NXP Semiconductors
User's Guide

Document Number: IMXVIRTUG
Rev. 1, 05/2019

i.MX Virtualization User's Guide

1. Release Description

Virtualization includes a wide range of technologies, such as hypervisor, desktop virtualization, and Unikernel. This documentation focuses on Jailhouse and Xen, which are implementations of hypervisor technologies.

The instructions on how to configure the Yocto Project tools to build the image with Xen or Jailhouse support can be found in the *Yocto Project User's Guide* (IMXLXYOCTOUG) or the *Android User's Guide* (AUG). This User’s Guide describes how Xen and Jailhouse work and how to start them on the boards.

For details, see the documentation of Linux L4.14.98_2.0.0_ga and Android P9.0.0_2.1.0-auto-ga.

Contents

2. Jailhouse

2.1. Overview

Jailhouse is a static partitioning hypervisor based on Linux OS. It can run bare-metal applications and several operating systems, including Linux Kernel OS. For simplicity, it only supports CPU partitioning, but not support virtual CPU scheduling. It configures CPUs and devices into different domains, called "cells". The guest OS or bare-metal application running inside the cell is called an “inmate.”

When Jailhouse is activated, it takes full control over the hardware and needs no external support. However, in contrast to other hypervisors, it is loaded and configured by Linux system. Its management interface is based on the Linux infrastructure. Therefore, users need to boot Linux OS first, then enable Jailhouse, and finally split off parts of the system's resources and assign them to additional cells.

2.2. Boot workflow

The following figure shows the workflow from board power-on to multi-OS bootup. Jailhouse relies on the Linux OS (another boot loader) to do Jailhouse kernel loading and initialization.

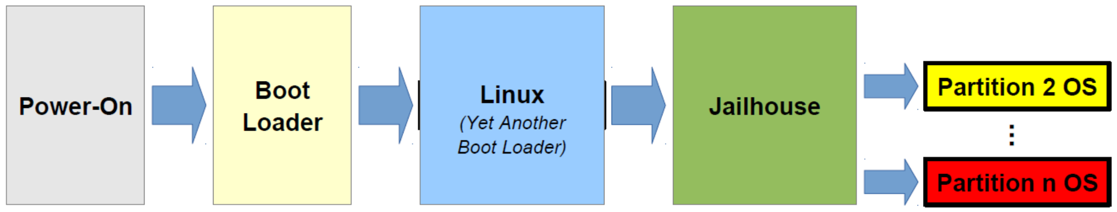


Figure 1. **Boot workflow**

2.3. Managing interfaces

Jailhouse provides several user-space interfaces as follows:

```
#jailhouse
Usage: jailhouse { COMMAND | --help | --version }
```

Available commands:

```
   enable SYSCONFIG
   disable
   console [-f | --follow]
   cell create CELLCONFIG
   cell list
   cell load { ID | [--name] NAME } { IMAGE | { -s | --string } "STRING" }
             [-a | --address ADDRESS] ...
   cell start { ID | [--name] NAME }
```

```
cell shutdown { ID | [--name] NAME }
cell destroy { ID | [--name] NAME }
cell linux CELLCONFIG KERNEL [-i | --initrd FILE]
           [-c | --cmdline "STRING"] [-w | --write-params FILE]
cell stats { ID | [--name] NAME }
config create [-h] [-g] [-r ROOT] [--mem-inmates MEM_INMATES]
              [--mem-hv MEM_HV] FILE
config collect FILE.TAR
hardware check
```

Command descriptions are as follows:

- Enable: To load the Jailhouse kernel to the memory and complete initialization.
- Console: To show the Jailhouse kernel log.
- Cell create: To create a new cell.
- Cell list: To list the cells of the system.
- Cell load: To load the cell image to the cell that specified by ID or name.
- Cell start: To start a cell.
- Cell shutdown: To shut down a cell.
- Cell destroy: To destroy a cell, and the resources of the cell will be released.
- Cell stats: To show the status of vmexits_cp15/hypercall/mmio/psci and etc.
- Cell Linux: To create/load/start a Linux cell.

2.4. Using Jailhouse with a prebuilt image

Prebuilt images are only provided for i.MX 8M Mini and i.MX 8M Quad.

NOTE

This section takes i.MX 8M Mini as an example. For i.MX 8M Quad, replace "8mm" with "8mq" in the following steps.

