AX 快速启动使用说明

文档版本：**V1.8**

发布日期：**2024/8/26**

目 录

前 言 6
修订历史 7
1 快速启动基本概念 8
1.1 基本概念 8
1.2 优化 8
1.2.1 系统优化 8
1.2.2 APP 优化 9
1.2.3 KO 全部编译进入内核 9
1.2.4 快启应用依赖的 so 放到 ramdisk 中 9
1.2.5 RISCV 搬运 ramdisk 9
1.3 系统 DDR 内存的划分 10
1.3.1 DDR 整体分布 10
1.3.2 RISCV 的 DDR 空间调整 10
1.4 FLASH 存储空间的划分 11
1.4.1 FLASH 整体分布 11
1.4.2 RISCV 的 FLASH 空间调整 12
2 RTT 编译准备工作 13
2.1 RTT 工具链安装 13
2.2 安装 python 与 scons 13
2.3 代码 14
2.4 编译 14
3 RISCV 模块 16

3.1 中断 16
3.2 clock 和 pinmux 16
3.2.1 clock 16
3.2.2 pinmux 16
3.3 UART 17
3.3.1 UART 配置 17
3.3.2 UART pinmux 配置 17
3.3.3 UART clock 配置 17
3.4 GPIO 18
3.4.1 gpio_set_mode 18
3.4.2 gpio_set_value 18
3.4.3 gpio_get_value 19
3.4.4 gpio_set_raw_int_mode 20
3.4.5 gpio_get_raw_int_status 20
3.4.6 gpio_clear_raw_int_status 21
3.5 I2C 21
3.5.1 i2c_init 22
3.5.2 i2c_wrtie_reg 22
3.5.3 i2c_read_reg 23
3.6 PWM 24
3.6.1 pwm_init 24
3.6.2 pwm_start 25
3.6.3 pwm_stop 26
3.6.4 pwm_deinit 26
3.7 mailbox 27
3.7.1 mbox_auto_send_message 27

3.7.2 mbox_register_callback....28
3.7.3 mbox_unregister_callback....29
3.8 timer64....29
3.8.1 t64_get_val....29
3.8.2 t64_calc_dur_us....30
3.8.3 t64_calc_dur_ms....30
3.8.4 t64_udelay(ax_udelay)....31
3.8.5 t64_mdelay(ax_mdelay)....32
4 Sensor....33
4.1 Sensor 配置....33
4.1.1 数据结构....33
AX_SNS_CONFIG_T....33
AX_SNS_CONFIG_PARAM_T....34
AX_SNS_CONFIG_ATTR_T....34
AX_SNS_PIPE_ATTR_T....36
4.2 Sensor 调试....36
5 2A STAT....37
5.1 AE STAT 配置....37
5.1.1 数据结构....37
5.2 AWB STAT 配置....37
6 RISCV log 配置....38
6.1 ax_env.sh 工具....38
6.1.1 写命令....38
6.1.2 读命令....38
6.2 使用 ax_env 在 Linux 侧修改 riscv log 配置....39

6.2.1 串口 log 开关 riscv_uart_log_toggle 39
6.2.2 log 级别设置 riscv_log_level 39
6.2.3 串口通道设置 riscv_uart_channel 39
6.2.4 串口波特率设置 riscv_uart_baudrate 40
7 QSDemo 41
7.1 Pipeline 41
7.2 编译 41
7.3 快启演示 43
8 Engine Demo 45
8.1 Engine 45
8.2 Engine Demo 45

权利声明

注意

您购买的产品、服务或特性等应受商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非商业合同另有约定，本公司对本文档内容不做任何明示或默示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

前 言

适用产品

AX620E 系列产品（AX630C、AX620Q）

适读人群

- 软件开发工程师
- 技术支持工程师

符号与格式定义

符号/格式	说明
Xxx	表示您可以执行的命令行。
☞ 说明/备注：	表示您在使用产品的过程中，我们向您说明的事项。
！ 注意：	表示您在使用产品的过程中，我们需要您注意的事项。

修订历史

文档版本	发布时间	修订说明
V0.1	2023/11/13	文档初版
V1.0	2024/1/24	更新工程名
V1.1	2024/3/12	调整 QSDemo 章节的位置，增加 AOV 的说明，修改 Pipeline
V1.2	2024/3/19	AE STAT 章节修改为 2A STAT，增加 AWB STAT 配置，补充 Sensor 配置
V1.3	2024/5/17	更新 AOV 的章节描述
V1.4	2024/5/30	更新章节 1.4 RISCV 的 FLASH 空间调整注意事项（V1.5 已删除）
V1.5	2024/6/5	更新 QSDemo AOV 演示章节，增加帧率切换的描述
V1.6	2024/6/12	更新 QSDemo 快启演示章节，增加 log 说明
V1.7	2024/6/26	删除 AOV 章节
V1.8	2024/8/26	QSDemo 删除 AOV 演示章节

1 快速启动基本概念

1.1 基本概念

快速启动目的是为了减少系统的启动时间，以减少系统功耗的目的，主要包括下面 3 个功能。

- 快速抓图像，目前的方案是在 rtos 上实现；
- 对于抓取的图像进行检测，目前方案是在 linux 上实现；
- 对于抓取的图像进行编码输出，这部分目前也是在 linux 上实现的；

