Ameba AI Glass Documents

Release v1.0

REALTEK SG

Mar 18, 2025

CATEGORY LIST

CHAPTER

ONE

COMPILE ENV SETUP

1.1 Compile Env Setup

1.1.1 Set up developing environment

Step 1. OS environment

AI Wearable glass currently supports Windows OS 64-bits (Windows 10 and above), Linux OS (Ubuntu22 and above) and MacOS (Intel and Apple Silicon). To have the best experiences, please use the latest version of OS.

Step 2. Installing the Driver

First, connect AI Wearable glass to the computer via Micro USB:

Check the COM port number in Device Manager of computer:

Step 3. Set up GCC Building Environment (window)

Installing mingw with ASDK and setting up the CMake

---

(1) Download and extract msys64_v10_3.7z from https://github.com/Ameba-AIoT/ameba-tool-rtos-pro2/releases/tag/msys64_v10_3

(2) Double click "msys2_shell.cmd" from mysys64 folder

(3) After setting up mingw, you need to install cmake. Download cmake in https://github.com/Kitware/CMake/releases/download/v3.20.0-rc1/cmake-3.20.0-rc1-windows-x86_64.msi and install it

(4) Add location of cmake.exe to PATH of msys2_shell by using vim ~/.bashrc and appending path of cmake.exe to environment variable PATH or using editor to directly append the path to file "msys64/home/<USER_FOLDER>/.bashrc"

```
export PATH=/c/Program\ Files/CMake/bin:$PATH
```

Caution

If your PATH contains space characters, remember to use "\" to escape

Note

For the first time adding the CMake PATH, after adding the PATH, you need to re-open the msys2_shell and check the version by:

```
$ cmake --version
cmake version 3.20.0-rc1

CMake suite maintained and supported by Kitware (kitware.com/cmake).
```

Adding toolchain to msys2

(1) Download and extract asdk-10.3.0-mingw32-newlib-build-3633-x86_64.zip from https://github.com/Ameba-AIoT/ameba-toolchain/releases/download/V10.3.0-amebe-rtos-pro2/asdk-10.3.0-mingw32-newlib-build-3633-x86_64.zip

(2) Like adding PATH for cmake, user can add or change the toolchain in "msys64/home/<USER_FOLDER>/.bashrc".

(3) Add toolchain PATH by "export PATH=<path to toolchain>:$PATH".

```
if [ -d "../../asdk-10.3.0" ]; then
    echo "asdk-10.3.0 exist"
    export PATH=/asdk-10.3.0/mingw32/newlib/bin:$PATH
```

Note

Recommand to use the latest provided toolchain or use the version after 10.3.0

Building the project

(1) Open mingw by double clicking "msys2_shell.cmd".

(2) Create folder "build" in project/realtek_amebapro2_v0_example/GCC-RELEASE.

(2) Enter the project location: project/realtek_amebapro2_v0_example/GCC-RELEASE/build.

(4) Run "cmake .. -G"Unix Makefiles" -DCMAKE_TOOLCHAIN_FILE=../toolchain.cmake -DSCENARIO=ai_glass" to create the makefile.

(5) Run "cmake –build . –target flash" to build and generate flash binary.

Note

If building the project successfully, you can see flash_ntz.bin in the 'build' folder

Step 4. Set up GCC Building Environment (LINUX)

Add toolchain to the linux PATH

(1) Dowload and extract asdk-10.3.0-linux-newlib-build-3638-x86_64.tar.bz2 from https://github.com/Ameba-AIoT/ameba-toolchain/releases/download/V10.3.0-amebe-rtos-pro2/asdk-10.3.0-linux-newlib-build-3638-x86_64.tar.bz2

```
tar -jxvf <PATH_TO_YOUR_TOOLCHAIN.tar.bz2> -C <DIR_TO_EXTRACT>
```

(2) Add toolchain to PATH:

```
export PATH=<PATH_TO_YOUR_TOOLCHAIN>/asdk-10.3.0/linux/newlib/bin:$PATH
```

Note

You can add PATH to ~/.bash_profile

Installing cmake for linux

(1) Install cmake using terminal (like "sudo apt-get -y install cmake"), if the installation is successful, you can get the version by "cmake –version".

Building the project

(1) Create folder "build" in project/realtek_amebapro2_v0_example/GCC-RELEASE.

(2) Open linux terminal and enter the project location: project/realtek_amebapro2_v0_example/GCC-RELEASE/build.

(3) Run "cmake .. -G"Unix Makefiles" -DCMAKE_TOOLCHAIN_FILE=../toolchain.cmake -DSCENARIO=ai_glass" to create the makefile.

(4) Run "cmake –build . –target flash" to build and generate flash binary.

Note

- If building the project successfully, you can see flash_ntz.bin in the 'build' folder
- If the 'build' folder has been used by others, you can remove 'build' folder first to have clean build
- If there's some permission issues, you can do "chmod -R 777 <PATH_TO_YOUR_SDK>"

CHAPTER

TWO

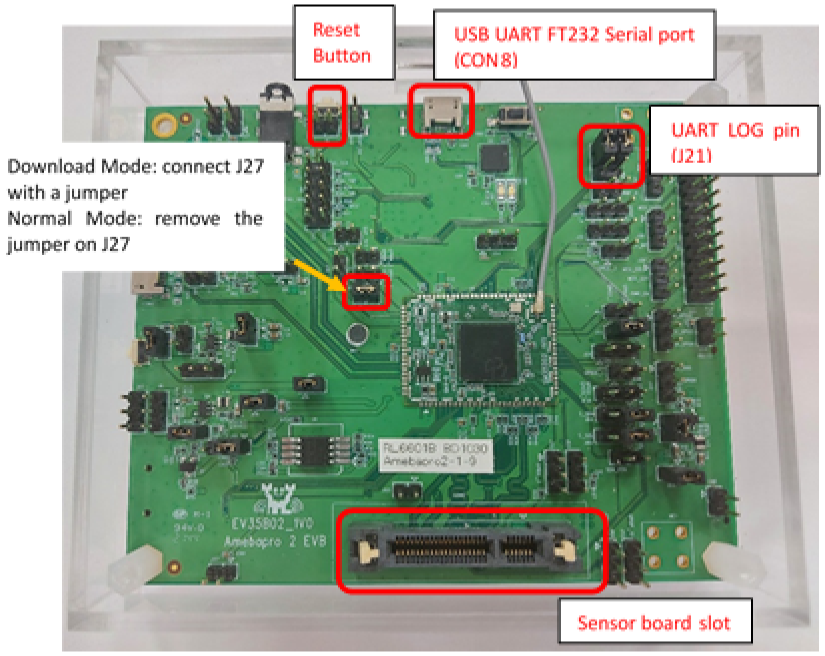
UPLOAD PROCESS

2.1 Upload process

2.1.1 Step 1. Log UART Settings

(1) To use AmebaPro2 log UART, the user needs to connect jumpers to **J21** for **FT232 (CON8)**.

(2) After using CON8 to connect to PC, you can use console tools (like tera term, MoBaxterm) to get log from EVB by setting baud rate as **115200**.



Reset Button
USB UART FT232 Serial port (CON8)
UART LOG pin (J21)
Download Mode: connect J27 with a jumper
Normal Mode: remove the jumper on J27
Sensor board slot

2.1.2 Step 2. Set up Pro2_PG_tool

CHAPTER

THREE

AI GLASS APPLICATIONS

3.1 AI Wearable Glass

3.1.1 Introduction

AI Wearable glass is an easy-to-program platform powered with RTL8735B and BT SoC, which designed to enable a wide range of IoT applications.

This advanced wearable technology integrates multiple peripheral interfaces, including WiFi, Bluetooth Low Energy (BLE), I2S, I2C, and UART, making it versatile and suitable for diverse use cases.

Three primary scenarios define the capabilities of AI Wearable Glass such as Lifetime Snapshot, Lifetime Recording and AI Snapshot.

The data collected through AI Wearable Glass can be uploaded via Bluetooth, enabling integration with applications on smart devices to fully realize IoT implementations.

Powering the device is simple and convenient through a standard Micro USB connection, commonly available with most smart devices, ensuring accessibility and ease of use.

3.1.2 Get Started

This guide will help you set up and configure AI Wearable Glass for building and compilation. Follow these steps to modify the necessary parameters and settings.

Modify File System Configuration

Modify FATFS for SD Card

File: `\component\file_system\fatfs\fatfs_sdcard_api.c`

- Comment out fatfs_sd_close in the function fatfs_sd_init.

Update Module_MP4 Configuration

File: `\component\media\mmfv2\module_mp4.c`

- Update macro definitions:
  1. Undefine `FATFS_SD_CARD`
  2. Undefine `FATFS_RAM`
  3. Define `VFS_ENABLE`
- Modify *mp4_destroy* function:
  1. Comment out:

```
// vfs_user_unregister("sd", VFS_FATFS, VFS_INF_SD);
```

- Modify *mp4_create* function:
  1. Comment out and replace:

```
//vfs_init(NULL);
memcpy(ctx->mp4_muxer->_drv, "aiglass:/", strlen("aiglass:/")); // Set tag
ctx->mp4_muxer->vfs_format_enable = 1; // Enable the VFS format
// if (vfs_user_register("sd", VFS_FATFS, VFS_INF_SD) < 0) {
//     goto mp4_create_fail;
// }
```

Modify System Boot Configuration

Update User Boot Configuration

File: `\component\soc\8735b\misc\platform\user_boot.c`

- **set bl_log_cust_ctrl to DISABLE**

```
uint8_t bl_log_cust_ctrl = DISABLE;
```

Modify Video Boot Stream Configuration

File: `\component\video\driver\RTL8735B\video_user_boot.c`

Update ISP Pre-Setting

- **Open flag ISP_CONTROL_TEST**

```
#define ISP_CONTROL_TEST
```

Modify Video Boot Stream Configuration

Modify the following parameters while keeping others unchanged

- Modify *video_boot_stream_t video_boot_stream* Settings by replacing:
  1. Video Channel 0

```
.video_enable[STREAM_V1] = 1,
.video_snapshot[STREAM_V1] = 0,
.video_drop_frame[STREAM_V1] = 0,
.video_params[STREAM_V1] = {
    .stream_id = STREAM_ID_V1,
    .type = CODEC_H264,
    .resolution = 0,
    .width = 176,
    .height = 144,
    .bps = 1024 * 1024,
    .fps = 15,
    .gop = 15,
    .rc_mode = 2,
    .minQp = 25,
    .maxQp = 48,
    .jpeg_qlevel = 0,
    .rotation = 0,
    .out_buf_size = V1_ENC_BUF_SIZE,
    .out_rsvd_size = 0,
    .direct_output = 0,
    .use_static_addr = 0,
    .fcs = 1 // Enable FCS for channel 0
},
```

  2. Video Channel 1

```
.video_enable[STREAM_V2] = 1,
.video_snapshot[STREAM_V2] = 1,
.video_drop_frame[STREAM_V2] = 0,
.video_params[STREAM_V2] = {
    .stream_id = STREAM_ID_V2,
    .type = CODEC_H264,
    .resolution = 0,
    .width = sensor_params[USE_SENSOR].sensor_width,
```

(continues on next page)

(continued from previous page)

```
    .height = sensor_params[USE_SENSOR].sensor_height,
    .bps = 2 * 1024 * 1024,
    .fps = sensor_params[USE_SENSOR].sensor_fps,
    .gop = sensor_params[USE_SENSOR].sensor_fps,
    .rc_mode = 2,
    .minQp = 25,
    .maxQp = 48,
    .jpeg_qlevel = 0,
    .rotation = 0,
    .out_buf_size = V2_ENC_BUF_SIZE,
    .out_rsvd_size = 0,
    .direct_output = 0,
    .use_static_addr = 0,
    .fcs = 0,
},
```

3. Disable Video Stream 4

```
.video_enable[STREAM_V4] = 0,
```

4. Modify *user_boot_config_init* Settings by adding:

```
video_boot_stream.init_isp_items.enable = 1;
video_boot_stream.init_isp_items.init_brightness = 0;
video_boot_stream.init_isp_items.init_contrast = 50;
video_boot_stream.init_isp_items.init_flicker = 1;
video_boot_stream.init_isp_items.init_hdr_mode = 0;
video_boot_stream.init_isp_items.init_mirrorflip = 0xf3; // Mirror and flip
video_boot_stream.init_isp_items.init_saturation = 50;
video_boot_stream.init_isp_items.init_wdr_level = 50;
video_boot_stream.init_isp_items.init_wdr_mode = 0;
video_boot_stream.init_isp_items.init_mipi_mode = 0
```

Modify Sensor Configuration

File: `\project\realtek_amebapro2_v0_example\inc\sensor.h`

- Modify in *sensor_params* entry by replacing:

```
[SENSOR_SC5356] = {2592, 1944, 24},
```

- Modify *sen_id* entry by replacing:

  SENSOR_GC2053 to SENSOR_SC5356

```
static const unsigned char sen_id[SENSOR_MAX] = {
    SENSOR_DUMMY,
    SENSOR_SC5356,
    SENSOR_GC4653,
    SENSOR_GC4023,
    SENSOR_SC2333
};
```

- Set USE_SENSOR to SENSOR_SC5356

```
#define USE_SENSOR            SENSOR_SC5356
```

- Modify *manual_iq* entry by replacing:

  iq_gc2053 to iq_sc5356

```
static const      char manual_iq[SENSOR_MAX][64] = {
    "iq",
    "iq_sc5356",
    "iq_gc4653",
    "iq_gc4023",
    "iq_sc2333",
};
```

- Set ENABLE_FCS to 1

```
#define ENABLE_FCS            1
```

Modify RAM Disk Size

File: `\component\file_system\fatfs\fatfs_ramdisk_api.c`

- Set RAM Disk Size:

```
#define RAM_DISK_SIZE (1024 * 1024 * 2)
```

Note

This allocates 2MB to store a 720P JPEG image (~1.3MB required).

Modify FAT Time Function

File: `\component\file_system\fatfs\r0.14\diskio.c`

- Modify *DWORD get_fattime(void)* function:

```
attribute((weak)) DWORD get_fattime(void)
```

Set up GCC Building Environment (window)

Installing mingw with ASDK and setting up the CMake

---

(1) Download and extract msys64_v10_3.7z from https://github.com/Ameba-AIoT/ameba-tool-rtos-pro2/releases/tag/msys64_v10_3

(2) Double click "msys2_shell.cmd" from mysys64 folder

(3) After setting up mingw, you need to install cmake. Download cmake in https://github.com/Kitware/CMake/releases/download/v3.20.0-rc1/cmake-3.20.0-rc1-windows-x86_64.msi and install it

(4) Add location of cmake.exe to PATH of msys2_shell by using vim ~/.bashrc and appending path of cmake.exe to environment variable PATH or using editor to directly append the path to file "msys64/home/<USER_FOLDER>/.bashrc"

```
export PATH=/c/Program\ Files/CMake/bin:$PATH
```

Caution

If your PATH contains space characters, remember to use "\" to escape

Note

For the first time adding the CMake PATH, after adding the PATH, you need to re-open the msys2_shell and check the version by:

```
$ cmake --version
cmake version 3.20.0-rc1

CMake suite maintained and supported by Kitware (kitware.com/cmake).
```

Adding toolchain to msys2

(1) Download and extract asdk-10.3.0-mingw32-newlib-build-3633-x86_64.zip from https://github.com/Ameba-AIoT/ameba-toolchain/releases/download/V10.3.0-amebe-rtos-pro2/asdk-10.3.0-mingw32-newlib-build-3633-x86_64.zip

(2) Like adding PATH for cmake, user can add or change the toolchain in "msys64/home/<USER_FOLDER>/.bashrc".

(3) Add toolchain PATH by "export PATH=<path to toolchain>:$PATH".

```
if [ -d "../../asdk-10.3.0" ]; then
    echo "asdk-10.3.0 exist"
    export PATH=/asdk-10.3.0/mingw32/newlib/bin:$PATH
```

Note

Recommand to use the latest provided toolchain or use the version after 10.3.0

Building the project

(1) Open mingw by double clicking "msys2_shell.cmd".

(2) Create folder "build" in project/realtek_amebapro2_v0_example/GCC-RELEASE.

(2) Enter the project location: project/realtek_amebapro2_v0_example/GCC-RELEASE/build.

(4) Run "cmake .. -G"Unix Makefiles" -DCMAKE_TOOLCHAIN_FILE=../toolchain.cmake -DSCENARIO=ai_glass" to create the makefile.

(5) Run "cmake –build . –target flash" to build and generate flash binary.

Note

If building the project successfully, you can see flash_ntz.bin in the 'build' folder

Set up GCC Building Environment (LINUX)

Add toolchain to the linux PATH

(1) Dowload and extract asdk-10.3.0-linux-newlib-build-3638-x86_64.tar.bz2 from https://github.com/Ameba-AIoT/ameba-toolchain/releases/download/V10.3.0-amebe-rtos-pro2/asdk-10.3.0-linux-newlib-build-3638-x86_64.tar.bz2

```
tar -jxvf <PATH_TO_YOUR_TOOLCHAIN.tar.bz2> -C <DIR_TO_EXTRACT>
```

(2) Add toolchain to PATH:

```
export PATH=<PATH_TO_YOUR_TOOLCHAIN>/asdk-10.3.0/linux/newlib/bin:$PATH
```

Note

You can add PATH to ~/.bash_profile

Installing cmake for linux

(1) Install cmake using terminal (like "sudo apt-get -y install cmake"), if the installation is successful, you can get the version by "cmake –version".

Building the project

(1) Create folder "build" in project/realtek_amebapro2_v0_example/GCC-RELEASE.

(2) Open linux terminal and enter the project location: project/realtek_amebapro2_v0_example/GCC-RELEASE/build.

(3) Run "cmake .. -G"Unix Makefiles" -DCMAKE_TOOLCHAIN_FILE=../toolchain.cmake -DSCENARIO=ai_glass" to create the makefile.

(4) Run "cmake –build . –target flash" to build and generate flash binary.

Note

- If building the project successfully, you can see flash_ntz.bin in the 'build' folder
- If the 'build' folder has been used by others, you can remove 'build' folder first to have clean build
- If there's some permission issues, you can do "chmod -R 777 <PATH_TO_YOUR_SDK>"

3.1.3 Media Scenarios

This section describes the different Media Scenarios supported by the AI Wearable Glass.

Supported Media Features

Lifetime Snapshot

This feature allows the AI Wearable Glass to capture a JPEG image from the on-board camera sensor (SC5356) and save it on the EMMC.

- **Image Capture**: High-resolution JPEG image capture using the SC5356 camera sensor.

(1) The life snapshot is triggered each time the push button is pressed.

(2) The image is saved on the EMMC with a unique filename in the format: lifesnap_xxxxx.jpg.

Lifetime Recording

This feature allows the AI Wearable Glass to record video and audio data using the on-board camera sensor (SC5356) and audio codec.

- **Video Capture**: Up to 2K video capture using the SC5356 camera sensor.
- **Audio Capture**: High-quality audio input through the onboard microphone.

(1) A lifetime recording is initiated when the push button is pressed and held.

(2) The recorded data is saved as an MP4 file on the EMMC with a unique filename in the format: liferecord_xxxxx.mp4.

AI Snapshot

This feature allows the AI Wearable Glass to capture a JPEG image from the on-board camera sensor (SC5356), save it on the EMMC and RAM, and transmit it from RAM to a connected application.

- **AI Processing**: Captured images are sent to a mobile app, which process them on an AI server for tasks like AI analysis.

(1) The AI snapshot is triggered each time the push button is "tbc".

(2) The image is saved on the EMMC with a unique filename in the format: ai_snapshot.jpg.

(3) The image is sent to the application from RAM and processes the image on an AI server for tasks like AI analysis."tbc: facial recognition or object detection"

Media Scenarios Public API reference

Media Parameter Related API

1. - **Function**

   ```
   void initial_media_parameters(void);
   ```

   - **Description:**

     This function initialize the snapshot and recording system from the parameters stored in the flash and pre-open the video channel.

   - **Parameters:**

     NA

   - **Return:**

     NA

---

2. - **Function**

   ```
   void deinitial_media(void);
   ```

   - **Description:**

     This function de-initialize media and close the pre-opened video channel.

   - **Parameters:**

     NA

   - **Return:**

     NA

---

3. - **Function**

   ```
   int media_update_record_params(const ai_glass_record_param_t *params);
   ```

   - **Description:**

     This function is use to update the parameters for the lifetime recording.

   - **Parameters:**

     – params – a const pointer for ai_glass_record_param_t structure, used for updating recording parameters.

   - **Return:**

     – MEDIA_OK – update the parameter and storing update result to flash successfully.

     – MEDIA_FAIL – update the parameter failed.

   - **Key Call:**

     1. **media_set_record_params(params)**

     – If **return MEDIA_OK**, this means the parmeters is valid and **call media_update_record_params_to_flash** to save the result to the flash.

   - If **return is not MEDIA_OK**, this means the parmeters is not valid and **return MEDIA_FAIL**.

---

4. - **Function**

```
int media_update_ai_snapshot_params(const ai_glass_snapshot_param_t *params);
```

- **Description:**

  This function is use to update the snapshot parameters for ai recognition.

- **Parameters:**

  - params – a const pointer for ai_glass_snapshot_param_t structure, used for updating snapshot parameters for ai recognition.

- **Return:**

  - MEDIA_OK – update the parameter and storing update result to flash successfully.
  - MEDIA_FAIL – update the parameter failed.

- **Key Call:**

  1. **media_set_ai_snapshot_params(params)**

  - If **return MEDIA_OK**, this means the parmeters is valid and **call media_update_ai_snapshot_params_to_flash** to save the result to the flash.
  - If **return is not MEDIA_OK**, this means the parmeters is not valid and **return MEDIA_FAIL**.

---

5. - **Function**

```
int media_update_life_snapshot_params(const ai_glass_snapshot_param_t *params);
```

- **Description:**

  This function is use to update the snapshot parameters for lifetime.

- **Parameters:**

  - params – a const pointer for ai_glass_snapshot_param_t structure, used for updating snapshot parameters for lifetime.

- **Return:**

  - MEDIA_OK – update the parameter and storing update result to flash successfully.
  - MEDIA_FAIL – update the parameter failed.

- **Key Call:**

  1. **media_set_life_snapshot_params(params)**

  - If **return MEDIA_OK**, this means the parmeters is valid and **call media_update_life_snapshot_params_to_flash** to save the result to the flash.
  - If **return is not MEDIA_OK**, this means the parmeters is not valid and **return MEDIA_FAIL**.

---

6. - **Function**

```
int media_get_record_params(ai_glass_record_param_t *params);
```

- **Description:**

  This function is use to get the parameters for the lifetime recording.

- **Parameters:**

  - params – a pointer to an allocated ai_glass_record_param_t structure, used for storing recording parameters.

- **Return:**

  - MEDIA_OK – get the parameter successfully.
  - MEDIA_FAIL – get the parameter failed.

---

7. - **Function**

```
int media_update_record_time(uint16_t record_length);
```

- **Description:**

  This function updates the recording time based on the specified record length.

- **Parameters:**

  - record_length – the length of the record in seconds.

- **Return:**

  - MEDIA_OK – update the record length successfully.
  - MEDIA_FAIL – update the record length failed.

---

8. - **Function**

```
int media_get_ai_snapshot_params(ai_glass_snapshot_param_t *params);
```

- **Description:**

  This function is use to get the snapshot parameters for ai recognition.

- **Parameters:**

  - params – a pointer to an allocated ai_glass_record_param_t structure, used for storing get the snapshot parameters for ai recognition.

- **Return:**

  - MEDIA_OK – get the parameter successfully.
  - MEDIA_FAIL – get the parameter failed.

---

9. - **Function**

```
int media_get_life_snapshot_params(ai_glass_snapshot_param_t *params);
```

- **Description:**

  This function is use to get the snapshot parameters for lifetime.

- **Parameters:**
  - params – a pointer to an allocated ai_glass_record_param_t structure, used for storing get the snapshot parameters for lifetime.
- **Return:**
  - MEDIA_OK – get the parameter successfully.
  - MEDIA_FAIL – get the parameter failed.

