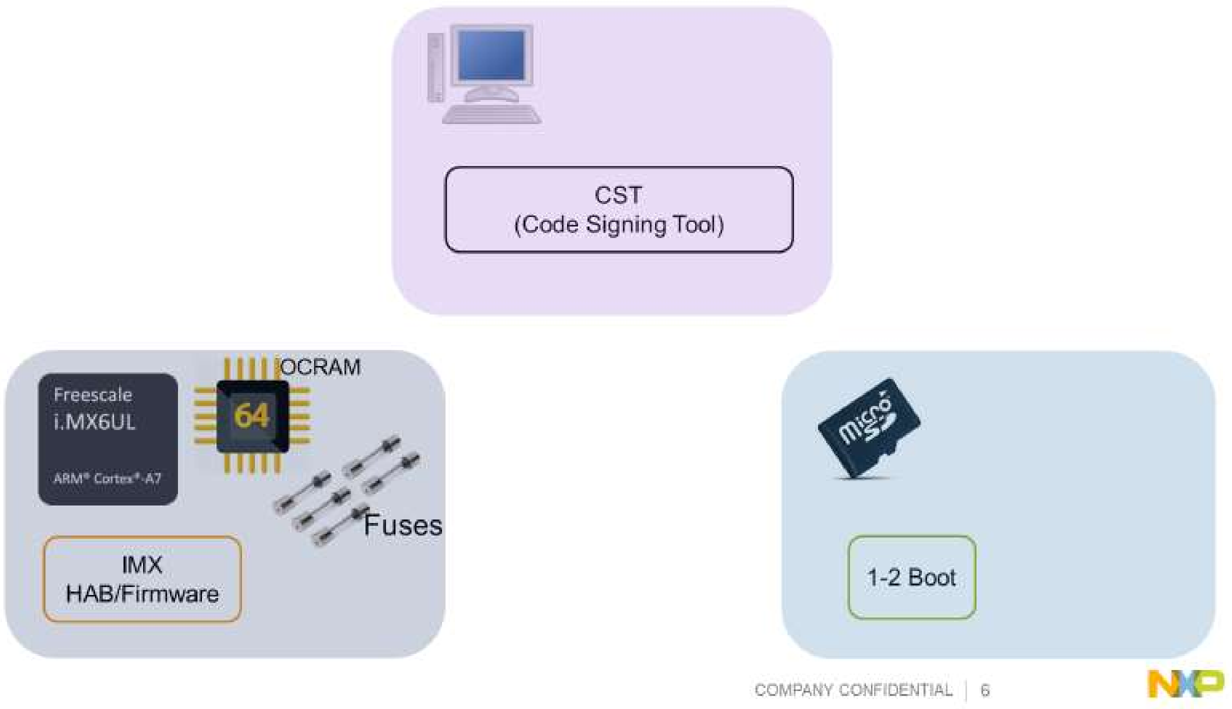
- CST (Code Signing Tool)
- HAB (High Assurance Boot)
- 1-2 Boot
- U-Boot
- OPTEE OS
- Linux kernel

The sequence of components that execute is the following:



5.2.1. CST

The CST manufacturing tool is an application written by NXP to generate authentication certificates and sign the binaries used while booting the i.MX6 system.



The Yocto recipe which builds and generates the certificates is available in the meta-fsl-miss repository at:

```
meta-fsl-miss/recipes-pos/cst-imx/cst-imx6_2.3.3.bb
```

The tool first generates a CA (Certificate Authority) key and a set of 4 SRK (Super Root Keys). Using this key, the tool then generates a Fuse table with which, at manufacturing stage, the physical fuses on the i.MX6UL need to be programed (electrically blown). The fuses will hold the key used to authenticate the signed binaries at boot time.

