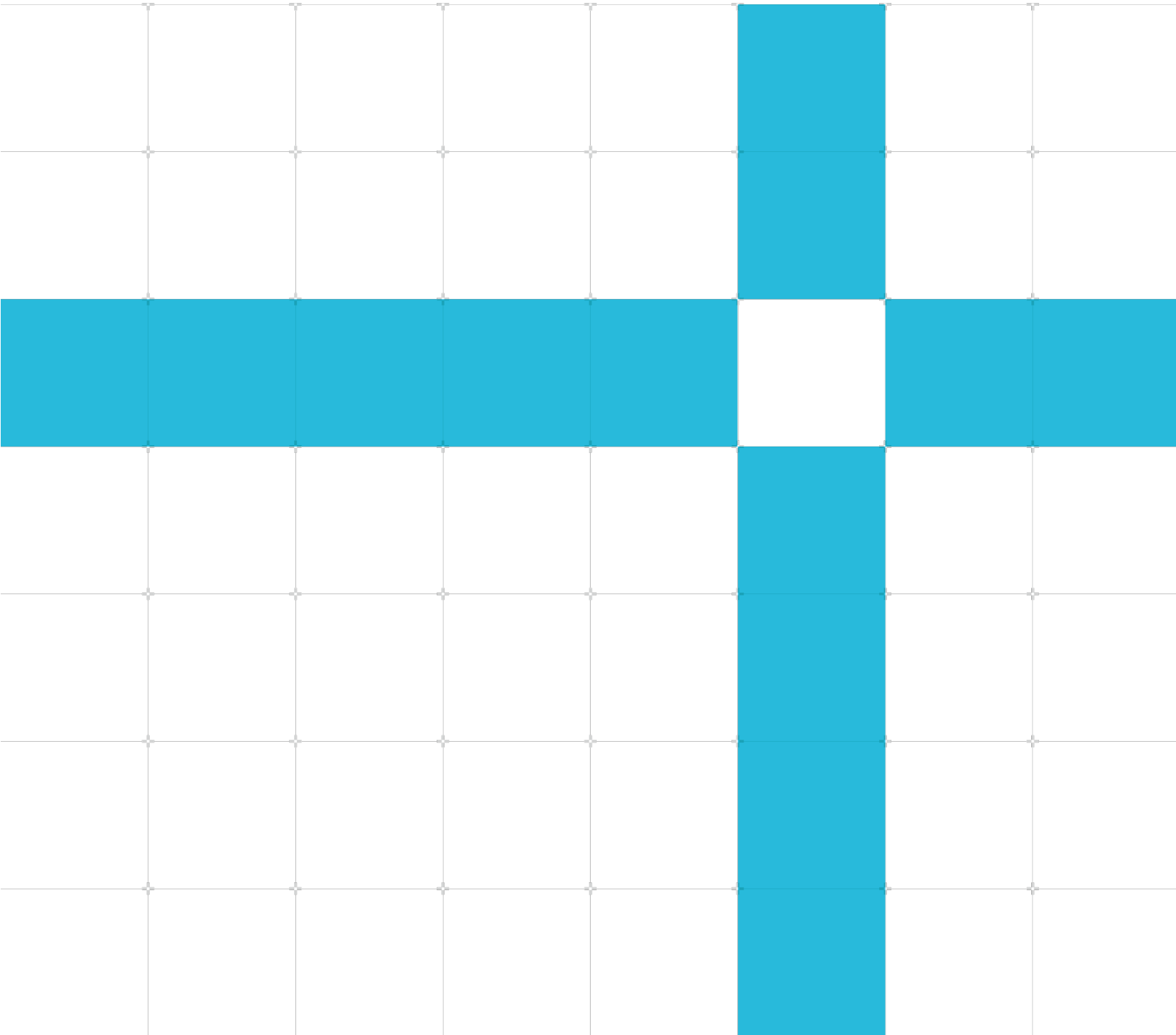
Arm® Mali™-C78AE Image Signal Processor

Guide to Port ISP Driver to Linux Platform

Non-Confidential

Issue 01
109539



Arm® Mali™-C78AE Image Signal Processor

Guide to port ISP driver to Linux platform

Release information

Document history

Issue	Date	Confidentiality	Change
01	14-12-2023	Non-Confidential	First release Version 1.0

Non-Confidential Proprietary Notice

Arm Limited. Company 02557590 registered in England.
110 Fulbourn Road, Cambridge, England CB1 9NJ.
(LES-PRE-20349)

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Feedback

Arm welcomes feedback on this product and its documentation. To provide feedback on Arm® Mali™-C78AE Image Signal Processor, create a ticket on **https://support**.developer**.arm.**com.

To provide feedback on the document, fill the following survey: **https://developer.arm.com/documentation-feedback-survey**.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

Contents

1. Introduction

1.1. About this guide

The purpose of this guide is to help you port the ISP driver to a Linux platform and avoid potential problems during the ISP driver porting.

This is an ISP software porting guide on Linux platforms. It introduces the detailed steps of how to port the ISP software on Linux platforms and how to do the software validation after porting done.

1.2. Product revision status

The rxpy identifier indicates the revision status of the product described in this book, for example, r1p2, where:

rx	identifies the major revision of the product, for example, r1.
py	identifies the minor revision or modification status of the product, for example, p2.

1.3. Intended audience

This guide is intended for software developers who want to configure the Arm® Mali™-C78AE Image Signal Processor (ISP) and port it to a Linux system.

This document assumes that you are familiar with the following:

- The C programming language and its syntax.
- Use of GNU make.
- The Linux operating system.