---

10. - **Function**

```
void print_record_data(const ai_glass_record_param_t *params);
```

- **Description:**

  This function prints the recording parameters pointed to by *params*.
- **Parameters:**
  - params – a pointer to a constant *ai_glass_record_param_t* structure, used for printing recording parameters.
- **Return:**

  NA

---

11. - **Function**

```
void print_snapshot_data(const ai_glass_snapshot_param_t *params);
```

- **Description:**

  This function prints the snapshot parameters pointed to by *params*.
- **Parameters:**
  - params – a pointer to a constant *ai_glass_snapshot_param_t* structure, used for printing snapshot parameters.
- **Return:**

  NA

Lifetime Snapshot API

1. - **Function**

```
int lifetime_snapshot_initialize(void);
```

- **Description:**

  This function initializes the lifetime snapshot system by setting up the necessary modules, configuring video parameters, and linking components together for capturing and saving images.
- **Parameters:**

  NA

- **Return:**
  - 0 – the needed modules for lifetime snapshot set up successfully.
  - -1 – the needed modules for lifetime snapshot set up fail.
  - -2 – the modules has already been set up.
- **Key Call:**
  1. **if (is_file_saved != LIFESNAP_IDLE)**
  - If **is_file_saved is not LIFESNAP_IDLE**, return an error **(ret = -2)**.
  2. **if (ls_filesaver_ctx)**
  - If **ls_filesaver_ctx** is true, assigns **lifetime_snapshot_file_save** as its handler.
  3. **media_get_life_snapshot_params(&life_snap_param)**
  - To retrieve the desired snapshot width, height, and quality.
  4. **Configure Video Parameters**
  - If the requested resolution is within sensor limits, no scaling is required **(need_scaleup = 0)**.
  - If the requested resolution exceeds sensor width or height, scaling is needed **(need_scaleup = 1)**.
  5. **Video module configuration**

     5.1. **mm_module_ctrl(ls_snapshot_ctx, CMD_VIDEO_SET_PARAMS, (int) & (ls_video_params.params));**
     - Trigger a command to the video module to set video parameters.

     5.2. **mm_module_ctrl(ls_snapshot_ctx, MM_CMD_SET_QUEUE_LEN, 2)**
     - Trigger a command to the video module to set the length of the video queue to 2.

     5.3. **mm_module_ctrl(ls_snapshot_ctx, MM_CMD_INIT_QUEUE_ITEMS, MMQI_FLAG_DYNAMIC)**
     - Trigger a command to the video module to initializes the queue for dynamic handling of video frames.
  6. **Create SISO (Single Input Single Output) Link**

     6.1. **siso_ctrl(ls_siso_snapshot_filesaver, MMIC_CMD_SET_SECURE_CONTEXT, 1, 0)**
     - Trigger a command to ensure that the data processing inside the SISO pipeline is handled securely, which prevent unauthorized access or tampering.

     6.2. **siso_ctrl(ls_siso_snapshot_filesaver, MMIC_CMD_ADD_INPUT, (uint32_t)ls_snapshot_ctx, 0)**
     - Trigger a command to connect video module **(ls_snapshot_ctx)** as Input, which tells the pipeline where to get the lifetime snapshot from.

     6.3. **siso_ctrl(ls_siso_snapshot_filesaver, MMIC_CMD_ADD_OUTPUT, (uint32_t)ls_filesaver_ctx, 0)**
     - Trigger a command to connect file saver **(ls_filesaver_ctx)** as Output, which tells the pipeline where to send the lifetime snapshot after capturing.

     6.4. **siso_ctrl(ls_siso_snapshot_filesaver, MMIC_CMD_SET_TASKPRIORITY, LIFE_SNAP_PRIORITY, 0)**

  - Trigger a command to set processing priority, which sets how important this process is compared to others running on the system.

  6.5. **siso_start(ls_siso_snapshot_filesaver)**

  - Trigger a command to activate the pipeline, allowing lifetime snapshots to flow from the video module to the file saver.

7. **mm_module_ctrl(ls_snapshot_ctx, CMD_VIDEO_APPLY, ls_video_params.params.stream_id)**

- Trigger a command to start video stream.

8. **is_file_saved = LIFESNAP_START**

- Updates the **status** to **LIFESNAP_START**, which indicates that the lifetime snapshot process has started successfully.

9. **goto endoflifesnapshot**

- If any step fails, the function calls lifetime_snapshot_deinitialize() and returns an error.

---

2. - **Function**

```
int lifetime_snapshot_take(const char *file_name);
```

- **Description:**

  This function initiates a lifetime snapshot process, saves the captured image to a specified file, and waits for the process to complete. It ensures that a lifetime snapshot is only taken if the system is in the LIFESNAP_START state.

- **Parameters:**

  - file_name – Pass in the unique filename.

- **Return:**

  - 0 – take snapshot successfully.
  - -1 – take snapshot fail.

- **Key Call:**

  1. **if (is_file_saved == LIFESNAP_START)**

  - Ensures that a lifetime snapshot can only be taken if the system is in the correct state **(LIFESNAP_START)**.

  2. **snprintf(snapshot_name, MAXIMUM_FILE_SIZE, "%s", file_name)**

  - Saves the provided file name into snapshot_name (ensuring it does not exceed **MAXIMUM_FILE_SIZE**).

  3. **mm_module_ctrl(ls_filesaver_ctx, CMD_FILESAVER_SET_SAVE_FILE_PATH, (int)snapshot_name)**

  - Trigger a command to set file path for saving.

  4. **is_file_saved = LIFESNAP_TAKE**

  - Updates the **status** to **LIFESNAP_TAKE**, which indicates that the lifetime snapshot is in progress.

  5. **mm_module_ctrl(ls_snapshot_ctx, CMD_VIDEO_YUV, 1)**

  - Trigger a command to the video module **(ls_snapshot_ctx)** to capture a one shot lifetime snapshot.

  6. **Wait for Snapshot Completion**

   – Ensures that the lifetime snapshot has been processed.

7. **The function returns**

   – **0** if the lifetime snapshot taken successfully.

   – **-1** Lifetime snapshot was not taken because the system was not in the correct state **(LIFESNAP_START)**.

---

3. - **Function**

```
int lifetime_snapshot_deinitialize(void);
```

- **Description:**

  The lifetime_snapshot_deinitialize function is responsible for deinitializing the lifetime snapshot module. It ensures that no lifetime snapshot is in progress before releasing resources such as filesaver, video snapshot module, and the linker.

- **Parameters:**

  NA

- **Return:**

  – 0 – de-initialize lifetime snapshot modules successfully.

- **Key Call:**

  1. **if (is_file_saved != LIFESNAP_TAKE)**

  – Ensures that lifetime snapshot is not in progress before proceeding with deinitialization.

  2. **siso_pause(ls_siso_snapshot_filesaver)**

  – Ensures that **ls_siso_snapshot_filesaver** module is not empty before pausing the data flow between modules.

  3. **mm_module_ctrl(ls_snapshot_ctx, CMD_VIDEO_STREAM_STOP, 0)**

  – Ensures that **ls_snapshot_ctx** module is not empty before triggering a command to stops video module from capturing limetime snapshots.

  4. **siso_delete(ls_siso_snapshot_filesaver)**

  – Ensures that **ls_siso_snapshot_filesaver** module is not empty before removing the pipeline handling lifetime snapshot transfers.

  5. **ls_siso_snapshot_filesaver = NULL**

  – Reset **ls_siso_snapshot_filesaver** context.

  6. **ls_snapshot_ctx = mm_module_close(ls_snapshot_ctx)**

  – Ensures that **ls_snapshot_ctx** module is not empty before freeing the video module.

  7. **ls_filesaver_ctx = mm_module_close(ls_filesaver_ctx)**

  – Ensures that **ls_filesaver_ctx** module is not empty before freeing the file-saving module.

  8. **Reset Lifetime Snapshot Status**

  – is_file_saved = LIFESNAP_IDLE

Lifetime Recording API

1. - **Function**

```
void lifetime_recording_initialize(void);
```

- **Description:**

  The lifetime_recording_initialize function are responsible on the full process of recording such as setting video and audio parameters, capturing video and audio data, video and audio encode, and storing to EMMC storage device.

- **Parameters:**

  NA

- **Return:**

  NA

- **Key Call:**

  1. **Generates a unique filename for the recording**

  – Uses extdisk_generate_unique_filename() to create a timestamped MP4 filename.

  2. **If ENABLE_GET_GSENSOR_INFO is enable:**

  – Calls **lr_gyro_init**().

  3. **If GSENSOR_RECORD_FAST is enable:**

  – Calls **start_gyro_to_exdisk_process**().

  4. **Fetch & Set Recording Parameters**

  – Calls media_get_record_params() to get members of **ai_glass_record_param_t**.

  – Updates MP4 parameters and video encoding parameters.

  5. **Initialize Audio Module**

  – Opens the audio module (audio_module or i2s_module).

  – Sets sample rate, queue length, and parameters.

  6. **Initialize AAC Module**

  – Opens Opens the aac_module and configures it.

  – Sets sample rate, queue length, and parameters.

  7. **Initialize MP4 Recording**

  – Opens mp4_module and sets parameters.

  – Assigns callbacks for stop, end, and error handling.

  – Starts MP4 recording with CMD_MP4_START.

  8. **Initialize Video Module**

  – Opens video_module and configures encoding.

  – Adjusts queue size to prevent data loss.

  9. **Create & Start Pipelines**

  – SISO (Single Input Single Output) for Audio to AAC.

- MISO (Multiple Input Single Output) for Video & Audio to MP4.

10. **Apply Settings & Start Recording**

- Sets current state to STATE_RECORDING.
- Calls CMD_VIDEO_APPLY and (CMD_AUDIO_APPLY or CMD_I2S_APPLY).

---

2. - **Function**

```
void lifetime_recording_deinitialize(void);
```

- **Description:**

  The lifetime_recording_deinitialize function is responsible for deinitializing the lifetime recording module. It ensures that no lifetime recording is in progress before releasing resources such as audio module, video snapshot module, and the linker.

- **Parameters:**

  NA

- **Return:**

  NA

- **Key Call:**

  1. **siso_pause(lr_siso_audio_aac)**

  - Pausing the audio AAC SISO linker.

  2. **miso_pause(lr_miso_video_aac_mp4, MM_OUTPUT0)**

  - Pauses the MP4 recording/RTSP stream MISO linker.

  3. **mm_module_ctrl(lr_video_ctx, CMD_VIDEO_STREAM_STOP, lr_video_params.stream_id)**

  - Ensures that **lr_video_ctx** module is not empty before triggering a command to stops video stream.

  4. **If AUDIO_SRC==AUDIO_INTERFACE**

  - mm_module_ctrl(lr_audio_ctx, CMD_AUDIO_SET_TRX, 0)
  - Ensures that **lr_audio_ctx** module is not empty before triggering a command to stop audio transmission.

  5. **If AUDIO_SRC!=AUDIO_INTERFACE**

  - mm_module_ctrl(lr_audio_ctx, CMD_I2S_SET_TRX, 0)
  - Ensures that **lr_audio_ctx** module is not empty before triggering a command to stop I2S transmission.

  6. **mm_module_ctrl(lr_aac_ctx, CMD_AAC_STOP, 0)**

  - Ensures that **lr_aac_ctx** module is not empty before triggering a command to stops AAC encoding.

  7. **mm_module_ctrl(lr_mp4_ctx, CMD_MP4_STOP_IMMEDIATELY, 0)**

  - Ensures that **lr_mp4_ctx** module is not empty before triggering a command to immediately stops MP4 recording.

  8. **siso_delete(lr_siso_audio_aac)**

  - Deletes the audio SISO linker and sets it to NULL.

  9. **miso_delete(lr_miso_video_aac_mp4)**

   - Deletes the video MISO linker and sets it to NULL.

10. **Close All Active Modules and set to null**

   - mm_module_close(lr_audio_ctx)
   - lr_audio_ctx = NULL
      * mm_module_close(lr_aac_ctx)
   - lr_aac_ctx = NULL
   - mm_module_close(lr_mp4_ctx)
   - lr_mp4_ctx = NULL
   - mm_module_close(lr_video_ctx)
   - lr_video_ctx = NULL

11. **Sets current_state = STATE_IDLE**

AI Sanpshot API

1. - **Function**

```
int ai_snapshot_initialize(void);
```

   - **Description:**

     This function initializes the AI snapshot module by allocating memory for the AI snapshot context, configuring video parameters, initializing synchronization mechanisms (semaphore and mutex), and setting up the video snapshot module. If the initialization is successful, it registers the snapshot callback and applies the necessary parameters. If any step fails, it deinitializes the snapshot context and returns an error.

   - **Parameters:**

     NA

   - **Return:**

     - 0 – the needed modules for ai recognition set up successfully.
     - -1 – the needed modules for ai recognition snapshot set up fail.
     - -2 – the modules has already been set up.

   - **Key Call:**

     1. **Allocating memory for snapshot context and configuring video parameters**

     2. **rtw_init_sema(&ai_snap_ctx->snapshot_sema, 0)**

     - Initializes a semaphore for synchronizing snapshot operations.

     3. **rtw_mutex_init(&ai_snap_ctx->snapshot_mutex)**

     - Initializes a mutex for thread-safe access.

     4. **ai_snap_ctx->snapshot_write = aisnapshot_write_picture**

     - Assigns a function for writing snapshots.

5. **Video module configuration**

   5.1. **mm_module_ctrl(ai_snap_ctx->video_snapshot_ctx, CMD_VIDEO_SNAPSHOT_CB, (int)video_snapshot_cb)**

   - Trigger a command to the video module to registers the snapshot callback.

   5.2. **mm_module_ctrl(ai_snap_ctx->video_snapshot_ctx, CMD_VIDEO_SET_PARAMS, (int) & (ai_snap_ctx->video_snapshot_params))**

   - Trigger a command to the video module to set video parameters.

   5.3. **mm_module_ctrl(ai_snap_ctx->video_snapshot_ctx, CMD_VIDEO_APPLY, ai_snap_ctx->video_snapshot_params.stream_id)**

   - Trigger a command to the video module to applies configuration.

6. **If video module initialization fails**

- Deinitialize AI snapshot.

7. **The function returns**

- **0** if successful.
- **-1** if video module fails.
- **-2** if AI snapshot is already in processing.

---

2. - **Function**

```
int ai_snapshot_take(const char *file_name);
```

- **Description:**

  This function captures an AI snapshot and saves it to the specified file. It ensures thread safety using a mutex and controls the video snapshot module.

- **Parameters:**
  - file_name – Pass in the filename, **"ai_snapshot.jpg"**.
- **Return:**
  - 0 – take snapshot successfully.
  - -1 – take snapshot fail.
- **Key Call:**

  1. **if (ai_snap_ctx)**

  - Ensures that the snapshot context exists before proceeding.

  2. **rtw_mutex_get(&ai_snap_ctx->snapshot_mutex)**

  - This function obtains a mutex semaphore.

  3. **ai_snap_ctx->take_snapshot = 1**

  - Set to **1**, indicates that an AI snapshot is on process.

  4. **mm_module_ctrl(ai_snap_ctx->video_snapshot_ctx, CMD_VIDEO_SNAPSHOT, 1)**

  - Trigger a command to the video module to take a snapshot.

  5. **video_capture_snapshot(file_name)**

- Passes file_name to the function to proceed with image saving
- Returns the **success (0)** or **failure (-1)** status.

6. **ai_snap_ctx->take_snapshot = 0**

- Set to **0**, indicates the snapshot process as complete.

7. **rtw_mutex_put(&ai_snap_ctx->snapshot_mutex)**

- This function releases a mutex semaphore.

8. **The function returns**

- **0** if successful.
- **-1** if the AI snapshot context is not initialized.

---

3. - **Function**

```
int ai_snapshot_deinitialize(void);
```

- **Description:**

  The ai_snapshot_deinitialize function is responsible for deinitializing the AI snapshot module. It ensures that no AI snapshot is in progress before releasing resources such as semaphores, mutexes, and the video snapshot module.

- **Parameters:**

  NA

- **Return:**

  - 0 – de-initialize ai snapshot modules successfully.

- **Key Call:**

  1. **if (ai_snap_ctx)**

  - Ensures that the AI snapshot context exists before proceeding.

  2. **if (ai_snap_ctx->take_snapshot)**

  - Ensures that the function does not deinitialize the AI snapshot module if an AI snapshot is currently in progress.

  3. **Clears and stops all configuration**

     3.1. **rtw_free_sema(&ai_snap_ctx->snapshot_sema)**

     - This function deletes the semaphore

     3.2. **rtw_mutex_free(&ai_snap_ctx->snapshot_mutex)**

     - This function deletes the mutex semaphore.

     3.3. **mm_module_ctrl(ai_snap_ctx->video_snapshot_ctx, CMD_VIDEO_STREAM_STOP, ai_snap_ctx->video_snapshot_params.stream_id)**

     - Trigger a command to stops the video stream for the AI snapshot.

     3.4. **mm_module_close(ai_snap_ctx->video_snapshot_ctx)**

     - Closes the video snapshot module.

     3.5. **Free allocated memory and reset the context**

- free(ai_snap_ctx).
- ai_snap_ctx = NULL.

4. **The function returns**

- **0** if AI snapshot module deinitialized successfully.
- **-1** if AI snapshot is currently in progress, deinitialization aborted.

Media Scenarios Internal Functions reference

Media Parameter Related Internal Functions

1. - **Function**

```
static int record_data_check(const ai_glass_record_param_t *params);
```

- **Description:**

  This function checks whether the parameters supplied for a recording session are valid.

- **Parameters:**
  - params – a pointer to a structure (ai_glass_record_param_t) containing the configuration parameters for a recording session.

- **Return:**
  - MEDIA_OK – all parameters are valid.
  - MEDIA_FAIL – unknown error.
  - others – describe which parameter is invalid.

- **Key Call:**
  1. Parameter Null Check:
  - If params is NULL, the function returns MEDIA_FAIL. This is an immediate termination to prevent dereferencing a null pointer, which could lead to a crash.
  2. Parameter Validations:
  - The function checks each aspect of params using presumed validation macros or functions. These check whether the parameter values conform to expected ranges or types. These checks involve:
  - IS_VALID_RECORD_TYPE(params->type): Validate the recording type.
  - IS_VALID_RECORD_WIDTH(params->width): Validate the recording width.
  - IS_VALID_RECORD_HEIGHT(params->height): Validate the recording height.
  - IS_VALID_RECORD_BPS(params->bps): Validate the bits per second rate.
  - IS_VALID_RECORD_FPS(params->fps): Validate the frame rate.
  - IS_VALID_RECORD_GOP(params->gop): Validate the Group of Pictures size.
  - IS_VALID_RECORD_RCMODE(params->rc_mode): Validate the rate control mode.
  - IS_VALID_RECORD_RECTIME(params->record_length): Validate the maximum recording length.

2. - **Function**

```
static int ai_snapshot_data_check(const ai_glass_snapshot_param_t *params);
```

- **Description:**

  This function checks whether the parameters supplied for a ai snapshot session are valid.

- **Parameters:**
  - params – a pointer to a structure (ai_glass_snapshot_param_t) containing the configuration parameters for a snapshot session.

- **Return:**
  - MEDIA_OK – all parameters are valid.
  - MEDIA_FAIL – unknown error.
  - others – describe which parameter is invalid.

- **Key Call:**
  1. Parameter Null Check:
  - If params is NULL, the function returns MEDIA_FAIL. This is an immediate termination to prevent dereferencing a null pointer, which could lead to a crash.
  2. Parameter Validations:
  - The function checks each aspect of params using presumed validation macros or functions. These check whether the parameter values conform to expected ranges or types. These checks involve:
  - IS_VALID_SNAP_TYPE(params->type): Validate the snapshot type.
  - IS_VALID_LIFESNAP_WIDTH(params->width): Validate the snapshot width.
  - IS_VALID_LIFESNAP_HEIGHT(params->height): Validate the snapshot height.
  - IS_VALID_SNAP_QVALUE(params->bps): Validate q value for jpeg file.

3. - **Function**

```
static int life_snapshot_data_check(const ai_glass_snapshot_param_t *params);
```

- **Description:**

  This function checks whether the parameters supplied for a lifetime snapshot session are valid.

- **Parameters:**
  - params – a pointer to a structure (ai_glass_snapshot_param_t) containing the configuration parameters for a snapshot session.

- **Return:**
  - MEDIA_OK – all parameters are valid.
  - MEDIA_FAIL – unknown error.
  - others – describe which parameter is invalid.

- **Key Call:**

1. Parameter Null Check:

- If params is NULL, the function returns MEDIA_FAIL. This is an immediate termination to prevent dereferencing a null pointer, which could lead to a crash.

2. Parameter Validations:

- The function checks each aspect of params using presumed validation macros or functions. These check whether the parameter values conform to expected ranges or types. These checks involve:
- IS_VALID_SNAP_TYPE(params->type): Validate the snapshot type.
- IS_VALID_LIFESNAP_WIDTH(params->width): Validate the snapshot width.
- IS_VALID_LIFESNAP_HEIGHT(params->height): Validate the snapshot height.
- IS_VALID_SNAP_QVALUE(params->bps): Validate q value for jpeg file.

---

4. - **Function**

```
static int record_data_update_if_valid(ai_glass_record_param_t *ori_params,␣
↪const ai_glass_record_param_t *params);
```

- **Description:**

  This function checks and updates a set of recording parameters if the new parameters are valid and different from the current ones.

- **Parameters:**
  - ori_params – a pointer to the original recording parameters, which may be updated if new valid parameters are provided.
  - params – a pointer to the proposed new recording parameters that need validation.

- **Return:**
  - MEDIA_OK – the parameters in ori_params has been updated.
  - MEDIA_NO_NEED_TO_UPDATE – there is no parameters in ori_params been updated.

- **Key Call:**

  1. Initialization of Update Flag:

     - int need_update = 0;: A flag to determine if any parameter needs updating. If set to 1 after checks, it indicates changes were made to the original parameters.

     2. Validation and Conditional Update:

     - Each parameter is checked for validity using the IS_VALID_* macros before comparing it with the original parameter:
     - Type: Updated if valid and different.
     - Resolution (Width & Height): Updated if both width and height are valid and different. Additionally, updates the Region of Interest (ROI) if the new dimensions exceed current ROI constraints.
     - ROI: Updated if valid and new ROI constraints fit within the current resolution.
     - Bitrate (BPS), Frames Per Second (FPS), Group of Pictures (GOP), Rate Control Mode (RC Mode), Recording Time: Each parameter updates the original parameters if valid and different.

---

5. - **Function**

```
static int ai_snapshot_update_if_valid(ai_glass_snapshot_param_t *ori_params,␣
↪const ai_glass_snapshot_param_t *params);
```

- **Description:**

  This function checks and updates a set of ai snapshot parameters if the new parameters are valid and different from the current ones.

- **Parameters:**

  – ori_params – a pointer to the original ai snapshot parameters, which may be updated if new valid parameters are provided.

  – params – a pointer to the proposed new ai snapshot parameters that need validation.

- **Return:**

  – MEDIA_OK – the parameters in ori_params has been updated.

  – MEDIA_NO_NEED_TO_UPDATE – there is no parameters in ori_params been updated.

- **Key Call:**

  1. Initialization of Update Flag:

  – int need_update = 0;: This flag is used to track whether any updates to the original parameters are needed. If the flag is set to 1, it indicates at least one parameter has been updated.

  2. Validation and Conditional Update:

  – Snapshot Type: It checks if the type is valid and different from the original. If so, an update is performed.

  – Resolution (Width & Height): Both the width and height are validated. If either of them changes, the function updates them and ensures the ROI fits within the new dimensions.

  – ROI adjustment happens only if the new width or height exceeds the current ROI range, resetting the ROI to start from (0,0).

  – Region of Interest (ROI): Validates the ROI against sensor dimensions and checks if it fits within the new snapshot dimensions. It ensures the x and y dimensions are positive and updates them if the conditions are met.