The jailhouse.ko and uio_ivshmem.ko files are under:

```
/lib/modules/<linux_kernel_version>/extra/driver
```

The cells and inmates are under:

```
/usr/share/jailhouse
```

1. Boot from the SD card. In U-Boot, run `run jh_netboot` or `run jh_mmcboot`. It loads a dedicated DTB for Jailhouse usage.
2. After Linux OS boots up, execute the following commands to use Jailhouse with an ivshmem demo:

```
#insmod /lib/modules/<linux_kernel_version>/extra/driver/jailhouse.ko
#insmod /lib/modules/<linux_kernel_version>/extra/driver/uio_ivshmem.ko
```

```
#jailhouse enable /usr/share/jailhouse/cells/imx8mm.cell
#jailhouse cell create /usr/share/jailhouse/cells/imx8mm-ivshmem-demo.cell
#jailhouse cell load 1 /usr/share/jailhouse/inmates/ivshmem-demo.bin
#jailhouse cell start 1
```

Then, the following logs are displayed:

```
Started cell "ivshmem-demo"
root@imx8mmevk:~/jailhouse# IVSHMEM: Found 1af4:1110 at 00:00.0
IVSHMEM: shmem is at 0x00000000bba00000
IVSHMEM: bar0 is at 0x00000000bbb00000
IVSHMEM: mapped shmem and bars, got position 0x0000000000000001
IVSHMEM: Hello from bare-metal ivshmem-demo inmate!!!
IVSHMEM: Enabled IRQ:0x6c
IVSHMEM: Enabling IVSHMEM_IRQs
IVSHMEM: Done setting up...
IVSHMEM: waiting for interrupt.
```

3. Check whether the device node exists. Run `/dev/uio0 and cat /proc/interrupts | grep uio`. You could see interrupts triggered once.
4. Use the following commands to enable the communication between the Linux OS and ivshmem-demo. Open two shells, using SSH to log into the board. Execute uio_read in one shell and execute uio_send in the other shell.

```
#uio_send /dev/uio0 [count] 0 0
#uio_read /dev/uio0 [count]
```

2.5. Building instructions

This section provides information for building Jailhouse hypervisor, kernel modules, and userspace tools for using Jailhouse. If you use Yocto to build i.MX Jailhouse, ignore this section. You could download the source code from the https://source.codeaurora.org/external/imx/imx-jailhouse branch: imx_4.14.98_2.0.0_ga.

1. Install the L4.14.98 GA Yocto toolchain and execute `$export KDIR=<linux kernel source>`.
2. Execute `$source <toolchain-installation-path>/environment-setup-aarch64-poky-linux #replace` with your own path.

```
unset CFLAGS
unset LDFLAGS
make ARCH=arm64 CROSS_COMPILE=aarch64-poky-linux- KDIR=../linux-kernel
CC="aarch64-poky-linux-gcc  --sysroot=$SDKTARGETSYSROOT" clean
make ARCH=arm64 CROSS_COMPILE=aarch64-poky-linux- KDIR=../linux-kernel
CC="aarch64-poky-linux-gcc  --sysroot=$SDKTARGETSYSROOT"
```

3. Copy `driver/jailhouse.ko`, `driver/ivshmem.ko`, `hypervisor/jailhouse.bin`, `inmates/demos/arm64/*.bin`, `configs/arm64/imx8*.cell`, and the executables under tools to `/home/root/jailhouse` on your board.
4. Copy `inmates/tools/arm64/linux-loader.bin` to `/home/root/inmates/tools/arm64/` on your board.

2.6. Demo usage

This section shows how to use GIC, IVSHMEM, and Linux demos.

2.6.1. GIC demo

The following is a GIC demo example on i.MX 8MQuad. It only contains the non-root cell part. For information on Jailhouse usage, see Section 2.4 “ Using Jailhouse with prebuilt image”.

```
# jailhouse cell create /usr/share/jailhouse/cells/imx8mq-gic-demo.cell
[ 1588.415166] CPU3: shutdown
[ 1588.417885] psci: CPU3 killed.
Created cell "gic-demo"
Page pool usage after cell creation: mem 66/994, remap 336/131072
[ 1588.447694] Created Jailhouse cell "gic-demo"
# jailhouse cell load 1 /usr/share/jailhouse/inmates/gic-demo.bin
Cell "gic-demo" can be loaded
#jailhouse cell start 1   #log, now you see it is running
Timer fired, jitter:    999 ns, min:    999 ns, max:   3624 ns
Timer fired, jitter:    999 ns, min:    999 ns, max:   3624 ns
Timer fired, jitter:    999 ns, min:    999 ns, max:   3624 ns
Timer fired, jitter:    999 ns, min:    999 ns, max:   3624 ns
Timer fired, jitter:    999 ns, min:    999 ns, max:   3624 ns
Timer fired, jitter:   1124 ns, min:    999 ns, max:   3624 ns
#./jailhouse cell shutdown 1 or #./jailhouse cell destroy 1
```

2.6.2. IVSHMEM demo

The following is an IVSHMEM demo example on i.MX 8MQuad. It only contains non-root cell part. For information on Jailhouse usage, see Section 2.4 “ Using Jailhouse with prebuilt image”.

