Application Note

UART Log Debug System on Jacinto 7 SoC

Kangjia Dong and Kevin Peng

ABSTRACT

TI latest automotive processor Jacinto 7 series covers the automotive applications in ADAS and Gateway under different scenarios. It contains both TDA4X and DRA82X series which are mainly used for ADAS and Gateway respectively. These processors are based on multi-core heterogeneous architecture, consisting of Cortex®-R5, Cortex-A72, Cortex-M3/M4, DSP and some common peripherals. There is a high reusability within these processors. Normally, this series processors need to run an operating system and the related application threads on each core, involving the internal data transmission and peripheral calls. Hence, there is a pretty high complexity for the applications of Jacinto™ 7 series processors. Sometimes when a problem occurs, it is not that easy to debug.

This application note demonstrates the basic information on the hardware and software levels of the universal asynchronous receiver/transmitter (UART) logging system in the reference design provided by TI. It includes the way to customize customers' own log output serial port based on the default SDK and reference design, and the method of debugging multi-core heterogeneous system-on-chips (SoCs) by printing the UART logs to solve the problems.

Table of Contents

1 UART Introduction......2
1.1 Jacinto 7 UART Overview......2
1.2 Jacinto 7 UART Features......2
1.3 Jacinto 7 UART Functional Introduction......3
2 UART Usage Overview......4
2.1 WKUP_UART0 Usage......4
2.2 MCU_UART0 Usage......6
2.3 MAIN_UARTx Usage......6
3 Log Level Design on Software Module......9
4 Change UART Instance......11
4.1 Change MAIN_UARTx for MAIN Domain......11
4.2 Set Standalone UART Port for DSP/MCU......15
5 Summary......18
6 References......18

Trademarks

1 UART Introduction

UART is a common peripheral in processors and is usually used for system log information output, low-cost human-computer interaction, communication between devices, and so forth. Particularly common in vehicle communications, the Lin bus often uses UART as a low-cost serial communication protocol. UART does not need both clock synchronization and master-slave settings, only the configuration of the start bit and stop bit is required, since it is an asynchronous communication, and it can send and receive data at any time. TX and RX are sufficient for the external hardware connections when flow control and level conversion are not required. However, in common debugging processes, log information often needs to be output to the computer. Therefore, in hardware design, a USB serial port chip is required to convert the TTL level into a USB serial port protocol and output it to the computer port. Software tools on PC need properly set the serial port parameters applied in the processor driver, including baud rate, start bit, data bit, parity bit, stop bit, and so on, then the log information can be received for debugging.

1.1 Jacinto 7 UART Overview

Jacinto 7 series processors all have the same UART IP, so the functions and usage methods of UART on different processors in this series are basically the same. Table 1-1 shows that each processor has totally 11 UART interfaces, one of which is in the WKUP domain, one is in the MCU domain, and the remaining nine are in the MAIN domain. After all the domains are powered on normally, each core can access any of these UART through software. However, in the system software architecture, multiple cores should not access a UART at the same time. This may lead to some system conflicts, causing a certain core to hang.

Table 1-1. UART Allocation Across Device Domains

Instance	Domain		
	WKUP	MCU	MAIN
WKUP_UART0	√	-	-
MCU_UART0	-	√	-
UART0	-	-	√
UART1	-	-	√
UART2	-	-	√
UART3	-	-	√
UART4	-	-	√
UART5	-	-	√
UART6	-	-	√
UART7	-	-	√
UART8	-	-	√
UART9	-	-	√

1.2 Jacinto 7 UART Features

Jacinto 7 UART includes the following features:

- 16C750-compatible
- RS-485 external transceiver auto flow control support
- 64-byte FIFO buffer for receiver and 64-byte FIFO buffer for transmitter
- Programmable interrupt trigger levels for FIFOs
- Programmable sleep mode
- The 48 MHz functional clock is default option and allows baud rates up to 3.6 Mbps
- Auto-baud between 1200 bits/s and 115.2 Kbits/s (only when 48 MHz function clock is used)
- Optional multi-drop transmission
- Configurable time-guard feature
- Configurable data format:
  - Data bit: 5, 6, 7, 8, or 9 bits
  - Parity bit: Even, odd, none
  - Stop-bit: 1, 1.5, 2 bit(s)

- Flow control: Hardware (RTS/CTS) or software (XON/XOFF)
- False start bit detection
- Line break generation and detection
- Fully prioritized interrupt system controls
- Internal test and loopback capabilities
- Modem control functions (CTS, RTS)
- Module instance has extended modem control signals (DCD, RI, DTR, DSR)

1.3 Jacinto 7 UART Functional Introduction

Figure 1-1 shows the Jacinto 7 UART functional block diagram. When the processor needs to send data, it only needs to write the output data into the FIFO through CPU/DMA. Then, the data is automatically transmitted to TX pin by UART_THR register and converted into TTL level. The data transmission is completed if the UART_THR register is empty. During the data receiving process, the TTL level is firstly converted into bit data through the pin, and is written to the FIFO by UART_RHR. When the FIFO reaches a threshold (maximum 64 Bytes), a CPU/DMA interrupt will be triggered to write the data to the memory. When enough data is read and the data in the FIFO is lower than the threshold, the interrupt condition disappears and the data receiving is finished.

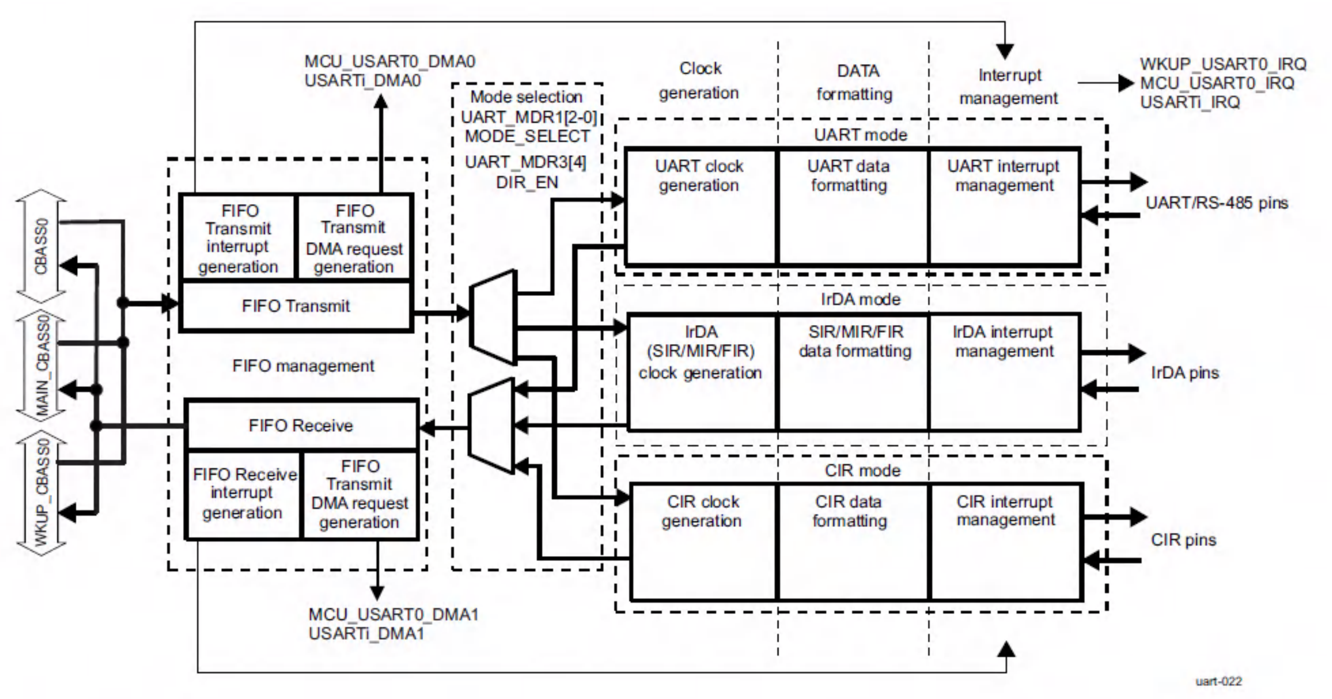


Figure 1-1. UART Functional Block Diagram

2 UART Usage Overview

The EVM boards of Jacinto 7 series processors all have multiple serial ports, which can be seen from Table 2-1. Normally, during hardware design, at least three serial ports are reserved for log information debugging, and the remaining serial ports can be used for communication with external devices. By default, the software parameters of all serial ports are the same, the baud rate is 115200 bit/s, the start bit is 0, there are 8 data bits, the parity bit is None, and the stop bit is 1.