该文档以 SDK 的 AX620Q_fastnor_arm32_k419 工程作为参考，说明下快速启动的流程。快速启动基本流程如下图：

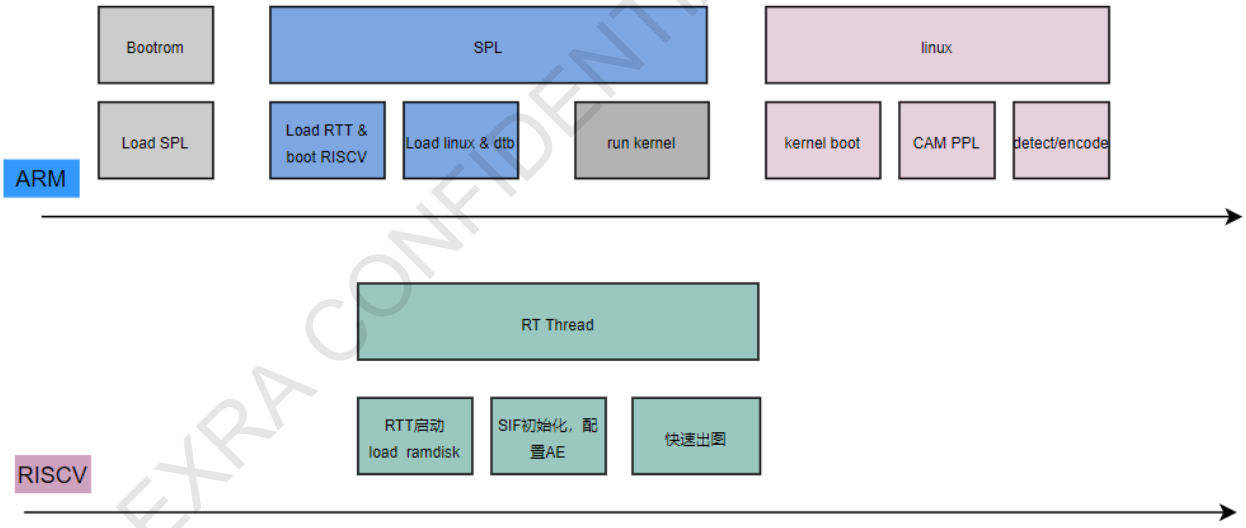

图1-1 快启流程

1.2 优化

1.2.1 系统优化

build/projects/ AX620Q_fastnor_arm32_k419/ project.mak 的宏

AX_BOOT_OPTIMIZATION_SUPPORT，控制优化，默认已经打开了。

```
AX_BOOT_OPTIMIZATION_SUPPORT := TRUE
```

图1-2 启动优化

1.2.2 APP 优化

快启的 Demo 是 qsdemo_s，采用链接 sdk 的静态库编译。

APP 自动启动：修改 build/projects/AX620Q_fastnor_arm32_k419/rootfs/etc/init.d/rcS

```
/etc/init.d/axsyslogd start
/etc/init.d/axklogd start
echo done > /dev/kmsg
export PATH="/bin:/sbin:/usr/bin:/usr/sbin:/opt/bin:/opt/usr/bin:/opt/scripts:/soc/bin:/soc/scripts:/usr/local/bin"
export LD_LIBRARY_PATH="/usr/local/lib:/usr/lib:/opt/lib:/opt/usr/lib:/soc/lib"

if [ "$DEBUG_BOOT_TIME" = "true" ]; then
    rcS_end=`devmem 0x4820000`
    devmem 0x938 32 $rcS_end
fi

/opt/bin/qsdemo_s &
insmod /soc/ko/spi-axera-module.ko
mount -t jffs2 /dev/mtdblock10 /customer
```

图1-3 APP 自动启动

1.2.3 KO 全部编译进入内核

KO 编译进内核，Linux 启动时执行，比 insmod 快。

1.2.4 快启应用依赖的 so 放到 ramdisk 中

采用 ramdisk 后 so 库在 DDR，加快执行响应时间。

1.2.5 RISCV 搬运 ramdisk

Linux 启动过程中 RISCV 同时搬运 ramdisk。不用等待 ramdisk 搬运完了再启动 Linux，并行执行节省了从 flash 读取 RAMDISK 的时间。

1.3 系统 DDR 内存的划分

1.3.1 DDR 整体分布

系统内存划分为 5 部分，参考下图：

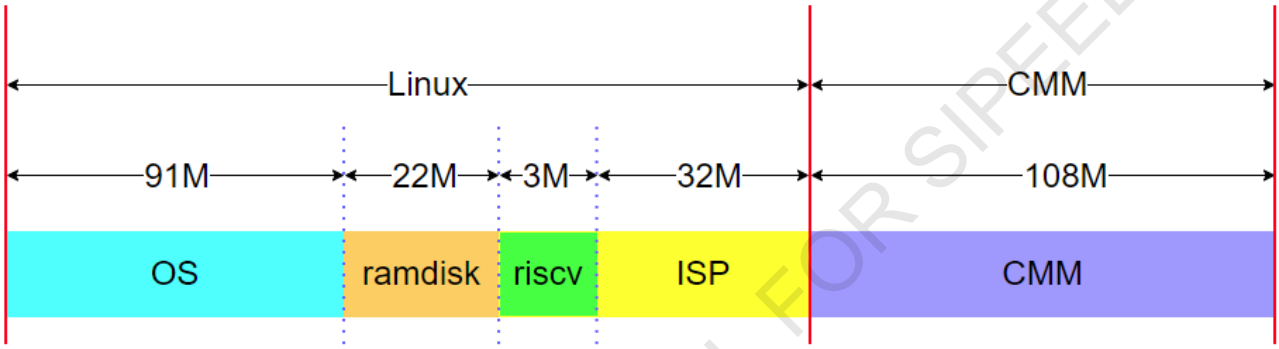

图1-4 DDR 分布示意图

其配置可以参考 build/projects/AX620Q_fastnor_arm32_k419/ project.mak 中的配置。
目前的配置大小和功能参考下面的表格：

表1-1 DDR 分布信息

区域名称	大小 (MB)	用途
OS(包含 ramdisk/riscv/ISP)	148	Linux 系统使用的内存
ramdisk	22	Ramdisk 所占用的空间
riscv	3	Riscv 的 rtos 占用的内存空间
ISP	32	在快速启动的时候，riscv 侧 ISP 运行保存的图像内存
CMM	108	Linux 的媒体内存

1.3.2 RISCV 的 DDR 空间调整

如 1.3.1 介绍，RISCV 占用 3M 空间，如需调整可修改 build/projects/AX620Q_fastnor_arm32_k419/project.mak 的宏 RISCV_MEM_SIZE_MB

```
RISCV_MEM_SIZE_MB                := 3 #MB
```

图1-5 RISCV DDR 空间

比如将上图的宏 RISCV_MEM_SIZE_MB 从 3 改成 4 以后，RISCV 空间将从 3M 变成 4M。对应的 Linux OS 内存大小将会减少 1M。