  – JPEG Quality Level: If the JPEG quality level is valid and different, the jpeg_qlevel is updated accordingly.

---

6. - **Function**

```
static int life_snapshot_update_if_valid(ai_glass_snapshot_param_t *ori_params,
↪ const ai_glass_snapshot_param_t *params);
```

- **Description:**

  This function checks and updates a set of lifetime snapshot parameters if the new parameters are valid and different from the current ones.

- **Parameters:**

  – ori_params – a pointer to the original lifetime snapshot parameters, which may be updated if new valid parameters are provided.

  – params – a pointer to the proposed new lifetime snapshot parameters that need validation.

- **Return:**

- MEDIA_OK – the parameters in ori_params has been updated.
- MEDIA_NO_NEED_TO_UPDATE – there is no parameters in ori_params been updated.

- **Key Call:**

1. Initialization of Update Flag:

- int need_update = 0;: This flag is used to track whether any updates to the original parameters are needed. If the flag is set to 1, it indicates at least one parameter has been updated.

2. Validation and Conditional Update:

- Snapshot Type: It checks if the type is valid and different from the original. If so, an update is performed.
- Resolution (Width & Height): Both the width and height are validated. If either of them changes, the function updates them and ensures the ROI fits within the new dimensions.
- ROI adjustment happens only if the new width or height exceeds the current ROI range, resetting the ROI to start from (0,0).
- Region of Interest (ROI): Validates the ROI against sensor dimensions and checks if it fits within the new snapshot dimensions. It ensures the x and y dimensions are positive and updates them if the conditions are met.
- JPEG Quality Level: If the JPEG quality level is valid and different, the jpeg_qlevel is updated accordingly.

---

7. - **Function**

```
static int media_set_record_params(const ai_glass_record_param_t *params);
```

- **Description:**

  This function is aim to update global record parameters based on new input parameters.

- **Parameters:**
  - params – a constant pointer to a structure containing the new record parameters that need to be applied after validation.

- **Return:**
  - MEDIA_OK – the parameters in ori_params has been updated.
  - MEDIA_NO_NEED_TO_UPDATE – there is no parameters in ori_params been updated.

- **Key Call:**

1. Temporary Copy Creation:

- "ai_glass_record_param_t temp_params = {0};": Initializes a temporary structure to hold a copy of the current record parameters.
- "memcpy(&temp_params, &record_params, sizeof(ai_glass_record_param_t));": Copies the current global record parameters into the temporary structure. This ensures that changes can be evaluated without directly affecting the global state.

2. Validation and Update Check:

- "int ret = record_data_update_if_valid(&temp_params, params);": Calls record_data_update_if_valid to determine if any changes to temp_params are necessary based on the new input params. The return value ret indicates whether an update is applied (e.g., MEDIA_OK) or not needed (MEDIA_NO_NEED_TO_UPDATE).

- If ret is MEDIA_OK, it signifies that the new parameters are valid and different from the original. The function proceeds to update the global record_params with the changes made in temp_params.

3. Final Update of Global Parameters:

- "memcpy(&record_params, &temp_params, sizeof(ai_glass_record_param_t));": If updates are valid, this line copies the validated and potentially modified parameters from temp_params back to the global record_params, effectively applying the changes.

---

8. - **Function**

```
static int media_set_ai_snapshot_params(const ai_glass_snapshot_param_t␣
↪*params);
```

- **Description:**

  This function is aim to update global ai snapshot parameters based on new input parameters.

- **Parameters:**
  - params – a constant pointer to a structure containing the new snapshot parameters that need to be applied after validation.
- **Return:**
  - MEDIA_OK – the parameters in ori_params has been updated.
  - MEDIA_NO_NEED_TO_UPDATE – there is no parameters in ori_params been updated.
- **Key Call:**

  1. Temporary Copy Creation:

  - "ai_glass_snapshot_param_t temp_params = {0};": Initializes a temporary structure to hold a copy of the current snapshot parameters.
  - "memcpy(&temp_params, &ai_snapshot_params, sizeof(ai_glass_snapshot_param_t));": Copies the current global record parameters into the temporary structure. This ensures that changes can be evaluated without directly affecting the global state.

  2. Validation and Update Check:

  - "int ret = ai_snapshot_update_if_valid(&temp_params, params);": Calls ai_snapshot_update_if_valid to determine if any changes to temp_params are necessary based on the new input params. The return value ret indicates whether an update is applied (e.g., MEDIA_OK) or not needed (MEDIA_NO_NEED_TO_UPDATE).
  - If ret is MEDIA_OK, it signifies that the new parameters are valid and different from the original. The function proceeds to update the global record_params with the changes made in temp_params.

  3. Final Update of Global Parameters:

  - "memcpy(&ai_snapshot_params, &temp_params, sizeof(ai_glass_snapshot_param_t));": If updates are valid, this line copies the validated and potentially modified parameters from temp_params back to the global ai_snapshot_params, effectively applying the changes.

---

9. - **Function**

```
static int  media_set_life_snapshot_params(const ai_glass_snapshot_param_t␣
↪*params);
```

- **Description:**

  This function is aim to update global lifetime snapshot parameters based on new input parameters.

- **Parameters:**

  - params – a constant pointer to a structure containing the new snapshot parameters that need to be applied after validation.

- **Return:**

  - MEDIA_OK – the parameters in ori_params has been updated.
  - MEDIA_NO_NEED_TO_UPDATE – there is no parameters in ori_params been updated.

- **Key Call:**

  1. Temporary Copy Creation:

  - "ai_glass_snapshot_param_t temp_params = {0};": Initializes a temporary structure to hold a copy of the current snapshot parameters.
  - "memcpy(&temp_params, &life_snapshot_params, sizeof(ai_glass_snapshot_param_t));": Copies the current global record parameters into the temporary structure. This ensures that changes can be evaluated without directly affecting the global state.

  2. Validation and Update Check:

  - "int ret = life_snapshot_update_if_valid(&temp_params, params);": Calls life_snapshot_update_if_valid to determine if any changes to temp_params are necessary based on the new input params. The return value ret indicates whether an update is applied (e.g., MEDIA_OK) or not needed (MEDIA_NO_NEED_TO_UPDATE).
  - If ret is MEDIA_OK, it signifies that the new parameters are valid and different from the original. The function proceeds to update the global record_params with the changes made in temp_params.

  3. Final Update of Global Parameters:

  - "memcpy(&life_snapshot_params, &temp_params, sizeof(ai_glass_snapshot_param_t));": If updates are valid, this line copies the validated and potentially modified parameters from temp_params back to the global ai_snapshot_params, effectively applying the changes.

---

10. - **Function**

```
static int media_update_record_params_to_flash(const ai_glass_record_param_t␣
↪*params);
```

- **Description:**

  This function is aim to save record parameters to flash memory.

- **Parameters:**

  - params – a constant pointer to the structure containing the record parameters to be saved to flash memory.

- **Return:**

  - MEDIA_OK – indicating the operation was successful.
  - MEDIA_FAIL – indicating the operation was fail.

- **Key Call:**

  1. Buffer Allocation:

- "unsigned char *record_buf = malloc(FLASH_REC_BLOCK_SIZE);": Allocates a buffer of size FLASH_REC_BLOCK_SIZE, typically 2KB, to temporarily store the data to be written to flash memory.

2. Flash Address Determination:

- "unsigned int flash_addr = 0;": Initializes the flash address variable.
- The flash address is set, depending on the boot selection state retrieved from sys_get_boot_sel(). Currently, it only accommodates one scenario with a placeholder for future extension (like NAND flash).

3. Null Check for Buffer:

- Checks if record_buf is NULL (allocation failed), and returns MEDIA_FAIL.

4. Buffer Preparation:

- "memset(record_buf, 0x00, FLASH_REC_BLOCK_SIZE);": Clears the allocated buffer.
- "record_buf[0] = 'R'; … record_buf[3] = 'D';": Sets a header/tag in the buffer for identification as record parameters ("RECD").

5. Data Copying:

- "memcpy(record_buf + 4, params, sizeof(ai_glass_record_param_t));": Copies the record parameters into the buffer, starting after the identification tag.

6. Flash Write and Read Operations:

- "ftl_common_write(flash_addr, record_buf, FLASH_REC_BLOCK_SIZE);": Writes the prepared buffer to the determined flash address.
- Clears the buffer again with 0xff to ensure it reflects only the data read from flash.
- "ftl_common_read(flash_addr, record_buf, FLASH_REC_BLOCK_SIZE);": Reads the data back from flash memory to verify the write.

7. Data Verification and Logging:

- Reads the record parameters from the buffer (record_buf + 4 due to the identification tag).
- Outputs a detailed log with the parameters to provide verification and traceability, ensuring the correct data was written and read.

8. Buffer Deallocation:

- Frees the allocated buffer to prevent memory leaks.

---

11. - **Function**

```
static int media_update_ai_snapshot_params_to_flash(const ai_glass_snapshot_
↪param_t *params);
```

- **Description:**

  This function is aim to save ai snapshot parameters to flash memory.

- **Parameters:**
  - params – a constant pointer to the structure containing the ai snapshot parameters to be saved to flash memory.

- **Return:**
  - MEDIA_OK – indicating the operation was successful.

- MEDIA_FAIL – indicating the operation was fail.

- **Key Call:**

  1. Buffer Allocation:

  - "unsigned char *ai_snap_buf = malloc(FLASH_AI_SNAP_BLOCK_SIZE);": Allocates a buffer of size FLASH_AI_SNAP_BLOCK_SIZE, typically 2KB, to temporarily store the data to be written to flash memory.

  2. Flash Address Determination:

  - "unsigned int flash_addr = 0;": Initializes the flash address variable.
  - The flash address is set, depending on the boot selection state retrieved from sys_get_boot_sel(). Currently, it only accommodates one scenario with a placeholder for future extension (like NAND flash).

  3. Null Check for Buffer:

  - Checks if record_buf is NULL (allocation failed), and returns MEDIA_FAIL.

  4. Buffer Preparation:

  - "memset(record_buf, 0x00, FLASH_REC_BLOCK_SIZE);": Clears the allocated buffer.
  - "record_buf[0] = 'A'; … record_buf[5] = 'P';": Sets a header/tag in the buffer for identification as record parameters ("AISNAP").

  5. Data Copying:

  - "memcpy(ai_snap_buf + 6, params, sizeof(ai_glass_snapshot_param_t));": Copies the ai snapshot parameters into the buffer, starting after the identification tag.

  6. Flash Write and Read Operations:

  - "ftl_common_write(flash_addr, ai_snap_buf, FLASH_AI_SNAP_BLOCK_SIZE);": Writes the prepared buffer to the determined flash address.
  - Clears the buffer again with 0xff to ensure it reflects only the data read from flash.
  - "ftl_common_read(flash_addr, ai_snap_buf, FLASH_AI_SNAP_BLOCK_SIZE);": Reads the data back from flash memory to verify the write.

  7. Data Verification and Logging:

  - Reads the ai snapshot parameters from the buffer (ai_snap_buf + 6 due to the identification tag).
  - Outputs a detailed log with the parameters to provide verification and traceability, ensuring the correct data was written and read.

  8. Buffer Deallocation:

  - Frees the allocated buffer to prevent memory leaks.

---

12. - **Function**

```
static int media_update_life_snapshot_params_to_flash(const ai_glass_snapshot_
↪param_t *params);
```

- **Description:**

  This function is aim to save lifetime snapshot parameters to flash memory.

- **Parameters:**

    - params – a constant pointer to the structure containing the lifetime snapshot parameters to be saved to flash memory.

- **Return:**
    - MEDIA_OK – indicating the operation was successful.
    - MEDIA_FAIL – indicating the operation was fail.
- **Key Call:**
    1. Buffer Allocation:
    - "unsigned char *lifetime_snap_buf = malloc(FLASH_LIFE_SNAP_BLOCK_SIZE);": Allocates a buffer of size FLASH_LIFE_SNAP_BLOCK_SIZE, typically 2KB, to temporarily store the data to be written to flash memory.
    2. Flash Address Determination:
    - "unsigned int flash_addr = 0;": Initializes the flash address variable.
    - The flash address is set, depending on the boot selection state retrieved from sys_get_boot_sel(). Currently, it only accommodates one scenario with a placeholder for future extension (like NAND flash).
    3. Null Check for Buffer:
    - Checks if record_buf is NULL (allocation failed), and returns MEDIA_FAIL.
    4. Buffer Preparation:
    - "memset(record_buf, 0x00, FLASH_REC_BLOCK_SIZE);": Clears the allocated buffer.
    - "record_buf[0] = 'L'; … record_buf[7] = 'P';": Sets a header/tag in the buffer for identification as record parameters ("LIFESNAP").
    5. Data Copying:
    - "memcpy(lifetime_snap_buf + 8, params, sizeof(ai_glass_snapshot_param_t));": Copies the ai snapshot parameters into the buffer, starting after the identification tag.
    6. Flash Write and Read Operations:
    - "ftl_common_write(flash_addr, lifetime_snap_buf, FLASH_LIFE_SNAP_BLOCK_SIZE);": Writes the prepared buffer to the determined flash address.
    - Clears the buffer again with 0xff to ensure it reflects only the data read from flash.
    - "ftl_common_read(flash_addr, lifetime_snap_buf, FLASH_LIFE_SNAP_BLOCK_SIZE);": Reads the data back from flash memory to verify the write.
    7. Data Verification and Logging:
    - Reads the lifetime snapshot parameters from the buffer (lifetime_snap_buf + 8 due to the identification tag).
    - Outputs a detailed log with the parameters to provide verification and traceability, ensuring the correct data was written and read.
    8. Buffer Deallocation:
    - Frees the allocated buffer to prevent memory leaks.

---

13. - **Function**

```
static int media_get_record_params_from_flash(ai_glass_record_param_t *params);
```

- **Description:**

  This function is aim to retrieve record parameters from flash memory.

- **Parameters:**

  - params – a pointer to the structure where the retrieved record parameters should be stored.

- **Return:**

  - MEDIA_OK – indicating the operation was successful.
  - MEDIA_FAIL – indicating the operation was fail.

- **Key Call:**

  1. Input Validation:

  - "if (params == NULL)": Checks if the input pointer params is NULL. If it is, return MEDIA_FAIL. This prevents dereferencing a NULL pointer, which would cause a segmentation fault.

  2. Buffer Allocation:

  - "unsigned char *record_buf = malloc(FLASH_REC_BLOCK_SIZE);": Allocates a buffer (typically 2KB) used to temporarily hold the data read from flash memory.

  3. Determine Flash Address:

  - "unsigned int flash_addr = 0;": Initializes a variable for the flash address.
  - Based on sys_get_boot_sel(), it sets the flash address for parameter retrieval, supporting future extensions (like NAND flash).

  4. Buffer Allocation Check:

  - Checks if record_buf is NULL (allocation failed), logs an error using AI_GLASS_ERR, and returns MEDIA_FAIL.

  5. Clear Buffer and Read from Flash:

  - "memset(record_buf, 0x00, FLASH_REC_BLOCK_SIZE);": Clears the buffer to ensure it starts empty.
  - "ftl_common_read(flash_addr, record_buf, FLASH_REC_BLOCK_SIZE);": Reads the data from the specified flash address into record_buf.

  6. Validate and Copy Data:

  - Checks if the first four bytes of record_buf match the expected header tag ("RECD"). This validates the integrity and correctness of the stored data.
  - If the header matches, copies the parameters from record_buf to params and sets ret to MEDIA_OK.
  - If the header does not match, logs a warning with AI_GLASS_WARN and sets ret to MEDIA_FAIL.

  7. Buffer Deallocation:

  - Frees the allocated buffer to prevent memory leaks.

---

14. - **Function**

```
static int media_get_ai_snapshot_params_from_flash(ai_glass_snapshot_param_t␣
↪*params);
```

- **Description:**

  This function is aim to retrieve ai snapshot parameters from flash memory.

- **Parameters:**
  - params – a pointer to the structure where the retrieved ai snapshot parameters should be stored.
- **Return:**
  - MEDIA_OK – indicating the operation was successful.
  - MEDIA_FAIL – indicating the operation was fail.
- **Key Call:**
  1. Input Validation:
  - "if (params == NULL)": Checks if the input pointer params is NULL. If it is, return MEDIA_FAIL. This prevents dereferencing a NULL pointer, which would cause a segmentation fault.
  2. Buffer Allocation:
  - "unsigned char *ai_snap_buf = malloc(FLASH_AI_SNAP_BLOCK_SIZE);": Allocates a buffer (typically 2KB) used to temporarily hold the data read from flash memory.
  3. Determine Flash Address:
  - "unsigned int flash_addr = 0;": Initializes a variable for the flash address.
  - Based on sys_get_boot_sel(), it sets the flash address for parameter retrieval, supporting future extensions (like NAND flash).
  4. Buffer Allocation Check:
  - Checks if ai_snap_buf is NULL (allocation failed), logs an error using AI_GLASS_ERR, and returns MEDIA_FAIL.
  5. Clear Buffer and Read from Flash:
  - "memset(ai_snap_buf, 0x00, FLASH_AI_SNAP_BLOCK_SIZE);": Clears the buffer to ensure it starts empty.
  - "ftl_common_read(flash_addr, ai_snap_buf, FLASH_AI_SNAP_BLOCK_SIZE);": Reads the data from the specified flash address into record_buf.
  6. Validate and Copy Data:
  - Checks if the first four bytes of record_buf match the expected header tag ("AISNAP"). This validates the integrity and correctness of the stored data.
  - If the header matches, copies the parameters from record_buf to params and sets ret to MEDIA_OK.
  - If the header does not match, logs a warning with AI_GLASS_WARN and sets ret to MEDIA_FAIL.
  7. Buffer Deallocation:
  - Frees the allocated buffer to prevent memory leaks.

---

15. - **Function**

```
static int media_get_life_snapshot_params_from_flash(ai_glass_snapshot_param_t␣
↪*params);
```

- **Description:**

  This function is aim to retrieve lifetime snapshot parameters from flash memory.
- **Parameters:**

   - params – a pointer to the structure where the retrieved lifetime snapshot parameters should be stored.

- **Return:**
   - MEDIA_OK – indicating the operation was successful.
   - MEDIA_FAIL – indicating the operation was fail.
- **Key Call:**
   1. Input Validation:
   - "if (params == NULL)": Checks if the input pointer params is NULL. If it is, return MEDIA_FAIL. This prevents dereferencing a NULL pointer, which would cause a segmentation fault.
   2. Buffer Allocation:
   - "unsigned char *lifetime_snap_buf = malloc(FLASH_LIFE_SNAP_BLOCK_SIZE);": Allocates a buffer (typically 2KB) used to temporarily hold the data read from flash memory.
   3. Determine Flash Address:
   - "unsigned int flash_addr = 0;": Initializes a variable for the flash address.
   - Based on sys_get_boot_sel(), it sets the flash address for parameter retrieval, supporting future extensions (like NAND flash).
   4. Buffer Allocation Check:
   - Checks if ai_snap_buf is NULL (allocation failed), logs an error using AI_GLASS_ERR, and returns MEDIA_FAIL.
   5. Clear Buffer and Read from Flash:
   - "memset(lifetime_snap_buf, 0x00, FLASH_LIFE_SNAP_BLOCK_SIZE);": Clears the buffer to ensure it starts empty.
   - "ftl_common_read(flash_addr, lifetime_snap_buf, FLASH_LIFE_SNAP_BLOCK_SIZE);": Reads the data from the specified flash address into record_buf.
   6. Validate and Copy Data:
   - Checks if the first four bytes of record_buf match the expected header tag ("LIFESNAP"). This validates the integrity and correctness of the stored data.
   - If the header matches, copies the parameters from record_buf to params and sets ret to MEDIA_OK.
   - If the header does not match, logs a warning with AI_GLASS_WARN and sets ret to MEDIA_FAIL.
   7. Buffer Deallocation:
   - Frees the allocated buffer to prevent memory leaks.

Lifetime Snapshot Internal Functions

1. - **Function**

```
static void lifetime_snapshot_file_save(char *file_path, uint32_t data_addr,␣
↪uint32_t data_size);
```

- **Description:**

  This function captures and saves a lifetime snapshot image to a file. It processes raw NV12 image data, optionally scales it up, encodes it into a JPEG format, and writes it to an external storage file.

- **Parameters:**

  – file_path – a pointer to the path of the file where the JPEG image will be saved.

  – data_addr – the memory address of the raw image data.

  – data_size – the size of the raw image data.

- **Return:**

- **Key Call:**

  1. **if (is_file_saved == LIFESNAP_TAKE)**

  – Ensures that the function only proceeds when a snapshot capture request has been made.

  2. **If (ls_video_params.need_scaleup)**

  – If scaling is needed, it allocates memory with new buffer **(uint32_t nv12_size = ls_video_params.jpg_width * ls_video_params.jpg_height / 2 * 3)** and resizes the image using **custom_resize_for_nv12**.

  – If scaling is not needed, it directly assigns **data_addr** to **input_nv12_buf**.

  3. **output_jpg_buf = malloc(nv12_size)**

  – The output JPEG buffer (output_jpg_buf) is always the same size, regardless of scaling requirement.

  4. **FILE *life_snapshot_file = extdisk_fopen(file_path, "wb")**

  – Opens the output file for writing the **JPEG data**.

  5. **custom_jpegEnc_from_nv12(input_nv12_buf, ls_video_params.jpg_width, ls_video_params.jpg_height, output_jpg_buf, ls_video_params.jpg_qlevel, nv12_size, &jpg_size)**

  – Converts the **NV12 image** to **JPEG format**.

  6. **fwrite(output_jpg_buf + i, 1, ((i + BSIZE) >= jpg_size) ? (jpg_size - i) : BSIZE, life_snapshot_file)**.

  – Writes the encoded JPEG data in chunks.

  7. **Close File and Free Buffers**

     – extdisk_fclose(life_snapshot_file)

     – free(input_nv12_buf)

     – input_nv12_buf = NULL

     – free(output_jpg_buf)

     – output_jpg_buf = NULL

     8. **Update Lifetime Snapshot Status**

  - **is_file_saved = LIFESNAP_FAIL**: If any step fails (e.g., memory allocation, file opening, or encoding), ret is set to **-1**.
  - **is_file_saved = LIFESNAP_DONE**: If everything is successful, ret remains **0**.

Lifetime Recording Internal Functions

1. - **Function**

```
static audio_sr audio_samplerate2index(int samplerate);
```

- **Description:**

  This function converts an audio sample rate **(Hz)** into a corresponding audio sample rate index **(ASR_XXKHZ)**. If an invalid sample rate is provided, it defaults to **ASR_16KHZ (16 kHz)** and prints a warning.

- **Parameters:**
  - samplerate – an audio sample rate in Hz

- **Return:**
  - the index format of audio interface for the corresponding sample rate

---

2. - **Function**

```
static int i2s_samplerate2index(int samplerate);
```

- **Description:**

  This function converts an I2S sample rate **(Hz)** to an I2S sample rate index **(SR_XXKHZ)**. If an invalid sample rate is provided, it defaults to **SR_16KHZ (16 kHz)** and prints a warning.

- **Parameters:**
  - samplerate – an audio sample rate in Hz

- **Return:**
  - the index format of i2s interface for the corresponding sample rate

---

3. - **Function**

```
static int lr_mp4_end_cb(void *parm);
```

- **Description:**

  This function is a callback that is executed when MP4 recording ends. It manages gyro sensor data and updates the recording state.

- **Parameters:**

  NA

- **Return:**

  NA

- **Key Call:**
  1. **If GSENSOR_RECORD_FAST is enabled:**
  - Waits for GYRO_SAVE_STOP state before proceeding.
  - Deletes the entire gyro data list (delete_whole_list(&gyro_list)).
  2. **If GSENSOR_RECORD_FAST is disable:**
  - Saves gyro data to a CSV file (**save_gyro_data_to_file**()).
  - Sets current_state = **STATE_END_RECORDING**.