WKUP_UART0 is reserved for DSMC debugging, which is common when the system accidentally triggers the firewall. For SBL boot, MCU_UART0 is used for the serial port output of MCU1_0. For UART boot, MCU_UART0 is used to print the "C" character to determine whether the processor is working normally and to debug whether the HS key burning is successful. The A72 cores log in DRA821 is printed to MAIN_UARTx. The log of all the cores in TDA4X is printed into MAIN_UARTx, except the MCU1_0 log is printed in to MCU_UART0 in the SBL boot.

Therefore, when designing the board, at least WKUP_UART0, MCU_UART0 and one MAIN_UARTx should be reserved. Moreover, they are suggested to have the same Pin configuration with TI reference design. Especially for MCU_UART0, if the Pin is changed then it cannot use “C” character printing to debug in the early stages of project development, because this feature is implemented in the ROM code with the default Pin setting.

When connecting to external devices, it is required to connect the flow control. It is always recommended to connect hardware flow control lines for UART communication. Also, software should enable hardware flow control explicitly. Otherwise, there will be data loss and data corruption.

CAUTION

The reference design provided by TI uses USB to UART for serial printing. When connecting on Windows, it is required to install an additional driver.

Table 2-1. UART Pinout on the Reference Design

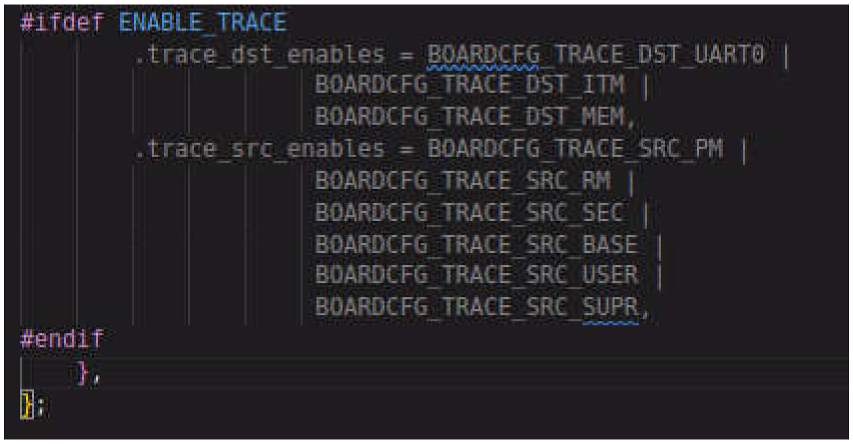
Instance	Device			
	DRA821	**TDA4VM**	**TDA4VL/Eco/AL**	**TDA4VH**
WKUP_UART0	√	√	√	√
MCU_UART0	√	√	√	√
UART0	√	√	-	-
UART1	√	√	-	-
UART2	-	√	√	√
UART3	√	-	-	√
UART4	-	√	-	-
UART5	-	-	√	√
UART6	-	-	-	-
UART7	-	-	-	-
UART8	-	-	√	√
UART9	-	-	-	-

2.1 WKUP_UART0 Usage

Jacinto 7 series processors use WKUP_UART0 to print DMSC (Device Management and Security Control) log, this log can be used to check if there is any error in the firewall or SYSFW (System Firmware). By default, the WKUP_UART0 log output is not sufficient, some additional steps are needed to obtain the full log. The steps are provided below in details.

1. For SPL Boot:
   a. Enable the ENABLE_TRACE macro in the ti-processor-sdk-linux-xxxx-evm-0x_0x_00_xx/board-support/k3-image-gen-xxxxxxxx/soc/j7xxxx/evm/board-cfg.c
   b. Recompile board configure under Linux SDK home directory, $make sysfw-image
   c. Recompile tiboot3.bin, $make u-boot

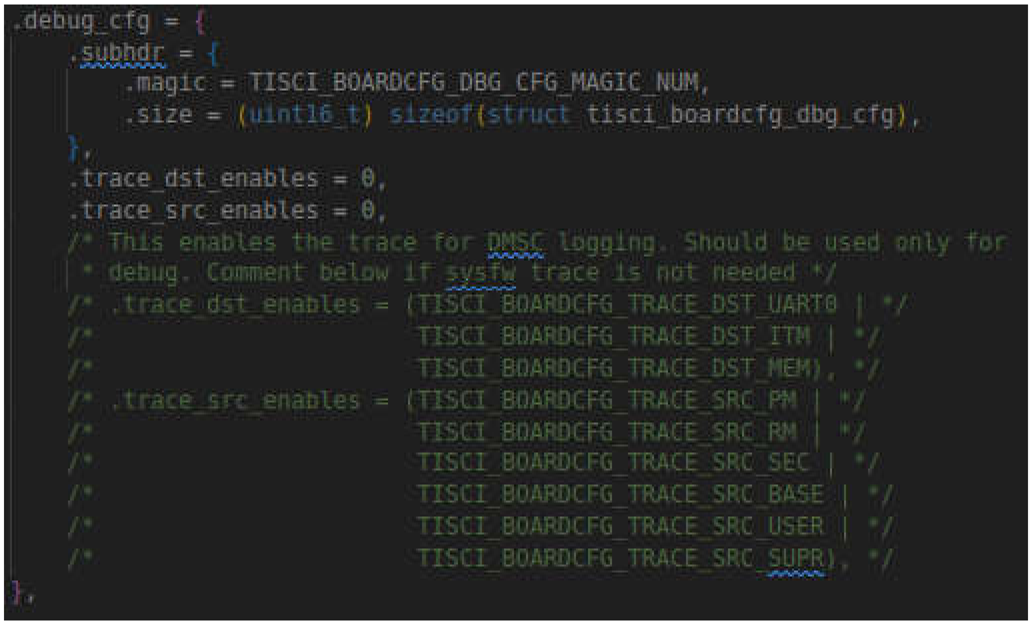
d. Copy tiboot3.bin and sysfw.itb to SD BOOT partition
e. Boot up board, and copy the WUKUP_UART0 log (on the screen) to input_log.txt file
f. There is a script parser sysfw_trace_parser.py in RTOS SDK
g. ./sysfw_trace_parser.py -I input_log.txt -o output_log.txt



```
#ifdef ENABLE_TRACE
        .trace_dst_enables = BOARDCFG_TRACE_DST_UART0 |
                             BOARDCFG_TRACE_DST_ITM |
                             BOARDCFG_TRACE_DST_MEM,
        .trace_src_enables = BOARDCFG_TRACE_SRC_PM |
                             BOARDCFG_TRACE_SRC_RM |
                             BOARDCFG_TRACE_SRC_SEC |
                             BOARDCFG_TRACE_SRC_BASE |
                             BOARDCFG_TRACE_SRC_USER |
                             BOARDCFG_TRACE_SRC_SUPR,
#endif
    },
};
```

Figure 2-1. SPL Boot Board configure

2. For SBL Boot:
   a. Enable the code comments in ti-processor-sdk-rtos-j7xxxx-evm-xx_xx/pdk_xxxx/packages/ti/drv/sciclient/soc/Vx/sciclient_defaultBoardcfg.c
   b. Recompile board configure under pdk_xxxx/packages/ti/build, $make sciclient_boardcfg
   c. Update PDK library under pdk_xxxx/packages/ti/build, $make pdk_libs_allcores BOARD=j7xxx_evm SOC=j7xxx
   d. Recompile sbl_mmcsd_img_mcu1_0_release.tiimage under pdk_xxxx/packages/ti/build $make -j BOARD=j7xxx_evm CORE=mcu1_0 BUILD_PROFILE=release sbl_mmcsd_img
   e. Copy the sbl_mmcsd_img_mcu1_0_release.tiimage to SD BOOT partition as tiboot3.bin $cp pdk_xxxx/packages/ti/boot/sbl/binary/j7xxx_evm/mmcsd/bin/sbl_mmcsd_img_mcu1_0_release.tiimage /media/BOOT/tiboot3.bin
   f. Boot up board, and copy the WUKUP_UART0 log (on the screen) to input_log.txt file
   g. There is a script parser sysfw_trace_parser.py in RTOS SDK
   h. ./sysfw_trace_parser.py -I input_log.txt -o output_log.txt



```
.debug_cfg = {
    .subhdr = {
        .magic = TISCI_BOARDCFG_DBG_CFG_MAGIC_NUM,
        .size = (uint16_t) sizeof(struct tisci_boardcfg_dbg_cfg),
    },
    .trace_dst_enables = 0,
    .trace_src_enables = 0,
    /* This enables the trace for DMSC logging. Should be used only for
     * debug. Comment below if sysfw trace is not needed */
    /* .trace_dst_enables = (TISCI_BOARDCFG_TRACE_DST_UART0 | */
    /*                       TISCI_BOARDCFG_TRACE_DST_ITM | */
    /*                       TISCI_BOARDCFG_TRACE_DST_MEM), */
    /* .trace_src_enables = (TISCI_BOARDCFG_TRACE_SRC_PM | */
    /*                       TISCI_BOARDCFG_TRACE_SRC_RM | */
    /*                       TISCI_BOARDCFG_TRACE_SRC_SEC | */
    /*                       TISCI_BOARDCFG_TRACE_SRC_BASE | */
    /*                       TISCI_BOARDCFG_TRACE_SRC_USER | */
    /*                       TISCI_BOARDCFG_TRACE_SRC_SUPR), */
},
```