To program the fuses the user must first dump the fuse table and convert to the proper endianness:

```
hexdump -e '/4 "0x"' -e '/4 "%X""\n"' < SRK_1_2_3_4_fuse.bin
```

The table will look like the following:

```
0xfc66e8e8

0x7af49289

0x7af49289

0xd174bd3b

0xd57defc8
```

```
0xeafa7d44

0xb0185ec3

0x49f8f3ee

0xc450c945
```

Then at the U-Boot prompt the fuses can be programmed with the following command:

```
imx6ul_iwg18m_twr_pos> fuse prog 3 0 0xfc66e8e8

Programming bank 3 word 0x00000000 to 0xfc66e8e8...

Warning: Programming fuses is an irreversible operation!

This may brick your system.

Use this command only if you are sure of what you are doing!

Really perform this fuse programming? <y/N>

y
```

5.2.2. HAB

The first component which starts when the CPU is first powered on is the HAB unit of the i.MX6UL processor. The unit is responsible for authenticating the 1-2 Boot image which is loaded from the first sector of the SDCard. If the image is successfully authenticated then the 1-2 Boot code is executed to further continue the boot process.

The HAB unit relies on physical burnt fuses on the i.MX6UL processor, which hold the public key, to authenticate the binaries loaded from the SDCard.

5.2.3. The 1-2 Boot and U-boot

The 1-2 Boot binary is loaded by default from the first sector of the SDCard directly into the OCRAM (On-Chip RAM). The OCRAM is only 64KB in size and allows 1-2 Boot to configure the TZASC unit (Trust-Zone Address Space Controller). The TZASC unit will filter any memory access and allow/disallow access to Secure Memory regions.

The 1-2 Boot code will also load the rest of the components into RAM (U-Boot, OPTEE OS and the Linux kernel). Before 1-2 Boot gives control to U-Boot, the code authenticates the U-Boot binary by calling HAB API functions which will use the same key from the fuses to authenticate U-Boot.

After U-Boot performs its initialization steps, U-Boot calls the HAB API functions to authenticate OPTEE OS and if authentication is successful then it gives control to OPTEE OS.

The location of the U-Boot Yocto recipe which signs the 1-2 Boot binary and the U-Boot binary is at:

```
meta-fsl-arm-extra/recipes-bsp/u-boot/u-boot-iwg18_2016.03.bb
```

5.2.4. The OPTEE OS and the Linux kernel

OPTEE OS uses a separate authentication set of certificates, for more flexibility in using the signing/authentication phase.

While building the POS project, the Linux Yocto recipe first generates a new set of certificates, with their private and public key pair. The kernel image is then signed with the private key and the public key is exported as a C header file and installed in the following location:

```
build/build-imx6ul-iwg18m-twr-pos/tmp/sysroots/imx6ul-iwg18m-twr-
    pos/usr/include/pos_modulus_auth/pos_modulus_auth.h
```

When the OPTEE OS is being built, the C header is used to store the public key in the OPTEE OS binary and used to authenticate the Linux image at boot time.

Updating the OPTEE/Linux keys

The first time the POS project is built, a new set of keys is generated for the OPTEE/Linux signing process. The private key is generated in the following location, which contains in the path the GIT version of the latest commit, so the path might be different on newer versions of the build:

```
build/build-imx6ul-iwg18m-twr-pos/tmp/work/imx6ul_iwg18m_twr_pos-poky-linux-gnueabi/linux-
    iwg18/1.0+gitAUTOINC+69780be956-r0/build/private.key
```

Re-using an existing key to rebuild and resign the binaries is possible by copying a private.key file to a specific location. The Yocto recipe for the Linux kernel looks for a pre-existing key one level up the build folder:

```
build/build-imx6ul-iwg18m-twr-pos/tmp/work/imx6ul_iwg18m_twr_pos-poky-linux-gnueabi/linux-
    iwg18/1.0+gitAUTOINC+69780be956-r0
```

To trigger a regeneration of the Linux kernel signed image, the SDCard image can be rebuilt with the following commands:

```
cd build

source setup-environment build-imx6ul-iwg18m-twr-pos
```