Destroy the GIC demo if you already create it in the previous section.

```
#jailhouse cell destroy 1
```

Page pool usage after cell destruction: mem 55/994, remap 336/131072

```
[ 1743.019179] Detected VIPT I-cache on CPU3
[ 1743.019213] GICv3: CPU3: found redistributor 3 region 0:0x00000000388e0000
[ 1743.019274] CPU3: Booted secondary processor [410fd034]
[ 1743.050367] Destroyed Jailhouse cell "gic-demo"
#jailhouse cell creates imx8mq-ivshmem-demo.cell
[ 1867.115170] CPU3: shutdown
[ 1867.117891] psci: CPU3 killed.
Adding virtual PCI device 00:00.0 to cell "ivshmem-demo"
Shared memory connection established: "ivshmem-demo" <--> "imx8mq"
Created cell "ivshmem-demo"
Page pool usage after cell creation: mem 69/994, remap 336/131072
[ 1867.181641] Created Jailhouse cell "ivshmem-demo"
#insmod uio_ivshmem.ko  #see following section on how to build uio_ivshmem.ko

#jailhouse cell load 1 ivshmem-demo.bin
Cell "ivshmem-demo" can be loaded

#jailhouse cell start 1
Started cell "ivshmem-demo"
root@imx8mqevk:~/jailhouse# IVSHMEM: Found 1af4:1110 at 00:00.0
IVSHMEM: shmem is at 0x00000000bfe00000
IVSHMEM: bar0 is at 0x00000000bfe01000
IVSHMEM: mapped shmem and bars, got position 0x0000000000000001
IVSHMEM: Hello from bare-metal ivshmem-demo inmate!!!
IVSHMEM: Enabled IRQ:0x8d
IVSHMEM: Enabling IVSHMEM_IRQs
IVSHMEM: Found 1af4:1110 at 00:01.0
IVSHMEM: shmem is at 0x00000000bfd00000
IVSHMEM: bar0 is at 0x00000000bfd01000
IVSHMEM: mapped shmem and bars, got position 0x0000000000000001
IVSHMEM: Hello from bare-metal ivshmem-demo inmate!!!
IVSHMEM: Enabled IRQ:0x8e
IVSHMEM: Enabling IVSHMEM_IRQs
IVSHMEM: Done setting up...
[ 1930.428350] ivshmem_handler ivshmem_handler 234
#uio_read /dev/uio0 1
#uio_send /dev/uio0 1 0 0
 [UIO] opening file /dev/uio0
```

```
    [UIO] count is 1

    [UIO] writingV 0

    [UIO] ping #0

    ISHMEM: handle_irq(irqn:141) - interrupt #0

    [ 2086.064878] ivshmem_handler ivshmem_handler 234

    IVSHMEM: waiting for interrupt.

    [UIO] Exiting...

    #cat /proc/interrupts  | grep uio   #Each time, execute `./uio_send  /dev/uio0 1
0 0`, the interrupts trigger times will increase.

    234:          6          0          0     GICv3  88 Edge      uio_ivshmem
```

When `uio_send` is being executed, `uio_read` could read data from ivshmem demo.

NOTE

uio_read and uio_send need to be started in different shells.

2.6.3. Linux demo

Destroy the previous setup if you already create GIC or IVSHMEM demo. For root-cell, use `*-veth.cell` and then export `/usr/share/jailhouse/tools` into PATH.

#export PATH=$PATH:/usr/share/jailhouse/tools

```
#jailhouse cell destroy 1

#jailhouse disable

#jailhouse enable imx8mq-veth.cell
```

Now start Linux demo.

```
# jailhouse cell linux /usr/share/jailhouse/cells/imx8mq-linux-demo.cell
/run/media/mmcblk1p1/Image -d /run/media/mmcblk1p1/fsl-imx8mq-evk-inmate.dtb -c
"clk_ignore_unused console=ttymxc1,115200 earlycon=ec_imx6q,0x30860000,1115200
root=/dev/mmcblk0p2 rootwait rw"
```

The image is a copy of your root cell Linux Image. fsl-imx8mq-evk-inmate.dtb is generated when you build your Linux Kernel.

Then, you could see the Linux inmates print out earlycon log on root cell Linux UART and print out more logs on another UART on the board.