---

4. - **Function**

```
static int lr_mp4_stop_cb(void *parm);
```

- **Description:**

  This function is a callback that is executed when MP4 recording stops. It updates the recording state, manages gyro sensor data, and retrieves the video timestamp.

- **Parameters:**

  NA

- **Return:**

  NA

- **Key Call:**
  1. **If ENABLE_GET_GSENSOR_INFO is enabled:**
  - Sets recording state to **STATE_IDLE**.
  - Deinitializes gyro sensor.
  - Prints stored gyro data.
  - Retrieves and prints the MP4 video timestamp.
  2. **If ENABLE_GET_GSENSOR_INFO is disable:**
  - Sets recording state to **STATE_IDLE**.

---

5. - **Function**

```
static int lr_mp4_error_cb(void *parm);
```

- **Description:**

  This function is a callback that gets triggered when an error occurs during MP4 recording. It updates the system state to an error state and deinitializes the gyro sensor if enabled.

- **Parameters:**

  NA

- **Return:**

  NA

- **Key Call:**

1. **If ENABLE_GET_GSENSOR_INFO is enabled:**

– Sets recording state to **STATE_ERROR**.

– Deinitializes gyro sensor.

2. **If ENABLE_GET_GSENSOR_INFO is disable:**

– Sets recording state to **STATE_ERROR**.

AI Sanpshot Internal Functions

1. - **Function**

```
static int video_snapshot_cb(uint32_t jpeg_addr, uint32_t jpeg_len);
```

- **Description:**

  This function is registered as a callback and is triggered when an AI snapshot is captured by the video module. It allocates memory to store the JPEG image, copies the captured data, and signals the completion of the AI snapshot using a semaphore.

- **Parameters:**

  – jpeg_addr – The memory address where the JPEG image data starts.

  – jpeg_len – The length (size) of the JPEG image data in bytes.

- **Return:**

  NA

- **Key Call:**

  1. **ai_snap_ctx->take_snapshot = 1**

  – Set the snapshot flag to **1**.

  2. **ai_snap_ctx->dest_addr = (uint32_t) malloc(jpeg_len)**

  – Memory is allocated for storing the JPEG image.

  3. **memcpy((void *)ai_snap_ctx->dest_addr, (const void *)jpeg_addr, jpeg_len)**

  – Copies the JPEG data from jpeg_addr to **ai_snap_ctx->dest_addr**.

  4. **ai_snap_ctx->dest_actual_len = jpeg_len**

  – Stores the actual size of the AI snapshot.

  5. **rtw_up_sema(&ai_snap_ctx->snapshot_sema)**

  – Signals the semaphore to notify that the AI snapshot is ready.

  6. **The function returns 0**.

---

2. - **Function**

```
static int video_snapshot_get_buffer(jpeg_buffer_t *video_buf, uint32_t␣
↪timeout_ms);
```

- **Description:**

  This function retrieves the captured AI snapshot buffer. It waits for the semaphore snapshot_sema to be signaled within the specified timeout. If the AI snapshot is ready, it assigns the stored JPEG data and size to video_buf. Otherwise, it sets the buffer to NULL and returns an error.

- **Parameters:**
  - video_buf – a pointer to a structure where the function will store the snapshot buffer and its size.
  - timeout_ms – the maximum number of milliseconds to wait for the snapshot to be available.

- **Return:**
  - 0 – upon successful buffer acquisition, indicating that the snapshot buffer information is now stored in video_buf.
  - -1 – when the operation fails, either due to a timeout or some other issue in acquiring the semaphore.

- **Key Call:**

  1. **rtw_down_timeout_sema(&ai_snap_ctx->snapshot_sema, timeout_ms)**

  - Waits for the semaphore snapshot_sema to be signaled within the given timeout.
  - If the AI snapshot is ready, execution continues. Otherwise, it times out.

  2. **video_buf->output_buffer = (uint8_t *) ai_snap_ctx->dest_addr**

  - Assigns the AI snapshot memory address to **video_buf->output_buffer**.

  3. **video_buf->output_size = ai_snap_ctx->dest_actual_len**

  - Stores the actual AI snapshot size in **video_buf->output_size**.

  4. **If the semaphore times out**

  - The **video_buf->output_buffer** is set to **NULL**.
  - The **video_buf->output_size** is set to **0**.

  5. **The function returns**

  - **0** if the AI snapshot is successfully retrieved.
  - **-1** if the operation fails due to a timeout.

---

3. - **Function**

```
static int video_capture_snapshot(const char *filename);
```

- **Description:**

  This function captures an AI snapshot and saves it to a file. It first retrieves the snapshot buffer using video_snapshot_get_buffer().

  If successful, it writes the snapshot data to the specified file using snapshot_write(). Once completed, it resets the AI snapshot flag and returns the result.

- **Parameters:**
  - filename – The name of the file where the snapshot data will be saved.

- **Return:**
  - 0 – success

  - -1 – failure.

- **Key Call:**

  1. **!video_snapshot_get_buffer(&ai_snap_ctx->video_buf, SNAPSHOT_TIMEOUT)**

  - Attempts to retrieve the AI snapshot buffer within the defined timeout.
  - If successful, the AI snapshot data and size are stored in **ai_snap_ctx->video_buf**.

  2. **ai_snap_ctx->snapshot_write(ai_snap_ctx->video_buf.output_buffer, ai_snap_ctx->video_buf.output_size, filename)**

  - Writes the captured AI snapshot data to the specified file.
  - If successful, **ret** is set to **0** (**success**).

  3. **ai_snap_ctx->take_snapshot = 0**

  - Resets the AI snapshot flag after saving the image.

  4. **If video_snapshot_get_buffer() fails**

  - The function returns **-1** indicating failure.

  5. **The function returns**

  - **0** if the AI snapshot is successfully captured and saved.
  - **-1** if the AI snapshot retrieval fails.

---

4. - **Function**

```
static int aisnapshot_write_picture(uint8_t *buf, uint32_t len, const char␣
↪*filename);
```

- **Description:**

  Firsty, this function pointer snapshot_write is assigned to aisnapshot_write_picture (**ai_snap_ctx->snapshot_write = aisnapshot_write_picture;**), allowing indirect calls to this function for writing AI snapshots.

  This function writes a captured AI snapshot (JEG image) to a file in RAM disk storage.

- **Parameters:**

  - buf – a pointer to the buffer containing the JPEG data.
  - len – the length of the JPEG data in bytes.
  - filename – The name of the file where the JPEG data should be written.

- **Return:**

  - 0 – success
  - -1 – failure.

- **Key Call:**

  1. **if (ai_snap_ctx->dest_addr)**

  - Ensures that a valid buffer exists before opening the file.

  2. **m_file = ramdisk_fopen(filename, "w")**

  - Opens a file in RAM disk storage with the given filename in write mode.

3. **if (m_file)**

– Checks if the file was successfully opened.

4. **fwrite(buf, 1, len, m_file)**

– Writes the **JPEG data buf** to the file.

5. **fclose(m_file)**

– Closes the file to ensure data is properly saved.

6. **free((void *)ai_snap_ctx->dest_addr)**

– Frees the allocated memory for the AI snapshot buffer after writing.

7. **ai_snap_ctx->dest_addr = 0**

– Resets the buffer address to **0**.

8. **The function returns**

   – **0** if the file image is opened and written successfully.

   – **-1** if the file opening fails.

3.1.4 WiFi Scenario

This section describes the WiFi scenario supported by AI Wearable Glass. In the WiFi usage scenario, the main process involves enabling AP mode to allow the app to connect, using HTTP (or HTTPS) protocol as the primary means of transmission.

WiFi Scenario Public API reference

WiFi Scenario Common API

1. - **Function:**

```
int wifi_enable_ap_mode(const char *ssid, const char *password, int channel,␣
↪int timeout);
```

- **Description:**

  This function is used to start the AP mode and the also the http server. User can customize their SSID, password, channel and timeout.

- **Parameters:**

  – ssid – a const string represents the Service Set Identifier (SSID) that other devices will see when they scan for available WiFi networks. Should not be a null pointer.

  – password – a const string represents password required to connect to the WiFi network. Should not be a null pointer.

  – channel – specifies the WiFi channel to be used.

  – timeout – this denote the time duration in milliseconds for which the AP mode.

- **Return:**

  – WLAN_SET_OK – the ap mode and the http server is opened correctly.

  - WLAN_SET_FAIL – the ap mode and the http server is not opened correctly.

- **Key Call:**

  1. This function can be called via AT command using **CMDSETWIFIMODE=1**.

---

2. - **Function:**

```
int wifi_disable_ap_mode(void);
```

- **Description:**

  This function is used to stop the AP mode.

- **Parameters:**

  NA

- **Return:**

  - WLAN_SET_OK – close http server and wifi correctly.
  - WLAN_SET_FAIL – fail to close http server and wifi.

- **Key Call:**

  1. This function can be called via AT command using **CMDSETWIFIMODE=0**.

---

3. - **Function:**

```
int wifi_get_ap_setting(rtw_softap_info_t *wifi_cfg);
```

- **Description:**

  This function is to retrieve the AP wifi configuration.

- **Parameters:**

  - wifi_cfg – a pointer to a allocated structure rtw_softap_info_t, which is use to store current ap information include ssid, password, secure type and channel.

- **Return:**

  - WLAN_SET_OK – get the ap mode inofrmation successfully.
  - WLAN_SET_FAIL – fail to get the ap mode inofrmation successfully.

- **Key Call:**

  1. This function is usually called after the AP mode has been triggered to find out what is the existing wifi configuration such as SSID, password.

---

4. - **Function:**

```
int wifi_get_connect_status(void);
```

- **Description:**

  This function is get the current wifi status. There are three possible status: WLAN_STAT_IDLE, WLAN_STAT_HTTP_IDLE and WLAN_STAT_HTTP_CONNECTED. WLAN_STAT_IDLE means that the HTTPD server is not running and AP mode is OFF. WLAN_STAT_HTTP_CONNECTED means that the HTTPD server is running, AP mode ON and there are at least 1 connected clients. WLAN_STAT_HTTP_IDLE means that the HTTPD server is running, AP mode is ON and there is no connected clients.

- **Parameters:**

  NA

- **Return:**

  - WLAN_STAT_HTTP_IDLE – the ap mode is on and the http server is on, but no client has connected to the http server.
  - WLAN_STAT_HTTP_CONNECTED – the ap mode is on and the http server is on, and at least one client has connected to the http server.
  - WLAN_STAT_IDLE – both ap mode and http server is off.

- **Key Call:**

  1. This function is called to retrieve what is the current connection status.

---

5. - **Function:**

```
void wifi_set_up_file_delete_flag(uint8_t flag);
```

- **Description:**

  This function is use to set to the flag which represents the file will be deleted or remained after download process.

- **Parameters:**

  - flag – 0 means the file will be remained while others means the file will be deleted after download process.

- **Return:**

- **Key Call:**

  1. This function is called to retrieve what is the current connection status.

WiFi Scenario Internal Functions reference

WiFi Scenario Internal Function

1. - **Function:**

```
static void pingpong_cb(struct httpd_conn *conn);
```

- **Description:**

  This function handles HTTP requests of the http page /pingpong using a connection object (denoted as conn), checking the HTTP request method and responding with appropriate HTTP headers.

- **Parameters:**

  - conn – a pointer to a structure representing an HTTP connection.

- **Return:**

  NA

- **Key Call:**

  1. "httpd_request_get_header_field"

  - retrieve the "User-Agent" header from the HTTP request.
  - If successful, it logs the User-Agent value and then frees the allocated memory for the header value and use httpd_request_is_method to check the HTTP method.

  2. "httpd_request_is_method"

  - If the HTTP method is "GET", the server responds with a 200 OK status and content type text/plain, which will also set the Access-Control-Allow-Origin header to *, allowing any origin to access the resource for typical behavior for CORS policies.
  - If the HTTP method is "OPTIONS", it typically suggests a pre-flight CORS request to check supported methods and headers. The server responds with a 204 No Content status.
  - If the HTTP is not HTTP method other than "GET" and "OPTIONS", the server returns a 405 Method Not Allowed status, indicating the request method is recognized but not supported by the server.

  3. "httpd_conn_close":

  - Regardless of the request method, the server closes the connection using httpd_conn_close, finishing the client's request-response transaction.

---

2. - **Function:**

```
static void media_list_cb(struct httpd_conn *conn);
```

- **Description:**

  This function handles HTTP requests of the http page /media-list using a connection object (denoted as conn), checking the HTTP request method and responding with appropriate HTTP headers.

- **Parameters:**

  - conn – a pointer to a structure representing an HTTP connection.

- **Return:**

  NA

- **Key Call:**

  1. "httpd_request_get_header_field"

  - retrieve the "User-Agent" header from the HTTP request.
  - If successful, it logs the User-Agent value and then frees the allocated memory for the header value and use httpd_request_is_method to check the HTTP method.

  2. "httpd_request_is_method"

  - If the HTTP method is "GET", the server will call function "extdisk_get_filelist" to get the file list and then use "cJSON_Print" to covert the json object to string.
  - After getting the file list, it will respond with a 200 OK status and content type text/plain, which will also set the Access-Control-Allow-Origin header to *, allowing any origin to access the resource for typical behavior for CORS policies. The body will be the file list in the file system.

- If the HTTP method is "OPTIONS", it typically suggests a pre-flight CORS request to check supported methods and headers. The server responds with a 204 No Content status.
- If the HTTP is not HTTP method other than "GET" and "OPTIONS", the server returns a 405 Method Not Allowed status, indicating the request method is recognized but not supported by the server.

3. "httpd_conn_close":

- Regardless of the request method, the server closes the connection using httpd_conn_close, finishing the client's request-response transaction.

---

3. - **Function:**

```
static int httpd_request_get_path_key(struct httpd_conn *conn, const char *key,
↪ char **value);
```

- **Description:**

  This function aims to parse the HTTP request path to locate a specific key and extract its associated value.

- **Parameters:**
  - conn – a pointer to a structure representing an HTTP connection.
  - key – a constant character string representing the key to search for within the request path.
  - value – a pointer to a character pointer, which will be used to return the extracted value associated with the specified key. The function allocates memory dynamically for this value, which should be freed by the caller.

- **Return:**
  - 0 – if the path have some string matched the key string.
  - -1 – the path do not have words match the key string.

---

4. - **Function:**

```
static void media_getfile_queue_cb(struct httpd_conn *conn);
```

- **Description:**

  This function handles HTTP requests of the http page /media/* (like /media/123.mp4, /media/456.jpg) using a connection object (denoted as conn), checking the HTTP request method and responding with appropriate HTTP headers.

- **Parameters:**
  - conn – a pointer to a structure representing an HTTP connection.

- **Return:**

- **Key Call:**

  1. "httpd_request_get_header_field"

  - retrieve the "User-Agent" header from the HTTP request.
  - If successful, it logs the User-Agent value and then frees the allocated memory for the header value and use httpd_request_is_method to check the HTTP method.

  2. "httpd_request_is_method"

- If the HTTP method is "GET", the server will open the file in the file system and send via http.
- After getting the file list, it will respond with a 200 OK status and content type text/plain, which will also set the Access-Control-Allow-Origin header to *, allowing any origin to access the resource for typical behavior for CORS policies. The body will be the file list in the file system.
- If the HTTP method is "OPTIONS", it typically suggests a pre-flight CORS request to check supported methods and headers. The server responds with a 204 No Content status.
- If the HTTP is not HTTP method other than "GET" and "OPTIONS", the server returns a 405 Method Not Allowed status, indicating the request method is recognized but not supported by the server.

3. "httpd_conn_close":

- Regardless of the request method, the server closes the connection using httpd_conn_close, finishing the client's request-response transaction.

---

5. - **Function:**

```
static void transfer_file_normal_internal(struct httpd_conn *conn, char␣
↪*filename);
```

- **Description:**

  This function reads a file from an external disk and sends its contents over an HTTP connection.

- **Parameters:**
  - conn – a pointer to a structure representing the HTTP connection through which data will be sent to the client.
  - filename – a string specifying the name of the file to be transferred.

- **Return:**

  NA

- **Key Call:**

  1. "extdisk_fopen(filename, "r")"

  - open the file in read mode. If the file cannot be opened, it returns immediately, preventing further processing.

  2. Data Reading and Sending Loop:

  2.1 A loop repeatedly reads chunks from the file and sends them over the HTTP connection:

  - extdisk_fread reads data into the buffer data_buf with a size defined by HTTP_DATA_BUF_SIZE.
  - if the read (br) returns less than zero, it logs an error and breaks the loop.
  - a send timeout (send_timeout) of 3000 milliseconds is set to ensure that the socket doesn't block indefinitely.
  - the read data is then sent over the network using httpd_response_write_data.
  - if the send operation returns zero or negative, an error is logged, and the loop exits.
  - if the number of bytes read (br) is less than HTTP_DATA_BUF_SIZE, it indicates the end of the file, so the loop breaks.

  3. File closure

  - After the loop completes, it ensures the file is closed.

---

6. - **Function:**

```
static void http_file_send_thread(void *pvParameters);
```

- **Description:**

  This function acts as a "reader task" that reads data from a specified file and sends it through a message queue. It communicates status (such as errors or end-of-file conditions) to a "writer task" using task notifications.

- **Parameters:**

  - pvParameters – point to a character string containing the filename to be processed.

- **Return:**

  NA

- **Key Call:**

  1. File Opening:

  - The file specified by filename is opened using extdisk_fopen in read mode.
  - If opening fails, error messages are logged, and a task notification is sent to signal an error (TASK_NOTIFY_ERROR).

  2. Data Reading and Queueing Loop:

  - The task sets the file position to the start and enters a loop to read data into message.message (presumably a buffer within the Message_t structure), up to QUEUE_ITEM_SIZE.

  3. Error handling and EOF detection:

  - If reading returns a negative value, an error is recorded, and the loop exits.
  - If zero bytes are read, indicating EOF, EOF status is recorded, and the loop exits. The number of bytes read (br) is used as message.id. Each read chunk is sent to the queue with xQueueSend. If queueing fails, an error is logged, and the loop breaks.

  4. Notification and Message Handling:

     4.1 When reading terminates, either by error or EOF:

     - Errors result in logging and a task notification to signal the error.
     - EOF results in logging, followed by sending a special zero-length message to indicate the end of the data stream to the queue.
     - Notifications are sent using xTaskNotify, differentiating between EOF and errors.

  5. File and Task Cleanup:

  - The file is closed with extdisk_fclose if it is open.
  - Delete itself using vTaskDelete(NULL).

---

7. - **Function:**

```
static void http_file_read_thread(void *pvParameters);
```

- **Description:**

This function receives data from a queue, which acts as an intermediary between this "receiver task" and a file reading task. It sends the received data over an HTTP connection and manages communication with a main task through task notifications.

- **Parameters:**
  - pvParameters – point to struct httpd_conn, which represents the HTTP connection to the client.
- **Return:**

  NA
- **Key Call:**
  1. Initialization:
  - The conn pointer is initialized from the task parameters, representing the HTTP client connection.
  - Variables writer_status and total_bw are initialized to track the writer task status and total bytes written, respectively.
  2. Queue Receiving Loop:
  - Continuously waits and receives messages from the file_queue using xQueueReceive, blocking indefinitely until a message is available.
  3. Upon receiving a message:
  - If receivedMessage.id is greater than zero, it increments total_bw to count the total bytes processed.
  - A zero receivedMessage.id indicates EOF, setting the writer status and breaking the loop, logging the total bytes processed.
  4. Sending Data over HTTP:
  - Prior to sending, sets a send timeout on the socket to manage network delays.
  - Uses httpd_response_write_data to send the message data over the HTTP connection.
  - If sending fails (returns zero or negative), it sets the writer status to indicate an error and exits the loop.
  5. Completion and Notification:
  - Once terminated, either by EOF or error, it sends a task notification to the main task (core_taskhandle):
  - Sends TASK_NOTIFY_END if EOF is encountered.
  - Sends TASK_NOTIFY_ERROR if an error was reported by the reader.
  - Sends TASK_NOTIFY_WERROR if writing data over HTTP failed.
  6. Task Cleanup:
  - Delete itself using vTaskDelete(NULL).

---

8. - **Function:**

```
static void media_getfile_cb(struct httpd_conn *conn);
```

- **Description:**

  This function handles HTTP requests of the http page /media/* (like /media/123.mp4, /media/456.jpg) using a connection object (denoted as conn), checking the HTTP request method and responding with appropriate HTTP headers.
- **Parameters:**

  - conn – a pointer to a structure representing an HTTP connection.

- **Return:**
- **Key Call:**

  1. "httpd_request_get_header_field"

  - retrieve the "User-Agent" header from the HTTP request.
  - If successful, it logs the User-Agent value and then frees the allocated memory for the header value and use httpd_request_is_method to check the HTTP method.

  2. "httpd_request_is_method"

  - If the HTTP method is "GET", the server will open the file in the file system and send via http.
  - After getting the file list, it will respond with a 200 OK status and content type text/plain, which will also set the Access-Control-Allow-Origin header to *, allowing any origin to access the resource for typical behavior for CORS policies. The body will be the file list in the file system.
  - If the HTTP method is "OPTIONS", it typically suggests a pre-flight CORS request to check supported methods and headers. The server responds with a 204 No Content status.
  - If the HTTP is not HTTP method other than "GET" and "OPTIONS", the server returns a 405 Method Not Allowed status, indicating the request method is recognized but not supported by the server.

  3. "httpd_conn_close":

  - Regardless of the request method, the server closes the connection using httpd_conn_close, finishing the client's request-response transaction.

---

9. - **Function:**

```
static void deinit_dhcp(void);
```

- **Description:**

  This function deinitializes the DHCP server when the system is configured to operate in WiFi AP mode.

- **Parameters:**
- **Return:**
- **Key Call:**

  1. Preprocessor Conditional Check:

  - "Call to dhcps_deinit()": This function call stops the DHCP server process. In AP mode, the device typically assigns IP addresses to clients connecting to the network. dhcps_deinit disables this functionality, perhaps to switch to a static IP configuration or to shift roles.

  2. Set Static IP Configuration:

  - IP Address: Constructed from predefined macros AI_GLASS_AP_IP_ADDR{0-3} using WIFI_MAKEU32. This allows a static IP configuration.
  - Netmask and Gateway: Similarly constructed from respective macros AI_GLASS_AP_NETMASK_ADDR{0-3} and AI_GLASS_AP_GW_ADDR{0-3}.

  3. LwIP_SetIP: This function is used to apply the static IP, netmask, and gateway address to the network interface, effectively assigning a fixed network configuration to the device.

HTTP download file from file system

The feature allows the AI Wearable glass to transfer the media data in the file system (EMMC / SD / RAM) to other devices via HTTP. The AI Wearable glass will enable the AP mode and set up the HTTP file server.

The user can use any browser to request the resource required in the file system of the AI Wearable glass. For instance, there is a MP4 file named "test.mp4" in the EMMC. The user can enter the URL **http://192.168.43:1:$(AI_GLASS_AP_PORT)/media/test.mp4** in the browser to download the file as long as the PC is also connected to the AP set up by the AI Wearable glass. The port can be set in the file wlan_scenario.h. The default port is set as 8080. You may modify the value of AI_GLASS_AP_PORT to the port value that you desire.

HTTP Function

1. - **Function:**

   ```
   void httpd_setup_priority(uint8_t priority);
   ```

   - **Description:**

     This function is used to setup httpd server priority.

   - **Parameters:**

     – priority – the default priority is defined to 1. In this scenario, we set as 5.

   - **Return:**

     NA

---

2. - **Function:**

   ```
   void httpd_setup_idle_timeout(int idle_timeout);
   ```

   - **Description:**

     This function is used to setup connection idle timeout for server.

   - **Parameters:**

     – idle_timeout: timeout in seconds. In this scenario, we use HTTPD_CONNECT_TIMEOUT for user to confiure the setting.

   - **Return:**

     NA

---

3. - **Function:**

   ```
   void httpd_setup_idle_timeout(int idle_timeout);
   ```

   - **Description:**

     This function is used to setup connection idle timeout for server.

   - **Parameters:**

     – idle_timeout: timeout in seconds. In this scenario, we use HTTPD_CONNECT_TIMEOUT for user to confiure the setting.