Figure 2-2. SBL Boot Board Configure

CAUTION
Different SOC and SDK versions have different code path, so here using xxxxx to represent. The value of BOARD includes j7200_evm/j721e_evm/j721s2_evm/j784s4_evm, and the value of SOC includes j7200/j721e/j784s4/j721s2

2.2 MCU_UART0 Usage

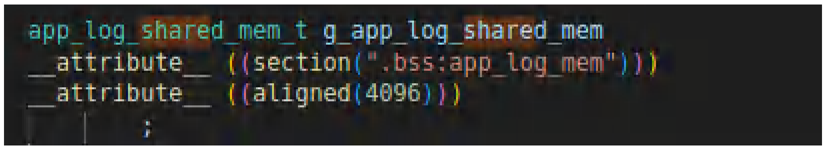
1. After MCU_UART0 is reserved, it can be used to check whether the processor is working normally. The method is to set the board to UART boot mode firstly, then the MCU_UART0 serial port will print out the "CCCCC" string, which can detect whether the power supply of the board is normal.
2. MCU_UART0 also can be used to burn OSPI flash according to this link.
3. For HS processors, MCU_UART0 also can be used to check if efuse burns the key successfully, the steps in details could be referred to in the *TDA4 HS Prime* chapter of this document.
4. For SBL boot, MCU_UART0 will be used as the debugging serial port of SBL and MCU1_0 firmware to print logs. The main function entry in sbl_main.c configures and initializes this serial port.

2.3 MAIN_UARTx Usage

1. The default SDK will use MAIN_UARTx to print the boot log of HLOS.
2. For TDA4X series processors, MAIN_UARTx is used to print the APP log and the boot log of A72, R5F and DSP cores.

The software-level design of printing multi-core log to a serial port is a complex process. Understanding this multi-core log output system helps to customize the design of own system output. By default, U-BOOT and kernel will initialize a serial port for log output. For the drivers and other related information in details, see this U-BOOT / Kernel document. The control of this serial port is always on the A72 side. The log of A72 cores will be printed out directly through this serial port. However, the log of other cores is firstly put into a shared memory, and then it will be read out by the A72 application. The specific process is as follows:

1. The OS boot of every core except A72 cores will call appInit function to initialize a 256KB shared memory shown in Figure 2-3 for storing the log.



```
app_log_shared_mem_t g_app_log_shared_mem
__attribute__ ((section(".bss:app_log_mem")))
__attribute__ ((aligned(4096)))
        ;
```

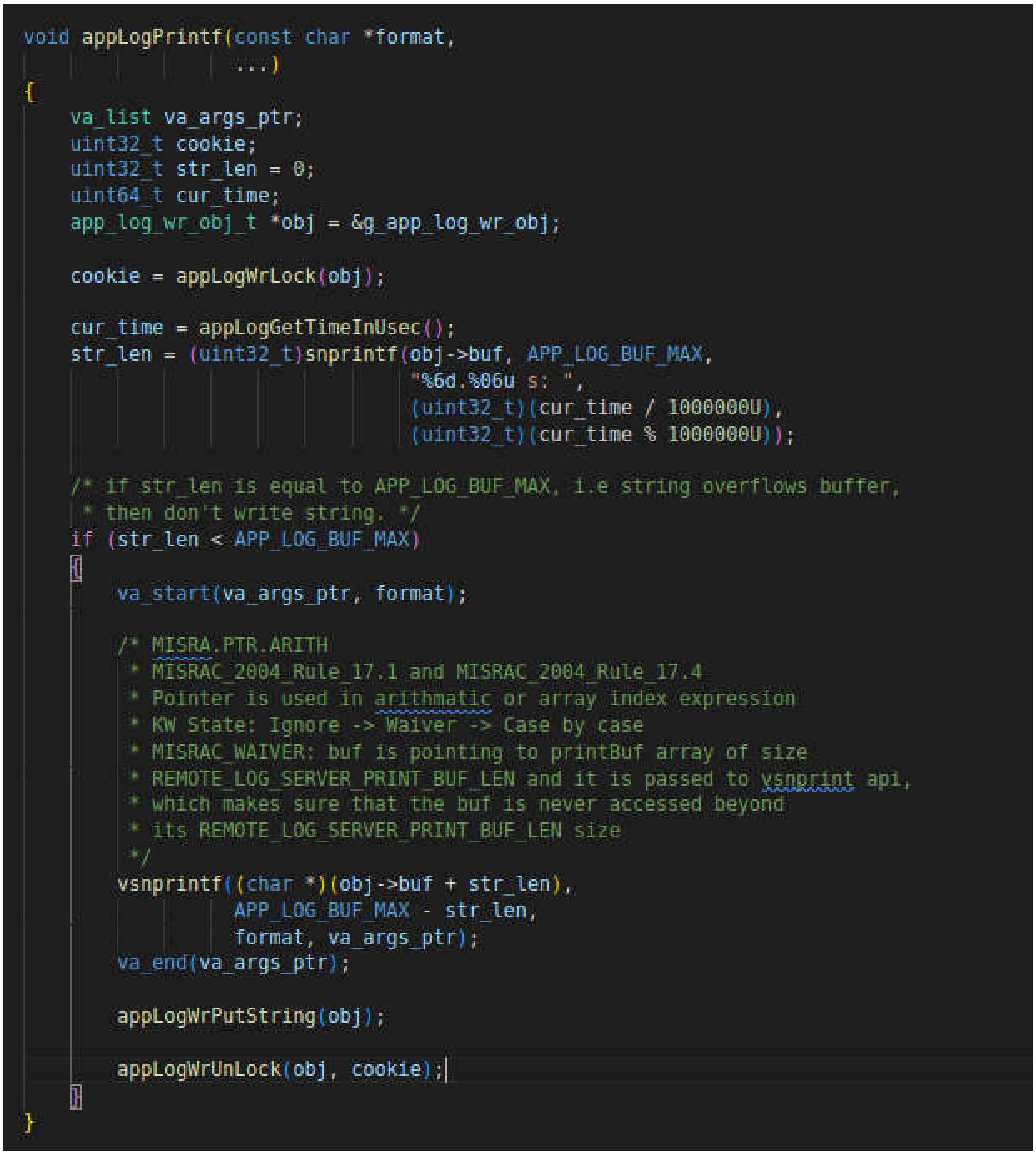
Figure 2-3. Log Share Memory Definition on the Code

2. This shared memory will be divided into 16 parts. Every part has 16 KB in total, and the first 32 Bytes for each part is used to store a struct shown the following code block. This struct is used to show the position of pointers in reading and writing respectively. The rest of this shared memory is used for log output.

```
typedef struct {

 /**< Init by reader to 0 */

    uint32_t log_rd_idx;


/**< Init by writer to 0 */
    uint32_t log_wr_idx;

/**<  Init by writer to APP_LOG_AREA_VALID_FLAG.
 reader will ignore this CPU shared mem log
 until the writer sets this
 to APP_LOG_AREA_VALID_FLAG */

    uint32_t log_area_is_valid;

 /**< CPU sync state */
    uint32_t log_cpu_sync_state;

/**< Init by writer to CPU name, used by reader to add a prefix when writing to console device
*/
    uint8_t  log_cpu_name[APP_LOG_MAX_CPU_NAME];
```

```
        /**< memory into which logs are written by this CPU */

        uint8_t  log_mem[APP_LOG_PER_CPU_MEM_SIZE];
    } app_log_cpu_shared_mem_t;
```