1.4. Using this manual

This manual is organized into the following sections:

Basic firmware (FW) porting

This section describes how to do the ISP kernel mode driver porting and simple verification.

Linux user driver(application) 3A porting

This section describes the user mode ISP driver (AE/AWB/Iridix algorithms) porting.

Enable Test pattern generator (TPG) to verify the driver

This section describes how to enable the ISP test pattern generator as the signal input to verify the ISP driver.

V4L2 memory to memory test

This section describes how to use Memory to Memory mode to verify the ISP driver.

Build the ISP multiple contexts software environment

This section refers you to information on how to build the ISP multiple contexts software to verify the driver.

Sensor driver porting and integration

This section describes the sensor driver porting and integration with the kernel mode ISP driver.

3A algorithm customization

This section describes how to customize the 3A algorithm in the user mode driver.

2 Guide to port ISP driver to Linux platform

This guide describes the following tasks:

1. Basic firmware (FW) porting
2. Linux user driver(application) 3A porting
3. Enable Test pattern generator (TPG) to verify the driver
4. V4L2 memory to memory test
5. Build the ISP multiple contexts software environment
6. Sensor driver porting and integration
7. 3A algorithm customization

2.1 Basic firmware (FW) porting

Perform the following steps for basic FW porting:

2.1.1. Step 1: Set up software development environment to compile ISP kernel driver

1. Compile the ISP driver.

Ensure that you have the correct cross-compile toolchain and matched kernel header files (KDIR) in the host PC.

2. Go to the ISP kernel driver folder `linux_kernel`.

The following three options are mandatory to compile the ISP kernel driver for the target platform:

- `ARCH` defines the target system architecture.
- `CROSS_COMPILE` defines the toolchain.
- `KDIR` defines the path to the Linux kernel headers.

You can set these options in the `make` command option, for example:

```
make ARCH=arm64 CROSS_COMPILE=/exports/kernels/juno-bsp_kernel-
4.9/toolchain/bin/aarch64-linux-gnu-  KDIR=/exports/kernels/juno-bsp_kernel-
4.9/out/juno/mobile/kernel
```

For simplicity, you can also change the default setting of these options in `linux_kernel/Makefile` so that you do not need to set them every time when compiling. For example:

```
ifeq ($(ARCH),)
        _ARCH=arm64
else
        _ARCH=$(ARCH)
Endif
```

Similarly, you can set `_CROSS_COMPILE` and `_KDIR` in the same way as you do for `_ARCH`. In this way, you only need to execute the `make` command under the folder `linux_kernel` to compile the ISP kernel driver.

3. Run the ISP driver.

After compiling, you can copy and install the driver (`isp.ko`) on the target platform to ensure that you use the correct toolchain and software environment.

If the toolchain or kernel KDIR does not match, running the ISP driver might report the following error message:

```
"isp: disagrees about version of symbol module_layout
insmod: ERROR: could not insert module isp.ko: Invalid module format"
```

If you get the following error message while running `insmod isp.ko`, it indicates that the device tree of the ISP driver is not correctly configured. This incorrect configuration causes the kernel module initialization failure. Therefore, you must add the ISP configuration to the DTS file.

```
"insmod: ERROR: could not insert module isp.ko: No such device"
```

2.1.2. Step 2: Add the ISP driver setting in the DTS (Device tree setting) file

1. Get required information for setting the DTS file.

The DTS file configuration is as follows:

```
        isp: isp@0x60400000 {
                compatible = "arm,isp";
                reg = <0x0 0x60400000 0x0 0x00B00000>;
                interrupts = <0 168 1>;
                interrupt-names = "ISP";
                memory-region = <&isp_reserved>;
                userptr-memory-addr = <0x40 0x00000000>;
        };
        reserved-memory {
                isp_reserved: frame_buffer@0x61000000 {
                        compatible = "shared-dma-pool";
                        no-map;
                        #address-cells = <2>;
                        #size-cells = <2>;
                        reg = <0x0 0x61000000 0x0 0x7000000>;
                };
        };
```

For the previous DTS configuration, you can check with hardware engineers to get the following information of the target platform:

- Physical base address of the ISP register.
- Reserved physically continued frame buffer memory base address and length.
- Interrupt number of the ISP hardware.

2. Use these platform parameters to set the DTS file correctly.

Consider the previous sample DTS file as an example to explain some parameters in DTS file. You can get the following information:

- 0x60400000 is the physical base address of the ISP register.
- 0x61000000 is reserved physically continued frame buffer memory base address. The ISP raw buffers and output buffers are allocated from this region.
  168 is interrupt number of ISP hardware. Get the information from the hardware engineer to confirm whether the ISP interrupt is SPI or PPI, and set it correctly.

If there are v4l2 sub-devices, you must also configure the sub-device items. In the C78AE kernel driver, there is no sub-device, so you do not need to configure these items.

The sub-device configuration is listed below for your reference only.

```
        sensor: soc_sensor@0x60604000 {
                compatible = "soc,sensor";
                reg = <0x0 0x60604000 0x0 0x001000>;
        };
        lens: soc_lens@0x60600000 {
                compatible = "soc,lens";
                reg = <0x0 0x60600000 0x0 0x001000>;
        };
        iq: soc_iq@0x60606000 {
                compatible = "soc,iq";
                reg = <0x0 0x60606000 0x0 0x001000>;
        };
```

3. Recompile the DTS file to generate a DTB file, program the DTB file to the target platform, and install the driver module again.