This will make the “bitbake” command available in the terminal. Next, we can recompile the recipe.

```
bitbake -f -c compile linux-iwg18 && bitbake linux-pos-image
```

6. Secure screen generation

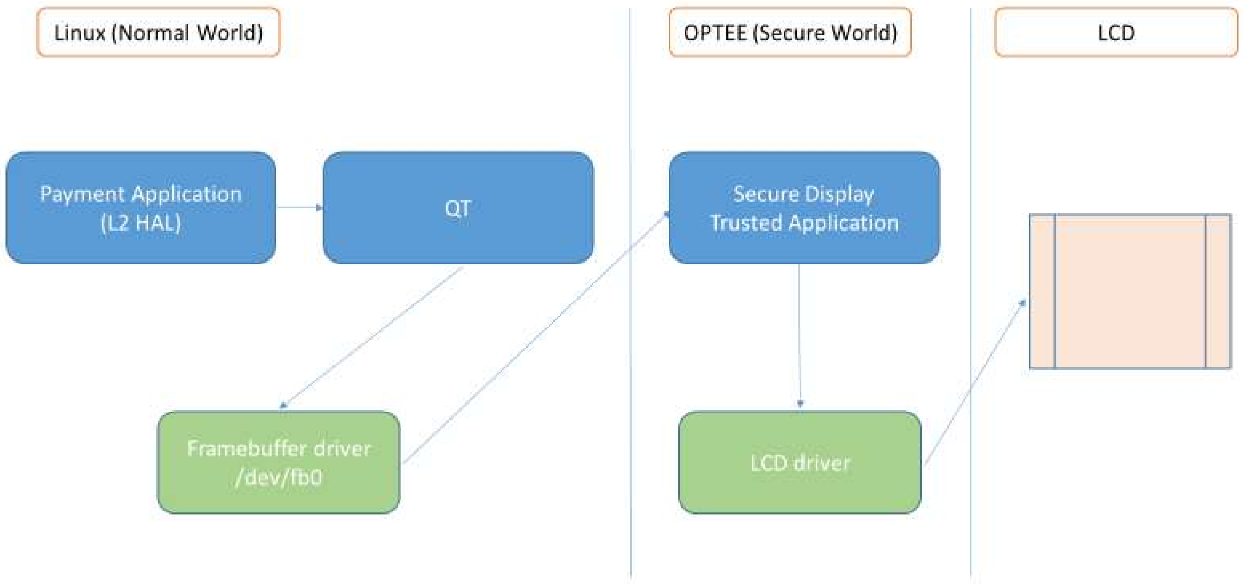
Introduction

The Linux POS Reader needs to protect the information displayed on the LCD, to prevent any malicious usage of the Payment Application. Besides the Payment Flow protection offered by Secure Boot and ISDM, the LCD images are protected by the Secure World LCD driver which uses pre-saved image hashes to compare and authenticate the displayed images.

Implementation

There are three components which are used to secure the display:

- L2 HAL library
- OPTEE OS secure display driver
- Secure Display TA (Trusted Application)


Linux (Normal World)
OPTEE (Secure World)
LCD
Payment Application (L2 HAL)
QT
Secure Display Trusted Application
Framebuffer driver /dev/fb0
LCD driver

There are two parts of the Payment Flow which involve displaying the Secure Screens of the Payment Application. First, all the images are generated in Linux (Normal World) after which they are copied to Secure World for authentication and display.

When the Payment Application enters the Payment flow, the LCD device is commuted to only accept images from Secure World LCD driver. Any Linux application which wants to display images on the LCD are ignored.

The Payment Application, linked to L2 HAL library, uses the QT graphical library to display the image on the LCD. Since it runs in Normal World (Linux) it uses /dev/fb0 as the framebuffer device driver.

The images that the Payment Application displays on /dev/fb0 are only stored in Linux memory but not yet displayed on the LCD, as the LCD is now receiving data only from Secure World driver.