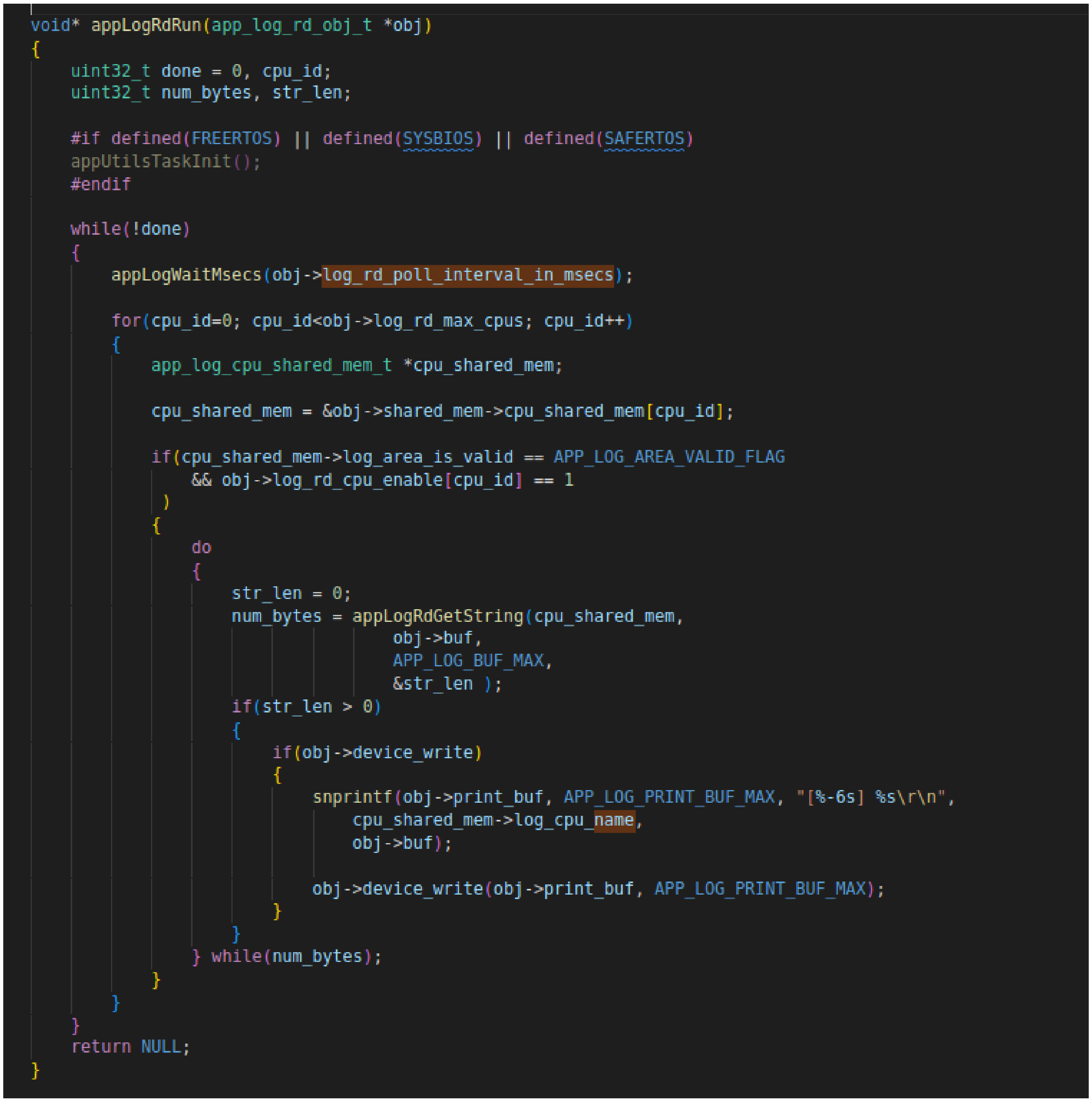
3. Apart from A72 cores, the log of other cores will be written to this shared memory by appLogPrintf, no matter using printf or UART_print. At the same time, the timestamp of this output log will also be written to this shared memory shown in Figure 2-4.



```
void appLogPrintf(const char *format,
                  ...)
{
    va_list va_args_ptr;
    uint32_t cookie;
    uint32_t str_len = 0;
    uint64_t cur_time;
    app_log_wr_obj_t *obj = &g_app_log_wr_obj;

    cookie = appLogWrLock(obj);

    cur_time = appLogGetTimeInUsec();
    str_len = (uint32_t)snprintf(obj->buf, APP_LOG_BUF_MAX,
                                 "%6d.%06u s: ",
                                 (uint32_t)(cur_time / 1000000U),
                                 (uint32_t)(cur_time % 1000000U));

    /* if str_len is equal to APP_LOG_BUF_MAX, i.e string overflows buffer,
     * then don't write string. */
    if (str_len < APP_LOG_BUF_MAX)
    {
        va_start(va_args_ptr, format);

        /* MISRA.PTR.ARITH
         * MISRAC_2004_Rule_17.1 and MISRAC_2004_Rule_17.4
         * Pointer is used in arithmatic or array index expression
         * KW State: Ignore -> Waiver -> Case by case
         * MISRAC_WAIVER: buf is pointing to printBuf array of size
         * REMOTE_LOG_SERVER_PRINT_BUF_LEN and it is passed to vsnprint api,
         * which makes sure that the buf is never accessed beyond
         * its REMOTE_LOG_SERVER_PRINT_BUF_LEN size
         */
        vsnprintf((char *)(obj->buf + str_len),
                  APP_LOG_BUF_MAX - str_len,
                  format, va_args_ptr);
        va_end(va_args_ptr);

        appLogWrPutString(obj);

        appLogWrUnLock(obj, cookie);
    }
}
```

Figure 2-4. MCU/DSP Core Write Log to Share Memory

4. The A72 application vx_app_arm_remote_log.out will map this shared memory from Linux. It reads the memory every second to extract the log of each core except A72 cores, and add the name of the corresponding core and output it to the serial port shown in Figure 2-5.



```
void* appLogRdRun(app_log_rd_obj_t *obj)
{
    uint32_t done = 0, cpu_id;
    uint32_t num_bytes, str_len;

    #if defined(FREERTOS) || defined(SYSBIOS) || defined(SAFERTOS)
    appUtilsTaskInit();
    #endif

    while(!done)
    {
        appLogWaitMsecs(obj->log_rd_poll_interval_in_msecs);

        for(cpu_id=0; cpu_id<obj->log_rd_max_cpus; cpu_id++)
        {
            app_log_cpu_shared_mem_t *cpu_shared_mem;

            cpu_shared_mem = &obj->shared_mem->cpu_shared_mem[cpu_id];

            if(cpu_shared_mem->log_area_is_valid == APP_LOG_AREA_VALID_FLAG
                && obj->log_rd_cpu_enable[cpu_id] == 1
             )
            {
                do
                {
                    str_len = 0;
                    num_bytes = appLogRdGetString(cpu_shared_mem,
                                        obj->buf,
                                        APP_LOG_BUF_MAX,
                                        &str_len );
                    if(str_len > 0)
                    {
                        if(obj->device_write)
                        {
                            snprintf(obj->print_buf, APP_LOG_PRINT_BUF_MAX, "[%-6s] %s\r\n",
                                cpu_shared_mem->log_cpu_name,
                                obj->buf);

                            obj->device_write(obj->print_buf, APP_LOG_PRINT_BUF_MAX);
                        }
                    }
                } while(num_bytes);
            }
        }
    }
    return NULL;
}
```

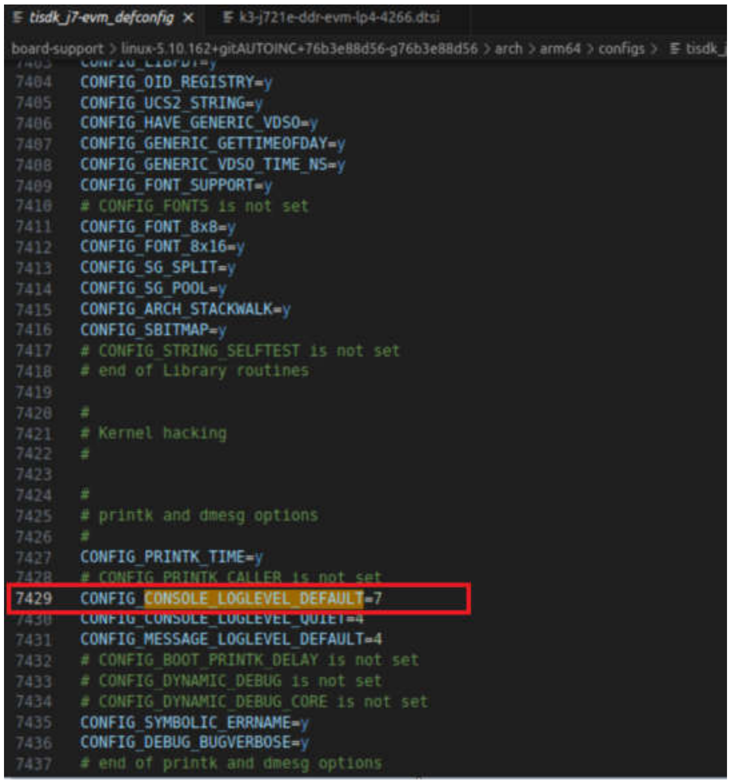
Figure 2-5. A72 Core Read Log From Share Memory

3 Log Level Design on Software Module

In large software projects, there is usually a log level used for debugging, and TI's SDK also has some commonly used log levels for controlling and printing more useful information to assist debugging. For more details, see the following steps.