If everything works fine, you are expected to get a correct log as follows:

```
[   61.847705] isp: loading out-of-tree module taints kernel.
[   61.872874] arm,isp 60400000.isp: assigned reserved memory node frame_buffer@0x60800000
[   61.880832] [NOTICE] main.c:287: CMA region: start = 0x60800000, size = 0xf5d0000
[   61.889975] [NOTICE] main.c:293: ISP irq = 37, flags = 0x404 60400000.isp id: -1!
[   61.899396] [NOTICE] main.c:138: Copying default register space to CDMA spaces...
[   61.908563] [NOTICE] main.c:147: - Product ID: REG = 0, BACKUP = 0
[   61.940276] [NOTICE] main.c:159: - Base address for CTX#00 to 6fdf0000...
[   61.948727] [NOTICE] acamera.c:93: DRIVER: 2.0 (1a36a296)
[   61.955802] [NOTICE] acamera.c:94: HW GEN: 1.2 (r101397)
[   62.164180] [NOTICE] acamera_isp_ctx.c:330: Loading custom settings CTX[0]
[   62.269783] [NOTICE] system_cma.c:63: system_cma: used = 0x800000 (8 MiB), remaining =
0xedd0000 (237 MiB); 1 node(s)
[   62.379063] [NOTICE] system_cma.c:63: system_cma: used = 0x1000000 (16 MiB), remaining =
0xe5d0000 (229 MiB); 2 node(s)
[   62.427870] [NOTICE] system_cma.c:63: system_cma: used = 0x1400000 (20 MiB), remaining =
0xe1d0000 (225 MiB); 3 node(s)
[   62.476671] [NOTICE] system_cma.c:63: system_cma: used = 0x1800000 (24 MiB), remaining =
0xddd0000 (221 MiB); 4 node(s)
[   62.599824] [ERROR]  module_mcfe_service_input.c:426: Failed to turn on input port 0 after
100 ms.
[   62.610602] [ERROR]  module_mcfe_usecase_tdmf.c:468: Error, failed to start video 0
(context id: 0).
```

4. Try uninstalling or installing the driver module to confirm that the driver load or unload work correctly.

You can use the `rmmod isp` command to uninstall the driver module, and use the `insmod isp.ko` command to install the driver module.

After *Step 1: Set up software development environment to compile ISP kernel driver* and *Step 2: Add the ISP driver setting in the DTS (Device tree setting) file*, you can confirm the following:

- The toolchain and KDIR that you use are correct.
- ISP items are added to the DTS file. However, you cannot confirm whether you set the correct value of the ISP items in the DTS file.

2.1.3. Step 3: Platform porting implementation

1. Configure the `src/acamera_configuration.h` as described in the following table:

Definitions	Notes
`#define PHY_ADDR_ISP 0x60400000`	60400000 must be replaced with ISP register base address of target platform.
`#define PHY_ADDR_CDMA 0x6fdd0000`	Define the start of the Configuration DMA (CDMA) buffer in the physical address. Need to be statically allocated.
`#define FIRMWARE_CONTEXT_NUMBER 1`	Use context number 1 to verify the driver first.

`#define PHY_ADDR_FPGA 0x60600000`
`#define ISPAS_MINUS_SYSPHY 0x1F800000`
`#define PHY_ADDR_BUFF_POOL 0x60800000`
	If the target platform is not Arm Juno platform, you can ignore these definitions.
	
	If the target platform is Arm Juno platform, you must contact Arm support to ensure that you set the correct value.

Notes:

- The `PHY_ADDR_ISP` value must be same with the `PHY_ADDR_ISP` value defined in the DTS file.
- `PHY_ADDR_CDMA` is for MCFE multiple slots, and it is independent of reserved frame buffer configuration in the DTS file. This memory must be statically allocated in the system and reserved. Its size depends on the slot number you want to use. The actual size is calculated as follows:
  `Total_CDMA_size =slot_number * (LEN_ADDR_ISP + LEN_ADDR_META)`
  `LEN_ADDR_ISP and LEN_ADDR_META are defined in src/acamera_configuration.h.`

2. Port the platform BSP.

The following table lists all the platform APIs that you must implement for the platform porting. There are two properties for each category:

- Whether it has default implementation in the released reference ISP software.
- Whether it is mandatory for ISP kernel driver porting.

Many platform APIs are already implemented in the release package. These APIs are marked as Y in the Default implementation column of the table.

To do the platform porting, you need to do platform-specific implementations, which are marked as N in the Default Implementation column of the table.

Module	Default implementation	Optional or Mandatory	Notes
system_assert.h	Y	Mandatory	-
system_chardev.h	Not used	-	-
system_cache.h	Not used	-	-
system_cma.h	Y	Mandatory	The current default implementation must work with the driver and set the frame buffer in kernel DTS file.
system_control.h	Y	Mandatory	The current bsp_init calls back the acamera_interrupt_xxx driver API.
system_ext_mem_io.h	Not used	-	Not used in the current reference driver
system_file_io.h	Not used	-	Not used in the current reference driver
system_i2c.h	N	Mandatory	For sensor or lens control.
system_interrupts.h	Y	Mandatory	Combined with driver code.
system_isp_io.h	N	Mandatory	Platform specific register access.
system_log.h	Y	Mandatory	• The current function is printk. • The default function is printf.
system_platform.h	N	Optional	The driver does not call APIs in this module.

system_platform_ device_numbers.h	Not used	-	-
system_profiler.h	Not used	-	-
system_semaphore.h	Y	Mandatory	-
system_spi.h	N	Optional	Platform specific SPI, for some sensors control.
system_spinlock.h	Y	Mandatory	-
system_stdlib.h	Y	Mandatory	Might need reimplementation with the platform
system_timer.h	Y	Mandatory	Might need reimplementation

According to the previous table, you implement the mandatory items with N in the default implementation column. You can follow the below sequence to implement the platform porting layer:

a. System_isp_io.h
b. System_interrupts.h, which to be reimplemented according to the target platform.