1.4 FLASH 存储空间的划分

1.4.1 FLASH 整体分布

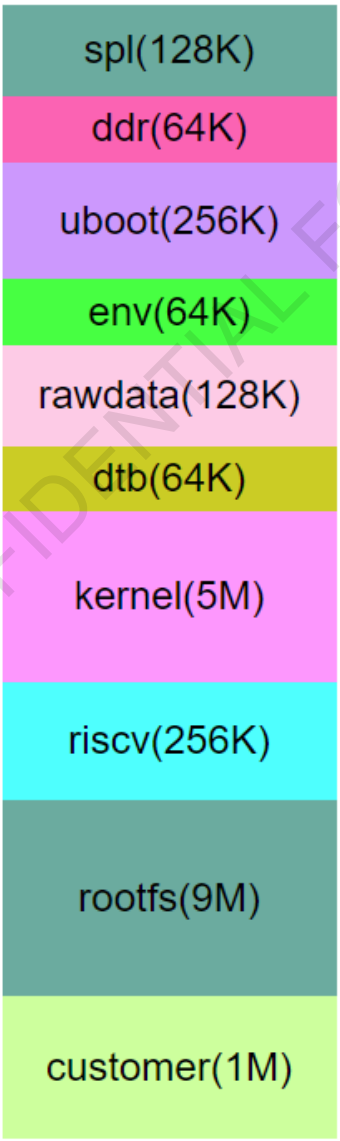

图1-6 FLASH 分布示意图

其配置可以参考 build/projects/AX620Q_fastnor_arm32_k419/ project.mak 中的配置。

表1-2 FLASH 分布信息

区域名称	大小	用途
SPL	128K	存放 SPL 启动程序
DDR	64K	存放 DDR 初始化信息
UBOOT	256K	用于存放 UBOOT 程序
ENV	64K	用于存放 UBOOT 使用的 env
RAWDATA	128K	用于存放客户使用的 env 的空间
DTB	64K	Linux dts 文件
Kernel	5M	Linux 系统
RISCV	256K	RISCV 固件
Rootfs	9M	rootfs 文件系统
customer	1M	用户可用空间

1.4.2 RISCV 的 FLASH 空间调整

RISCV 的 FLASH 空间在够用的情况下，一般不建议您修改。如果 riscv 压缩后的 rtthread_signed.bin 文件大于 256K，可以修改 build/projects/AX620Q_fastnor_arm32_k419/project.mak 的宏 RISCV_PARTITION_SIZE

```
49 RISCV_PARTITION_SIZE   := 256K
```

图1-7 RISCV FLASH 空间

比如将 RISCV_PARTITION_SIZE 改成 384K，RISCV 的 FLASH 空间将增加 128K，变成 384K。

！注意：

FLASH 是 32K 对齐的，所以调整间隔应当是 32K 的倍数。

2 RTT 编译准备工作

参照 SDK 使用说明，安装对应依赖项以后，再安装如下内容。

2.1 RTT 工具链安装

riscv 需要自己的 gcc 工具链编译 rtthread 代码。

步骤1 将 riscv64-elf-gnu-amd64.tar.gz 解压放到/usr/local/目录/usr/local/riscv64-elf-gnu-amd64

☞ 说明：

如您是第一次使用，可向对应技术支持人员获取该包

步骤2 工具链安装完成后，在任意目录下执行 `riscv64-unknown-elf-gcc -v` 命令。如果工具链安装成功，系统将会显如下工具链的版本号：

```
Using built-in specs.
COLLECT_GCC=riscv64-unknown-elf-gcc
COLLECT_LTO_WRAPPER=/usr/local/riscv64-elf-gnu-amd64/bin/../libexec/gcc/riscv64-unknown-elf/10.2.0/lto-wrapper
Target: riscv64-unknown-elf
Configured with: /ldhome/software/toolsbuild/slave2/workspace/Toolchain/release-riscv-0/build/../source/riscv/riscv-gcc/configure --target=riscv64-unknown-elf --with-gmp=
/ldhome/software/toolsbuild/slave2/workspace/Toolchain/release-riscv-0/lib-for-gcc-x86_64-linux/ --with-mpfr=/ldhome/software/toolsbuild/slave2/workspace/Toolchain/releas
e-riscv-0/lib-for-gcc-x86_64-linux/ --with-mpc=/ldhome/software/toolsbuild/slave2/workspace/Toolchain/release-riscv-0/lib-for-gcc-x86_64-linux/ --with-libexpat-prefix=/ld
home/software/toolsbuild/slave2/workspace/Toolchain/release-riscv-0/lib-for-gcc-x86_64-linux/ --with-pkgversion='T-HEAD RISCV Tools V2.0.1 B20210512' --enable-libgcctf --
prefix=/ldhome/software/toolsbuild/slave2/workspace/Toolchain/release-riscv-0/install --disable-shared --disable-threads --enable-languages=c,c++ --with-system-zlib --ena
ble-tls --with-newlib --with-sysroot=/ldhome/software/toolsbuild/slave2/workspace/Toolchain/release-riscv-0/install/riscv64-unknown-elf --with-native-system-header-dir=/i
nclude --disable-libmudflap --disable-libssp --disable-libquadmath --disable-libgomp --disable-nls --disable-tm-clone-registry --src=../../source/riscv/riscv-gcc --enable
-multilib --with-abi=lp64d --with-arch=rv64gcxthead 'CFLAGS_FOR_TARGET=-Os   -mcmodel=medany' 'CXXFLAGS_FOR_TARGET=-Os   -mcmodel=medany'
Thread model: single
Supported LTO compression algorithms: zlib
gcc version 10.2.0 (T-HEAD RISCV Tools V2.0.1 B20210512)
```

图2-1 工具链版本号

2.2 安装 python 与 scons

rtthread 使用 scons 编译，需安装 python 和 scons。

步骤1 安装 python，建议 3.10 或以上版本。

```
sudo apt-get install python3
```

步骤2 安装 scons。

```
sudo apt-get install scons
```

2.3 代码

参照 SDK 使用说明解压 SDK 包以后，AX620E_SDK_V…/riscv 目录即 RISCV 代码。

- applications：应用程序及 main 函数入口
- config：配置文件
- drivers：axera 的外设驱动
- libs：.a 库
- risc-v：riscv 的头文件
- rt-thread：rtthread 系统的头文件及 tool 工具

2.4 编译

编译前确认工具链所在目录，在 AX620E_SDK_V…/riscv/config/rtconfig.py 修改工具链目录。

```
 6 ARCH        ='risc-v'
 7 CPU         ='e9xx'
 8 CPUNAME     ='e907'
 9 VENDOR      ='t-head'
10 CROSS_TOOL  ='gcc'
11
12 if os.getenv('RTT_CC'):
13     CROSS_TOOL = os.getenv('RTT_CC')
14
15 if  CROSS_TOOL == 'gcc':
16     PLATFORM    = 'gcc'
17     EXEC_PATH   = r'/usr/local/riscv64-elf-gnu-amd64/bin'
18 else:
19     print ('Please make sure your toolchains is GNU GCC!')
20     exit(0)
```

图2-2 工具链目录修改

编译命令和其他模块编译命令相同，进入 AX620E_SDK_V…/riscv 目录：

```
make p=AX620Q_fastnor_arm32_k419 clean all install.
```