1. Linux kernel log level:

   During the boot stage, Linux kernel could configure different log levels on the bootargs in the device tree(*k3-j721e-common-proc-board.dts if use TDA4VM*). By adding the loglevel=8 parameter in the Linux kernel configure, where the value ranges from 0 (least detailed) to 8 (most detailed). By default, the loglevel is 7 shown in Figure 3-1.


tisdk_j7-evm_defconfig
k3-j721e-ddr-evm-lp4-4266.dtsi
board-support > linux-5.10.162+gitAUTOINC+76b3e88d56-g76b3e88d56 > arch > arm64 > configs > tisdk_

```
CONFIG_OID_REGISTRY=y
CONFIG_UCS2_STRING=y
CONFIG_HAVE_GENERIC_VDSO=y
CONFIG_GENERIC_GETTIMEOFDAY=y
CONFIG_GENERIC_VDSO_TIME_NS=y
CONFIG_FONT_SUPPORT=y
# CONFIG_FONTS is not set
CONFIG_FONT_8x8=y
CONFIG_FONT_8x16=y
CONFIG_SG_SPLIT=y
CONFIG_SG_POOL=y
CONFIG_ARCH_STACKWALK=y
CONFIG_SBITMAP=y
# CONFIG_STRING_SELFTEST is not set
# end of Library routines

#
# Kernel hacking
#

#
# printk and dmesg options
#
CONFIG_PRINTK_TIME=y
# CONFIG_PRINTK_CALLER is not set
CONFIG_CONSOLE_LOGLEVEL_DEFAULT=7
CONFIG_CONSOLE_LOGLEVEL_QUIET=4
CONFIG_MESSAGE_LOGLEVEL_DEFAULT=4
# CONFIG_BOOT_PRINTK_DELAY is not set
# CONFIG_DYNAMIC_DEBUG is not set
# CONFIG_DYNAMIC_DEBUG_CORE is not set
CONFIG_SYMBOLIC_ERRNAME=y
CONFIG_DEBUG_BUGVERBOSE=y
# end of printk and dmesg options
```

Figure 3-1. Linux Default Kernel Log Level

2. SBL BOOT log level:

   The SBL log level ranges from 0 (least detailed) to 3 (most detailed). The following describes the steps needed to set the log level to 3 and remove the log output restriction.

   Changes DSBL_LOG_LEVEL=3 in mcusw/mcuss_demos/boot_app_mcu_rtos/makefile

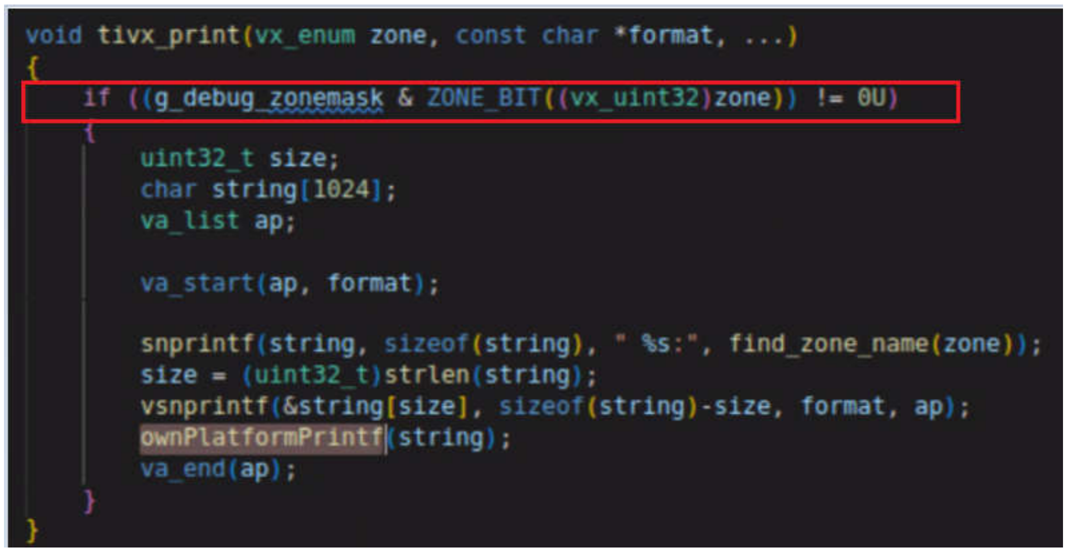
   Changes DSBL_LOG_LEVEL=3 in pdk_xxxx /packages/ti/boot/sbl/sbl_component.mk

   Changes in pdk_xxxx/packages/ti/boot/sbl/soc/k3/sbl_log.h show as following *#define SBL_log(dbg_level, ...) if (1) { UART_printf(__VA_ARGS__); }*

   Changes DSBL_LOG_LEVEL=3 in pdk_xxxx/packages/ti/build/makerules/build_config.mk

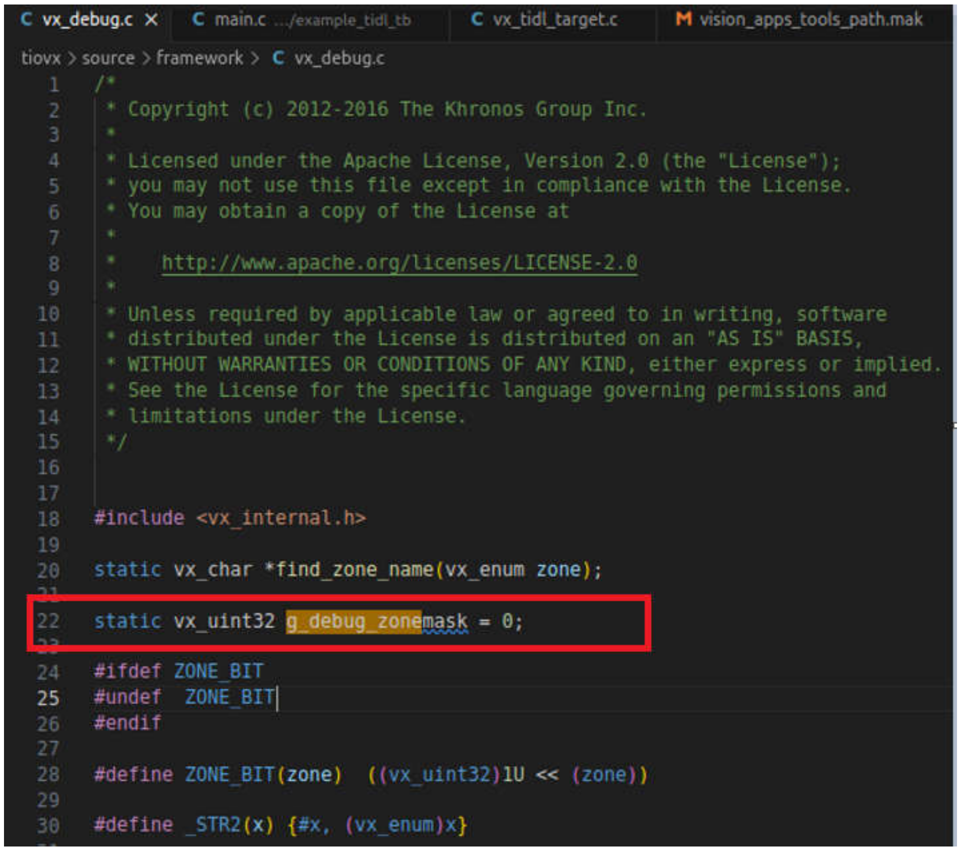
3. Openvx log level:

   As an important part of the TDA4X applications, Openvx also provides log level for debugging. From Figure 3-2, the g_debug_zonemask (default 0) is used to calculate the current log level. To make it easier for printing Openvx log level, the following if condition could be simply commented out.



```
void tivx_print(vx_enum zone, const char *format, ...)
{
    if ((g_debug_zonemask & ZONE_BIT((vx_uint32)zone)) != 0U)
    {
        uint32_t size;
        char string[1024];
        va_list ap;

        va_start(ap, format);

        snprintf(string, sizeof(string), " %s:", find_zone_name(zone));
        size = (uint32_t)strlen(string);
        vsnprintf(&string[size], sizeof(string)-size, format, ap);
        ownPlatformPrintf(string);
        va_end(ap);
    }
}
```

Figure 3-2. Openvx Low Level Print API



```
C vx_debug.c ×    C main.c .../example_tidl_tb    C vx_tidl_target.c    M vision_apps_tools_path.mak
tiovx > source > framework > C vx_debug.c
/*
 * Copyright (c) 2012-2016 The Khronos Group Inc.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *    http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */


#include <vx_internal.h>

static vx_char *find_zone_name(vx_enum zone);

static vx_uint32 g_debug_zonemask = 0;

#ifdef ZONE_BIT
#undef  ZONE_BIT
#endif

#define ZONE_BIT(zone)  ((vx_uint32)1U << (zone))

#define _STR2(x) {#x, (vx_enum)x}
```

Figure 3-3. Openvx Log Level Set

4. TIDL log level:

   During the import and inference stage of TIDL, the debugTraceLevel=3 could be used to configure different log level ranging from 0 (least detailed) to 3 (most detailed) in the TIDL inference configure file.
5. Memory allocation log level:

   The C file in this directory ti-processor-sdk-rtos-j7xxxx-evm-xx_xx_xx_xx/app_utils/utils/mem/src/ , when the **APP_MEM_DEBUG** (default is not defined) macro is defined, the memory allocation log will be printed out.