- **Return:**

  NA

---

1. - **Function:**

```
int httpd_reg_page_callback(char *path, void (*callback)(struct httpd_conn␣
↪*conn));
```

- **Description:**

  This function allows users to set a path to call on the browser and define a callback function to perform when a particular HTTP action (POST,GET,OPTIONS) is performed.

- **Parameters:**

  NA

- **Return:**

  – WLAN_STAT_HTTP_IDLE – the ap mode is on and the http server is on, but no client has connected to the http server.

  – WLAN_STAT_HTTP_CONNECTED – the ap mode is on and the http server is on, and at least one client has connected to the http server.

  – WLAN_STAT_IDLE – both ap mode and http server is off.

- **Key Call:**

  1. `httpd_reg_page_callback((char *)"/media/*", media1_getfile_cb);`

  – Downloads the file from storage system to PC.

  2. `httpd_reg_page_callback((char *)"/media-list", media_list_cb);`

  – Lists the file in the storage system.

3.1.5 File System

This section describes the storage capabilities of the AI Wearable Glass:

- **Storage**: The device supports saving media directly to the onboard EMMC storage, which provides reliable and fast data access.
- **File Management**: Files are stored with unique names (e.g., lifesnap_xxxxx.jpg and liferecord_xxxxx.mp4) to ensure that each media capture is saved without overwriting existing files.
- **Access and Retrieval**: The image data stored on the RAM can be accessed via Bluetooth UART for easy retrieval and integration with other smart devices or applications.

File System Public API reference

File System Commmon API

1. - **Function:**

```
int media_filesystem_init(void);
```

- **Description:**

  Initializes the filesystem and ensures mutual exclusion when accessing external and RAM disks.

- **Parameters:**

  NA

- **Return:**

  - 0 – initializing the media system successfully.
  - -1 – initializing the media system fail.

- **Key Call:**

  1. **vfs_init(NULL)**

  - Initializes the filesystem.

  2. **extdisk_mutex = xSemaphoreCreateMutex()**

  - Create a mutex for safe access to external disk.
  - If fail clean up filesystem by deinitializing the **VFS (vfs_deinit(NULL))**.

  3. **ramdisk_mutex = xSemaphoreCreateMutex()**

  - Create mutex for safe access to RAM disk.
  - If fail cleans up external disk and filesystem by deinitializing the **VFS (vfs_deinit(NULL))**.

---

2. - **Function:**

```
const char *media_filesystem_get_current_time_string(void);
```

- **Description:**

  This function retrieves the current media timestamp and converts it into a formatted time string. It combines GPS-based time **(gps_timeinfo)** with the media time offset **(mm_read_mediatime_ms())**, then formats it into a human-readable string.

- **Parameters:**

  NA

- **Return:**

  - a pointer to a string representing the current time. Note that the caller is responsible for freeing this memory to avoid memory leaks. If memory allocation for the time string fails, it returns NULL.

- **Key Call:**

  1. **time_t cur_time = gps_timeinfo + mm_read_mediatime_ms() / 1000**

  - The function adds the GPS time and media time **(converted to seconds)** to get the current timestamp.

2. **malloc(CUR_TIME_BUFFER_SIZE)**

– Allocates memory for storing the formatted time string.

3. **time_transfer_to_string(cur_time, time_buffer, CUR_TIME_BUFFER_SIZE)**

– Converts the calculated timestamp (**cur_time**) into a formatted string and stores it in **time_buffer**.

4. **Return the formatted time string pointer**

---

3. - **Function:**

```
time_t media_filesystem_gpstime_to_time_t(uint32_t gps_week, uint32_t gps_
↪seconds);
```

- **Description:**

  Converts GPS time to a **Unix timestamp (time_t)**, based on the **GPS epoch (1980-01-06 00:00:00 UTC)**.

- **Parameters:**
  - gps_week – Specifies the number of weeks since the GPS system started (January 6, 1980). This parameter is used to calculate the total number of full weeks from the GPS start to the target time.
  - gps_seconds – Specifies the number of seconds within the current week. This includes the seconds elapsed from the beginning of the week to the desired time.

- **Return:**
  - a value of type time_t that represents the number of seconds relative to the UNIX epoch time (January 1, 1970). This value corresponds to the UTC time for the given GPS time.

- **Key Call:**

  1. **mktime(&gps_start_time)**

  – Convert GPS start time to **time_t format**.

  2. Return utcform time_t.

---

4. - **Function:**

```
void media_filesystem_setup_gpstime(uint32_t gps_week, uint32_t gps_seconds);
```

- **Description:**

  Sets the system's time based on the provided GPS week and second values. It converts the GPS time into a Unix timestamp (time_t) and displays the UTC time.

- **Parameters:**
  - gps_week – Specifies the number of weeks since the GPS system started (January 6, 1980). This parameter is used to calculate the total number of full weeks from the GPS start to the target time.
  - gps_seconds – Specifies the number of seconds within the current week. This includes the seconds elapsed from the beginning of the week to the desired time.

- **Return:**

  NA

- **Key Call:**

1. **media_filesystem_gpstime_to_time_t(gps_week, gps_seconds)**
2. **show_utc_format_time(gps_timeinfo)**

---

5. - **Function:**

```
int media_filesystem_setup_gpscoordinate(float latitude, float longitude,␣
↪float altitude);
```

- **Description:**

  Sets the system's location based on the provided GPS coordinate, which will be can be use to store in the snapshot or recording file.

- **Parameters:**
  - latitude – Specifies the latitude the GPS system.
  - longitude – Specifies the longitude the GPS system.
  - altitude – Specifies the altitude the GPS system.
- **Return:**
  - 0: set up the GPS coordinate successfully.

---

6. - **Function:**

```
void media_filesystem_get_gpscoordinate(float *latitude, float *longitude,␣
↪float *altitude);
```

- **Description:**

  Get the system's GPS coordinate.

- **Parameters:**
  - latitude – a pointer to a float, which means the latitude in setting the current system.
  - longitude – a pointer to a float, which means the longitude in setting the current system.
  - altitude – a pointer to a float, which means the altitude in setting the current system.
- **Return:**

Ram Disk API

The RAM disk is used to temporarily store information that is only needed during the current boot cycle. It is currently primarily used to temporarily store JPEG images for AI recognition.

1. - **Function:**

```
int ramdisk_filesystem_init(const char *disk_tag);
```

- **Description:**

  Initializes a RAM disk filesystem for AI Glass storage. It ensures exclusive access using a mutex, registers the RAM disk with the virtual filesystem (VFS), and marks the initialization as complete.

- **Parameters:**
  - disk_tag – a string representing the tag or identifier for the RAM disk. This is used to form the name of the RAM disk.
- **Return:**
  - 0 – the RAM disk has been initialized correctly.
  - -1 – this can occur if the mutex cannot be taken in a timely manner or if the RAM disk has already been initialized.
- **Key Call:**
  1. **xSemaphoreTake(ramdisk_mutex, portMAX_DELAY) != pdTRUE**
  - Acquires the mutex to ensure exclusive access.
  2. **vfs_user_register(disk_tag, VFS_FATFS, VFS_INF_RAM)**
  - Registers the RAM disk with the FATFS filesystem in VFS.
  3. **ai_glass_ramdisk_done = 1**
  - Set to **1** after registered the RAM disk.
  4. **xSemaphoreGive(ramdisk_mutex)**
  - Releases the mutex after initialization.

---

2. - **Function:**

```
int ramdisk_get_init_status(void);
```

- **Description:**

  Return a value 0 or 1.
- **Parameters:**

  NA
- **Return:**
  - 1 – the RAM disk has been set up.
  - 0 – the RAM disk has not been set up yet.
- **Key Call:**
  1. **return ai_glass_ramdisk_done**
  - ai_glass_ramdisk_done = **1** if already initialized.
  - ai_glass_ramdisk_done = **0** if not initialize.

---

3. - **Function:**

```
FILE *ramdisk_fopen(const char *filename, const char *mode);
```

- **Description:**

  This function function opens a file on an ram disk using the specified mode.
- **Parameters:**

  - filename – a string representing the name of the file you want to open on the RAM disk.
  - mode – a string indicating the file opening mode (e.g., "r" for reading, "w" for writing, etc.).
- **Return:**
  - a pointer to a FILE object representing the opened file, or NULL if the file cannot be opened. If the RAM disk is not initialized, the function returns NULL.
- **Key Call:**
  1. **It first checks whether the ram disk is initialized**
  2. **It constructs the full file path by appending the filename to the ram disk tag (ai_glass_ramdisk_tag)**
  3. **Attempts to open the file using fopen**

---

4. - **Function:**

```
int ramdisk_fclose(FILE *stream);
```

- **Description:**

  This function closes a file that was opened on an ram disk.
- **Parameters:**
  - stream – a pointer to a FILE object representing the open file stream that you wish to close.
- **Return:**
  - 0 on successful closure of the file stream, and EOF (a non-zero value, typically -1) on failure.
- **Key Call:**
  1. **It first checks whether the ram disk is initialized**
  2. **Attempts to close the file (stream) using fclose**

---

5. - **Function:**

```
size_t ramdisk_fread(void *ptr, size_t size, size_t count, FILE *stream);
```

- **Description:**

  This function function reads data from a file on an ram disk into a buffer.
- **Parameters:**
  - ptr – a pointer to a block of memory where the read data should be stored.
  - size – the size, in bytes, of each element to be read.
  - count – the number of elements, each of the specified size, to be read from the file.
  - stream – a pointer to a FILE object that identifies the input file stream from which data is to be read.
- **Return:**
  - the total number of elements successfully read, which may be less than count if a reading error occurs or the end of the file is reached. If the RAM disk is not initialized, the function returns -1.
- **Key Call:**
  1. **It first checks whether the ram disk is initialized**

2. **Attempts to reads data from the file (stream) into the provided buffer (ptr) using fread**

---

6. - **Function:**

```
size_t ramdisk_fwrite(const void *ptr, size_t size, size_t count, FILE␣
↪*stream);
```

- **Description:**

  This function writes data from the buffer ptr to the file.

- **Parameters:**
  - ptr – a pointer to the block of memory containing the data to write to the file.
  - size – the size, in bytes, of each element to be written.
  - count – the number of elements, each of the specified size, to be written to the file.
  - stream – a pointer to a FILE object representing the output file stream where data will be written.

- **Return:**
  - the total number of elements successfully read, which may be less than count if an error occurs during writing. If the RAM disk is not initialized, the function returns -1.

- **Key Call:**
  1. **It first checks whether the ram disk is initialized**
  2. **Attempts to writes data from buffer (ptr) into the file (stream) using fwrite**

---

7. - **Function:**

```
int ramdisk_fseek(FILE *stream, long int offset, int origin);
```

- **Description:**

  This function moves the file pointer in stream to a new position based on offset and origin.

- **Parameters:**
  - stream – a pointer to a FILE object representing the file stream whose position is to be set.
  - offset – the number of bytes to offset from origin.
  - origin – the position from which offset is added.

- **Return:**
  - 0 – on successful repositioning of the file pointer.
  - -1 – the function failed or the RAM disk is not initialized.

- **Key Call:**
  1. **It first checks whether the ram disk is initialized**
  2. **offset specifies how far to move the pointer**
  3. **origin specifies the reference point (e.g., SEEK_SET, SEEK_CUR, SEEK_END)**

---

8. - **Function:**

```
int ramdisk_ftell(FILE *stream);
```

- **Description:**

  This function gets the current position of the file pointer in stream.

- **Parameters:**
  - stream – a pointer to a FILE object representing the file stream for which the current file pointer position is to be determined.

- **Return:**
  - the current file position indicator as a long integer. If the function fails to retrieve the position or the RAM disk is not initialized, it returns -1.

- **Key Call:**
  1. **It first checks whether the ram disk is initialized**
  2. **Returns the current position in bytes**

---

9. - **Function:**

```
int ramdisk_feof(FILE *stream);
```

- **Description:**

  This function checks if the end of the file (EOF) has been reached.

- **Parameters:**
  - stream – a pointer to a FILE object representing the file stream for which the EOF status is to be determined.

- **Return:**
  - 0 – the EOF indicator is not set.
  - -1 – the RAM disk is not initialized.
  - others – the EOF indicator is set for the given stream, indicating that the end of the file has been reached.

- **Key Call:**
  1. **It first checks whether the ram disk is initialized**
  2. **This function returns**

  - Return **0** (indicate file not pointing to **(EOF)**).
  - Return **1** (indicate file pointing to **(EOF)**).

---

10. - **Function:**

```
int ramdisk_remove(const char *filename);
```

- **Description:**

  This function deletes a file from the ram disk.

- **Parameters:**
  - filename – a pointer to a string representing the name of the file to be removed.

- **Return:**
  - 0 – remove the file successfully.
  - -1 – if the RAM disk is not initialized or if any error occurs during the file removal process.
- **Key Call:**
  1. **It first checks whether the ram disk is initialized**
  2. **It constructs the full file path by appending the filename to the ram disk tag (ai_glass_ramdisk_tag)**
  3. **Attempts to deletes the file using fremove**

---

11. - **Function:**

```
void ramdisk_generate_unique_filename(const char *base_name, const char *time_
↪str, const char *extension_name, char *new_name, uint32_t size);
```

- **Description:**

  This function generates a unique filename for saving a file on an ram disk. It ensures the filename does not conflict with an existing file by appending an incrementing number if needed.

- **Parameters:**
  - base_name – a pointer to a string representing the based name, which user want to set.
  - time_str – a pointer to a string representing the time name, which user want to set.
  - extension_name – a pointer to a string representing the extension name, which user want to set.
  - new_name – a pointer to a allocated buffer to store the final result of the file name.
  - size – the size of the allocated buffer of the new_name
- **Return:**
- **Key Call:**
  1. **Constructs the Initial Filename**
  - Combines base_name and time_str to form the initial filename.
  - If the buffer new_name is too small, it prints an error and exits.
  2. **if (!ai_glass_ramdisk_done)**
  - Checks if ram disk is ready.
  3. **Generates a Unique Filename**
  4. **Checks for File Existence**
  5. Creates a New Name with an Incrementing Counter
  - Adds a incrementing numeric suffix (_1, _2, etc.) to avoid name conflicts.
  - Ensures the buffer new_name is large enough.
  6. **Updates temp_path for the Next Iteration**

External Disk API

The external disk is used to store information that needs to be retained for a longer period and will not disappear when the system is turned on or off. It is currently designed to be registered as either an SD card or eMMC. It is currently primarily used to store information related to daily life photography and videography.

1. - **Function:**

  ```
  int extdisk_filesystem_init(const char *disk_tag, int vfs_type, int interface);
  ```

  - **Description:**

    Initializes a external disk filesystem for AI Glass storage. It ensures exclusive access using a mutex, registers the external disk with the virtual filesystem (VFS), and marks the initialization as complete.

  - **Parameters:**

    - disk_tag – a string representing the tag or identifier for the external disk. This is used to form the name of the external disk.
    - vfs_type – the file system type of the external disk.
    - interface – the file system interface of the external disk, like SD card or EMMC.

  - **Return:**

    - 0 – the external disk has been initialized correctly.
    - -1 – this can occur if the mutex cannot be taken in a timely manner or if the external disk has already been initialized.

  - **Key Call:**

    1. **xSemaphoreTake(extdisk_mutex, portMAX_DELAY) != pdTRUE**

    - Acquires the mutex to ensure exclusive access.

    2. **vfs_user_register(disk_tag, vfs_type, interface)**

    - Registers the external disk with the FATFS filesystem in VFS.

    3. **ai_glass_extdisk_done = 1**

    - Set to **1** after registered the external disk.

    4. **xSemaphoreGive(extdisk_mutex)**

    - Releases the mutex after initialization.

---

2. - **Function:**

  ```
  int extdisk_get_init_status(void);
  ```

  - **Description:**

    Return a value 0 or 1.

  - **Parameters:**

    NA

  - **Return:**

    - 1 – the RAM disk has been set up.

  - 0 – the RAM disk has not been set up yet.

- **Key Call:**

  1. **return ai_glass_extdisk_done**

  - ai_glass_extdisk_done = **1** if already initialized.
  - ai_glass_extdisk_done = **0** if not initialize.

---

3. - **Function:**

```
FILE *extdisk_fopen(const char *filename, const char *mode);
```

- **Description:**

  This function function opens a file on an external disk using the specified mode.

- **Parameters:**

  - filename – a string representing the name of the file you want to open on the RAM disk.
  - mode – a string indicating the file opening mode (e.g., "r" for reading, "w" for writing, etc.).

- **Return:**

  - a pointer to a FILE object representing the opened file, or NULL if the file cannot be opened. If the RAM disk is not initialized, the function returns NULL.

- **Key Call:**

  1. **It first checks whether the external disk is initialized**
  2. **It constructs the full file path by appending the filename to the external disk tag (ai_glass_extdisk_tag)**
  3. **Attempts to open the file using fopen**

---

4. - **Function:**

```
int extdisk_fclose(FILE *stream);
```

- **Description:**

  This function closes a file that was opened on an external disk.

- **Parameters:**

  - stream – a pointer to a FILE object representing the open file stream that you wish to close.

- **Return:**

  - 0 on successful closure of the file stream, and EOF (a non-zero value, typically -1) on failure.

- **Key Call:**

  1. **It first checks whether the external disk is initialized**
  2. **Attempts to close the file (stream) using fclose**

---

5. - **Function:**

```
size_t extdisk_fread(void *ptr, size_t size, size_t count, FILE *stream);
```

- **Description:**

  This function function reads data from a file on an external disk into a buffer.

- **Parameters:**
  - ptr – a pointer to a block of memory where the read data should be stored.
  - size – the size, in bytes, of each element to be read.
  - count – the number of elements, each of the specified size, to be read from the file.
  - stream – a pointer to a FILE object that identifies the input file stream from which data is to be read.

- **Return:**
  - the total number of elements successfully read, which may be less than count if a reading error occurs or the end of the file is reached. If the external disk is not initialized, the function returns -1.

- **Key Call:**
  1. **It first checks whether the external disk is initialized**
  2. **Attempts to reads data from the file (stream) into the provided buffer (ptr) using fread**

---

6. - **Function:**

```
size_t extdisk_fwrite(const void *ptr, size_t size, size_t count, FILE␣
↪*stream);
```

- **Description:**

  This function writes data from the buffer ptr to the file.

- **Parameters:**
  - ptr – a pointer to the block of memory containing the data to write to the file.
  - size – the size, in bytes, of each element to be written.
  - count – the number of elements, each of the specified size, to be written to the file.
  - stream – a pointer to a FILE object representing the output file stream where data will be written.

- **Return:**
  - the total number of elements successfully read, which may be less than count if an error occurs during writing. If the external disk is not initialized, the function returns -1.

- **Key Call:**
  1. **It first checks whether the external disk is initialized**
  2. **Attempts to writes data from buffer (ptr) into the file (stream) using fwrite**

---

7. - **Function:**

```
int extdisk_fseek(FILE *stream, long int offset, int origin);
```

- **Description:**

  This function moves the file pointer in stream to a new position based on offset and origin.

- **Parameters:**
  - stream – a pointer to a FILE object representing the file stream whose position is to be set.
  - offset – the number of bytes to offset from origin.
  - origin – the position from which offset is added.

- **Return:**
  - 0 – on successful repositioning of the file pointer.
  - -1 – the function failed or the external disk is not initialized.

- **Key Call:**
  1. **It first checks whether the external disk is initialized**
  2. **offset specifies how far to move the pointer**
  3. **origin specifies the reference point (e.g., SEEK_SET, SEEK_CUR, SEEK_END)**

---

8. - **Function:**

```
int extdisk_ftell(FILE *stream);
```

- **Description:**

  This function gets the current position of the file pointer in stream.

- **Parameters:**
  - stream – a pointer to a FILE object representing the file stream for which the current file pointer position is to be determined.

- **Return:**
  - the current file position indicator as a long integer. If the function fails to retrieve the position or the RAM disk is not initialized, it returns -1.

- **Key Call:**
  1. **It first checks whether the external disk is initialized**
  2. **Returns the current position in bytes**

---

9. - **Function:**

```
int extdisk_feof(FILE *stream);
```

- **Description:**

  This function checks if the end of the file (EOF) has been reached.

- **Parameters:**
  - stream – a pointer to a FILE object representing the file stream for which the EOF status is to be determined.

- **Return:**
  - 0 – the EOF indicator is not set.

  - -1 – the RAM disk is not initialized.
  - others – the EOF indicator is set for the given stream, indicating that the end of the file has been reached.

- **Key Call:**

  1. **It first checks whether the external disk is initialized**
  2. **This function returns**

  - Return **0** (indicate file not pointing to (**EOF**)).
  - Return **1** (indicate file pointing to (**EOF**)).

---

10. - **Function:**

```
int extdisk_remove(const char *filename);
```

- **Description:**

  This function deletes a file from the external disk.

- **Parameters:**

  - filename – a pointer to a string representing the name of the file to be removed.

- **Return:**

  - 0 – remove the file successfully.
  - -1 – if the RAM disk is not initialized or if any error occurs during the file removal process.

- **Key Call:**

  1. **It first checks whether the external disk is initialized**
  2. **It constructs the full file path by appending the filename to the external disk tag (ai_glass_extdisk_tag)**
  3. **Attempts to deletes the file using fremove**

---

11. - **Function:**

```
void extdisk_generate_unique_filename(const char *base_name, const char *time_
↪str, const char *extension_name, char *new_name, uint32_t size);
```

- **Description:**

  This function generates a unique filename for saving a file on an external disk. It ensures the filename does not conflict with an existing file by appending an incrementing number if needed.

- **Parameters:**

  - base_name – a pointer to a string representing the based name, which user want to set.
  - time_str – a pointer to a string representing the time name, which user want to set.
  - extension_name – a pointer to a string representing the extension name, which user want to set.
  - new_name – a pointer to a allocated buffer to store the final result of the file name.
  - size – the size of the allocated buffer of the new_name

- **Return:**

  NA

- **Key Call:**
  1. **Constructs the Initial Filename**
  - Combines base_name and time_str to form the initial filename.
  - If the buffer new_name is too small, it prints an error and exits.
  2. **if (!ai_glass_extdisk_done)**
  - Checks ifexternal disk is ready.
  3. **Generates a Unique Filename**
  4. **Checks for File Existence**
  5. Creates a New Name with an Incrementing Counter
  - Adds a incrementing numeric suffix (_1, _2, etc.) to avoid name conflicts.
  - Ensures the buffer new_name is large enough.
  6. **Updates temp_path for the Next Iteration**

---

12. - **Function:**

```
void extdisk_count_filenum(const char *dir_path, const char **extensions,␣
↪uint16_t *ext_counts, uint16_t num_extensions, const char *exclude_filename);
```

- **Description:**

  This function counts the number of files with specified extensions in a given directory on an external disk. It also allows excluding a specific filename from the count.

- **Parameters:**
  - dir_path – a pointer to a string that represents the path of the directory where the files are counted.
  - extensions – a pointer to an array of strings, where each string is a file extension considered during the count.
  - ext_counts – a pointer to an array of 16-bit unsigned integers where the function stores the counts of files matching each extension.
  - num_extensions – the number of file extensions provided in the extensions array.
  - exclude_filename – a pointer to a string representing a filename that should be excluded from the count.

- **Return:**

  NA

- **Key Call:**
  1. **if (!ai_glass_extdisk_done)**
  - Ensures that external disk is initialized.
  2. **snprintf(ai_glass_path, sizeof(ai_glass_path), "%s%s", ai_glass_extdisk_tag, dir_path)**
  - Combines the external disk's base tag **(ai_glass_extdisk_tag)** with the provided dir_path to construct a full path to search for files.
  3. **Initialize the extension count array**
  - Loops through **num_extensions** to set all values in **ext_counts[i]** to 0 before counting.

4. **extdisk_get_filenum(ai_glass_path, extensions, ext_counts, num_extensions, exclude_filename)**
   - Call this function for actual file counting.