生成的 bin 在

AX620E_SDK_V…/build/out/AX620Q_fastnor_arm32_k419/images/rtthread_signed.bin

3 RISCV 模块

下面介绍中断/clock/pinmux/UART/GPIO/I2C/PWM/mailbox/timer64/sensor 等模块。

3.1 中断

RISCV 和 ARM 对于外设中断有两种处理机制：一种是同时进入 RISCV 和 ARM，只要一边清中断，另一边就不会再触发中断，比如 DMAPER 中断；一种是分别触发 RISCV 和 ARM 的中断，清中断和屏蔽中断可以在 RISCV 和 ARM 分别单独配置，比如 GPIO 中断。

☞ 说明：

RISCV 和 ARM 都可以通过 APB 总线访问外设，外设的中断同时接入了 RISVC 和 ARM。

3.2 clock 和 pinmux

RISCV 和 ARM 都可以设置 clock 和 pinmux，所以在设置时需要注意避免冲突。

3.2.1 clock

RISCV 侧的 clock 在各个模块里面初始化，即使用相关模块之前开启相关的 clock。

ARM 侧在 Linux 启动时，会在 kernel/linux/linux-4.19.125/drivers/clk/axera/clk-ax620e.c 统一初始化一遍 clock，对于不使用的 clock 会关闭。所以，RISCV 在确认使用相关模块时，要避免 Linux 关闭对应 clock。

3.2.2 pinmux

RISCV 在 riscv/drivers/pinmux/drv_pinmux.c 统一初始化 pinmux,如果想使用对应 pinmux 的 function 需要提前在这里初始化。

ARM 侧在 Linux 启动时
build/projects/AX620Q_fastnor_arm32_k419/pinmux/AX620Q_DEMO_pinmux.h 统一初始化

pinmux，如果 RISCV 需要一直使用 pinmux 对应的 function 则 Linux 也应做一致的配置。

3.3 UART

头文件 drv_uart.h.

UART 用于打印 RISCV log.

3.3.1 UART 配置

可以配置 UART 为 uart0/uart1/uart2/uart3/uart4/uart5，设置波特率为 9600/19200/38400/57600/115200/921600 等。

riscv 串口 log，默认使用 UART1，波特率 921600。

☞ 说明：

UART log 的配置详细说明见第七章。

注意：

UART0 已经用作 ARM 侧打印串口 log，如您配置 RISCV 的串口为 uart_0 则需要把 ARM 侧的串口 log 设置其它通道。

3.3.2 UART pinmux 配置

按 3.3.1 配置以后还需要 riscv/drivers/pinmux/drv_pinmux.c 配置对应 UART 通道的 pinmux。

如 3.2.2 说明 ARM 侧会在 build/projects/AX620Q_fastnor_arm32_k419/pinmux/AX620Q_DEMO_pinmux.h 重新配置 pinmux，使用对应 UART 前需要确认 ARM 侧没有用到 UART 的 pin，并且在 AX620Q_DEMO_pinmux.h 做好对应设置。

3.3.3 UART clock 配置

按 3.3.1 配置以后 RISCV 已经设置好了 clock，但是 ARM 侧会在 kernel/linux/linux-5.15.73/drivers/clk/axera/clk-ax620e.c 重新设置 clock。

如 3.2.1 说明使用对应 UART 前需要确认 ARM 侧没有关闭对应 UART 的 clock。

3.4 GPIO

头文件 drv_gpio.h

注意：

使用 GPIO 前需要如 3.2.2 说明将对应 pinmux 配置成 gpio，还需要避免与 Linux 配置冲突。

3.4.1 gpio_set_mode

【描述】

设置输入输出

【语法】

```
int gpio_set_mode(uint32_t gpio_num, gpio_mode_e mode);
```

【参数】

参数名称	描述	输入/输出
gpio_num	gpio 号	输入
Mode	枚举 GPIO 模式	输入

【返回值】

返回值	描述
0	成功
非 0	失败，返回错误码

3.4.2 gpio_set_value

【描述】

GPIO 为 ouput 模式时，设置输出电平高低

【语法】

```
int gpio_set_value(uint32_t gpio_num, gpio_value_e value);
```

【参数】

参数名称	描述	输入/输出
gpio_num	gpio 号	输入
value	gpio_low / gpio_high	输入

【返回值】

返回值	描述
0	成功
非 0	失败，返回错误码

3.4.3 gpio_get_value

【描述】

GPIO 为 input 模式时，读取输入电平高低

【语法】

```
gpio_value_e gpio_get_value(uint32_t gpio_num);
```

【参数】

参数名称	描述	输入/输出
gpio_num	gpio 号	输入

【返回值】

返回值	描述
gpio_low	当前输入为低电平
gpio_high	当前输入为高电平

3.4.4 gpio_set_raw_int_mode

【描述】

设置 GPIO raw 中断模式

【语法】

```
int gpio_set_raw_int_mode(uint32_t gpio_num, gpio_int_mode_e int_mode);
```

【参数】

参数名称	描述	输入/输出
gpio_num	gpio 号	输入
int_mode	raw 中断模式	输入

【返回值】

返回值	描述
0	成功
非 0	失败，返回错误码

☞ **说明：**

设置 raw 中断时，已经将中断 mask 了，中断控制器不会收到中断，但可以通过读取 raw status 读取中断状态通过 eoi 清除中断状态。

3.4.5 gpio_get_raw_int_status

【描述】

当 GPIO 设置成 raw 中断模式后，读取 raw 中断状态

【语法】

```
int gpio_get_raw_int_status(uint32_t gpio_num);
```

【参数】

参数名称	描述	输入/输出
gpio_num	gpio 号	输入

【返回值】

返回值	描述
0	没有 raw 中断
1	读取到 raw 中断

3.4.6 gpio_clear_raw_int_status

【描述】

读取到 GPIO raw 中断后，清除中断状态

【语法】

```
int gpio_clear_raw_int_status(uint32_t gpio_num);
```

【参数】

参数名称	描述	输入/输出
gpio_num	gpio 号	输入

【返回值】

返回值	描述
0	清除中断成功
非 0	清除中断失败

3.5 I2C

头文件 drv_i2c.h.