Suppose you implement above two types of platform APIs and the register access and interrupt can work normally. In this case, you can install the driver successfully.

c. System_i2c.h

I2C is for sensor or lens control. You can also skip System_i2c.h APIs at this stage, and implement it later in the sensor porting stage.

3. Control the ISP driver log.

All platform layer code does not use the log control defined in system_log.h. Therefore, you do not need to change it. All the code in the ISP kernel driver uses the LOG(...), which is defined in util/logger/acamera_logger.h and util/logger/acamera_logger.c.

The default log level of the driver is controlled in src/acamera_configuration.h as follows:

```
#define SYSTEM_LOG_DEFAULT_VALUE LOG_LEVEL_NOTICE
```

If you want to change the default log level of the ISP driver, you can change the above definition.

4. Debug and verify the ISP driver.

You can compile the ISP kernel driver successfully and copy it to the target platform. After installing the ISP driver, suppose that you get the following log. Pay attention to main.c:131: - Product ID: REG = 4e, BACKUP = 4e in the log. If the product ID is same as the product ID described in the register map document. It means that the register read is correct.

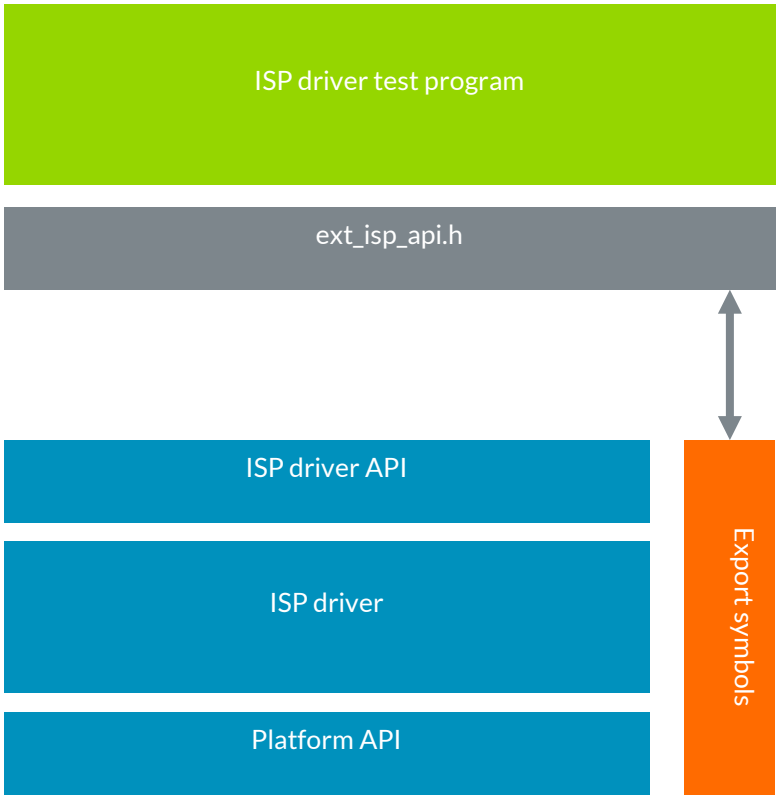
```
[  166.747277] arm,isp 60400000.isp: assigned reserved memory node
frame_buffer@0x60800000
[  166.755296] [NOTICE] main.c:272: CMA region: start = 0x60800000, size =
0xf5d0000
[  166.764500] [NOTICE] main.c:278: ISP irq = 37, flags = 0x404 60400000.isp id: -
1!
[  166.774002] [NOTICE] system_isp_io.c:31: ISP version from register = 0x1000003
[  166.783468] [NOTICE] main.c:122: Copying default register space to CDMA
spaces...
[  166.793443] [NOTICE] main.c:131: - Product ID: REG = 4e, BACKUP = 4e
[  166.825400] [NOTICE] main.c:144: - Base address for CTX#00 to 6fdf0000...
[  166.833904] [NOTICE] acamera.c:76: DRIVER: . (4be61c1f)
[  166.840850] [NOTICE] acamera.c:77: HW GEN: 2.0 (r102714)
[  167.088234] [NOTICE] awb_mesh_NBP_func.c:455: AWB USING 3A LIB
[  167.095850] [NOTICE] iridix8_func.c:65: iridix USING 3A LIB
[  167.168974] [NOTICE] acamera_isp_ctx.c:313: Loading custom settings CTX[0]
[  167.275071] [NOTICE] system_cma.c:46: system_cma: used = 0x800000 (8 MiB),
remaining = 0xedd0000 (237 MiB); 1 node(s)
[  167.384671] [NOTICE] system_cma.c:46: system_cma: used = 0x1000000 (16 MiB),
remaining = 0xe5d0000 (229 MiB); 2 node(s)
[  167.433778] [NOTICE] system_cma.c:46: system_cma: used = 0x1400000 (20 MiB),
remaining = 0xe1d0000 (225 MiB); 3 node(s)
[  167.482879] [NOTICE] system_cma.c:46: system_cma: used = 0x1800000 (24 MiB),
remaining = 0xddd0000 (221 MiB); 4 node(s)
[  167.496188] [NOTICE] isp_wrapper_func.c:573: - assigning ctx#0 (channels 1,
LOCAL) Exp.channel 0 to xbar16x4_out 0.
```

2.1.4. Step 4: ISP driver verification with test software

Arm provides a test program for you to verify whether the ISP driver porting is implemented correctly. The idea is to export some platform APIs from the ISP driver and the test-program driver calls these exported APIs to do some tests.