4 Change UART Instance

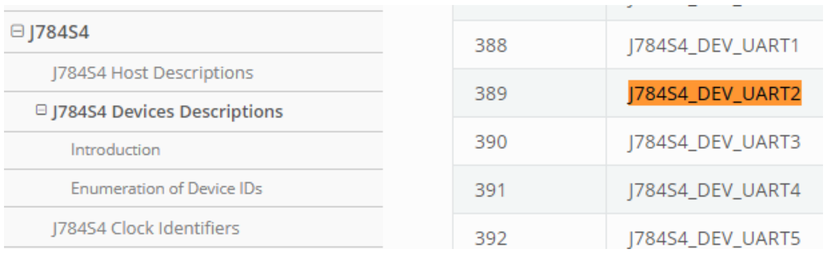
Normally, the default PINs configurations of MCU_UART0 and WKUP_UART0 serial ports are not suggested to be modified during the customers' hardware design stage. The reason is that ROM code uses the default pin setting of MCU_UART0 when doing OSPI burning, and SYSFW uses the default pin setting in WUKP_UART0. However, due to PIN conflicts and other reasons, MAIN_UARTX is often changed to a new serial port. Moreover, in multi-core debugging, a separate MAIN_UARTX serial port is often required to be configured for the DSP/ MAIN_R5F core for debugging purpose.

4.1 Change MAIN_UARTx for MAIN Domain

The default terminal output on the different Jacinto 7 series processors is not the same. For example, UART0 is used for the terminal output on the J721E and J7200 EVMs, but UART8 is the primary UART interface for the J721S2 and J784S4 EVMs. As the pins of UART ports can also be used by other applications, it is a common request to customize the UART port depending on customers' use cases. The following was an example of changing the default UART8 to UART2 on TDA4VH main domain base on SDK 8.6, and this also can be a reference of changing to other ports on other Jacinto 7 processors.

1. Add the clock setting for UART2.

   The 389 is the device ID of J784S4 series, and can be found from the link.



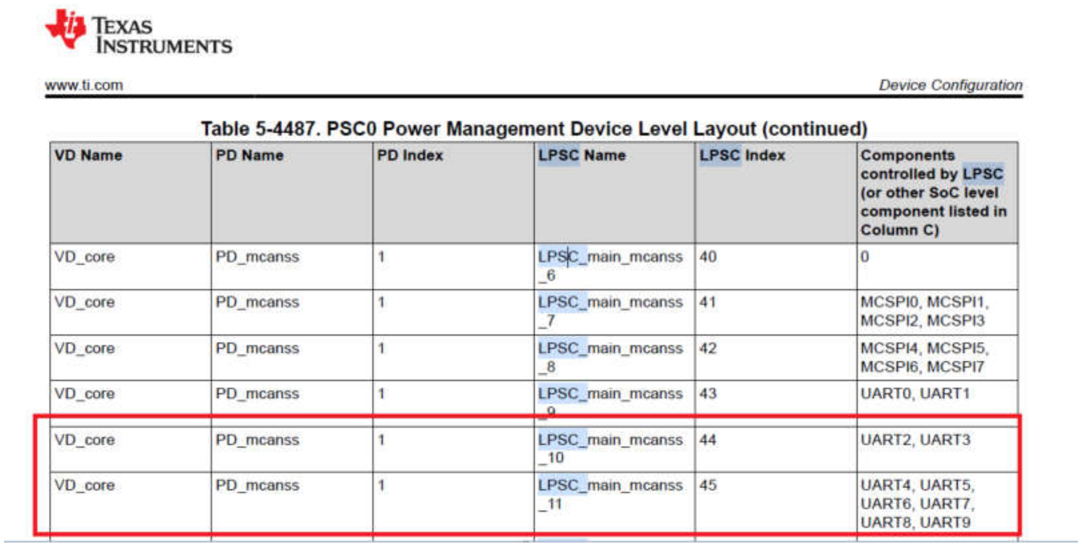
388	J784S4_DEV_UART1
389	J784S4_DEV_UART2
390	J784S4_DEV_UART3
391	J784S4_DEV_UART4
392	J784S4_DEV_UART5

Figure 4-1. TDA4VH UART Clock ID

```
diff --git a/arch/arm/mach-k3/j784s4/clk-data.c b/arch/arm/mach-k3/j784s4/clk-data.c
index a266735cc0..081c6d8970 100644
--- a/arch/arm/mach-k3/j784s4/clk-data.c
+++ b/arch/arm/mach-k3/j784s4/clk-data.c
@@ -283,7 +283,7 @@ static const struct clk_data clk_list[] = {
    CLK_MUX("emmcsd_refclk_sel_out1", emmcsd_refclk_sel_out1_parents, 4, 0x1080b4, 0, 2, 0),
    CLK_MUX("gtc_clk_mux_out0", gtc_clk_mux_out0_parents, 16, 0x108030, 0, 4, 0),
    CLK_DIV_DEFFREQ("usart_programmable_clock_divider_out0",
"hsdiv4_16fft_main_1_hsdivout0_clk", 0x1081c0, 0, 2, 0, 0, 48000000),
-   CLK_DIV("usart_programmable_clock_divider_out5", "hsdiv4_16fft_main_1_hsdivout0_clk",
0x1081d4, 0, 2, 0, 0),
+   CLK_DIV("usart_programmable_clock_divider_out2", "hsdiv4_16fft_main_1_hsdivout0_clk",
0x1081c8, 0, 2, 0, 0),
    CLK_DIV("usart_programmable_clock_divider_out8", "hsdiv4_16fft_main_1_hsdivout0_clk",
0x1081e0, 0, 2, 0, 0),
    CLK_DIV("k3_pll_ctrl_wrap_main_0_chip_div24_clk_clk",
"k3_pll_ctrl_wrap_main_0_sysclkout_clk", 0x41011c, 0, 5, 0, 0),
    CLK_DIV("k3_pll_ctrl_wrap_wkup_0_chip_div24_clk_clk",
"k3_pll_ctrl_wrap_wkup_0_sysclkout_clk", 0x4201011c, 0, 5, 0, 0),
@@ -405,8 +405,8 @@ static const struct dev_clk soc_dev_clk_data[] = {
    DEV_CLK(279, 2, "wkup_i2c_mcupll_bypass_out0"),
    DEV_CLK(279, 3, "hsdiv4_16fft_mcu_1_hsdivout3_clk"),
    DEV_CLK(279, 4, "gluelogic_hfosc0_clkout"),
-   DEV_CLK(392, 0, "usart_programmable_clock_divider_out5"),
-   DEV_CLK(392, 3, "k3_pll_ctrl_wrap_main_0_chip_div1_clk_clk"),
+   DEV_CLK(389, 0, "usart_programmable_clock_divider_out2"),
+    DEV_CLK(389, 3, "k3_pll_ctrl_wrap_main_0_chip_div1_clk_clk"),
    DEV_CLK(395, 0, "usart_programmable_clock_divider_out8"),
    DEV_CLK(395, 3, "k3_pll_ctrl_wrap_main_0_chip_div1_clk_clk"),
    DEV_CLK(398, 0, "k3_pll_ctrl_wrap_main_0_chip_div1_clk_clk"),
```

2. Add the device ID and LPSC setting in the list for UART2.

   The 44 is the LPSC index of UART2 for J784S4 series, and could be searched from device-specific TRM.



www.ti.com Device Configuration

Table 5-4487. PSC0 Power Management Device Level Layout (continued)

VD Name	PD Name	PD Index	LPSC Name	LPSC Index	Components controlled by LPSC (or other SoC level component listed in Column C)
VD_core	PD_mcanss	1	LPSC_main_mcanss_6	40	0
VD_core	PD_mcanss	1	LPSC_main_mcanss_7	41	MCSPI0, MCSPI1, MCSPI2, MCSPI3
VD_core	PD_mcanss	1	LPSC_main_mcanss_8	42	MCSPI4, MCSPI5, MCSPI6, MCSPI7
VD_core	PD_mcanss	1	LPSC_main_mcanss_9	43	UART0, UART1
VD_core	PD_mcanss	1	LPSC_main_mcanss_10	44	UART2, UART3
VD_core	PD_mcanss	1	LPSC_main_mcanss_11	45	UART4, UART5, UART6, UART7, UART8, UART9