注意：

如 3.2 说明，ARM 侧 Linux 会重新初始化 pinmux 设置，当 RISCV 要使用对应 pin 做 i2c 时需要在 Linux 侧也设置成 i2c 模式。

3.5.1 i2c_init

【描述】

初始化一个 i2c 通道

【语法】

```
int i2c_init(i2c_channel_e channel, char *name, i2c_freq_e freq);
```

【参数】

参数名称	描述	输入/输出
channel	i2c 通道: i2c_channel_0 ~ i2c_channel_7	输入
name	i2c 名字：注册以后使用 i2c 名字获取 i2c 设备	输入
freq	i2c 频率	输入

【返回值】

返回值	描述
0	初始化成功
非 0	初始化失败

3.5.2 i2c_wrtie_reg

【描述】

i2c master 设备往 i2c slave 设备寄存器写数据

【语法】

```
int i2c_wrtie_reg(struct rt_i2c_bus_device *i2c_dev, uint8_t addr, uint32_t reg, uint8_t* src, uint32_t len, uint32_t is_reg_addr_16bits);
```

【参数】

参数名称	描述	输入/输出
i2c_dev	i2c master 设备	输入
addr	i2c slave 设备地址	输入
reg	i2c slave 设备寄存器地址	输入
src	往 i2c slave 设备寄存器写入的数据	输入
len	往 i2c slave 设备寄存器写入数据的长度	输入
is_reg_addr_16bits	i2c slave 设备寄存器地址是否为 16 位	输入

【返回值】

返回值	描述
0	写入成功
非 0	写入失败

3.5.3 i2c_read_reg

【描述】

i2c master 设备从 i2c slave 设备寄存器读取数据

【语法】

```
int i2c_read_reg(struct rt_i2c_bus_device *i2c_dev, uint8_t addr, uint32_t reg,
uint8_t* dst, uint32_t len, uint32_t is_reg_addr_16bits);
```

【参数】

参数名称	描述	输入/输出
i2c_dev	i2c master 设备	输入
addr	i2c slave 设备地址	输入
reg	i2c slave 设备寄存器地址	输入

参数名称	描述	输入/输出
dst	从 i2c slave 设备寄存器读取的数据	输出
len	从 i2c slave 设备寄存器读取数据的长度	输入
is_reg_addr_16bits	i2c slave 设备寄存器地址是否为 16 位	输入

【返回值】

返回值	描述
0	读取成功
非 0	读取失败

3.6 PWM

头文件 drv_pwm.h

pwm 使用 pin 脚输出指定周期和占空比的波形。

注意：

- 如 3.2 说明，当 RISCV 要使用对应 pin 做 pwm 时，需要在 RISCV 和 Linux 侧都设置成 pwm 模式。
- pwm00 ~ pwm23 总共 12 个 pin，多数已经被用作其他功能，如需再设置成 pwm 需要确保不再使用其他功能。比如 PWM11 和 UART3_RXD 为同一个 pin，用作 pwm 以后就不能使用 UART3。

3.6.1 pwm_init

【描述】

初始化一个 pwm 功能，配 clock、pinmux、pwm 相关信息

【语法】

```
int pwm_init(pwm_e pwm, uint32_t freq_hz, uint8_t duty);
```

【参数】

参数名称	描述	输入/输出
pwm	pwm 号: pwm_00 ~ pwm_23	输入
freq_hz	pwm 频率	输入
duty	pwm 占空比	输入

【返回值】

返回值	描述
0	初始化成功
非 0	初始化失败

注意:

pwm23 已经用作调节 VDDCORE 电压，不可设置。如果设置 API 会返回错误。

3.6.2 pwm_start

【描述】

开始输出 pwm 信号

【语法】

```
int pwm_start(pwm_e pwm);
```

【参数】

参数名称	描述	输入/输出
pwm	pwm 号: pwm_00 ~ pwm_23	输入

【返回值】

返回值	描述
0	start 成功
非 0	start 失败

3.6.3 pwm_stop

【描述】

关闭 pwm 信号

【语法】

```
int pwm_stop(pwm_e pwm);
```

【参数】

参数名称	描述	输入/输出
pwm	pwm 号: pwm_00 ~ pwm_23	输入

【返回值】

返回值	描述
0	stop 成功
非 0	stop 失败

3.6.4 pwm_deinit

【描述】

注销一个 pwm 信号，回收相关资源

【语法】

```
int pwm_deinit(pwm_e pwm);
```

【参数】

参数名称	描述	输入/输出
pwm	pwm 号: pwm_00 ~ pwm_23	输入

【返回值】

返回值	描述
0	注销成功
非 0	注销失败

3.7 mailbox

头文件 drv_mailbox.h

mailbox 用于 RISCV 和 ARM 通信，通信同时携带 32 字节数据

☞ 说明：

- 为通信数据处理一致，mailbox 通信统一携带 32 字节数据；
- 32 个字节信息定义为 mbox_msg_t，其中 id 代码不同的信息类型，其余 31 个字节为数据。

3.7.1 mbox_auto_send_message

【描述】

RISCV 发送一个 32 字节的 message 给 ARM

【语法】

```
int mbox_auto_send_message(mbox_msg_t *msg);
```

【参数】

参数名称	描述	输入/输出
msg	数据结构 mbox_msg_t 第一个字节为 message id,剩余 31 个字节为数据	输入

【返回值】

返回值	描述
0	发送成功
非 0	发送失败

☞ 说明：

message id 在 drv_mailbox.h 有统一记录，增加通信 ID 以后可以记录到 drv_mailbox.h

```
16 #define MBOX_ID_SENSOR_RECV     0x00
17 #define MBOX_ID_SENSOR_SEND     0x05
18 #define MBOX_ID_VERIFY          0xff
```

3.7.2 mbox_register_callback

【描述】

注册一个回调函数，用于接收和处理 ARM 侧发过来的 mailbox 信息

【语法】

```
int mbox_register_callback(mbox_callback_t callback, void *data, uint8_t id);
```

【参数】

参数名称	描述	输入/输出
callback	回调函数	输入
data	回调函数参数	输入
id	message id	输入

【返回值】

返回值	描述
0	注册成功
非 0	注册失败

☞ 说明：

- 回调函数在中断处理函数内部执行不可调度，不建议执行耗时长的任务；
- 如 3.7.1 说明，增加通信 ID 以后可以记录到 drv_mailbox.h 避免 ID 冲突。