The following figure shows the relationship between the test program and the ISP driver:

- ISP driver components in blue (ISP Driver API, ISP Driver, and Platform API) remain unchanged.
- ISP driver test program components (ISP driver test program and `ext_isp_api.h`) are in green and gray.
- `Ext_isp_api.h` includes all the APIs that the test program calls to test. If you want to test some new APIs from the ISP driver, you can add the API prototype to the `ext_isp_api.h` header file.
- The Export symbols in orange export the symbols that are declared in `ext_isp_api.h` from the ISP driver. This block is automatically added by the test program script.



The test program verifies the following items:

- Check the consistency between the DTS setting and macro definition in ISP driver code.
- ISP registers read/write APIs.
- Frame buffer allocation and read/write.
- Interrupt handling.
- Use the ISP test pattern generator to check the ISP pipelines and output functionality.

Contact Arm to get the ISP driver test program. You can find some documents in the package. Consult these documents to do the ISP driver verification.

The steps to use the ISP driver test program is as follows:

1. Read the documents:
   - `README.md`
   - `Test-program/README.md`
   - `Script/README.md.`
2. Copy your porting-finished ISP driver source code, for example folder name `linux_kernel`, to the same folder with test-program and script. Make sure that the ISP driver Makefile is under the first level of the source code, for example `linux_kernel/Makefile`.
3. Configure `script/config/config.ini`.
   If the target platform supports `ssh` and `scp`, you can set the `autoload` to `enable`. The script in the test program can help copy the isp.ko and isp_drvport_test.ko to the target platform. You are recommended to set the `autoinstall` to `disable`.
4. Go to the folder `script` and use the following command to run the script:
   `bash ./run.sh;`

The script is run to check the consistency between ISP driver and test program, compile the ISP driver, and test program driver. If the `autoload` is set to `enable`, it copies the drivers to the target platform.
Make sure `bash ./run.sh` runs correctly. Otherwise, fix errors until it runs successfully.

5. If the `autoload` in config.ini is set to `disable`, copy the isp.ko and isp_drvport_test.ko to target platform individually.
6. In the target platform, install the driver to do the test:

```
Insmod isp.ko;
insmod isp_drvport_test.ko test_items = 1;   test the platform API first.
```

If there is any error reported from the log, try to fix the error in device tree setting or ISP driver platform porting layer. After that, go back to the "*Step 4 Go to the folder `script and …`*" to compile and test the driver again.

If all the platform API tests pass, go to the ISP test items as follows:

```
Rmmod isp_drvport_test.ko;
Rmmod isp.ko;
```

Because when you do the `test_items=1`, the system interrupt handler is changed, so you need to uninstall the ISP driver and ISP test program driver and reinstall them for the remaining tests:

```
Insmod isp.ko;
Insmod isp_drvport_test.ko test_items=2;
```

Check the output log information and make sure all tests are successful.

2.1.5. Step 5: Bring up the ACT tool.

After finishing the platform APIs porting, make sure that all the APIs work correctly. Then, you can proceed to bring up the ACT tool.

Using the ACT tool, you can control and monitor the ISP hardware and software.

Prerequisites to run ACT tool:

- The ISP driver runs in the Linux kernel.
- The target platform has internet access.

Consult the Arm Control Tool User Guide for ACT usage. For additional information about ACT server configuration, see the document **how to configure multiple instances of ACT servers**.

To run the ACT tool, first we need launch the ACT server in the target platform. The log information of the ACT server indicates the information to launch the ACT client, then in PC to launch the ACT client. For the target platform where Linux kernel runs, use `devmem` as the access channel.

You can use the ACT client with port 8000 to access the ISP registers. Running the ACT client requires the XML (ISP registers description) and JSON (ISP driver API command description) files. The XML and JSON files can be found in the release package ...`\control_tool_files`.

If the ACT client is launched correctly, you need to verify the ID registers value to confirm that the register access API works correctly. You can select tab **Hardware** and item **ID** to check the registers shown in the following figure:

If the value read from ACT client does not match with the value defined in the register map document, you must check whether the porting implementation of the register read API is correct.

The registers address range of the ISP Hardware Identification registers is from 0x0E000 to 0x0E00C. Consult the Mali-C71AE or Mali-C78AE register map document for details. `Product` and `Version` define default values in the register map document. You can use the default value of `Product` and `Version` to confirm the register reading correctness.

Notes:

- The ACT tool displays the register value with decimal by default. You can switch it to hexadecimal by clicking the **0x** icon before each value.
- Use the same version Mali-C71AE/Mali-C78AE register map document and hardware IP to compare these ID registers values. Otherwise, these values might not match.

After the register read verification is completed successfully, check the firmware revision. You can read the firmware revision from the ACT section **API** and item **TSELFTEST**. You can compare the read value of `FW_REVISION` and the definition in the `linux_kernel/src/revision.h` file, for example:
`#define FIRMWARE_REVISION 0x1a36a296u`

If all above verification is complete, it means that the ISP register access and other platform porting is complete. Then, you can try to verify the driver function by using the Test Pattern Generator as an input signal.

2.2 Linux user driver(application) 3A porting

Without the 3A algorithm application working, the ISP output image quality might not be acceptable. Therefore, you must make the 3A algorithm application take effect before verifying the ISP output image.

For details about the user mode driver architecture and design, see the sections *4.4 Linux split architecture* and *4.7 The 3A library* of the software TRM.