Figure 4-2. TDA4VH UART LPSC

```
diff --git a/arch/arm/mach-k3/j784s4/dev-data.c b/arch/arm/mach-k3/j784s4/dev-data.c
index e44afad3ec..b5ae132b4a 100644
--- a/arch/arm/mach-k3/j784s4/dev-data.c
+++ b/arch/arm/mach-k3/j784s4/dev-data.c
@@ -51,6 +51,7 @@ static struct ti_lpsc soc_lpsc_list[] = {
     [20] = PSC_LPSC(81, &soc_psc_list[2], &soc_pd_list[6], &soc_lpsc_list[18]),
     [21] = PSC_LPSC(120, &soc_psc_list[2], &soc_pd_list[7], &soc_lpsc_list[22]),
     [22] = PSC_LPSC(121, &soc_psc_list[2], &soc_pd_list[7], NULL),
+    [23] = PSC_LPSC(44, &soc_psc_list[2], &soc_pd_list[3], NULL),
 };
static struct ti_dev soc_dev_list[] = {
@@ -76,7 +77,7 @@ static struct ti_dev soc_dev_list[] = {
     PSC_DEV(141, &soc_lpsc_list[14]),
     PSC_DEV(140, &soc_lpsc_list[15]),
     PSC_DEV(146, &soc_lpsc_list[16]),
-    PSC_DEV(392, &soc_lpsc_list[17]),
+    PSC_DEV(389, &soc_lpsc_list[23]),
     PSC_DEV(395, &soc_lpsc_list[17]),
     PSC_DEV(198, &soc_lpsc_list[18]),
     PSC_DEV(202, &soc_lpsc_list[19]),
--
```

3. Set serial port and pin, this should set for both UBOOT dts and Kernel dts files.

```
diff --git a/arch/arm/dts/k3-j784s4-evm-u-boot.dtsi b/arch/arm/dts/k3-j784s4-evm-u-boot.dtsi
index 9d6f7dbbd5..3846d90f9a 100644
--- a/arch/arm/dts/k3-j784s4-evm-u-boot.dtsi
+++ b/arch/arm/dts/k3-j784s4-evm-u-boot.dtsi
@@ -12,7 +12,7 @@
     aliases {
         serial0 = &wkup_uart0;
         serial1 = &mcu_uart0;
-        serial2 = &main_uart8;
+        serial2 = &main_uart2;
         i2c0 = &wkup_i2c0;
         i2c1 = &mcu_i2c0;
         i2c2 = &mcu_i2c1;
@@ -105,6 +105,10 @@
     u-boot,dm-spl;
 };
```

```
+&main_uart2_pins_default {
+    u-boot,dm-spl;
+};
+
 &main_mmc1_pins_default {
     u-boot,dm-spl;
 };
@@ -132,6 +136,10 @@
     u-boot,dm-spl;
 };

+&main_uart2 {
+    u-boot,dm-spl;
+};
+
 &mcu_uart0 {
     u-boot,dm-spl;
 };
diff --git a/arch/arm/dts/k3-j784s4-evm.dts b/arch/arm/dts/k3-j784s4-evm.dts
index 5e213b2c11..7f8f507318 100644
--- a/arch/arm/dts/k3-j784s4-evm.dts
+++ b/arch/arm/dts/k3-j784s4-evm.dts
@@ -21,7 +21,7 @@
         stdout-path = "serial2:115200n8";
     };
     aliases {
-        serial2 = &main_uart8;
+        serial2 = &main_uart2;
         mmc0 = &main_sdhci0;
         mmc1 = &main_sdhci1;
         can0 = &mcu_mcan0;
@@ -402,6 +402,15 @@
         >;
     };

+    main_uart2_pins_default: main-uart2-pins-default {
+        pinctrl-single,pins = <
+            J784S4_IOPAD(0x0c4, PIN_INPUT, 11) /* (AD36) CTSn */
+            J784S4_IOPAD(0x0c8, PIN_OUTPUT, 11) /* (AJ32) RTSn */
+            J784S4_IOPAD(0x0dc, PIN_OUTPUT, 11) /* (AM36) TXD */
+            J784S4_IOPAD(0x0d8, PIN_INPUT, 11) /* (AM35) RXD*/
+        >;
+    };
+
     main_i2c3_pins_default: main-i2c3-pins-default {
         pinctrl-single,pins = <
             J784S4_IOPAD(0x064, PIN_INPUT_PULLUP, 13) /* (AF38) MCAN0_TX.I2C3_SCL */
@@ -743,11 +752,13 @@
     status = "disabled";
 };
-&main_uart1 {
-    status = "disabled";
+&main_uart2 {
+    status = "okay";
+    pinctrl-names = "default";
+    pinctrl-0 = <&main_uart2_pins_default>;
 };

-&main_uart2 {
+&main_uart1 {
     status = "disabled";
 };

diff --git a/arch/arm/dts/k3-j784s4-r5-evm.dts b/arch/arm/dts/k3-j784s4-r5-evm.dts
index 4a697e2738..154a07c802 100644
--- a/arch/arm/dts/k3-j784s4-r5-evm.dts
+++ b/arch/arm/dts/k3-j784s4-r5-evm.dts
@@ -13,7 +13,7 @@
 / {
     chosen {
         firmware-loader = &fs_loader0;
-        stdout-path = &main_uart8;
+        stdout-path = &main_uart2;
         tick-timer = &timer1;
     };

@@ -151,6 +151,15 @@
```

```
            >;
        };

+       main_uart2_pins_default: main-uart2-pins-default {
+           pinctrl-single,pins = <
+               J784S4_IOPAD(0x0c4, PIN_INPUT, 11) /* (AD36) CTSn */
+               J784S4_IOPAD(0x0c8, PIN_OUTPUT, 11) /* (AJ32) RTSn */
+               J784S4_IOPAD(0x0dc, PIN_OUTPUT, 11) /* (AM36) TXD */
+               J784S4_IOPAD(0x0d8, PIN_INPUT, 11) /* (AM35) RXD*/
+           >;
+       };
+
         main_mmc1_pins_default: main-mmc1-pins-default {
            pinctrl-single,pins = <
                J784S4_IOPAD(0x104, PIN_INPUT, 0) /* (AB38) MMC1_CLK */
@@ -253,6 +262,12 @@
        pinctrl-0 = <&main_uart8_pins_default>;
 };

+&main_uart2 {
+       status = "okay";
+       pinctrl-names = "default";
+       pinctrl-0 = <&main_uart2_pins_default>;
+};
```

4. Configure the boot command for UART2.

```
diff --git a/include/configs/j784s4_evm.h b/include/configs/j784s4_evm.h
index eb609100b0..942d6c3dbe 100644
--- a/include/configs/j784s4_evm.h
+++ b/include/configs/j784s4_evm.h
@@ -75,7 +75,7 @@
         "setenv fdtfile ${name_fdt}\0"                \
     "name_kern=Image\0"                        \
     "console=ttyS2,115200n8\0"                   \
-    "args_all=setenv optargs earlycon=ns16550a,mmio32,0x02880000 "    \
+    "args_all=setenv optargs earlycon=ns16550a,mmio32,0x02820000 "    \
         "${mtdparts}\0"                        \
     "run_kern=booti ${loadaddr} ${rd_spec} ${fdtaddr}\0"

--
```

5. Rebuild the OPTEE if it is Used.

   Just change 8 to 2 in export CFG_CONSOLE_UART=0x8 according to this link.
6. Change the Arm® Trust Firmware.

```
diff --git a/plat/ti/k3/include/platform_def.h b/plat/ti/k3/include/platform_def.h
index 690c68e5c..db083ca2f 100644
--- a/plat/ti/k3/include/platform_def.h
+++ b/plat/ti/k3/include/platform_def.h
@@ -91,14 +91,14 @@
 /* Platform default console definitions */
 #ifndef K3_USART_BASE
-#define K3_USART_BASE 0x02800000
+#define K3_USART_BASE 0x02820000
 #endif
```

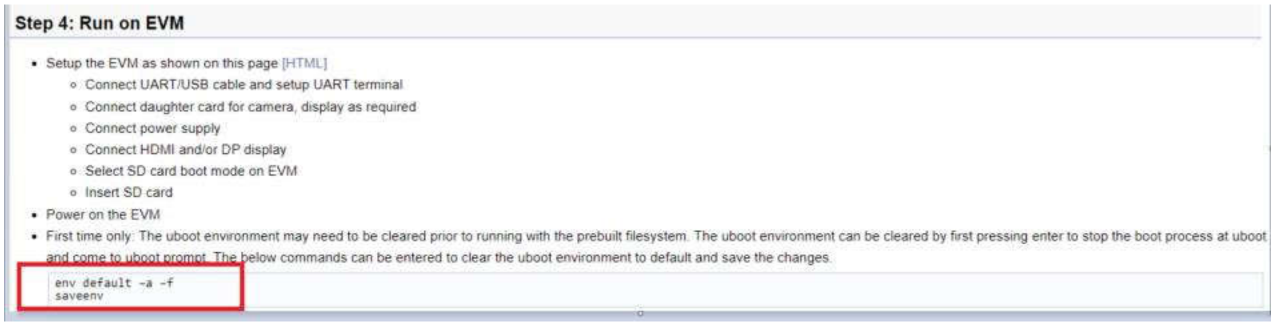
   After modifying the above K3_USART_BASE for UART2, then the following instructions are needed to recompile bl31.bin

   a. *cd $SDK_PATH/board-support/trusted-firmware-a-2.8+gitAUTOINC+2fcd408bb3*
   b. *make CROSS_COMPILE=aarch64-none-linux-gnu- ARCH=aarch64 PLAT=k3 TARGET_BOARD=generic SPD=opted*
   c. *cp ./build/k3/generic/release/bl31.bin ../prebuilt-images/*