You could configure the eth1 in root cell and eth0 in inmate cell and ping.

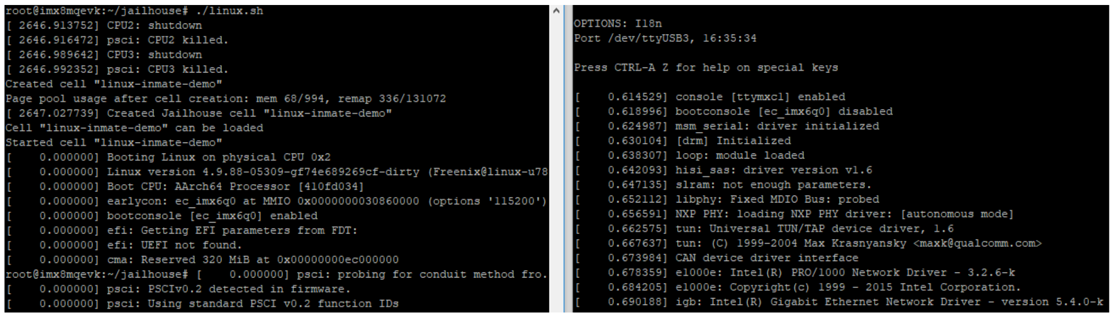
In inmate cell

```
#ifconfig eth0 192.168.0.2
```

In root cell

```
#ifconfig eth1 192.168.0.3
```

Now ping eth0 using eth1 `#ping 192.168.0.2` in root cell or ping eth1 using eth0 `ping 192.168.0.3` in inmate cell. It will work.



```
root@imx8mqevk:~/jailhouse# ./linux.sh
[ 2646.913752] CPU2: shutdown
[ 2646.916472] psci: CPU2 killed.
[ 2646.989642] CPU3: shutdown
[ 2646.992352] psci: CPU3 killed.
Created cell "linux-inmate-demo"
Page pool usage after cell creation: mem 68/994, remap 336/131072
[ 2647.027739] Created Jailhouse cell "linux-inmate-demo"
Cell "linux-inmate-demo" can be loaded
Started cell "linux-inmate-demo"
[    0.000000] Booting Linux on physical CPU 0x2
[    0.000000] Linux version 4.9.88-05309-gf74e689269cf-dirty (Freenix@linux-u78
[    0.000000] Boot CPU: AArch64 Processor [410fd034]
[    0.000000] earlycon: ec_imx6q0 at MMIO 0x0000000030860000 (options '115200')
[    0.000000] bootconsole [ec_imx6q0] enabled
[    0.000000] efi: Getting EFI parameters from FDT:
[    0.000000] efi: UEFI not found.
[    0.000000] cma: Reserved 320 MiB at 0x00000000ec000000
root@imx8mqevk:~/jailhouse# [    0.000000] psci: probing for conduit method fro.
[    0.000000] psci: PSCIv0.2 detected in firmware.
[    0.000000] psci: Using standard PSCI v0.2 function IDs
```

```

OPTIONS: I18n
Port /dev/ttyUSB3, 16:35:34

Press CTRL-A Z for help on special keys

[    0.614529] console [ttymxc1] enabled
[    0.618996] bootconsole [ec_imx6q0] disabled
[    0.624987] msm_serial: driver initialized
[    0.630104] [drm] Initialized
[    0.638307] loop: module loaded
[    0.642093] hisi_sas: driver version v1.6
[    0.647135] slram: not enough parameters.
[    0.652112] libphy: Fixed MDIO Bus: probed
[    0.656591] NXP PHY: loading NXP PHY driver: [autonomous mode]
[    0.662575] tun: Universal TUN/TAP device driver, 1.6
[    0.667637] tun: (C) 1999-2004 Max Krasnyansky <maxk@qualcomm.com>
[    0.673984] CAN device driver interface
[    0.678359] e1000e: Intel(R) PRO/1000 Network Driver - 3.2.6-k
[    0.684205] e1000e: Copyright(c) 1999 - 2015 Intel Corporation.
[    0.690188] igb: Intel(R) Gigabit Ethernet Network Driver - version 5.4.0-k
```

Figure 2. **Boot log from root and inmate cell Linux**

For i.MX 8M Quad:

```
# jailhouse cell linux /usr/share/jailhouse/cells/imx8mq-linux-demo.cell
/run/media/mmcblk1p1/Image -d /run/media/mmcblk1p1/fsl-imx8mq-evk-inmate.dtb -c
"clk_ignore_unused console=ttymxc1,115200 earlycon=ec_imx6q,0x30860000,1115200
root=/dev/mmcblk0p2 rootwait rw"
```