2.2.1. Step 1: Compile the Linux user driver

The ISP Linux user driver is User mode application that interacts with ISP driver to implement the 3A algorithms. Generally, it is not necessary to do any porting work to run on a standard Linux platform.

1. Go to the folder .../`linux_user`.
2. Configure these items CC/AR/ARNLIB in the Makefile, according to your toolchain. For example:

```
export CC := /exports/kernels/juno-bsp_kernel-4.9/toolchain/bin/aarch64-linux-gnu-gcc
```

3. Run the `make` command to compile the code.
   You get `isp_3a.elf` if the compilation is successful.

2.2.2. Step 2: Run the user driver

1. Run `insmod isp.ko`
2. Run `./isp_3a.elf &`
3. Check whether the 3A application runs correctly by running the following command:

```
ps -ef | grep isp_3a.elf
```

2.3 Enable Test pattern generator to verify the driver

You can enable the test pattern generator (TPG) by using method 1 or method 2.

- Method 1: Use register access (ACT tool).
- Method 2: Change software code.

2.3.1. Method 1: Use register access (ACT tool).

Mali-C71AE and Mali-C78AE support up to 16 contexts. There is a CDMA buffer, which includes the ISP register setting for each context. ISP hardware performs the DMA copy operation of the ISP register setting from CDMA buffer to ISP physical address for each frame data processing. Therefore, if you want to access the register setting for each context, you must access the CDMA buffer. The ACT configured for this purpose is called context ACT. Similarly, the ACT configured for ISP physical register access is called ISP ACT.

Normally, you configure the port 8000 for ISP ACT, port 8002 for context ACT. For details, see the document **How do I configure multiple instances of ISP ACT server?**

Before enabling the TPG via register access, make sure:
- The ISP driver is installed and works correctly.
- The ACT works correctly as verified with the ISP ID reading.

Take the following steps to enable the test pattern generator:
1. Use **ISP ACT** to make sure the setting as shown in the following table is correct. If the setting is not correct, you must correct it. Select the item **(frontend) Multichannel frontend** on the tab **Hardware**.

Item	Expected values	Register addresses
schedule mode	1	0x00414
slot mode	3	0x00430(bit0-3)
input mode	3	0x00680(bit 0-1)

The following screenshot shows slot mode and input mode. Slot 0 and input 0 are used, so the first one of slot mode is 3 and the first one of input mode is 3.

2. Use **ISP ACT** to set TPG before the MCFE.

   Select the item **(frontend) Pipeline frontend**, and set **Position Video test gen** to 1. The related register address is 0x0000C (bit3).

3. Use **context ACT** to set the registers in CDMA buffer0 as shown in the following table:

Items	Sub-items	Values	Register addresses
Pipeline	Bypass video test gen	0	0x0E040 (bit4)
Video Test Gen	test_pattern_off on	1	0x0E160
Video Test Gen	Generate mode	1	0x0E160
Video Test Gen	Video source	1	0x0E160
Video Test Gen	Frame request	1	0x0E164

4. Use **ISP ACT** to set the registers in the above table.

   Follow the top to bottom sequence in the table to set the registers.

2.3.2. Method 2: Change software code.

1. Add one API to `isp/fsm/control/control_func.c` as follows:

```
static void config_tpg_internal_timing( control_fsm_ptr_t p_fsm )
{
    acamera_isp_ctx_ptr_t p_ictx =
ACAMERA_FSM2ICTX_PTR( p_fsm );
    uint32_t isp_base = p_ictx->settings.isp_base;
    const uint32_t tpg_enable = get_context_param( p_ictx,
TEST_PATTERN_ENABLE_ID_PARAM );
    const uint8_t tpg_mode = get_context_param( p_ictx,
TEST_PATTERN_MODE_ID_PARAM );
    if ( tpg_mode != p_fsm->tpg_mode ) {
        acamera_isp_video_test_gen_pattern_type_write( isp_base, tpg_mode );
        p_fsm->tpg_mode = tpg_mode;
        /*to make all the TPG type can be detect by human*/
        if ( tpg_mode == 3 )
            acamera_isp_video_test_gen_r_backgnd_write( isp_base, 0xfffff);
        else
            acamera_isp_video_test_gen_r_backgnd_write( isp_base,
0 );
    }
```

```
    if ( tpg_enable != p_fsm->tpg_enable ) {
        acamera_isp_video_test_gen_test_pattern_off_on_write( isp_base,
tpg_enable );
        acamera_isp_video_test_gen_generate_mode_write( isp_base, 1 );  //free
run, continuous frames
        acamera_isp_video_test_gen_video_source_write( isp_base,
1 );  //internal video timing
        acamera_isp_pipeline_bypass_video_test_gen_write( isp_base, tpg_enable ?
0 : 1 );
        if ( tpg_enable == 1 )
            acamera_isp_video_test_gen_frame_request_write( isp_base, 1 );

        p_fsm->tpg_enable = tpg_enable;
        /*set the ISP registers to trigger the TPG*/
        isp_base = PHY_ADDR_ISP;
        acamera_frontend_pipeline_frontend_position_video_test_gen_write( isp_ba
se, tpg_enable ? 1 : 0 );
        acamera_isp_pipeline_bypass_video_test_gen_write( isp_base, tpg_enable ?
0 : 1 );
        acamera_isp_video_test_gen_test_pattern_off_on_write( isp_base,
tpg_enable ? 1 : 0 );
        acamera_isp_video_test_gen_generate_mode_write( isp_base, 1 );  //free
run, continuous frames
        acamera_isp_video_test_gen_video_source_write( isp_base, tpg_enable ?
1 : 0 );  //internal video timing
        //acamera_isp_pipeline_bypass_video_test_gen_write( isp_base,
tpg_enable ? 0 : 1 );
        acamera_isp_video_test_gen_frame_request_write( isp_base, tpg_enable ?
1 : 0 );
    }
}
```