---

13. - **Function:**

```
cJSON *extdisk_get_filelist(const char *list_path, uint16_t *file_number,␣
↪const char **extensions, uint16_t num_extensions, const char *exclude_
↪filename);
```

- **Description:**

  Retrieves a list of files from the external disk that match the specified extensions while excluding a specific filename. The function returns a JSON object containing the file list.

- **Parameters:**
  - list_path – a string representing the directory path from which files will be listed.
  - file_number – a pointer to a 16-bit unsigned integer that will store the number of files found that match the criteria.
  - extensions – a array of strings, where each string represents an extension used to filter files.
  - num_extensions – the number of extensions to consider in the extensions array.
  - exclude_filename – a string representing the filename to exclude from the results.

- **Return:**
  - a pointer to a cJSON object that represents the list of files. If the external disk isn't initialized or if there's a failure in listing the files, the function returns NULL.

- **Key Call:**
  1. **It first checks whether the external disk is initialized**
  2. **It constructs the full file path by appending the filename to the external disk tag (ai_glass_extdisk_tag)**
  3. **Returns a JSON object (cJSON *) containing the file list**
  4. file list JSON
  - file: it has field type, name and time.
  - folder: it has field type, name, time and ore more filed contents which is an array of file json object.
  - example:

```
{
        "type": "folder",
        "name": "aiglass:/",
        "time": "1980-01-06T00:00:00+0000",
        "contents":     [{
                        "type": "file",
                        "name": "lifesnap_19800106_000013.jpg",
                        "time": "1980-01-06T00:00:00+0000"
                }, {
                        "type": "file",
                        "name": "lifesnap_19800106_000020.jpg",
                        "time": "1980-01-06T00:00:00+0000"
```

(continues on next page)

(continued from previous page)

```
                }, {
                        "type": "file",
                        "name": "lifesnap_19800106_000026.jpg",
                        "time": "1980-01-06T00:00:00+0000"
                }, {
                        "type": "file",
                        "name": "lifesnap_19800106_000031.jpg",
                        "time": "1980-01-06T00:00:00+0000"
                }, {
                        "type": "file",
                        "name": "liferecord_19800106_000038.mp4",
                        "time": "1980-01-06T00:00:00+0000"
                }, {
                        "type": "file",
                        "name": "liferecord_19800106_000106.mp4",
                        "time": "1980-01-06T00:01:00+0000"
                }, {
                        "type": "file",
                        "name": "lifesnap_19800106_000128.jpg",
                        "time": "1980-01-06T00:01:00+0000"
                }, {
                        "type": "file",
                        "name": "lifesnap_19800106_000134.jpg",
                        "time": "1980-01-06T00:01:00+0000"
                }, {
                        "type": "file",
                        "name": "lifesnap_19800106_000139.jpg",
                        "time": "1980-01-06T00:01:00+0000"
                }]
}
```

14. - **Function:**

```
const char *extdisk_get_filesystem_tag_name(void);
```

- **Description:**

  This function returns the tag name of the external disk's filesystem, which is stored in ai_glass_extdisk_tag. If the external disk is not initialized, the function prints a warning and returns NULL.

- **Parameters:**

  NA

- **Return:**

  – a string of the external disk tag name.

- **Key Call:**

  1. **if (!ai_glass_extdisk_done)**

  – Ensures that external disk is initialized.

  2. **char *tag_name = malloc(MAX_TAG_LEN)**

  – Allocates Memory for the Tag Name.

3. **snprintf(tag_name, MAX_TAG_LEN, "%s", ai_glass_extdisk_tag)**
   - Copies the Filesystem Tag Name.
4. **Returns the Allocated String**

File System Internal Functions reference

1. - **Function:**

```
unsigned long get_fattime(void)
```

- **Description:**

  This function generates a FAT file system timestamp in a format compatible with FATFS **(File Allocation Table File System)**. It constructs a **32-bit timestamp** based on GPS time and media time offset.

- **Parameters:**

  NA

- **Return:**
  - return the time, which represents the raw time value, typically the number of seconds since the Unix epoch (January 1, 1970).

- **Key Call:**
  1. **Get the current time based on GPS and media time**
  2. **Convert time_t to UTC struct tm**
  3. **Construct a FATFS-compatible 32-bit timestamp**
  4. **Return the encoded FAT timestamp**

---

2. - **Function:**

```
static void time_transfer_to_string(time_t rawtime, char *buffer, uint32_t␣
↪buffer_size);
```

- **Description:**

  The function converts a given **time_t** value **(Unix timestamp)** into a formatted string representing the date and time.

- **Parameters:**
  - rawtime – represents the raw time value, typically the number of seconds since the Unix epoch (January 1, 1970).
  - buffer – a character buffer where the formatted time string will be stored.
  - buffer_size – the size of the buffer in bytes, used to ensure the formatted string does not exceed the buffer's capacity.

- **Return:**

  NA

- **Key Call:**

1. **struct tm *timeinfo = gmtime(&rawtime)**

- Converts time_t (epoch time) into **struct tm (UTC time)**.

2. **strftime(buffer, buffer_size, "%Y%m%d_%H%M%S", timeinfo)**

- Formats the time into a string using the pattern **(YYYYMMDD_HHMMSS)**.

---

3. - **Function:**

```
static void time_transfer_to_string_utcform(time_t rawtime, char *buffer,␣
↪uint32_t buffer_size);
```

- **Description:**

  The function converts a given time_t timestamp into a formatted UTC-based string, incorporating the timezone offset.

- **Parameters:**
  - rawtime – represents the number of seconds since the UNIX epoch (January 1, 1970, 00:00:00 UTC).
  - buffer – a character buffer to store the formatted time string.
  - buffer_size – the size of the buffer in bytes. This controls how much data can be safely written to the buffer.
- **Return:**
- **Key Call:**

  1. **struct tm *timeinfo = gmtime(&rawtime)**

  - Converts time_t (epoch time) into **struct tm (UTC time)**.

  2. **strftime(buffer, buffer_size, "%Y%m%d_%H%M%S", timeinfo)**

  - Formats the time into a string using the pattern **(YYYYMMDD_HHMMSS)**.

---

4. - **Function:**

```
static int show_utc_format_time(time_t rawtime);
```

- **Description:**

  Converts a given time_t value into a human-readable UTC timestamp and prints it. This function helps in displaying the system time in a structured format.

- **Parameters:**
  - rawtime – represents the number of seconds since the UNIX epoch (January 1, 1970, 00:00:00 UTC).
- **Return:**
  - 0 – indicating successful completion.
- **Key Call:**

  1. **gmtime(&rawtime)**

  - Converts time_t (epoch time) into **struct tm (UTC time)**.

  2. **asctime(timeinfo)**

  - Formats struct tm into a **human-readable string**.

3. **Prints the formatted UTC time**.

---

5. - **Function:**

```
static void ai_glass_init_external_disk(void);
```

- **Description:**

  Initializes the external disk if it has not been initialized. This ensures that the external disk is set up with the FATFS filesystem.

- **Parameters:**

  NA

- **Return:**

  NA

- **Key Call:**

  1. **!extdisk_get_init_status()**

  – Checks if the external disk is already initialized.

  2. **extdisk_filesystem_init(ai_glass_disk_name, VFS_FATFS, EXTDISK_PLATFORM)**

  – Sets up the external disk with the FATFS filesystem on the specified platform.

---

6. - **Function:**

```
static cJSON *get_filelist(const char *list_path, const char *folder_name,␣
↪uint16_t *file_number, const char **extensions, uint16_t num_extensions,␣
↪const char *exclude_filename);
```

- **Description:**

  Scans a specified directory **(list_path)** and recursively retrieves a list of files and folders, filtering results based on file extensions and excluding a specified filename.

- **Parameters:**

  – list_path – a string representing the directory path from which files will be listed.

  – folder_name – a string representing the folder name from which files will be listed.

  – file_number – a pointer to a 16-bit unsigned integer that will store the number of files found that match the criteria.

  – extensions – a array of strings, where each string represents an extension used to filter files.

  – num_extensions – the number of extensions to consider in the extensions array.

  – exclude_filename – a string representing the filename to exclude from the results.

- **Return:**

  – a pointer to a cJSON object that represents the list of files. If the external disk isn't initialized or if there's a failure in listing the files, the function returns NULL.

- **Key Call:**

  1. **Creates a JSON object for the directory**

2. **Opens the directory for reading**
   - Ensures the directory exists before processing.
3. **Recursively processes subdirectories**
4. **Filters and adds valid files**
   - Filters files based on extensions and exclusion rules.
   - Converts valid files into JSON objects and adds them to the list.
5. **Closes the directory and frees memory**
   - Ensure all files are properly closed and cleanup.

---

7. - **Function:**

```
static cJSON *create_json_folder_object(const char *list_path, const char␣
↪*foldername);
```

- **Description:**

  This function creates a cJSON object representing a folder, including its name, type, and last modification time.

- **Parameters:**
  - list_path – a string representing the directory path.
  - folder_name – a string representing the folder name.
- **Return:**
  - a pointer to a cJSON object that represents the folder.
- **Key Call:**
  1. **This initializes an empty JSON object**
  2. **Adds folder type and name to JSON**
  3. **Retrieves file system metadat**
  4. **Converts modification time to UTC format**
  5. **Returns the created JSON object**

---

8. - **Function:**

```
static cJSON *create_json_file_object(const char *list_path, const char␣
↪*filename);
```

- **Description:**

  This function creates a cJSON object representing a file, including its name, type, and last modification time.

- **Parameters:**
  - list_path – a string representing the directory path.
  - filename – a string representing the file name.
- **Return:**
  - a pointer to a cJSON object that represents the file.

- **Key Call:**
  1. **This initializes an empty JSON object**
  2. **Adds file type and name to JSON**
  3. **Constructs full file path dynamically**
  4. **Retrieves file system metadat**
  5. **Converts modification time to UTC format**
  6. **Frees dynamically allocated memory**
  - Prevent memory leak.
  7. **Returns the created JSON object**

---

9. - **Function:**

```
static void extdisk_get_filenum(const char *dir_path, const char **extensions,␣
↪uint16_t *ext_counts, uint16_t num_extensions, const char *exclude_filename);
```

- **Description:**

  This function recursively counts the number of files with specified extensions in a given directory, including subdirectories.

- **Parameters:**
  - dir_path – a pointer to a string that represents the path of the directory where the files are counted.
  - extensions – a pointer to an array of strings, where each string is a file extension considered during the count.
  - ext_counts – a pointer to an array of 16-bit unsigned integers where the function stores the counts of files matching each extension.
  - num_extensions – the number of file extensions provided in the extensions array.
  - exclude_filename – a pointer to a string representing a filename that should be excluded from the count.

- **Return:**

  NA

- **Key Call:**
  1. **file_dir = opendir(dir_path)**
  - Open the directory **(dir_path)** for reading.
  2. **Loop through directory entries (readdir)**
  - Reads each file or folder within the directory one by one.
  3. **if (entry->d_type == DT_DIR)**
  - If an entry is a subdirectory, it recursively calls **extdisk_get_filenum** to process its contents.
  4. **int extension_num = check_valid_file_and_remove(dir_path, entry->d_name, exclude_filename, extensions, num_extensions)**
  - Determines if the file matches one of the specified extensions and is not the excluded filename.
  5. **Increment the count for matching files**

- If the file is valid, it updates the corresponding index in **ext_counts**.

6. **Free allocated memory and close directory (closedir)**

- Ensures proper cleanup after processing the directory.

---

10. - **Function:**

```
static int check_valid_file_and_remove(const char *list_path, const char␣
↪*filename, const char *exclude_filename, const char **extensions, uint16_t␣
↪num_extensions);
```

- **Description:**

  This function checks if a given file is valid based on specified conditions, such as its size and extension. It also removes invalid files **(e.g., empty files or those with failed stat operations)**.

- **Parameters:**
  - list_path – the directory path where the file resides.
  - filename – the name of the file to be checked.
  - exclude_filename – a specific filename that needs to be excluded.
  - extensions – the array of valid file extensions.
  - num_extensions – the number of valid extensions in the extensions array.

- **Return:**
  - -1 – means the file is invalid
  - 0 – represents the extension index of the extension array

- **Key Call:**

  1. **if (is_excluded_file(filename, exclude_filename))**

  - Check if **filename** matches **exclude_filename**, the function returns **-1**, **indicating exclusion**.

  2. **Constructs the full file path (file_path) and retrieves file information using stat()**

  - snprintf(file_path, PATH_MAX + 1, "%s/%s", list_path, filename)
  - res = stat(file_path, &finfo)

  3. **if (finfo.st_size == 0)**

  - If the file size is 0, it is considered invalid and is deleted using remove().

  4. **if (check_extension(filename, extensions[i]))**

  - Compares the filename's extension with a list of valid extensions.
  - Returns the index of the matching extension if found.

  5. **Free allocated memory**

  - Ensures allocated memory for **file_path** is properly freed before returning.

---

11. - **Function:**

```
static int is_excluded_file(const char *filename, const char *exclude_
↪filename);
```

- **Description:**

  This function determines whether a given file should be excluded based on its filename suffix.

- **Parameters:**
  - filename – the name of the file being checked.
  - exclude_filename – the exclusion criterion, which is a string that the filename should end with to be considered excluded.

- **Return:**
  - 0 – means the file is not a excluded file
  - 1 – means the file is a excluded file

- **Key Call:**
  1. **Checks if the filename is shorter than the exclude_filename**
  - Check if **filename** matches **exclude_filename**, the function returns **1**, **indicating exclusion** else return **0**, **indicating non-exclusion**.

---

12. - **Function:**

```
static int check_extension(const char *filename, const char *ext);
```

- **Description:**

  This function verifies whether the given filename has the specified file extension.

- **Parameters:**
  - filename – the name of the file being checked.
  - ext – the file extension to check for at the end of the filename.

- **Return:**
  - 0 – means the file does not match the extension
  - 1 – means the file match the extension

- **Key Call:**
  1. **Ensures the filename is long than extension name**
  - Return **0** (indicate fail).
  - Return **1** (indicate success).

---

13. - **Function:**

```
static int file_exists(const char *filename);
```

- **Description:**

  This function verifies whether a file exists at the specified path. It attempts to open the file in read mode ("r").

- **Parameters:**
  - filename – the path to the file whose existence is being checked.
- **Return:**
  - 0 – the file is not exist
  - 1 – the file is exist
- **Key Call:**
  1. **Ensures the filename is long than extension name**
  - Return **0** (indicate file does not exist).
  - Return **1** (indicate the file exists).

3.1.6 UART System

This section describes the UART process of the AI Wearable Glass:

UART CMD Opcode

Type of UART commands	CMD code
UART_OPC_CMD_ACK	0x0000
UART_RX_OPC_RESP_START_BT_SOC_FW_UPGRADE	0x0631
UART_RX_OPC_RESP_FINISH_BT_SOC_FW_UPGRADE	0x0633
UART_OPC_CMD_QUERY_INFO	0x8400
UART_OPC_CMD_POWER_ON	0x8401
UART_OPC_CMD_POWER_DOWN	0x8402
UART_OPC_CMD_GET_POWER_STATE	0x8403
UART_OPC_CMD_UPDATE_WIFI_INFO	0x8404
UART_OPC_CMD_SET_GPS	0x8405
UART_OPC_CMD_SNAPSHOT	0x8406
UART_OPC_CMD_GET_FILE_NAME	0x8407
UART_OPC_CMD_GET_PICTURE_DATA	0x8408
UART_OPC_CMD_TRANS_PIC_STOP	0x8409
UART_OPC_CMD_RECORD_START	0x840A
UART_OPC_CMD_RECORD_CONT	0x840B
UART_RX_OPC_CMD_RECORD_SYNC_TS	0x840C
UART_OPC_CMD_RECORD_STOP	0x840D
UART_RX_OPC_CMD_GET_FILE_CNT	0x840E
UART_RX_OPC_CMD_DELETE_FILE	0x840F
UART_OPC_CMD_GET_SD_INFO	0x8411
UART_OPC_CMD_SET_WIFI_MODE	0x8412
UART_RX_OPC_CMD_GET_PICTURE_DATA_SLIDING_WINDOW	0x8413
UART_RX_OPC_CMD_GET_PICTURE_DATA_SLIDING_WINDOW_ACK	0x8414
UART_RX_OPC_CMD_SET_WIFI_FW_UPGRADE	0x8415
UART_OPC_CMD_SET_SYS_UPGRADE	0x8430

UART Response Opcode

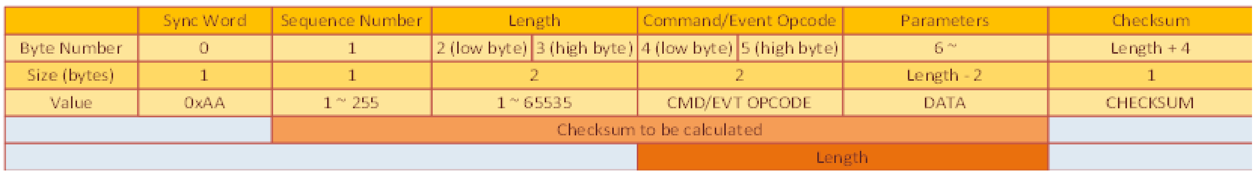
Type of UART commands	CMD code
UART_OPC_CMD_ACK	0x0000
UART_TX_OPC_CMD_START_BT_SOC_FW_UPGRADE	0x0631
UART_TX_OPC_CMD_TRANSFER_UPGRADE_DATA	0x0632
UART_TX_OPC_CMD_FINISH_BT_SOC_FW_UPGRADE	0x0633
UART_TX_OPC_RESP_QUERY_INFO	0x8400
UART_TX_OPC_RESP_POWER_ON	0x8401
UART_TX_OPC_RESP_GET_POWER_STATE	0x8403
UART_TX_OPC_RESP_UPDATE_WIFI_INFO	0x8404
UART_TX_OPC_RESP_SET_GPS	0x8405
UART_TX_OPC_RESP_SNAPSHOT	0x8406
UART_TX_OPC_RESP_GET_FILE_NAME	0x8407
UART_TX_OPC_RESP_GET_PICTURE_DATA	0x8408
UART_TX_OPC_RESP_TRANS_PIC_STOP	0x8409
UART_TX_OPC_RESP_RECORD_START	0x840A
UART_TX_OPC_RESP_RECORD_CONT	0x840B
UART_TX_OPC_RESP_RECORD_SYNC_TS	0x840C
UART_TX_OPC_RESP_RECORD_STOP	0x840D
UART_TX_OPC_RESP_GET_FILE_CNT	0x840E
UART_TX_OPC_RESP_DELETE_FILE	0x840F
UART_TX_OPC_RESP_DELETE_ALL_FILES	0x8410
UART_TX_OPC_RESP_GET_SD_INFO	0x8411
UART_TX_OPC_RESP_SET_WIFI_MODE	0x8412
UART_TX_OPC_RESP_GET_PICTURE_DATA_SLIDING_WINDOW	0x8413
UART_TX_OPC_RESP_SET_WIFI_FW_UPGRADE	0x8415
UART_OPC_RESP_REQUEST_SET_SYS_UPGRADE	0x8430

UART Error Code

Error Code	Descriptor
0x00	command complete : MCU can handle this command.
0x01	command disallow : MCU can not handle this command.
0x02	unknow CMD
0x03	parameters error
0x04	busy
0x05	process fail
0x06	one wire extend
0x07	TTS request (for watch)
0x08	music request (for watch)
0x09	version incompatible
0x0A	scenario error
0x11	GATT (for watch)
0x12	GATT error (for watch)
0x20	SD full
0x21	Device work scenario warning
0x30	OTA file not existed
0x31	OTA process failed
0x32	OTA failed due to Battery low
other	RFD

Transfer Data Format

The CMD/EVENT sequence is as follow:



	Sync Word	Sequence Number	Length		Command/Event Opcode		Parameters	Checksum
Byte Number	0	1	2 (low byte)	3 (high byte)	4 (low byte)	5 (high byte)	6 ~	Length + 4
Size (bytes)	1	1	2		2		Length - 2	1
Value	0xAA	1 ~ 255	1 ~ 65535		CMD/EVT OPCODE		DATA	CHECKSUM
		Checksum to be calculated						
					Length			

- Sync word: Serves as the starting point.
- Sequence number: The number representing the local sending sequence (1~255), in incrementing sequentially.
- Length: The total length of the Opcode and parameters combined.
- Checksum: The value calculated from the UART-transmitted data (starting from the sequence number) using app_util_calc_checksum.

The code is as follows:

```
uint8_t app_util_calc_checksum(uint8_t *dataPtr, uint16_t len)
{
  uint8_t check_sum;
  check_sum = 0;
  while (len)
  {
```

(continues on next page)

(continued from previous page)

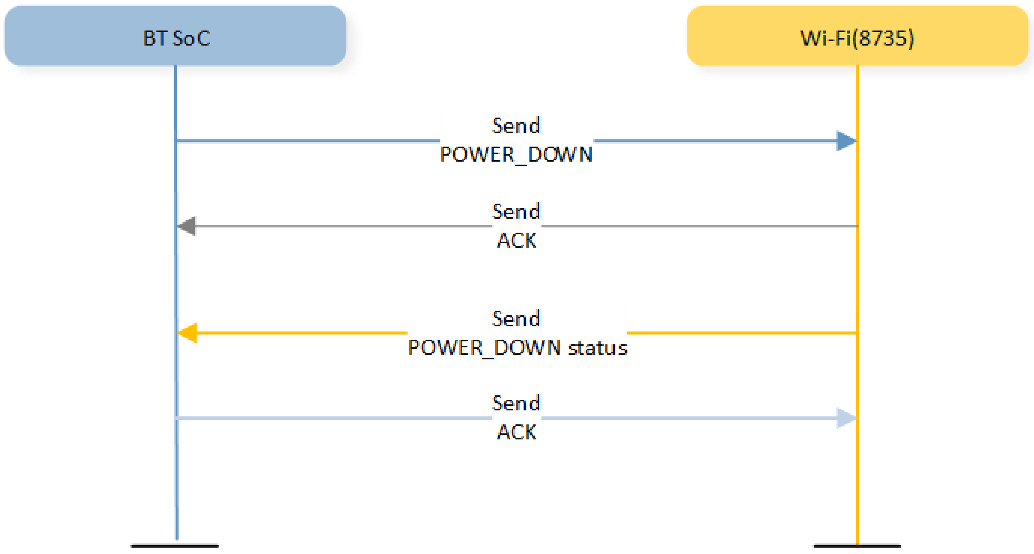
```
    check_sum += *dataPtr;
    dataPtr++;
    len--;
  }
  return (0xff - check_sum + 1); //((~check_sum)+1);
}
```

Checksum Rule: The summation **from Seq to CHK SUM** must equal 0x00. This field is only valid for the UART communication path.

Example:

- For the UART path: AA 01 04 00 08 00 01 00 F2
- For the SPP path: AA 01 04 00 08 00 01 00

When the BT SoC sends a command, the counterpart will respond with an ACK. Similarly, when the counterpart sends a command to the BT SoC, it will respond with an ACK. Generally, command exchanges between the BT SoC and the counterpart follow a one-to-one correspondence, as illustrated in the sequence flow below.



Uart communication Parameters between BT SoC and 8735

ACK

ACK(8735->BT SoC) (ID: 0x0000)		
Parameters	Byte0~Byte1	Event ID
	Byte2	Status
		Error code

ACK(BT SoC->8735) (ID: 0x0000)		
Parameters	Byte0~Byte1	CMD ID
	Byte2	Status Error code

The 8735 sends an mmi power_on CMD message to the BT SoC. After the BT SoC completes the execution, it responds with an event back to 8735.