3.7.3 mbox_unregister_callback

【描述】

注销一个 ID 的回调函数，回收相关资源

【语法】

```
int mbox_unregister_callback(uint8_t id);
```

【参数】

参数名称	描述	输入/输出
id	message id	输入

【返回值】

返回值	描述
0	注销成功
非 0	注销失败

3.8 timer64

头文件 drv_timer64.h

24M 的 timer64 计数值可以换算时间间隔

3.8.1 t64_get_val

【描述】

读取 timer64 计数值

【语法】

```
uint64_t t64_get_val(void);
```

【参数】

参数名称	描述	输入/输出
void	无	无

【返回值】

返回值	描述
uint64_t	此刻 timer64 计数值

3.8.2 t64_calc_dur_us

【描述】

将两个 timer64 计数值间隔换算成 us

【语法】

```
uint32_t t64_calc_dur_us(uint64_t start, uint64_t end);
```

需要说明最大值

【参数】

参数名称	描述	输入/输出
start	timer64 计数起始值	输入
end	timer64 计数结束值	

【返回值】

返回值	描述
uint32_t	时间间隔 us

3.8.3 t64_calc_dur_ms

【描述】

将两个 timer64 计数值间隔换算成 ms

【语法】

```
uint32_t t64_calc_dur_ms(uint64_t start, uint64_t end);
```

说明最大值

【参数】

参数名称	描述	输入/输出
start	timer64 计数起始值	输入
end	timer64 计数结束值	

【返回值】

返回值	描述
uint32_t	时间间隔 ms

3.8.4 t64_udelay(ax_udelay)

【描述】

使用计数值进行 delay

【语法】

```
void t64_udelay(uint32_t us);
```

【参数】

参数名称	描述	输入/输出
us	delay in us	输入

【返回值】

返回值	描述
void	无

☞ **说明：**

- t64_udelay 为忙等不会睡眠；
- 考虑代码可读性，建议您使用 ax_udelay 代替 t64_udelay，执行内容一样的。

3.8.5 t64_mdelay(ax_mdelay)

【描述】

使用计数值进行 delay

【语法】

```
void t64_mdelay(uint32_t ms);
```

【参数】

参数名称	描述	输入/输出
ms	delay ms	输入

【返回值】

返回值	描述
void	无

☞ **说明：**

- t64_mdelay 为忙等不会睡眠；
- 考虑代码可读性，建议您使用 ax_mdelay 代替 t64_mdelay，执行内容一样的。

4 Sensor

4.1 Sensor 配置

RISCV 端默认支持 os04a10，但是可以修改 ax_vin_config.h 中 AX_SNS_CONFIG_T，适配不同的 sensor。

4.1.1 数据结构

AX_SNS_CONFIG_T

【描述】

sensor/mipi/dev/pipe 等模块的配置信息

【语法】

```
typedef struct _AX_SNS_CONFIG_T_ {
    AX_U8              nSensorCnt;
    AX_U32             nCmmBase;
    AX_U32             nCmmEnd;
    AX_SNS_CONFIG_PARAM_T config[AX_SNS_CNT_MAX];
} AX_SNS_CONFIG_T;
```

【参数】

参数名称	描述
nSensorCnt	Sensor 个数，最大支持双摄
nCmmBase	RISCV 使用的物理内存起始地址

参数名称	描述
nCmmEnd	RISCV 使用的物理内存结束地址
config	Sensor 的配置参数，最大支持双摄

AX_SNS_CONFIG_PARAM_T

【描述】

sensor/mipi/dev/pipe 等模块的参数信息

【语法】

```
typedef struct _AX_SNS_CONFIG_PARAM_T_ {
    AX_SNS_CONFIG_ATTR_T tSnsAttr;
    AX_U8                mLaneNum;
    AX_U8                nDevId;
    AX_SNS_PIPE_ATTR_T tPipeAttr;
} AX_SNS_CONFIG_PARAM_T;
```

【参数】

参数名称	描述
tSnsAttr	Sensor 属性信息
mLaneNum	Mipi lane num
nDevId	Dev id
tPipeAttr	Pipe 属性信息

AX_SNS_CONFIG_ATTR_T

【描述】

sensor 的属性信息

【语法】

```
typedef struct _AX_SNS_CONFIG_ATTR_T_ {
    AX_SENSOR_REGISTER_FUNC_T  *pSnsHdl;
    AX_U8              mSnsId;
    AX_SNS_HDR_MODE_E   eSnsMode;
    AX_U8              mClkId;
    AX_U8              nI2cDev;
    AX_U8              nI2cAddr;
    AX_U8              nInitFps;
    AX_U8              nConvergeFps;
    AX_U32             nSettingIndex;
} AX_SNS_CONFIG_ATTR_T;
```

【参数】

参数名称	描述
pSnsHdl	Sensor 的回调函数对象
mSnsId	Sensor id
eSnsMode	Sensor HDR 模式选择（Linear/HDR_2X/HDR_3X），enum 类型
mClkId	Clock id
nI2cDev	I2c bus num
nI2cAddr	Sensor i2c 地址
nInitFps	Sensor 初始化帧率
nConvergeFps	Sensor 收敛之后的帧率
nSettingIndex	用于进行 Sensor 初始化序列的选择，在分辨率和帧率相同时，配置不同的 nSettingIndex 对应不同的初始化序列；其他情况，nSettingIndex 默认配置为 0，可通过 nWidth、nHeight 和 nFrameRate 进行初始化序列的选择。

AX_SNS_PIPE_ATTR_T

【描述】

pipe 的属性信息

【语法】

```
typedef struct _AX_SNS_PIPE_ATTR_T_ {
    AX_U32              nWidth;
    AX_U32              nHeight;
    AX_RAW_TYPE_E       eRawType;
    AX_IMG_FORMAT_E     ePixelFmt;
} AX_SNS_PIPE_ATTR_T;
```

【参数】

参数名称	描述
nWidth	Pipe 的宽
nHeight	Pipe 的高
eRawType	Pipe 的 Raw type
ePixelFmt	Pipe 的 format

4.2 Sensor 调试

参考 12 - AX Sensor 调试指南.docx

☞ 说明：

下列子项列出 RISCV 中存在差异的函数接口。

5 2A STAT

5.1 AE STAT 配置

RISCV 端 AE STAT 模块配置，具体参数可以修改 ax_ae_stat_config.h 中 AX_ISP_IQ_AE_STAT_PARAM_T。

5.1.1 数据结构

参考 15 - AX ISP API 文档.docx 中 3.3 数据结构

☞ 说明：

下列子项列出 RISCV 中存在差异的结构体成员。

eAEStatPos

只支持 before hdr

nGridMode

只支持 4ch

nHistMode

只支持 4ch

5.2 AWB STAT 配置

由于 AWB STAT 无法配置到 IFE 段，所以在 RISCV 中将 AE STAT 的统计经过转换后作为 AWB STAT 的统计，格子划分相关的参数配置和 AE STAT 相同。

6 RISCV log 配置

riscv 的 log 通过串口打印，串口的设置可以在 Linux 侧通过 ax_env.sh 工具配置。

6.1 ax_env.sh 工具

Linux 通过 ax_env.sh 写入 flash 的 RAWDATA 分区，在 SPL 启动时把分区的内容读到 ocm，riscv 从 ocm 读取分区信息。