For i.MX 8M Mini

```
# jailhouse cell linux /usr/share/jailhouse/cells/imx8mm-linux-demo.cell
/run/media/mmcblk1p1/Image -d /run/media/mmcblk1p1/fsl-imx8mm-evk-inmate.dtb -c
"clk_ignore_unused console=ttymxc3,115200 earlycon=ec_imx6q,0x30890000,115200
root=/dev/mmcblk2p2 rootwait rw"
```

For i.MX 8QuadXPlus, the UART in the base board RS232 UART (conflict with Arm Cortex-M4 UART).

```
#jailhouse cell linux imx8qxp-linux-demo.cell Image -d fsl-imx8qxp-mek-inmate.dtb -
c "clk_ignore_unused console=ttyLP2,115200 earlycon=lpuart32,0x5a060000,115200
root=/dev/mmcblk0p2 rootwait rw"
```

For other platforms, the inmate dtb and bootargs need to be updated accordingly.

2.7. Jailhouse internals

This section shows the changes required when enabling Jailhouse on the custom platform.

2.7.1. Linux DTS changes

In fsl-<soc-name>-<board-name>-root.dts, there are pieces of memory reserved for Jailhouse hypervisor/inmate/loader/shmem/pci configuration space. Take fsl-imx8mq-evk-root.dts as an example:

```
  &resmem {
  jh_reserved: jh@0xfdc00000 {                                      (1)
                      no-map;
                      reg = <0 0xfdc00000 0x0 0x400000>;
          };
```

```
        inmate_reserved: inmate@0xc0000000 {                      (2)
                no-map;
                reg = <0 0xc0000000 0x0 0x3dc00000>;
        };

        loader_reserved: loader@0xbff00000 {                      (3)
                no-map;
                reg = <0 0xbff00000 0x0 0x00100000>;
        };

        ivshmem_reserved: ivshmem@0xbfe00000 {                    (4)
                no-map;
                reg = <0 0xbfe00000 0x0 0x00100000>;
        };

        ivshmem2_reserved: ivshmem2@0xbfd00000 {                 (5)
                no-map;
                reg = <0 0xbfd00000 0x0 0x00100000>;
        };

        pci_reserved: pci@0xbfc00000 {                           (6)
                no-map;
                reg = <0 0xbfb00000 0x0 0x00200000>;
        };
};
```

(1) Used by Jailhouse hypervisor.

(2) Used by inmate. The reserved memory is large enough to boot a second LinuxOS. If you do not need such a big memory, shrink the memory as required, and rearrange the memory of (3)(4)(5)(6).

(3) Inmate loader reserved memory.

(4)(5) SHMEM for cross-cell communication.

(6) Virtual PCI space.