Add this API into `control_config()` to be called as follows:

```
void control_config( control_fsm_ptr_t p_fsm )
{
      …
      config_tpg_internal_timing( p_fsm );
}
```

2. Add flags for context parameters `TEST_PATTERN_ENABLE_ID_PARAM` and `TEST_PATTERN_MODE_ID_PARAM` in `isp/params/context_params.c`

   ```
   params->params[93].flags = ( PARAM_FLAG_READ | PARAM_FLAG_WRITE |
   PARAM_FLAG_RECONFIG );
   ```

3. Recompile the ISP driver and upload it to the target platform.
   Use ISP ACT to enable the test pattern in the ACT client with the `API/TSYSTEM/TEST_PATTERN_ENABLE_ID` path.

In the previous two methods, to enable test pattern generator requires the ACT to work correctly. However, on some platforms, the ACT might not be enabled in the early stage. Therefore, you cannot enable the test pattern by the ACT. In this case, you can also enable test pattern by directly calling function at the end of function `isp_fw_init_kthread_func` in `main.c`, to add the following function call:

```
extern void *get_ctx_ptr_by_id( uint32_t ctx_id );
 acamera_isp_ctx_ptr_t p_ictx = get_ctx_ptr_by_id( 0 );
system_timer_usleep(1000000);  //sleep 1s to wait the FSM initialize and context
initialize finish.
set_context_param( p_ictx, TEST_PATTERN_ENABLE_ID_PARAM, 1 );
  …
```

2.3.3. Verify the TPG feature

After you enable the TPG by using method 1 or method 2, you can use one of the following ways to verify the TPG feature:

- The test pattern picture displays output through the HDMI output and is shown on screen on the Arm ISP reference platform.
- Arm provides a tool to dump the ISP output buffer image, you can request the tool from Arm to dump the ISP output image to check in the PC.
- If there is no ISP output display device, you can check whether the TPG works correctly by checking Frame Counter register values in the pipeline. If the TPG works correctly, these frame counters keep increasing. Otherwise, they remain unchanged.

  The following table shows the Frame Counter registers addresses:

Frame counter names in ACT tool	Register addresses	Notes
(frontend) Frame counter sensor 1-4	0x00744 - 0x00750	
(frontend) Frame counter ISP start	0x00754	
(frontend) Frame counter ISP VTPG	0x00758	
(frontend) Frame counter ISP Iridix	0x0075c	
(frontend) Frame counter ISP Demosaic	0x00760	
(frontend) Frame counter ISP out 1-3	0x00764 - 0x0076c	For stream output
(frontend) Frame counter ISP out AXI 1-3	0x00770 - 0x00778	For memory output

- Check the system interrupt counter with Linux `proc` system.

  ```
  Cat /proc/device/isp
  ```

2.4 V4L2 memory to memory test

You can use the TPG feature to verify that the ISP driver works with the pipeline. In this guide, V4L2 test application memory-to-memory(M2M) mode is used to verify most of the ISP driver and features.

The reasons to use v4l2 memory to memory are as follows:

- It can verify almost all the ISP features include both hardware and software.
- It only depends on the ISP driver and does not rely on external devices, such as sensor devices.

The purposes of the V4L2 memory-to-memory test application are as follows:

- Verify raw buffer, output buffer, MCFE, and ISP pipelines.
- Input raw image from memory and output ISP processed image to memory.

2.4.1. Step 1: Enable the V4L2 in ISP driver build

1. Enable V4L2 in the ISP driver source code.
   Enable V4L2 feature definition in `src/acamera_configuration.h` as below.

   ```
   #define V4L2_INTERFACE_BUILD 1
   ```

2. Enable the Linux V4L2 framework in the Linux kernel configuration file to use the 4VL2 interface of the driver.

   If V4L2 is not supported in the target platform Linux kernel, see the ISP software Technical Reference Manual 6.2 for detailed information.

   If the kernel is not configured correctly, a compilation error occurs because some V4L2 symbols are missing in the Linux kernel.

3. Compile the ISP kernel driver with V4L2 support successfully and install it on the target platform, you can find some device node `/dev/videoxraw`, `/dev/videoxout`, `/dev/videoxmeta`. For information about the device node, see the ISP software TRM section 6.1.

4. Test for the live sensor streaming, after installing the V4L2-enabled ISP driver, the ISP driver does not automatically start streaming. You must launch the V4L2 application to start streaming.

   Note: The ISP released reference kernel driver is verified with Linux kernel 4.9. If your kernel version is 5.x, see the document **how do I modify the ISP driver to support Linux Kernel 5.x?** for the code modification.

2.4.2. Step 2: Build the v4l2 test application.

After you enable the V4L2 in the ISP driver, the ISP driver can support standard V4L2 APIs. Not all the V4L2 APIs are supported. Check the ISP driver source code for details.

Arm can provide a reference V4L2 test application. If you want this reference application source code, contact Arm to get the source code.

There is a sample Makefile in the V4L2 reference test application folder. To compile it, you only need to set the correct cross-compile toolchain for your target platform.

After the `make` command is executed successfully, you get the V4L2 test application `v4l2_test.elf`.

2.4.3. Step 3: Run the v4l2 m2m program

To run the V4L2 test program memory to memory mode, request a raw image from Arm and put that raw image in the same folder with `isp-v4l2.ko` and `v4l2_test.elf`. Also, create the folder with name `out` to save the processed output image.