After the image is available in Linux in /dev/fb0, the L2 HAL library makes a Secure World call to copy the image to Secure LCD. Secure World authenticates the image and if the authentication is successful then the image is printed on the Secure LCD screen.

Adding/Updating a Secure Screen

The HAL API can be invoked with different text messages to be displayed on the screen. The HAL functions which show text on the LCD screen are:

- HAL_PIN_GetPin(), which, for example, displays the "ENTER PIN" text
- HAL_LCD_UserInput(), which, for example, is used to display text as "ENTER AMOUNT" or "ENTER DATE AND TIME"
- HAL_TwoLineMessageToDisp()
- HAL_LCD_Message(), which, for example, can print message such as "Tap or Insert card"

In any usage of the above functions, the L2 HAL library needs to obtain an identification number (ID) for the Secure Screen it needs to display on the Secure LCD. This ID is obtained based on the arguments received by the invoked functions and then passed to Secure World for authentication. The code which identifies the screen is part of the linux-pos-reader repository at:

linux-pos-reader/pos/modules/ui/src/auth_id.c

For example, to know that the Payment Application displays the "Tap or Insert card" screen, the text is identified as SECDISPLAY_SCREEN_INSERT_OR_TAP_CARD, ID which is then sent to Secure World as part of the OPTEE-OS call.

The OPTEE-OS call receives this ID inside the core code at:

optee-os/core/tee/tee_svc_framebuffer.c

The received image is hashed using the SHA256 algorithm and compared with the pre-saved set of hashes. The ID of the image is used to obtain the pre-saved hash, as for example the "Tap or Insert card" hash is defined as:

```
{ SECDISPLAY_SCREEN_INSERT_OR_TAP_CARD,

    { 0xba, 0x90, 0x81, 0x53, 0xe5, 0x6f, 0x5f, 0x1a,

    0x01, 0xaa, 0x2c, 0x76, 0x7b, 0xba, 0x2c, 0xd5,

    0x57, 0xd0, 0x46, 0x5d, 0xdd, 0x4f, 0x5a, 0x3b,

    0x92, 0xcd, 0xfc, 0xd6, 0x1f, 0xf9, 0x0a, 0xc7 }

},
```

If the hashes do not match, as for example in the case of a different text that needs to be displayed, the following error will be printed:

```
ERROR: TEE-CORE: Failed to authenticate image id 4

ERROR: TEE-CORE: Received image hash

MESSAGE: [0x0] TEE-CORE:print_image_hash:260: 0x53, 0xcb, 0x5e, 0x33, 0xc3, 0x20, 0x1a, 0x83,

MESSAGE: [0x0] TEE-CORE:print_image_hash:260: 0x97, 0xd2, 0xd6, 0xb2, 0xab, 0xd4, 0xeb, 0xc6,

MESSAGE: [0x0] TEE-CORE:print_image_hash:260: 0x0d, 0x7f, 0x2f, 0x83, 0x44, 0x62, 0x49, 0x5d,

MESSAGE: [0x0] TEE-CORE:print_image_hash:260: 0xeb, 0x27, 0x51, 0x9f, 0xab, 0x74, 0x22, 0x85,

ERROR: TEE-CORE: Presaved image hash

MESSAGE: [0x0] TEE-CORE:print_image_hash:260: 0xba, 0x90, 0x81, 0x53, 0xe5, 0x6f, 0x5f, 0x1a,

MESSAGE: [0x0] TEE-CORE:print_image_hash:260: 0x01, 0xaa, 0x2c, 0x76, 0x7b, 0xba, 0x2c, 0xd5,

MESSAGE: [0x0] TEE-CORE:print_image_hash:260: 0x57, 0xd0, 0x46, 0x5d, 0xdd, 0x4f, 0x5a, 0x3b,

MESSAGE: [0x0] TEE-CORE:print_image_hash:260: 0x92, 0xcd, 0xfc, 0xd6, 0x1f, 0xf9, 0x0a, 0xc7,

Failed to authenticate the image in Secure World, res ffff000f, err_origin 4
```

To update the hash for this screen, copy-paste the "Received image hash", similar to the above log, over the pre-saved hash in tee_svc_framebuffer.c.