2.7.2. Root cell configuration

The following provides an example of commands for how to configure a root cell:

```
struct {
        struct jailhouse_system header;
        __u64 cpus[1];
        struct jailhouse_memory mem_regions[70];
        struct jailhouse_irqchip irqchips[3];
        struct jailhouse_pci_device pci_devices[1];
} __attribute__((packed)) config = {
        .header = {
                .signature = JAILHOUSE_SYSTEM_SIGNATURE,
                .revision = JAILHOUSE_CONFIG_REVISION,
                .hypervisor_memory = {
                        .phys_start = 0xfdc00000, (a) hypervisor start memory, 
see (1)
                        .size =       0x00400000,
                },
                .debug_console = {
                        .address = 0x30860000,  (b) The debug uart address
                        .size = 0x1000,
                        .flags = JAILHOUSE_CON1_TYPE_IMX |
                                 JAILHOUSE_CON1_ACCESS_MMIO |
                                 JAILHOUSE_CON1_REGDIST_4 |
                                 JAILHOUSE_CON2_TYPE_ROOTPAGE,
                },
                .platform_info = {
                        /*
                         * .pci_mmconfig_base is fixed; if you change it,
                         * update the value in mach.h
                         * (PCI_CFG_BASE) and regenerate the inmate library
                         */
                        .pci_mmconfig_base = 0xbfb00000, (c) virtual pci space, 
see(6)
                        .pci_mmconfig_end_bus = 0,
                        .pci_is_virtual = 1,
                        .arm = {
                                .gic_version = 3,
                                .gicd_base = 0x38800000,
                                .gicr_base = 0x38880000,
                                .maintenance_irq = 25,
                        },
                },
                .root_cell = {
                        .name = "imx8mq",

                        .num_pci_devices = ARRAY_SIZE(config.pci_devices),
                        .cpu_set_size = sizeof(config.cpus),
                        .num_memory_regions = ARRAY_SIZE(config.mem_regions),
                        .num_irqchips = ARRAY_SIZE(config.irqchips),
/* Not include 32 base */
                        .vpci_irq_base = 56,  (d) doorbell interrupt for root
cell
                },
        },

        .cpus = {
                0xf,
        },

        .mem_regions = {
                /* SAI3 */ {
                        .phys_start = 0x30030000,
                        .virt_start = 0x30030000,
                        .size =       0x10000,
                        .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE |
                                JAILHOUSE_MEM_IO,
                },
                /* SAI5 */ {
                        .phys_start = 0x30040000,
                        .virt_start = 0x30040000,
                        .size =       0x10000,
                        .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE |
                                JAILHOUSE_MEM_IO,
                },
                [...ignore part code...]
                /* DDR_PMU */ {
                        .phys_start = 0x3d800000,
                        .virt_start = 0x3d800000,
                        .size =       0x400000,
                        .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE |
                                JAILHOUSE_MEM_EXECUTE,
                },
                /* RAM */ {
                        .phys_start = 0x40000000,
                        .virt_start = 0x40000000,
                        .size = 0x7fb00000,
                        .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE |
                                JAILHOUSE_MEM_EXECUTE,
                },
                /* IVHSMEM shared memory region for 00:00.0 */ {
  /* (e) The shmem region, same as (4)(5) */
                        .phys_start = 0xbfd00000,
                        .virt_start = 0xbfd00000,
                        .size = 0x200000,
                        .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE ,
                },
                /* Linux Loader */{
                        .phys_start = 0xbff00000,
                        .virt_start = 0xbff00000,
                        .size = 0x100000,
                        .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE |
                                JAILHOUSE_MEM_EXECUTE,
                },
                /* Inmate memory */{
                        .phys_start = 0xc0000000,
                        .virt_start = 0xc0000000,
                        .size = 0x3fc00000,
                        .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE |
                                JAILHOUSE_MEM_EXECUTE,
                },
        },

        .irqchips = {
                /* GIC */ {
                        .address = 0x38800000,
                        .pin_base = 32,
                        .pin_bitmap = {
                                0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff,
                        },
                },
                /* GIC */ {
                        .address = 0x38800000,
                        .pin_base = 160,
                        .pin_bitmap = {
                                0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff,
                        },
                },
                /* GIC */ {
                        .address = 0x38800000,
                        .pin_base = 288,
                        .pin_bitmap = {
                                0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff,
                        },
                },
        },

        .pci_devices = {
                {
                        .type = JAILHOUSE_PCI_TYPE_IVSHMEM,
                        .bdf = 0x0 << 3,
                        .bar_mask = {
                                0xffffff00, 0xffffffff, 0x00000000,
                                0x00000000, 0x00000000, 0x00000000,
                        },

                        /*num_msix_vectors needs to be 0 for INTx operation*/
                        .num_msix_vectors = 0,
                        .shmem_region = 67,
  /*(f) The shmem region index, see (e)*/
                        .shmem_protocol = JAILHOUSE_SHMEM_PROTO_VETH,     /*(g)
  The protocol, ivshmem-net needs xx_PROTO_VETH*/
                        .domain = 0x0,
                },
        },
};
```