8735 ->> BT SoC: AA 07 04 00 04 00 00 54 9D

syncWord: AA, seqn: 07, len: 0x0004, opcode: 0004, params : 0x54 (power_on mmi), status: 0, check_sum: 0x9D(data: 07 04 00 04 00 00 54)

8735 <<- BT SoC: AA 01 05 00 00 00 04 00 00 F6 // ACK

syncWord: 0xAA, seqn: 0x01, len: 0x0005, opcode: 0000, event_id: 0x0004 status: 0, check_sum: 0xF6(data: 01 05 00 00 00 04 00 00)

8735 <<- BT SoC: AA 02 03 00 07 00 01 F3

syncWord: AA,seqn: 02, len: 0x0003, opcode: 0007, params: 0x01, check_sum: 0xF3

8735 ->> BT SoC: AA 08 05 00 00 00 07 00 00 EC //ACK

syncWord: AA,seqn: 08, len: 0x0005, opcode: 0000, event_id: 0x0007 status: 0, check_sum: 0xEC

Power on

The Wi-Fi chip powered through a pin enable. After the Wi-Fi UART is ready, it sends a command to notify the BT SoC.

POWER_ON(8735->BT SoC) (ID: 0x8401)		
Parameters	Byte0	Result Status

Init Info

QUERY_INFO(BT SoC->8735) (ID: 0x8400)		
Parameters	Byte0~Byte1	**Packet size** The size of each packet used for transmitting image data.
	Byte2~Byte3	**Buffer check size** Calculate the data length of buffer check.
	Reserve (28 bytes)	

After ACK by 8735, it sends some parameter information together with Some parameter its own version information.

QUERY_INFO_RESP(8735->BT SoC) (ID: 0x8400)		
Parameters (Total 32 bytes)	Byte0~Byte1	**Packet size** The size of each packet used for transmitting image data.
	Byte2~Byte3	**Buffer check size** Calculate the data length of buffer check.
	Byte4	**Protocol version** Protocol version number. After SoC adds new functions or fixes bugs, the version number increases, = 0, = 1, = 2, …
	Byte5	**WIFI IC type**
	Reserve (26 bytes)	

Power down

POWER_DOWN(BT SoC->8735) (ID: 0x8402)
Parameters

Get power state

BT Soc acquire the power state of 8735 chip.

After the BT SoC sends a power down command to the 8735. The 8735 first orderly shuts down the device and responds back to the BT SoC. Subsequently, the BT SoC will turn off the power.

GET_POWER_STATE(BT SoC->8735) (ID: 0x8403)
Parameters

8735 respond with its power states back to BT SoC:

GET_POWER_STATE_RESP(8735->BT SoC) (ID: 0x8403)		
Parameters	Byte0	**Result** 0x01:power on 0x02:DLPS 0x03:power down 0x04:AP mode 0x05:HTTP transmit mode others: fail

Update Wi-Fi parameter

BT SoC updates parameters to 8735:

UPDATE_WIFI_INFO(BT SoC->8735) (ID: 0x8404)		
Parameters	Byte0	Mode 1:snapshot default parameter, 2:record default parameter; 3.max record time (s)
	Byte1~Byte2	Param_data length
	Byte3~ByteN	Param_data

REPORT_UPDATE_WIFI_PARAM(8735->BT SoC) (ID: 0x8404)		
Parameters	Byte0	Status

Set GPS

Get GPS information from mobile phone, and update to 8735 for setting GPS metadata to picture and video.

SET_GPS(BT SoC->8735) (ID: 0x8405)		
Parameters	Byte0~Byte7	**GPS value** 0x00000000 none GPS

Snapshot

Notify the 8735 to initiate the snapshot function. After completion of snapshot function, 8735 will report either a success status or a failure status. Subsequently, the BT SoC will play a tone and display the status, followed by starting to transmit the picture.

SNAPSHOT(BT SoC->8735) (ID: 0x8406)		
Parameters	Byte0	Mode; 0 life, 1 AI,
	Byte1~Byte127	Parameter data;

SNAPSHOT_RESP(8735->BT SoC) (ID: 0x8406)		
Parameters	Byte0	**Result** Error code

Picture transmits

Firstly, the 8735 will send the image name to BT SoC before transmitting image data. Additionally, the BT SoC will then send it to the host, where the host will create an image folder and return the result back to the BT SoC while continuing to retrieve image information.

Get file name

Obtain the number of files and the total length of the file names.

Before starting to transmit the picture, first send the picture name to BT SoC, and BT SoC forwards it to the Host (there is a unified limit here, the maximum file name is 128 bytes).

GET_FILE_NAME(BT SoC->8735) (ID: 0x8407)
Parameters

GET_FILE_NAME_RESP(8735->BT SoC) (ID: 0x8407)		
Parameters	Byte0	**Result** Error code
	Byte1~Byte2	**Length** The number of bytes occupied by the image name.
	Byte3~Byte6	**Length of the whole file** The data length of the entire image file.
	Byte7~ByteN	**File name**

Get file data

Protocol Version 0

One-Command, Multi-Event. The picture data may be divided into multiple packets for transmission. The BT SoC actively retrieves multiple data packets.

GET_PICTURE_DATA(BT SoC->8735) (ID: 0x8408)		
Parameters	Byte0~BYTE3	**The current offset ofthe offset file.**
	Byte4	**Packet num**

The 8735 can issue this command multiple times, each with image data.

GET_PICTURE_DATA_RESP(8735->BT SoC) (ID: 0x8408)		
Parameters	Byte0	**cnt (from 0 to Packet num)** **0xFF: end (The flag bit of the last transmitted data)**
	Byte1~ByteN	**Byte1~Byte2:** **data length** **Byte3~ByteN** **file data**

Transfer stop

Notify the 8735 to stop transmission.

TRANS_PICTURE_STOP(BT SoC->8735) (ID: 0x8409)
Parameters

TRANS_PICTURE_STOP_RESP(8735->BT SoC) (ID: 0x8409)		
Parameters	Byte0	**Result** Error code

Record Video

Record Start

Notify the 8735 to initiate the recording function. After completion of recording function, 8735 will report either a success status or a failure status.

RECORD_START(BT SoC->8735) (ID: 0x840A)
Parameters

RECORD_START_RESP(8735->BT SoC) (ID: 0x840A)		
Parameters	Byte0	**Result** Error code

Continuously report record status

8735 regularly reports recording status, such as SD full status.

RECORD_CONTINUE(8735->BT SoC) (ID: 0x840B)		
Parameters	Byte0	**Result** Error code

Record Stop

Notify the 8735 to terminal the recording function. After termination of recording function, 8735 will report either a success status or a failure status.

RECORD_STOP(BT SoC->8735) (ID: 0x840D)
Parameters

RECORD_STOP_RESP(8735->BT SoC) (ID: 0x840D)		
Parameters	Byte0	**Result** Error code

Get SD Information

The host app needs to display the SD card capacity, so it needs to retrieve the SD card information status.

GET_SD_INFO(BT SoC->8735) (ID: 0x8411)
Parameters

GET_SD_INFO_RESP(8735->BT SoC) (ID: 0x8411)		
Parameters	Byte0~Byte3	Used size
	Byte4~Byte7	**Total size**

Set Wi-Fi mode

Set wifi mode.

SET_WIFI_MODE(BT SoC ->8735) (ID: 0x8412)		
Parameters	Byte0	MODE 0 IDLE 1 AP mode

SET_WIFI_MODE_RESP(8735->BT SoC) (ID: 0x8408)		
Parameters	Byte0	Result 0:set success Others:fail
	Byte1	SSID length(max length:33)
	Byte2~Byte34	SSID_VAL
	Byte35	resv
	Byte36~ Byte39	security_type
	Byte40	channel
	Byte41	password_len(max length:65)
	Byte42~ Byte106	password
	Byte107~Byte111	resv

UART Public API reference

1. - **Function**

```
int uart_send_packet(uint16_t resp_opcode, uart_params_t *params_head, bool␣
↪ignore_ack, int timeout);
```

- **Description:**

  This function is responsible for sending a packet over UART while ensuring synchronization, mutex handling, and acknowledgment (ACK) reception if required. It constructs and transmits a packet with parameters and retries if an **acknowledgment is not received within a timeout period**, **unless ignore_ack is set to true**.

- **Parameters:**
  - resp_opcode – The response opcode indicating the type of packet being sent.
  - params_head – A pointer to a linked list of parameters that need to be sent as part of the UART packet.
  - ignore_ack – A boolean flag that determines whether to wait for an acknowledgment after sending the packet.
  - timeout – The time in milliseconds to wait for semaphore locks and acknowledgment before considering the transmission failed.

- **Return:**
  - UART_OK – Success.
  - UART_SEND_PACKET_TIMEOUT – Failed due to timeout.
  - UART_ACK_TIMEOUT – Failed to receive expected acknowledgment **(if ignore_ack is false)**.

- **Key Call:**

  1. **Semaphore Handling**

  - **xSemaphoreTake(tx_resp_mutex, timeout):** – Locks the response mutex for exclusive access.
  - **xSemaphoreTake(tx_pkt_sema, timeout):** – Locks the packet semaphore to prevent simultaneous transmissions.
  - **xSemaphoreTake(tx_ack_mutex, timeout):** – Waits for acknowledgment if ignore_ack is false.
  - **xSemaphoreGive(tx_pkt_sema):** – Releases the packet semaphore after sending data.
  - **xSemaphoreGive(tx_resp_mutex):** – Releases the response mutex after completing transmission.
  - **xSemaphoreGive(tx_ack_mutex):** – Releases the acknowledgment mutex once ACK is received.

  2. **Packet Construction & Sending**

  - **get_expected_tx_seq_number():** – Retrieves the expected transmission sequence number.
  - **CalculateChecksum():** – Computes a checksum for error detection.
  - **uart_send_buffer(&srobj.sobj, data, length, flag):** – Sends data over UART in chunks.

  3. **Acknowledgment Handling & Retransmission (if ignore_ack is false)**

  - **xQueueReceive(rx_uart_ack_tmp_recycle, (void *)&ack_packet, 0):** – Checks for old acknowledgment packets before waiting for a new one.
  - **xQueueReceive(rx_uart_ack_ready, (void *)&ack_packet, timeout / 3):** – Receives acknowledgment from the queue.

- **xQueueSend(rx_uart_ack_tmp_recycle, (void *)&ack_packet, 0):** – Recycles received acknowledgment packets.
- **Retry logic:** – If acknowledgment is not received within the timeout, the function retries sending the packet based on retry_time count.
- **Acknowledgment validation:** – Ensures the received acknowledgment matches the expected opcode before proceeding.

---

2. - **Function**

```
void uart_service_send_pwron_cmd(void);
```

- **Description:**

  This function sends a power-on command **(UART_OPC_RESP_POWER_ON)** over UART if the UART service is initialized. The command contains a single byte indicating the power status.

- **Parameters:**

  NA

- **Return:**

  NA

- **Key Call:**

  1. **Checks UART Initialization:**

  - If **srobj.uart_init** is **true**, it proceeds to send command.
  - if **srobj.uart_init** is **false**, it send a warning message.

  2. **Creates a UART Parameter Packet structure with the parmeters**
  3. **Calls uart_send_packet to Send the Command:**

  - Sends UART_OPC_RESP_POWER_ON with the created parameter packet.
  - Uses a timeout of 2000 ms for the operation.

---

3. - **Function**

```
int uart_service_init(PinName tx_pin, PinName rx_pin, int baudrate);
```

- **Description:**

  This function initializes a UART service by configuring the hardware, creating necessary queues and semaphores, and launching multiple FreeRTOS tasks for UART communication.

- **Parameters:**

  - tx_pin – The UART transmit **(TX)** pin.
  - rx_pin – The UART receive **(RX)** pin.
  - baudrate – The **baud rate** for UART communication.

- **Return:**

  - UART_OK – Successful initialization.
  - UART_SERVICE_INIT_FAIL – If initialization fails at any stage.

- **Key Call:**
  1. **UART Initialization:**
  - Initializes the UART hardware with the specified TX and RX pins.
  - Configures the UART baud rate.
  - Sets the UART format to 8 data bits, no parity, and 1 stop bit.
  - Registers an interrupt handler for UART events.
  - Sets the send completion handler for UART transmissions.
  2. **Stream Buffer Creation:**
  - Creates a stream buffer for UART reception.
  3. **Queue Creation:**
  - **xSemaphoreCreateBinary()** – Creates binary semaphores for transmission control.
  - **xSemaphoreCreateMutex()** – Creates mutexes for synchronizing access to shared resources.
  - **xSemaphoreGive()** – Releases semaphores to their initial state.
  4. **Task Creation:**
  - Creates threads for processing UART commands.
  - Creates a thread for handling critical UART operations.
  - Creates a thread for handling UART acknowledgments.
  - Creates a thread for receiving UART data.
  5. Final Initialization Flag:
  - Set **srobj.uart_init** = 1;

---

4. - **Function**

```
int uart_service_rx_cmd_reg(uint16_t uart_cmd_type, Callback_t uart_cmd_fun);
```

- **Description:**

  This function registers a callback function for a specific UART command type. It stores the command type and its associated callback function in a lookup table, allowing the system to execute the corresponding function when the command is received.

- **Parameters:**
  - uart_cmd_type – The command identifier/name for the UART message.
  - uart_cmd_fun – The callback function to be executed when the specified command is received.
- **Return:**
  - UART_OK – If the callback is successfully registered.
  - UART_REG_CALLBACK_FAIL – If the command type is invalid.
- **Key Call:**
  1. **IS_VALID_UART_CMD(uart_cmd_type)**
  - A macro or function that checks if uart_cmd_type falls within a valid range.

2. **Callback Registration:**

– Stores the command type in the uartcmdcb_table lookup table.

– Store the callback function in the uartcmdcb_table lookup table, executed when the command is received.

---

5. - **Function**

```
int uart_service_start(int send_power_start);
```

- **Description:**

  This function enables UART receive **(Rx)** and transmit **(Tx)** interrupts and **optionally** sends a **power-on command** when starting the UART service.

- **Parameters:**

  – send_power_start – A flag that determines whether to send a power-on command (**1 to send, 0 to skip**).

- **Return:**

  – UART_OK – If the service starts successfully.

  – UART_START_FAIL – If the UART service is not initialized.

- **Key Call:**

  1. **Checks UART Initialization:**

  – If **srobj.uart_init** is **true**, it proceeds to enable interrupts.

  – if **srobj.uart_init** is **false**, it returns UART_START_FAIL.

  2. **Enabling Interrupts:**

```
serial_irq_set(&(srobj.sobj), RxIrq, 1);

serial_irq_set(&(srobj.sobj), TxIrq, 1);
```

  3. **Sending Power-On Command (Optional):**

  – If **send_power_start** = 1, the function calls **uart_service_send_pwron_cmd()** to send a power-on command.

6. - **Function**

```
void uart_service_poweroff(void);
```

- **Description:**

  This function deinitializes the UART interface if it has been initialized. It is typically used during system shutdown or when UART communication is no longer needed.

- **Parameters:**

  NA

- **Return:**

  NA

- **Key Call:**

  1. **serial_free(&(srobj.sobj))**

- Releases resources and deinitializing the UART hardware.

7. - **Function**

```
void uart_service_deinit(void);
```

- **Description:**

  This function deinitializes the UART service, stopping related tasks and freeing allocated resources. This function ensures that all associated tasks, semaphores, and buffers are properly deleted before shutdown.

- **Parameters:**

  NA

- **Return:**

  NA

- **Key Call:**

  1. **UART Dinitialization:**

  - uart_service_poweroff() – Calls uart_service_poweroff to dinitialize UART hardware.
  - vTaskDelete(uartcmdtask[i]) – Deletes UART command handling tasks.
  - vTaskDelete(uartcrticaltask) – Deletes the UART critical command task.
  - vTaskDelete(uartacktask) – Deletes the UART acknowledgment task.
  - vTaskDelete(uartrecvtask) – Deletes the UART receive task.
  - vStreamBufferDelete(uart_rx_stream) – Deletes the UART receive stream buffer.
  - free_rx_uart_queue() – Frees the RX queue used for UART communication.
  - free_tx_uart_queue() – Frees the TX queue used for UART communication.
  - vSemaphoreDelete(uart_tx_free_sema) – Deletes the UART TX semaphore.
  - vSemaphoreDelete(tx_resp_mutex) – Deletes the TX response mutex.
  - vSemaphoreDelete(tx_pkt_sema) – Deletes the TX packet semaphore.
  - vSemaphoreDelete(tx_ack_mutex) – Deletes the TX acknowledgment mutex.

UART Internal Functions reference

1. - **Function**

```
static uint8_t CalculateChecksum(uint8_t sync_word, uint8_t seq_number, uint16_
↪t resp_opcode, uart_params_t *params_array, uint16_t *length);
```

- **Description:**

  This function is responsible for computing a checksum for a UART packet using the given parameters. It sums up various fields and applies a one's complement operation to return the final checksum.

- **Parameters:**

  - sync_word – The synchronization byte (currently excluded from checksum calculation).
  - seq_number – The sequence number of the UART packet.

  - resp_opcode – The response operation code, a 16-bit identifier for the UART command.
  - params_array – A linked list of UART parameters containing data and length.
  - length – A pointer to store the total length of the packet.

- **Return:**
  - Returns a **uint8_t checksum value**, calculated by summing packet components and applying a 1's complement operation.

- **Key Call:**

  1. **Initialize Checksum:**

  - **uint8_t checksum = 0** – Initializes the checksum variable.

  2. **Add Sequence Number:**

  - **checksum += seq_number;** – Includes the sequence number in the checksum.

  3. **Add Response Opcode (2 bytes):**

  - **checksum += (resp_opcode & 0xFF);** – Low byte of resp_opcode.
  - **checksum += ((resp_opcode >> 8) & 0xFF);** – High byte of resp_opcode.
  - ***length = 2;** – Initializes the length to account for the 2-byte opcode.

  4. **Process Parameter Array:**

  - Iterates over params_array.
  - ***length += param->length;** – Adds the parameter length to the total length.
  - **checksum += param->data[i];** – Adds each byte in param->data to the checksum.

  5. **Add Packet Length (2 bytes):**

  - **checksum += (*length & 0xFF);** – Low byte of total length.
  - **checksum += ((*length >> 8) & 0xFF);** – High byte of total length.

  6. **Apply 1's Complement:**

  - **return (0xff - checksum + 1);** – Returns the negative checksum in 1's complement form.

---

2. - **Function**

```
static uint8_t get_expected_tx_seq_number(void);
```

- **Description:**

  This function generates the next expected transmission sequence number in a cyclic manner, ensuring it stays within the range of 1 to 255.

- **Parameters:**

  NA

- **Return:**
  - tx_exp_seq – Return the next sequence number, ranging from **1 to 255** in **uint8_t** format.

- **Key Call:**

  1. **Increment and Cycle Sequence Number:**

- **tx_exp_seq = (tx_exp_seq % 255) + 1;** – Ensures the sequence number cycles from 1 to 255.

2. **Return Updated Sequence Number:**

- **return tx_exp_seq;** – Returns the newly computed sequence number.

---

3. - **Function**

```
static uint8_t get_expected_rx_seq_number(void);
```

- **Description:**

  This function generates the next expected reception sequence number in a cyclic manner, ensuring it stays within the range of 1 to 255.

- **Parameters:**

  NA

- **Return:**

  - rx_exp_seq – Return the next sequence number, ranging from **1 to 255** in **uint8_t** format.

- **Key Call:**

  1. **Increment and Cycle Sequence Number:**

  - **rx_exp_seq = (rx_exp_seq % 255) + 1;** – Ensures the sequence number cycles from 1 to 255.

  2. **Return Updated Sequence Number:**

  - **return rx_exp_seq;** – Returns the newly computed sequence number.

---

4. - **Function**

```
static void free_cmd_item(void *item);
```

- **Description:**

  This function frees the structure **(uartcmdpacket_t)** and frees data_buf, if it was allocated **(not NULL)**.

- **Parameters:**

  - item – A pointer to a uartcmdpacket_t structure.

- **Return:**

  NA

- **Key Call:**

  1. **free(cmd_pakcet->uart_pkt.data_buf):** – Frees the dynamically allocated data_buf.
  2. **free(cmd_pakcet):** – Frees the command packet structure.

---

5. - **Function**

```
static void free_pkt_item(void *item);
```

- **Description:**

  This function frees the structure **(uartpacket_t)** and frees data_buf, if it was allocated **(not NULL)**.

- **Parameters:**
  - item – A pointer to a uartpacket_t structure.
- **Return:**

  NA
- **Key Call:**
  1. **(pakcet->data_buf):** – Frees the dynamically allocated data_buf.
  2. **free(pakcet):** – Frees the packet structure.

---

6. - **Function**

```
static void free_ack_item(void *item);
```

- **Description:**

  This function frees the given item.
- **Parameters:**
  - item – A pointer to an acknowledgment packet.
- **Return:**

  NA
- **Key Call:**
  1. **free(item):** Frees the given item.

---

7. - **Function**

```
static void *init_cmd_item(void);
```

- **Description:**

  This function allocates and initializes a uartcmdpacket_t structure.
- **Parameters:**

  NA
- **Return:**
  - uartcmdpacket_t – Returns a pointer to the allocated structure.
  - NULL – If memory allocation fails.
- **Key Call:**
  1. **malloc(sizeof(uartcmdpacket_t)):** – Allocates memory for the command packet.
  2. **memset(cmd_pakcet, 0, sizeof(uartcmdpacket_t)):** – Initializes the allocated memory to 0.

---

8. - **Function**

```
static void *init_pkt_item(void);
```

- **Description:**

  This function allocates and initializes a uartpacket_t structure.

- **Parameters:**

  NA

- **Return:**
  - uartpacket_t – A pointer to the allocated structure.
  - NULL – If memory allocation fails.

- **Key Call:**
  1. **malloc(sizeof(uartpacket_t)):** – Allocates memory for the UART packet.
  2. **memset(ack_pakcet, 0, sizeof(uartpacket_t)):** – Initializes the allocated memory to 0.

---

9. - **Function**

```
static void *init_ack_item(void);
```

- **Description:**

  This function allocates and initializes a uartackpacket_t structure.

- **Parameters:**

  NA

- **Return:**
  - uartackpacket_t – A pointer to the allocated structure.
  - NULL – If memory allocation fails.

- **Key Call:**
  1. **malloc(sizeof(uartackpacket_t)):** – Allocates memory for the acknowledgment packet.
  2. **memset(ack_pakcet, 0, sizeof(uartackpacket_t)):** – Initializes the allocated memory to 0.

---

10. - **Function**

```
static void delete_queue_if_exists(QueueHandle_t queue, FreeItemFn free_item_
↪fn);
```

- **Description:**

  This function safely deletes a queue if it exists. It first removes and frees all items in the queue using the provided free_item_fn function, then deletes the queue itself.

- **Parameters:**
  - queue – The queue handle to be deleted. If queue is NULL, the function does nothing.
  - free_item_fn – A function pointer that frees each item removed from the queue. If free_item_fn is NULL, the items are not explicitly freed.

- **Return:**

  NA

- **Key Call:**
  1. **xQueueReceive(queue, &item, 0)** – Attempts to receive an item from queue and returns pdPASS if successful.
  2. **free_item_fn(item)** – Calls the provided function (free_item_fn) to free the item **(if free_item_fn is not NULL)**.
  3. **vQueueDelete(queue)** – Deletes the queue after all items have been removed.

---

11. - **Function**

```
static void free_rx_uart_queue(void);
```

- **Description:**

  This function frees and deletes multiple reception UART-related queues by dequeuing and freeing their allocated items before deleting the queue. It ensures that all pending reception UART command packets and acknowledgment packets are properly freed to prevent memory leaks.

- **Parameters:**

  NA

- **Return:**

  NA

- **Key Call:**
  1. **DeleteQueueIfExists(rx_uart_recycle, FreeCmdItem);** – Frees and deletes the queue for recycled reception UART command packets.
  2. **DeleteQueueIfExists(rx_uart_ready, FreeCmdItem);** – Frees and deletes the queue for ready-to-process reception UART command packets.
  3. **DeleteQueueIfExists(rx_critical_recycle, FreeCmdItem);** – Frees and deletes the queue for critical recycled reception UART command packets.
  4. **DeleteQueueIfExists(rx_critical_ready, FreeCmdItem);** – Frees and deletes the queue for critical ready reception UART command packets.
  5. **DeleteQueueIfExists(rx_uart_ack_recycle, FreeAckItem);** – Frees and deletes the queue for recycled reception UART acknowledgment packets.
  6. **DeleteQueueIfExists(rx_uart_ack_tmp_recycle, FreeAckItem);** – Frees and deletes the queue for temporary recycled reception UART acknowledgment packets.
  7. **DeleteQueueIfExists(rx_uart_ack_ready, FreeAckItem);** – Frees and deletes the queue for ready-to-process reception UART acknowledgment packets.