6.1.1 写命令

在 Linux 命令行执行：ax_env.sh set name value.重启系统后在 riscv 生效。

```
/root # ax_env.sh set name1 para1
/root #
/root # ax_env.sh set name2 para2
/root #
/root # ax_env.sh set name3 hello world
```

6.1.2 读命令

在 Linux 命令行执行：ax_env.sh print 可以立即查看或者重启系统后查看设置了哪些值。

```
/root # ax_env.sh print
name1=para1
name2=para2
name3=hello world
/root #
/root # ax_env.sh print name3
name3=hello world
```

6.2 使用 ax_env 在 Linux 侧修改 riscv log 配置

6.2.1 串口 log 开关 riscv_uart_log_toggle

用法举例，关闭串口 log： ax_env.sh set riscv_uart_log_toggle off

riscv_uart_log_toggle	
on	开启串口 log
off	关闭串口 log

6.2.2 log 级别设置 riscv_log_level

用法举例，设置 log 级别为 info: ax_env.sh set riscv_log_level info

☞ 说明：
log 级别只对 axera 头文件 ax_log.h 定义的 AX_LOG_DGB/ AX_LOG_INFO/ AX_LOG_NOTICE/AX_LOG_WARN/ AX_LOG_ERROR/ AX_LOG_CRIT 有效，对于其他 log api 无效。

riscv_log_level	
debug	设置 log 级别为 debug，只打印 debug 及以上级别的 log
info	设置 log 级别为 info，只打印 info 及以上级别的 log
notice	设置 log 级别为 notice，只打印 notice 及以上级别的 log
warn	设置 log 级别为 warning，只打印 warning 及以上级别的 log
error	设置 log 级别为 error，只打印 error 及以上级别的 log
critical	设置 log 级别为 critical，只打印 critical 及以上级别的 log

6.2.3 串口通道设置 riscv_uart_channel

用法举例，设置 uart 为通道 1：ax_env.sh set riscv_uart_channel uart_1

注意：

1. 修改串口通道需要注意 pinmux 和 clk 对应设置正确；
2. UART0 已经用作 ARM 侧打印串口 log，如您配置 RISCV 的串口为 uart_0 则需要把 ARM 侧的串口 log 设置其它通道。

riscv_uart_channel	
uart_0	使用 uart0 打印串口 log
uart_1	使用 uart1 打印串口 log
uart_2	使用 uart2 打印串口 log
uart_3	使用 uart3 打印串口 log
uart_4	使用 uart4 打印串口 log
uart_5	使用 uart5 打印串口 log

6.2.4 串口波特率设置 riscv_uart_baudrate

用法举例，设置串口波特率为 115200：ax_env.sh set riscv_uart_baudrate 115200

riscv_uart_baudrate	
9600	设置串口波特率为 9600
19200	设置串口波特率为 19200
38400	设置串口波特率为 38400
57600	设置串口波特率为 57600
115200	设置串口波特率为 115200
921600	设置串口波特率为 921600

7 QSDemo

本章介绍快启和 AOV 应用 QSDemo 的 Pipeline 和使用。

7.1 Pipeline

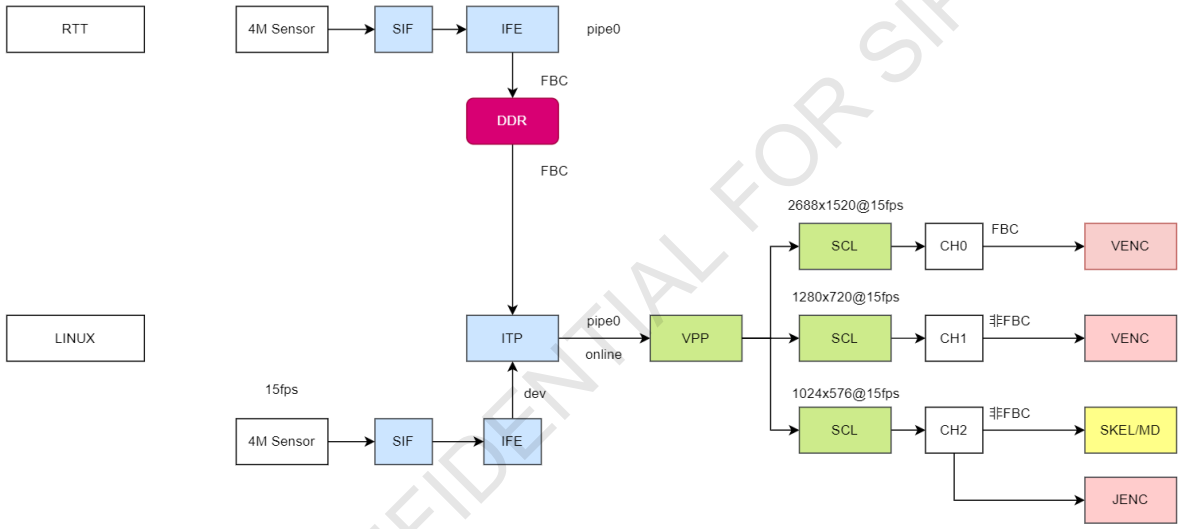

图7-1 QSDemo Pipeline

需要注意：

1. itp online vpp
2. FBC 的配置
3. 2M/4M/8M 的 sensor pipeline 是一样的，只是主码流的分辨率不一样，为了便于预览 VENC 编码后数据通过 rtsp 推流。
4. SKEL/MD 是指智能人形检测或者移动检测

7.2 编译

QSDemo 的代码在 app/demo/QSDemo 目录下。QSDemo 支持多种 Sensor，编译的时候

可以指定，不指定默认是 os04a10。

如果只是修改了 QSDemo，需要重新更新板端的固件，需要执行以下步骤：

1. 编译 QSDemo

   cd app/demo/QSDemo

   make p=AX620Q_fastnor_arm32_k419 clean all install

   说明：可以明确指定 sensor

   make p=AX620Q_fastnor_arm32_k419 sensor="os04a10" clean all install

   支持的 sensor 还有：

   sensor="sc850sl"

   sensor="sc500ai"

   sensor="sc200ai"

   sensor="sc200ai sc200ai"