2.7.3. Inmate example

Take imx8mq-ivshmem-demo.c as an example. It supports communication with root cell.

```
struct {
        struct jailhouse_cell_desc cell;
        __u64 cpus[1];
        struct jailhouse_memory mem_regions[4];
        struct jailhouse_irqchip irqchips[1];
        struct jailhouse_pci_device pci_devices[1];
} __attribute__((packed)) config = {
        .cell = {
                .signature = JAILHOUSE_CELL_DESC_SIGNATURE,
                .revision = JAILHOUSE_CONFIG_REVISION,
                .name = "ivshmem-demo",
                .flags = JAILHOUSE_CELL_PASSIVE_COMMREG,

                .cpu_set_size = sizeof(config.cpus),
                .num_memory_regions = ARRAY_SIZE(config.mem_regions),
                .num_irqchips = ARRAY_SIZE(config.irqchips),
                .num_pci_devices = ARRAY_SIZE(config.pci_devices),
                .vpci_irq_base = 109, /* Not include 32 base */
/* (I) doorbell for ivshmem-demo cell */
                .pio_bitmap_size = 0,
        },
        .cpus = {
                0x8,
/*(II)This is cpu mask, means cpu3 will be used for inmate*/
        },
        .mem_regions = {
                /* UART1 */ {
                        .phys_start = 0x30860000,
                        .virt_start = 0x30860000,
                        .size = 0x1000,
                        .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE |
                                JAILHOUSE_MEM_IO | JAILHOUSE_MEM_ROOTSHARED,
                },
                /* RAM: Top at 4GB Space */ {
                        .phys_start = 0xffaf0000,
                        .virt_start = 0,
                        .size = 0x00010000,
                        .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE |
                                JAILHOUSE_MEM_EXECUTE | JAILHOUSE_MEM_LOADABLE,
                },
                /* IVSHMEM */ {
                        .phys_start = 0xbfd00000,
                        .virt_start = 0xbfd00000,
                        .size = 0x200000,
                        .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE |
                                JAILHOUSE_MEM_ROOTSHARED,
                },
                /* communication region */ {
                        .virt_start = 0x80000000,
                        .size = 0x00001000,
                        .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE |
                                JAILHOUSE_MEM_COMM_REGION,
                },
        },

        .irqchips = {
                /* GIC */ {
                        .address = 0x38800000,
                        .pin_base = 128,
/* (III) pin_base is ((109 + 32) / 32) * 32 */
                        .pin_bitmap = {
                                0x1 << (109 + 32 - 128) /* SPI 109 */
/*(IV) same value as 1 << ((109 + 32) % 32)*/
                        },
                },
        },
        .pci_devices = {
                {
                        .type = JAILHOUSE_PCI_TYPE_IVSHMEM,
                        .bdf = 0x0 << 3,
                        .bar_mask = {
                                0xffffff00, 0xffffffff, 0x00000000,
                                0x00000000, 0x00000000, 0x00000000,
                        },

                        /*num_msix_vectors needs to be 0 for INTx operation*/
                        .num_msix_vectors = 0,
                        .shmem_region = 2,
/*(V) Index to shmem region */
                        .shmem_protocol = JAILHOUSE_SHMEM_PROTO_UNDEFINED,   /*(VI)
Change to xx_PROTO_VETH if use ivshmem-net*/
                        .domain = 0x0,
                },
        },
};
```

3. Xen

3.1. Overview

XEN project is a hypervisor using a microkernel design, providing services that allow multiple operating systems to run on the same hardware with hardware virtualization capability. It is widely used on Cloud and is also used in embedded systems.

For code downloading and building, see i.MX Yocto Project User's Guide (IMXLXYOCTOUG). For device passthrough, see https://xenbits.xen.org/docs/unstable/misc/arm/passthrough.txt.

3.2. Basic architecture

The following figure shows a basic architecture of Xen with three operating systems running.

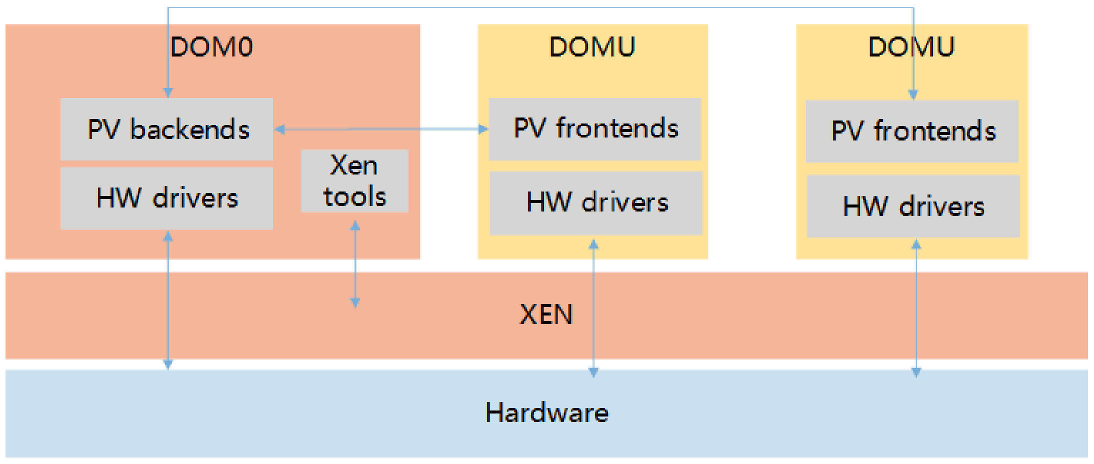


Figure 3. **Basic Xen architecture**

3.3. XEN xl

xl is a user-space tool that runs in Dom0 to manage domains and interact with the Xen hypervisor.

For details about Xen xl usage, see https://xenbits.xen.org/docs/unstable/man/xl.cfg.5.html.