In case a new screen is inserted in the Payment Flow, then a new ID needs to be created and a new hash needs to be inserted in "screen_hash" vector. The new ID needs to be part of the LCD TA header file located at:

optee-os/lib/libutee/include/ta_secdisplay.h

The last step is to update the OPTEE-POS-TA repository and insert the new ID in the set of calls that involve displaying an authenticated image. The function name is "TA_InvokeCommandEntryPoint()" and located at:

optee-pos-ta/ta/LCD_FB/src/secdisplay_ta.c

7. ISDM - Inter Services Domain Manager

Introduction

The ISDM is a module implemented in OPTEE OS core in order to make sure that the Payment Application flow is legit.

Implementation

The ISDM module is implemented using a "trie" tree (also known as prefix tree) and allows storing valid sequences which can be encountered in the Payment Flow.

When the L2 HAL library invokes Secure World TA (Truested Application) commads, the ISDM module intercepts the commands and checks them against the valid sequences trie, allowing or rejecting the command. Each issued command has a UUID associated with it, and ISDM checks this UUID and performs the following operations:

- Skip – if the TA is not specifically related to the transaction
- Add to the current sequence – if the TA is involved in a transaction
- Cancel transaction – if the current sequence doesn't belong to any of the allowed sequences

One correct sequence might look like this:

Module	Command
PINPAD	PIN_ENABLEPINPADINPUT_ID
CARDREADER	CR_RF_FIELD_ON_ID
CARDREADER	CR_WAITFORCARD_ID
CARDREADER	CR_EXCHANGEAPDU_ID
CRYPTO	CRYPTO_TRNG
CARDREADER	CR_EXCHANGEAPDU_ID
CRYPTO	CRYPTO_RSA_DECRYPT
CARDREADER	CR_EXCHANGEAPDU_ID
PINPAD	PIN_GETPIN_ID
PINPAD	PIN_ENCHIPHER_OFFLINE_ID
CARDREADER	CR_EXCHANGEAPDU_ID
CARDREADER	CR_POLLFORCARD_REMOVAL_ID
CARDREADER	CR_WAITFORCARD_REMOVE_ID
CARDREADER	CR_RF_FIELD_OFF_ID

The ISDM code can be found in the OPTEE OS repository at:

optee-os/core/kernel/isdm.c

8. Tamper

Introduction

The tamper module provides a mechanism to detect any physical device corruption (e.g. opening the enclosure). Detection of the temper on the system results in limited functionality or impossibility to boot the system.

Implementation

Tempering can be detected through the I/O TAMPER HEADER (J15) located on the board. The default TAMPER HEADER configuration is as following:

- Active Tamper:
  - SNVS_TAMPER3 to SNVS_TAMPER7

    - SNVS_TAMPER8 to SNVS_TAMPER9

- Passive Tamper:
    - SNVS_TAMPER0 to polarity 1
    - SNVS_TAMPER1 to polarity 1
    - SNVS_TAMPER2 to polarity 1

If the tamper is detected the system will have the following behavior:

- At boot time, the board will not boot
- When Payment Application is running, the application functions are stopped and the system will reboot
- To properly start back the system and the payment-application the tamper jumpers must be restored to the original state

9. Running Demo Scenarios

For information on using the Payment Demo to perform payment card transactions, please see the L2 Kernel Card Profiles and Demo Scenarios 03082017 v3.1 in the documentation folder from the release package.

10. Revision history

Table 1. Revision history

Revision number	Date	Substantive changes
1	01/2018	Initial release
2	04/2018	Release 3.0

How to Reach Us:

Home Page:
nxp.com

Web Support:
nxp.com/support

Document Number: LPOSRS003SUG
Rev. 2
04/2018