1. Run the `./insmod isp-v4l2.ko` command.

2. Run the command `./v4l2_test.elf -m m2m -i ./`
   If everything works fine, you get the following logs and the FPS values keep updating.

```
"[ Reading is in progress ] (files read: 1, files total: 1, FPS: 6.81) Choose
menu > "
```

Launch another shell of target Linux platform and start the 3A application.

3. Run the command `./isp_3a.elf &`
Now the correct 3A setting takes effect and ISP outputs correct image.

4. Exit the `v4l2_test.elf` and relaunch it by using the following command:

```
./v4l2_test.elf -m m2m -i ./ -o ./out -e batch
```

If everything works, you get the ISP output image in folder out. This image is in RGBA format and all the values of alpha channel are 0.

You can use some image viewer tools to preview the image. If there is no available image viewer to check the image, you can use command lines of ImageMagick, which is available in most Linux releases. The following command can convert an RGBA *.bin file to standard PNG file in the same folder.

```
find . -type f -name "*.bin" | xargs -I {} convert -size 1920x1080 -depth 8
BGRA:{} -channel BGRA -separate -delete 3 -combine {}.png
```

Note: You might wonder why not run the command `./v4l2_test.elf -m m2m -i ./ -o ./output/ -e batch` directly in step 2. This is because this command causes `v4l2_test.elf` to exit after the capture is done. At that time, however, the 3A might not take effect. In this case, you get an incorrect output image.

2.5 Build the ISP multiple contexts software environment

The KBA **How do I build ISP multiple contexts software?** describes how to build the ISP multiple contexts software environment with memory-to-memory mode. Consult the KBA for detailed instructions.

2.6 Sensor driver porting and integration

After you verify the ISP driver by using the TPG, it means that the ISP driver works. Then, you can start the actual sensor driver porting.

For details about the sensor driver porting, see the section *3.3 Sensor driver porting* of the software TRM. Before porting the sensor driver, you can first read the sensor document to understand it. You also need to get some board-level information of the sensor, such as:

- How is it connected with the ISP? (I2C or SPI)
- The number of stream inputs to the ISP?

In the ISP driver code `linux_kernel/isp/fsm/sensor/sensor_func.c`, you can check how the ISP driver interacts with the sensor driver. This driver code can help you understand the sensor driver behavior.

Perform the following steps for the sensor driver porting:

1. Port Sbus APIs.
   Finish the implementation of system_i2c.c, system_spi.c or both. Whether to implement `system_i2c.c` or `system_spi.c`depends on the sensor register access interface type.

2. Contact the sensor vendor to get the sensor initialization sequence and put it to `static const sensor_reg_t seq_table[] = {`. You can refer to `DUMMY_seq.h` for more information. Normally, this setting is loaded into sensor hardware in the `sensor_load_sequence` API, which is called in the sensor API `set_mode`.

3. Configure sensor setting.
   Read the sensor data sheet or get help from sensor vendor to get the sensor support mode list and put them in the sensor driver `static sensor_mode_t supported_modes[] = {`.

4. Refer to *Section 3.3 Sensor driver porting* of the software TRM to implement the sensor driver and its APIs.

5. Prepare calibration data. Each sensor must have at least one specific calibration data set, and it can also support multiple calibration sets. Normally, every sensor needs to do the IQ tuning to get the propriate calibration set for each mode. Refer to release code `get_calibrations_dummy()` in `acamera_get_calibrations_dummy.c` of how to support multiple calibration sets.

6. Integrate with driver code.
   The integration of sensor driver to ISP driver is in `./src/runtime_initialization_settings.h`.
   Set the correct sensor driver initialization and calibration interface in the structure variable shown as follows:
   `static acamera_settings settings[FIRMWARE_CONTEXT_NUMBER] = {`

```
#if ( FIRMWARE_CONTEXT_NUMBER >= 1 )
    {
        .sensor_init = sensor_init_ar0231,
        .sensor_name = "ar0231",

        .sensor_options = {
            .is_remote = 0,
            .preset_mode = 0,
        },

        .get_calibrations = get_calibrations_ar0231,
        .context_options = {
            .cmd_if_is_passive_mode = 0,
        },
        .isp_base = 0,
    },
#endif
#if ( FIRMWARE_CONTEXT_NUMBER >= 2 )
```

   Sensor_init_ar0231 must be replaced by the new sensor driver API. The `get_calibrations_ar0231` also must be replaced by the new sensor calibration API. For details about calibration implementation, see the code `linux_kernel/sensor/calibrations`.

2.7 3A algorithm customization

You can read the sections *4.4 Linux split architecture* and *4.7 The 3A library* of the ISP software Technical Reference Manual to understand the ISP user mode driver structure.

Arm provides all the source code of user mode driver, which includes the reference 3A algorithm. Arm also provides a flexible design for you to customize the 3A algorithm. You can use your own 3A library to replace the reference implementation.

The 3A interfaces are defined in the following files:

- AE interface: `3a_lib/include/3a/ae/ae_acamera_core.h`
- AWB interface: `3a_lib/include/3a/awb/awb_acamera_core.h`
- Iridix interface: `3a_lib/include/3a/iridix/iridix_acamera_core.h`

You can consult the following source code for reference implementation.

- `3a_lib/src/ae/ae_acamera_core.c`
- `3a_lib/src/awb/awb_acamera_core.c`
- `3a_lib/src/iridix/iridix_acamera_core.c`

If you want to customize the 3A algorithm, you only need to reimplement the previous code list. You can reuse all the other parts of the user mode driver.