---

12. - **Function**

```
static void free_tx_uart_queue(void);
```

- **Description:**

  This function frees and deletes multiple transmission UART-related queues by dequeuing and freeing their allocated items before deleting the queue. It ensures that all pending transmission UART command packets and acknowledgment packets are properly freed to prevent memory leaks.

- **Parameters:**

  NA

- **Return:**

  NA

- **Key Call:**

  1. **DeleteQueueIfExists(tx_uart_recycle, FreeCmdItem);** – Frees and deletes the queue for recycled transmission UART command packets.
  2. **DeleteQueueIfExists(tx_uart_ready, FreeCmdItem);** – Frees and deletes the queue for ready-to-process transmission UART command packets.
  3. **DeleteQueueIfExists(tx_critical_recycle, FreeCmdItem);** – Frees and deletes the queue for critical recycled transmission UART command packets.
  4. **DeleteQueueIfExists(tx_critical_ready, FreeCmdItem);** – Frees and deletes the queue for critical ready transmission UART command packets.
  5. **DeleteQueueIfExists(tx_uart_ack_recycle, FreeAckItem);** – Frees and deletes the queue for recycled transmission UART acknowledgment packets.
  6. **DeleteQueueIfExists(tx_uart_ack_ready, FreeAckItem);** – Frees and deletes the queue for ready-to-process transmission UART acknowledgment packets.

---

13. - **Function**

```
static QueueHandle_t create_queue_and_fill(int length, size_t item_size, void␣
↪**item_array, InitItemFn init_item_fn, FreeItemFn free_item_fn);
```

- **Description:**

  This function creates a FreeRTOS queue and fills it with preallocated items using an initialization function. If any item fails to initialize, it cleans up and deletes the queue to prevent memory leaks.

- **Parameters:**

  - length – The number of items the queue can hold.
  - item_size – The size of each item in the queue.
  - item_array – An array to store the initialized queue items.
  - init_item_fn – A function pointer to initialize each item before adding it to the queue.
  - free_item_fn – A function pointer to free allocated items in case of failure.

- **Return:**

  - QueueHandle_t – A handle to the created queue if successful.
  - NULL – If queue creation or item initialization fails.

- **Key Call:**

  1. **xQueueCreate(length, item_size);** – Creates a FreeRTOS queue.
  2. **init_item_fn();** – Allocates and initializes each item..
  3. **xQueueSend(queue, &item_array[i], 0);** – Adds initialized items to the queue.
  4. **Cleanup on Failure:**

- **free_item_fn(item_array[j]);** – Frees previously allocated items if an error occurs.
- **vQueueDelete(queue);** – Deletes the queue if initialization fails.

---

14. - **Function**

```
static int create_rx_uart_queue(int queue_length, int critical_queue_length,␣
↪int ack_queue_length);
```

- **Description:**

  This function creates and initializes multiple receive (RX) UART queues for command and acknowledgment packets.

- **Parameters:**
  - queue_length – Number of elements in the primary command queue.
  - critical_queue_length – Number of elements in the critical command queue
  - ack_queue_length – Number of elements in the acknowledgment queue.

- **Return:**
  - UART_OK – If multiple queues creation is success.
  - UART_QUEUE_CREATE_FAIL – If failure in creating one or more queues.

- **Key Call:**
  1. **CreateQueueAndFill()** – Creates a queue, initializes elements, and pre-fills it.
  2. **xQueueCreate()** – Creates a FreeRTOS queue.
  3. **CreateQueueAndFill(critical_queue_length, sizeof(uartcmdpacket_t *), (void **)cmd_item, InitCmdItem, FreeCmdItem);** – Adds initialized items to the queue.
  4. **FreeRxUartQueue()** – Cleans up resources if queue creation fails.

---

15. - **Function**

```
static int create_tx_uart_queue(int queue_length, int critical_queue_length,␣
↪int ack_queue_length);
```

- **Description:**

  This function creates and initializes transmit (TX) UART acknowledgment queues.

  Caution

  Currently unused of queue_length and critical_queue_length.

- **Parameters:**
  - queue_length – **(Unused in this implementation)**.
  - critical_queue_length – **(Unused in this implementation)**.
  - ack_queue_length – Number of elements in the acknowledgment queue.

- **Return:**
  - UART_OK – If acknowledgment queue creation is success.
  - UART_QUEUE_CREATE_FAIL – If failure in creating acknowledgment queue.
- **Key Call:**
  1. **CreateQueueAndFill()** – Creates a queue, initializes elements, and pre-fills it.
  2. **xQueueCreate()** – Creates a FreeRTOS queue.
  3. **CreateQueueAndFill(critical_queue_length, sizeof(uartcmdpacket_t *), (void **)cmd_item, InitCmdItem, FreeCmdItem);** – Adds initialized items to the queue.
  4. **FreeRxUartQueue()** – Cleans up resources if queue creation fails.

---

16. - **Function**

```
static int uart_send_ack(uint8_t recv_seq_number, uint16_t recv_opcode, uint8_
↪t status);
```

- **Description:**

  This function sends an acknowledgment (ACK) packet in response to a received UART command.

> **Caution**
>
> Currently unused of recv_seq_number.

- **Parameters:**
  - recv_seq_number – (**Unused in this implementation**).
  - recv_opcode – The opcode of the received command.
  - status – The status to be included in the acknowledgment packet.
- **Return:**
  - UART_OK – If acknowledgment queue creation is success.
  - UART_SEND_PACKET_FAIL – If failure to send ACK packet.
  - UART_ACK_PACKET_UNAVAILABLE – No available ACK packet in the tx_uart_ack_recycle queue.
- **Key Call:**
  1. **if (!IS_VALID_UART_CMD(recv_opcode))** – Ensures that recv_opcode is a valid UART command.
  2. **xQueueReceive(tx_uart_ack_recycle, (void *)&ack_packet, 0);** – Attempts to retrieve a pre-allocated acknowledgment packet from tx_uart_ack_recycle.
  3. **Initialize and populate ack_packet**
  4. **if (xQueueSend(tx_uart_ack_ready, (void *)&ack_packet, 0) != pdPASS)** – Pushes the prepared ACK packet into the tx_uart_ack_ready queue.

---

17. - **Function**

```
static void uart_service_receive_handler(uint8_t received_byte);
```

- **Description:**

  This function handles incoming UART data, constructs valid UART command packets, verifies checksums, and manages packet queuing for further processing. If an acknowledgment (ACK) packet is required, it prepares and queues the ACK response.

- **Parameters:**
  - received_byte – The incoming byte received via UART.

- **Return:**

  NA

- **Key Call:**
  1. **if (received_byte == UART_SYNC_WORD)** – Checks if the received byte matches the sync word.

---

18. - **Function**

```
static void process_uart_recv_thread(void *params);
```

- **Description:**

  This function continuously reads data from a UART stream buffer and processes each received byte using the UART_ReceiveHandler function.

- **Parameters:**
  - params – **(Unused in this implementation)**.

- **Return:**

  NA

- **Key Call:**
  1. **xStreamBufferReceive(uart_rx_stream, data, UART_MAX_BUF_SIZE, portMAX_DELAY);** – Reads incoming UART data from the stream buffer.
  2. **UART_ReceiveHandler(data[i]);** – Passing each received byte to the UART_ReceiveHandler for processing.

---

19. - **Function**

```
static void uart_service_irq(uint32_t id, SerialIrq event);
```

- **Description:**

  This function is the UART interrupt service routine (ISR) that handles UART receive (Rx) and transmit (Tx) interrupts request. It reads incoming data into a buffer and sends it to a stream buffer for further processing.

- **Parameters:**
  - id – Pointer to the serial_t structure.
  - event – The type of UART interrupt event **(RxIrq for receiving, TxIrq for transmitting)**.

- **Return:**

  NA

- **Key Call:**

  1. **serial_readable(sobj);** – Checks if there is available data in the UART receive buffer.
  2. **serial_getc(sobj);** – Reads a byte of data from the UART receive buffer.
  3. **xStreamBufferSendFromISR(uart_rx_stream, rx_uart_tmp_buf, xBytesGet, NULL);** – Sends received data from the temporary buffer to the stream buffer for processing.

---

20. - **Function**

```
static void process_uart_cmd_thread(void *params);
```

- **Description:**

  This function processes incoming UART command packets by retrieving them from the queue, executing the appropriate callback function, and recycling the packet.

- **Parameters:**

  – params – (**Unused in this implementation**).

- **Return:**

  NA

- **Key Call:**

  1. **xQueueReceive(rx_uart_ready, (void *)&recv_pkt, portMAX_DELAY)** – Retrieves a command packet from the queue.
  2. **uartcmdcb_table[recv_pkt->uart_pkt.opcode - UART_OPC_CMD_MIN].callback(recv_pkt)** – Calls the registered callback function from the received command.
  3. **free(recv_pkt->uart_pkt.data_buf)** – Frees allocated data buffer.
  4. **xQueueSend(rx_uart_recycle, (void *)&recv_pkt, 0)** – Recycles the processed packet by adding it back to the queue.

---

21. - **Function**

```
static void process_uart_critical_thread(void *params);
```

- **Description:**

  This function processes incoming critical UART command packets such as system shutdown, stopping streams, or stopping data transport.

- **Parameters:**

  – params – (**Unused in this implementation**).

- **Return:**

  NA

- **Key Call:**

1. **(xQueueReceive(rx_critical_ready, (void *)&recv_pkt, portMAX_DELAY)** – Retrieves a critical command packet from the queue.
2. **uartcmdcb_table[recv_pkt->uart_pkt.opcode - UART_OPC_CMD_MIN].callback(recv_pkt)** – Calls the registered callback function from the received critical command.
3. **free(recv_pkt->uart_pkt.data_buf)** – Frees allocated data buffer.
4. **xQueueSend(rx_uart_recycle, (void *)&recv_pkt, 0)** – Recycles the processed packet by adding it back to the queue.

---

22. - **Function**

```
static void process_uart_ack_thread(void *params);
```

- **Description:**

  This function processes incoming acknowledgment (ACK) packets when a command packet is received and sends an ACK response back through UART.

- **Parameters:**

  – params – **(Unused in this implementation)**.

- **Return:**

  NA

- **Key Call:**

  1. **xQueueReceive(tx_uart_ack_ready, (void *)&ack_packet, portMAX_DELAY)** – Retrieves a ACK packet from the queue.
  2. **uart_send_packet(UART_OPC_RESP_ACK, &opcode_params, 10000)** – Sends an ACK response over UART.
  3. **xQueueSend(rx_uart_recycle, (void *)&recv_pkt, 0)** – Recycles the ACK packet by adding it back to the queue.

---

23. - **Function**

```
static void uart_send_str_done(uint32_t id);
```

- **Description:**

  This function is called when a UART transmission is completed. It releases a semaphore to signal that the UART TX buffer is free and ready for the next transmission.

- **Parameters:**

  – id – The identifier of the UART instance that triggered the function.

- **Return:**

  NA

- **Key Call:**

  1. **xSemaphoreGiveFromISR(uart_tx_free_sema, &xHigherPriorityTaskWoken);** – Releases the semaphore from an ISR (Interrupt Service Routine) to indicate that UART TX is available.

---

24. - **Function**

```
static int uart_send_buffer(serial_t *sobj, uint8_t *pstr, uint16_t length,␣
↪uint8_t endbuf);
```

- **Description:**

  This function handlers UART transmission using Direct Memory Access (DMA). Additionally, ensures efficient data sending by managing buffer space, utilizing a semaphore for synchronization, and sending data in chunks when necessary.

- **Parameters:**

  – sobj – Pointer to the UART serial structure.

  – pstr – Pointer to the buffer containing data.

  – length – Number of bytes to send.

  – endbuf – If set, forces transmission of any remaining buffered data.

- **Return:**

  UART_OK – Data was successfully sent.

  UART_SEND_PACKET_FAIL – Transmission failed due to errors like semaphore acquisition failure or DMA issues.

- **Key Call:**

  1. **xSemaphoreTake(uart_tx_free_sema, portMAX_DELAY)** – Gets semaphore to ensure exclusive UART access.
  2. **memcpy(uart_txbuf + send_idx, pstr + input_idx, remaining_space);** – Copies data into UART buffer.
  3. **serial_send_stream_dma(sobj, (char *)uart_txbuf, tmp_uart_pic_size);** – Send the buffer over UART using DMA.
  4. **xSemaphoreGive(uart_tx_free_sema);** – Releases the semaphore if an error occurs or when transmission completes.

3.1.7 Configuration in AI Wearable Glass Scenario

This section outlines key configuration parameters and their default values across various files.

1. **main.c**

- **UART_LOG_BAUDRATE:** UART baud rate for the debug log (default: 3000000). Note: This baud rate significantly influences processing time.

2. **ai_glass_media.h**

- **FLASH_AI_SNAPSHOT_DATA:** Flash location to store AI snapshot parameters.
- **FLASH_RECORD_DATA:** Flash location to store lifetime recording parameters.
- **FLASH_SNAPSHOT_DATA:** Flash location to store lifetime snapshot parameters.
- **DEFAULT_RECORD_WIDTH:** Default width for sensor SC5356 is 2560 px. Users may edit to 1920 px if wanted.

- **DEFAULT_RECORD_HEIGHT:** Default Height for sensor SC5356 is 1440 px. Users may edit to 1080 px if wanted.
- **DEFAULT_LIFESNAP_WIDTH:** Default width for sensor SC5356 is 12M. Users may edit to 5M or 8M if wanted.
- **DEFAULT_LIFESNAP_HEIGHT:** Default Height for sensor SC5356 is 12M. Users may edit to 5M or 8M if wanted.

3. **ai_glass_initialize.c**

- **ENABLE_TEST_CMD:** Enables test command for the scenario (default: 1).
- **ENABLE_DISK_MASS_STORAGE:** Allows the tester to access the EMMC disk via USB mass storage (default: 0).
- **ENABLE_VIDEO_SEND_LATER:** Defers sending the video end command; requires both ENABLE_TEST_CMD and ENABLE_DISK_MASS_STORAGE (default: 0). When enabled, the process blocks after recording or a lifetime snapshot. Users need to issue the "SENDVIDEOEND" command.
- **DISK_PLATFORM:** External platform to store lifetime snapshots or recordings (default: VFS_INF_EMMC).
- **UART_TX:** UART TX pin for communication with another SoC (default: PA_2).
- **UART_RX:** UART RX pin for communication with another SoC (default: PA_3).
- **UART_BAUDRATE:** UART baud rate for communication with another SoC (default: 2000000). Note: This baud rate significantly influences processing time.

4. **lifetime_recording_initialize.c**

- **ENABLE_GET_GSENSOR_INFO:** Activates gyro sensor data capture during lifetime recording (default: 1).
- **AUDIO_SAMPLE_RATE:** Audio sample rate (default: 16000).
- **AUDIO_SRC:** Audio interface during recording (default: I2S_INTERFACE).
- **AUDIO_I2S_ROLE:** I2S role when using the I2S interface (default: I2S_MASTER).

5. **wlan_scenario.h**

- **AI_GLASS_AP_IP_ADDRx:** IP address in AP mode for 8735 (default: 192.168.43.1).
- **AI_GLASS_AP_NETMASK_ADDRx:** Netmask for AP mode (default: 255.255.255.0).
- **AI_GLASS_AP_GW_ADDR:** Gateway address in AP mode (default: 192.168.43.1).
- **AI_GLASS_AP_SSID:** SSID for AP mode (default: AI_GLASS_AP).
- **AI_GLASS_AP_PASSWORD:** Password for AP mode (default: 12345678).
- **AI_GLASS_AP_CHANNEL:** Channel for AP mode (default: 6).
- **AI_GLASS_AP_PORT:** Port number used for AP mode (default: 8080).

6. **uart_service.c**

- **UART_CMD_PRIORITY:** Priority for normal UART commands from another SoC (default: 5).
- **UART_CRITICAL_PRIORITY:** Priority for critical UART commands (default: 7).
- **UART_ACK_PRIORITY:** Priority for ACK processing threads (default: 8).
- **UART_THREAD_NUM:** Number of threads for normal UART command processing (default: 3).

7. **uart_service.h**

- **SEND_DATA_SHOW:** Display data sent to another SoC via UART (default: 0).

- **RECEIVE_ACK_SHOW:** Display received ACK data (default: 0).
- **SEND_ACK_SHOW:** Display sent ACK data (default: 0).
- **MAX_UART_QUEUE_SIZE:** Queue length for normal UART commands (default: 10).
- **MAX_CRITICAL_QUEUE_SIZE:** Queue length for critical UART commands (default: 10).
- **MAX_UARTACK_QUEUE_SIZE:** Queue length for ACK data (default: 20).

8. **gyrosensorgyrosensor_api.h**

- **GYROSENSOR_I2C_MTR_SDA:** Data pin for the gyro sensor I2C interface (default: PF_2).
- **GYROSENSOR_I2C_MTR_SCL:** Clock pin for the gyro sensor I2C interface (default: PF_1).

9. **media_filesystem.c**

- **ENABLE_FILE_TIME_FUNCTION:** Updates file timestamps using the get_fattime function in media_filesystem.c.

3.1.8 Code Flow Overview

This section explains the functional flow of the AI Wearable Glass firmware, starting from main.c and tracing the sequence of function calls across different modules.

Initialization Sequence

1. **Entry Point - main.c**

- **Function:**

```
void main(void)
```

- **Description:**

  This is the entry point of the firmware. The main function initializes the system and starts the AI Glass application.

- **Key Call:**

  The main function calls ai_glass_init() to initialize core features of the AI Wearable Glass.

2. **Initialization - ai_glass_initialize.c**

- **Function:**

```
void ai_glass_init(void)
```

- **Description:**

  The ai_glass_init() function is responsible for initializing various services needed for the AI Wearable Glass to operate. It creates a task for handling UART communication with BT SoC.

- **Key Call:**

  1. The ai_glass_init() function calls xTaskCreate() to create the ai_glass_service_thread task.
  2. The ai_glass_service_thread handles the core UART communication process for sending and receiving data between the AI Wearable Glass and external systems.

3. **ai_glass_service_thread - ai_glass_initialize.c**

- **Function:**

```
void ai_glass_service_thread(void *param)
```

- **Description:**

  The ai_glass_service_thread function is responsible for initializing various services required by the AI Wearable Glass system. For example, setting up media parameters, filesystems, UART communication and UART service function for the operation of system.

- **Key Call:**

  1. uart_fun_regist() and uart_service_start(1) are called to register and start the UART service for communication between the 8735 and BT SoC.
  2. The AI Wearable Glass system will enter a state where it waits for commands from the BT SoC via UART.

4. **uart_fun_regist - ai_glass_initialize.c**

- **Function:**

```
void uart_fun_regist(void)
```

- **Description:**

  The uart_fun_regist function is responsible for registers a series of UART operation commands. Each registered command corresponds to a specific UART operation, and each operation is associated with a handler function that will be executed when the corresponding command is received over UART.

- **Key Call:**

  1. The AI Wearable Glass system will enter a state where it waits for commands from the BT SoC via UART.
  2. Each of the commands corresponds to a specific action, and the handler functions. Here are some of the registered commands likely represent:

  – **UART_OPC_CMD_SNAPSHOT:** Command to trigger a snapshot. The handler would be uart_service_snapshot.

  – **UART_OPC_CMD_RECORD_START:** Command to start video recording. The handler would be uart_service_record_start.

5. **uart_service_snapshot - ai_glass_initialize.c**

- **Function:**

```
static void uart_service_snapshot(uartcmdpacket_t *param)
```

- **Description:**

  The uart_service_snapshot function was executed when the corresponding UART command "UART_OPC_CMD_SNAPSHOT" was received from the BT SoC.

- **Key Call:**

  1. The uart_service_snapshot function calls either (ai_snapshot_initialize() and ai_snapshot_deinitialize) or (lifetime_snapshot_initialize() and lifetime_snapshot_deinitialize) depending on the mode parameter of the UART packet's data_buf.

  – If the data buffer in the UART packet indicate mode "1", it will call ai_snapshot_initialize() and ai_snapshot_deinitialize.

- If the data buffer in the UART packet indicate mode "0", it will call lifetime_snapshot_initialize() and lifetime_snapshot_deinitialize.
- These functions ai_snapshot_initialize() or lifetime_snapshot_initialize() are responsible for setting up and initiating the snapshot process on the AI Wearable Glass, which involves configuring the camera or handling the snapshot image capture.
- These functions ai_snapshot_deinitialize() or lifetime_snapshot_deinitialize() are responsible for deinitializing the standard snapshot process, like stopping the camera or releasing resources used specifically for capturing a single snapshot.

6. **uart_service_record_start - ai_glass_initialize.c**

- **Function:**

```
static void uart_service_record_start(uartcmdpacket_t *param)
```

- **Description:**

  The uart_service_record_start function was executed when the corresponding UART command "UART_OPC_CMD_RECORD_START" was received from the BT SoC.

- **Key Call:**

  1. The uart_service_record_start function calls lifetime_recording_initialize() and lifetime_recording_deinitialize().

  - There functions lifetime_recording_initialize() are responsible for setting up and initiating the recording process on the AI Wearable Glass, which involves configuring the camera or handling the video recording capture.
  - These functions lifetime_recording_deinitialize(). are responsible for deinitializing the recording, like stopping the camera or releasing resources used specifically for a single video recording.

7. **lifetime_snapshot_initialize - lifetime_snapshot_initialize.c**

- **Function:**

```
int lifetime_snapshot_initialize(void)
```

- **Description:**

  The lifetime_snapshot_initialize function are responsible on the full process of snapshot such as checking picture size, scale up if needed, setting video parameters, jpeg encode, and storing to EMMC storage device.

- **Key Call:**

  1. Check if the desired width and height are larger than the default sensor's width and height, variable **"ls_video_params.need_scaleup"** is set to **1**; otherwise, it is set to **0**.
  2. If **"ls_video_params.need_scaleup"** is 1, scaling up will be performed in the **lifetime_snapshot_file_save** function.
  3. If **"ls_video_params.need_scaleup"** is 0, no scaling up will be performed in the **lifetime_snapshot_file_save** function.
  4. Setting up lifetime snapshot parameters **"ls_video_params"**.
  5. After completing the scale-up process, it will continue with the JPEG encoding process.
  6. Finally, after encoding, the file will be saved to the EMMC storage device.

7. **lifetime_recording_initialize - lifetime_recording_initialize.c**

- **Function:**

```
void lifetime_recording_initialize(void)
```

- **Description:**

  The lifetime_recording_initialize function are responsible on the full process of recording such as setting video and audio parameters, capturing video and audio data, video and audio encode, and storing to EMMC storage device.

- **Key Call:**

  1. Setting up audio parameters **"ir_audio_ctx"** and mp4 parameters **"lr_mp4_params"**.
  2. Capturing raw audio data from audio input and proceed on AAC (Advanced Audio Codec) on compressing raw audio data using the pre-set parameters in aac_params structure.
  3. Proceed on video capturing and encode.
  4. After completing both encoding, it will combined (multiplexed) into a single file MP4.
  5. Finally, the file will be saved to the EMMC storage device.

8. **ai_snapshot_initialize - ai_snapshot_initialize.c**

- **Function:**

```
int ai_snapshot_initialize(void)
```

- **Description:**

  The ai_snapshot_initialize function are responsible on the full process of AI snapshot such as setting snapshot parameters, jpeg encode, and storing to RAM storage device.

- **Key Call:**

  1. Setting up AI snapshot parameters **"snapshot_param"**.
  2. Proceed on JPEG encode.
  3. Finally, after encoding, the file will saved to the RAM storage device.