   最后一个支持双 2M 的 sensor。

   如果更改 sensor，则需要到 build 目录下重新全编译，带上 sensor 参数。

   因为不同的 sensor 使用的内存情况不一样，会优化 cmm，os，riscv 的内存配置。

2. 编译 rootfs

   cd rootfs

   make p=AX620Q_fastnor_arm32_k419 image

   QSDemo 的可执行文件打入到:

   build/out/AX620Q_fastnor_arm32_k419/images/rootfs.img

3. AXDL 工具打开 axp 包，在工具的设置界面替换 rootfs.img，然后重新下载。

7.3 快启演示

为了模拟检测到没有人的时候关机到检测到人开机这个过程，QSDemo 增加自动重启的逻辑，需要板端的 customer 分区创建 reboot flag 文件：

```
touch /customer/reboot
```

QSDemo 启动后连续 5 帧检测不到人形就会自动重启 reboot。

QSDemo 会保存第一帧 jepg 图到/tmp/jenc0.jpg

QSDemo 会保存最多 30 帧的视频文件到/tmp/venc0.264 （根据人形出现的情况而定）

- Log 说明：

```
[      356469][qsdemo] Build at Apr  9 2024 16:06:53
[      357278][qsdemo] sys and pool init done
[      357437][qsdemo] set itp online vpp done
[      359692][qsdemo] npu init done
[      359831][qsdemo] vin and mipi init start
[      372365][qsdemo] Vin Post Proc started
[      372505][qsdemo] link init done
[      376628][qsdemo] ivps init done
[      377285][qsdemo] vin and mipi init done
[      377745][qsdemo] vin priv pool init done
[      377937][qsdemo] current lux=98146
[      378366][qsdemo] video recorder init done
[      378654][qsdemo] cam[0] thread start
[      381507][qsdemo] pipe[0] AX_ISP_Create ++
[      387644][qsdemo] venc init done
[      398195][qsdemo] jenc init done
[      406045][qsdemo] pipe[0] AX_ISP_Create --
[      408034][qsdemo] pipe[0] AX_ISP_Open ++
[      423123][qsdemo] detect init done
[      425740][qsdemo] rgn init done
[      491031][qsdemo] pipe[0] AX_ISP_Open --
[      502077][qsdemo] pipe[0] AX_ISP_Start ++
[      502232][qsdemo] pipe[0] AX_ISP_Start --
[      502278][qsdemo] pipe[0] AX_ISP_StreamOn done
[      541541][qsdemo] Detect[0] 1st frame seqno: 5.
[      551723][qsdemo] Jenc[2]: 1st image is saved: /tmp/jenc0.jpg, seqno: 5, pts: 213432, size: 32731
[      561508][qsdemo] Venc[0]: H264 1st seqno: 5.
[      571559][qsdemo] main launch timestamp:  356430
[      571681][qsdemo] round[35] latency result: 1st Raw Image: 231340, 1st Detect Result: 570545, 1st Venc Frame: 561340, main launch: 356430.
[      571722][qsdemo] round[35] yuv latency from boot: 541650.
[      571781][qsdemo] round[35] vin latency sns[0]: started: 72869, opened: 143835, 1st raw ready: 231340, out next: 507592
[      571822][qsdemo] Detect[0]: 1st frame got detected result
[      809879][qsdemo] Init latency: total:145888, sys:905, npu:2426, isp:17591, cam open:124966, detect:53655
[     4232220][qsdemo] Venc[0]: H264 is saved: /tmp/venc0.264, size: 2302361 bytes
[     4709221][qsdemo] rtsp init done
```

QSDemo 启动后会输出如上的 log 信息，

```
[      571681][qsdemo] round[35] latency result: 1st Raw Image: 231340, 1st Detect Result: 570545, 1st Venc Frame: 561340, main launch: 356430.
```

1. 第一个中括号内的数字：时间，单位 us，表示从上电开始到当前的时间间隔，这个时间包含了上电固定延迟 30ms。
2. 1st Raw Image: 231340，表示 riscv 侧第一帧 AE 收敛后输出的 raw 的时间，单位 us

3. 1st Venc Frame: 561340，表示 venc 输出的第一帧编码包的时间，单位 us
4. 1st Detect Result: 570545，表示第一帧检测输出结果的时间，单位 us
5. main launch: 356430，表示从上电开始，到 qsdemo 里 main 函数执行的时间，单位 us

```
[        541541][qsdemo] Detect[0] 1st frame seqno: 5
```

这条 log 表示第一帧 yuv 输出时间 541541，帧号是 5，也可以确认 AE 收敛是第 5 帧收敛。

8 Engine Demo

本章介绍快启和 AOV 应用 QSDemo 的依赖 Engine 和 Engine 的 Demo。

8.1 Engine

RISCV 也实现了一份极轻量的 Engine，API 和 ARM 侧的 Engine 大体一致，开发过程也与 ARM 侧保持一致。进行 RISCV 的 Engine 开发时，用户可以参考《AX ENGINE API 使用说明》查看详细说明。

QSDemo 里调用了 Engine 进行场景检测，用户可以参考 QSDemo 进行二次开发，比如改用具有更多检测能力的模型进行厂商自定义快起场景检测(比如增加宠物、机动车辆等检测能力)。

8.2 Engine Demo

此外，为了方便用户自行修改模型测试，在 riscv/applications/test_engine 目录下准备了一份 Engine 测试 App，其中模型保存为 model.h，测试用图像保存为 image.h。

在 test_algo_det_raw.c 里有完整的实现过程，在单独测试 Engine 这个 Demo 时，需要修改 App 的入口处 riscv/applications/main.h；如果希望以 App 方式测试，则需要打开：

INIT_APP_EXPORT(test_engine)

或以 CMD 模式测试，则需要打开：

MSH_CMD_EXPORT(test_engine, test engine [STACK PRIORITY SLICE])

打开后，App 模式上电就会启动；CMD 模式则需要用户串口输入命令启动。为了避免连续打印干扰，还可以关闭其他 App。

```
[AX][INFO][load_rootfs_thread][51]: loadrootfs finished
[ALGO][Debug][entry_raw_detect 82]: Detected {2} objects:
[ALGO][Debug][entry_raw_detect 88]: [DET][Object] {77.3%} -> { 424,   71,  552,  376}.
[ALGO][Debug][entry_raw_detect 88]: [DET][Object] {64.3%} -> { 163,    6,  329,  355}.
```

如上图，测试成功后会打印检测到的物体和置信度。当模型检测算法类型修改时，比如修改为

anchor free 的 nanodet 时，需要一并修改 algo_det_yolo.c/algo_det_yolo.h 的实现。

此外，ARM 侧的检测用算法移植到 RISCV 时，需要注意 RISCV 启用的内存和存储空间有限，宜将其大幅简化并改用 C 语言重写，并在 ARM 侧测试通过后再考虑在 RISCV 测试，这样易于调试并发现问题。