3.4. How to boot multiple operating systems on i.MX 8QuadMax EVK

To boot dual Linux Kernel Distributions, partition the SD card (>=16 GB) with 3 partitions:

- The first FAT partition holds "xen", "Image", and "fsl-imx8qm-mek-dom0.dtb".
- The second and third partitions hold Linux Kernel distributions for Dom0 and DomU.
- Burn imx-boot-imx8qmmek-sd.bin-flash to the SD card offset 32 KB.

See uuu.xen-imx8qmmek in the i.MX 8 L4.14.98 GA release for instructions on how to burn the SD card. Only an SD card boot is supported.

In U-Boot stage:

Boot from the SD card:

```
=> run xenmmcboot
```

Boot from net:

```
run xennetboot
```

After the first Linux OS boots up, execute the following commands:

```
#umount /dev/mmcblk1p3
#xl create /etc/xen/domu-imx8qm-mek.cfg


#xl console DomU
Use `xl list` to see the running domains.
Use `xl console DomU` to attach to the console of DomU
Use `ctrl + ]` to exit console of DomU.
```

To boot Linux and Android Auto operating systems, use domu-imx8qm-mek-androidauto.cfg. See the Android P9.0.0_1.0.2-AUTO-beta documentation.

1. Burn image `fsl-image-validation-imx-imx8qmmek.sdcard` from `L_VIRT_4.14.98_2.0.0_4.11_0.10_ga_images_MX8QMMEK.zip` into the SD card.
2. Download Android Auto prebuilt image package `android_p9.0.0_2.1.0-auto-ga_image_8qmek.tar.gz` from the following location: https://www.nxp.com/webapp/Download?colCode=P9.0.0_2.1.0_AUTO_GA_DEMO_8Q&appType=license

   The prebuilt Android Auto image has a minor issue on EDMA and LPUART, which does not affect the usage of Xen. However, if you want to fix the issue, follow Steps 3-5 to rebuild the Android Auto image.

3. Download Android Auto source code. See the P9.0.0_2.1.0-auto-ga documentation.
4. Download the imx-xen source code from https://source.codeaurora.org/external/imx/imx-xen/log/?h=imx_4.14.98_2.0.0_ga and apply patches under directory patches to Android Auto Linux kernel source code.

   a) `0001-MLK-21443-dmaengine-fsl-edma-v3-clear-pending-irq-be.patch`

   b) `0001-MLK-21445-serial-fsl_lpuart-do-HW-reset-for-communic.patch`
5. Follow the P9.0.0_2.1.0-auto-ga documentation to build Android images.
6. Use UUU to burn images from the Android images into the board's eMMC with the following command:
   ```
   sudo ./uuu_imx_android_flash.sh -f imx8qm -d xen.
   ```
7. Copy the `spl-imx8qm-xen.bin` file in the package above into the SD card's FAT partition.
8. Use the `dd` command to burn the `u-boot-imx8qm-xen-dom0.imx` file in the package above into the SD card:
   ```
   sudo dd if=u-boot-imx8qm-xen-dom0.imx of=/dev/sdx seek=32 bs=1k && sync
   ```
9. Insert the SD card on the MEK board, and power on the board.
10. Press any key to enter U-Boot console, and execute the following commands:
    ```
    setenv domu-android-auto yes
    saveenv
    run xenmmcboot
    ```
11. After the board boots into Linux OS, use the following command to start DomU-Android-Auto:
    ```
    xl create /etc/xen/domu-imx8qm-mek-androidauto.cfg
    ```

3.5. Setting up a bridge network

If you need a bridge network, configure the bridge network before starting DomU.

1. insmod modules from /lib/modules/<linux_kernel_version>/.
   ```
   insmod xen-netback.ko
   insmod stp.ko
   insmod ipv6.ko
   insmod bridge.ko
   ```
2. Configure the bridge network.
   ```
   brctl addbr br0
   brctl addif br0 eth0
   ifconfig eth0 0.0.0.0
   udhcpc -i br0
   ```
3. Uncomment this line in /etc/xen/domu-*.cfg that you use.
   ```
   #vif = [ 'bridge=br0' ]
   ```

4. Create DomU.

4. Revision history

Table 1. **Revision history**

Revision number	Date	Substantive changes
0	02/2019	Initial release
1	05/2019	Updated Xen Android setup and Jailhouse inmate Linux setup for the Linux L4.14.98_2.0.0_ga and Android P9.0.0_2.1.0-auto-ga releases.

How to Reach Us:

Home Page:
nxp.com

Web Support:
nxp.com/support

Document Number: IMXVIRTUG
Rev. 1
05/2019