   Then based on the rebuilt bl31.bin, the below instructions are required to make it as part of tispl.bin, and copy tispl.bin and u-boot.img to the BOOT in the SD card.

   a. *cd ../../*
   b. *make u-boot-spl-jacinto*
   c. *cp board-support/u-boot_build/a72/tispl.bin board-support/u-boot_build/a72/u-boot.img /media/$USER/BOOT*

7. Clear the UBOOT Environment to the Default and Save the Changes.



Step 4: Run on EVM

- Setup the EVM as shown on this page [HTML]
  - Connect UART/USB cable and setup UART terminal
  - Connect daughter card for camera, display as required
  - Connect power supply
  - Connect HDMI and/or DP display
  - Select SD card boot mode on EVM
  - Insert SD card
- Power on the EVM
- First time only: The uboot environment may need to be cleared prior to running with the prebuilt filesystem. The uboot environment can be cleared by first pressing enter to stop the boot process at uboot and come to uboot prompt. The below commands can be entered to clear the uboot environment to default and save the changes.

```
env default -a -f
saveenv
```

Figure 4-3. UBOOT Environment Set

CAUTION
Step 7 is not requested anymore starting from SDK9.0. It is sufficient to apply the above 7 steps to change the default main UART port from 8 to 2 on TDA4VH.

4.2 Set Standalone UART Port for DSP/MCU

By default, the MCU and DSP cores of J7 SOC do not set a separate UART for serial port output. For TDA4X, by default the log of every core except A72 cores will be written to a shared memory, and then the A72 application will read it and print these logs to MAIN_UARTX serial port. However, in order to facilitate debugging, it is often necessary to print logs of multiple cores at the same time or to continue to print logs when core A is not working properly. In this case, it is necessary to configure a separate serial port for the MCU/DSP cores. The following uses TDA4VM as an example to set up a separate serial port for C7.

1. Add UART Initial configuration in MAIN function.

```
diff --git a/vision_apps/platform/j721e/rtos/c7x_1/main.c
b/vision_apps/platform/j721e/rtos/c7x_1/main.c
index 0dcfa4fd..e857838b 100755
--- a/vision_apps/platform/j721e/rtos/c7x_1/main.c
+++ b/vision_apps/platform/j721e/rtos/c7x_1/main.c
@@ -88,7 +88,8 @@
#include <ti/sysbios/family/c7x/Hwi.h>
#include <ti/sysbios/family/c7x/Mmu.h>
#endif
+#include <ti/drv/uart/UART.h>
+#include <ti/drv/uart/UART_stdio.h>
/* For J7ES/J721E/TDA4VM the upper 2GB DDR starts from 0x0008_8000_0000 */
/* This address is mapped to a virtual address of 0x0001_0000_0000 */
#define DDR_C7X_1_LOCAL_HEAP_VADDR (DDR_C7X_1_LOCAL_HEAP_ADDR)
@@ -96,18 +97,33 @@
+extern int uart_print_test(void);
+extern int uart_test(void);
@@ -181,9 +197,13 @@ int main(void)
{
TaskP_Params tskParams;
TaskP_Handle task;
OS_init();
+/* Set TDA4VM PINMUX UART2_RX(PIN Y1)&UART2_TX(PIN Y5)
+*  We can get the register address from the datasheet
+*/
+ *((int *)(0x00011C1DC))=0x50003;
+ *((int *)(0x00011C1E0))=0x10003;
+ uart_print_test();
appC7xClecInitDru();
```

2. Create UART instance for C7.

   Create c7_uart_print temp folder under the path *vision_apps/basic_demos/* and create c7_uart_print.c and concerto.mk file as initial configuration UART library.

   The following content is for c7_uart_print.c.

```
#include <ti/drv/uart/UART.h>
#include <ti/drv/uart/UART_stdio.h>
#include <ti/board/src/j721e_evm/include/board_utils.h>
#include <ti/board/board.h>
#include <ti/board/src/j721e_evm/include/board_cfg.h>
int uart_test(void)
{
   UART_printf("\n==============================================\n");
   UART_printf("\n**********c7x uart printf*********************\n");
   UART_printf("*                 UART Test                  *\n");
   UART_printf("**********************************************\n");
   return 0;
}

int uart_print_test(void)
{
    Board_initParams_t initParams;

    /* Verify the SoC UART0 */
    Board_getInitParams(&initParams);
    initParams.uartInst = 2;
    initParams.uartSocDomain = BOARD_SOC_DOMAIN_MAIN;
    Board_setInitParams(&initParams);
    Board_init(BOARD_INIT_UART_STDIO);

    uart_test();

return 0;
}
```

   The following content is for concerto.mk.

```
ifeq ($(TARGET_CPU),$(filter $(TARGET_CPU), x86_64 C71 C7120))

include $(PRELUDE)
TARGET      := c7_uart_print
TARGETTYPE  := library
CSOURCES    := $(call all-c-files)
CPPSOURCES  := $(call all-cpp-files)
CFLAGS+= -mv7100 --c11
ifeq ($(TARGET_CPU), x86_64)
IDIRS       += $(CGT7X_ROOT)/host_emulation/include/C7100
CFLAGS += --std=c++14 -D_HOST_EMULATION -pedantic -fPIC -w -c -g -o4
CFLAGS += -Wno-sign-compare
endif

include $(FINALE)

endif
```

3. Change output log from shared memory to UART FIFO.

```
diff --git a/vision_apps/utils/console_io/src/app_log_writer.c b/vision_apps/utils/
console_io/src/app_log_writer.c
index a02a785c..561d1434 100755
--- a/vision_apps/utils/console_io/src/app_log_writer.c
+++ b/vision_apps/utils/console_io/src/app_log_writer.c
@@ -220,6 +220,32 @@ int32_t  appLogWrPutString(app_log_wr_obj_t *obj)

     return status;
 }
+#if defined C71
+int32_t  c7x_appLogWrPutString(app_log_wr_obj_t *obj)
+{
+    int32_t status = 0;
+    volatile uint32_t copy_bytes,num_bytes;
+    volatile uint8_t *buf = (uint8_t*)obj->buf;
+
+
+    if (0 == status)
+    {
+        num_bytes = strlen((char*)buf);
+
+        if (num_bytes <= 0)
+        {
+            status = -1;
+        }
+    }
+
+    if (0 == status)
+    {
+        UART_puts(buf,num_bytes);
+    }
+
+    return status;
+}
+#endif

 void appLogPrintf(const char *format, ...)
 {
@@ -266,6 +292,8 @@ void appLogPrintf(const char *format, ...)
             printf(obj->buf);
             #endif
         }
+        #elif defined C71
+        c7x_appLogWrPutString(obj);
        #else
        appLogWrPutString(obj);
        #endif
```

4. Add Library path for compile binary.

```
diff --git a/vision_apps/platform/j721e/rtos/concerto_c7x_inc.mak b/vision_apps/platform/j721e/
rtos/concerto_c7x_inc.mak
index 4e3c5a29..4a9c94db 100755
--- a/vision_apps/platform/j721e/rtos/concerto_c7x_inc.mak
+++ b/vision_apps/platform/j721e/rtos/concerto_c7x_inc.mak
@@ -19,6 +19,10 @@ endif
 ifeq ($(RTOS),SAFERTOS)
        LDIRS += $(PDK_PATH)/packages/ti/osal/lib/safertos/$(SOC)/c7x/$(TARGET_BUILD)/
 endif
+
+LDIRS += $(PDK_PATH)/packages/ti/drv/uart/lib/$(SOC)/c7x/$(TARGET_BUILD)/
+LDIRS += $(PDK_PATH)/packages/ti/drv/i2c/lib/$(SOC)/c7x/$(TARGET_BUILD)/
+LDIRS += $(PDK_PATH)/packages/ti/board/lib/$(SOC)_evm/c7x/$(TARGET_BUILD)/
 LDIRS += $(PDK_PATH)/packages/ti/csl/lib/$(SOC)/c7x/$(TARGET_BUILD)/
 LDIRS += $(PDK_PATH)/packages/ti/drv/ipc/lib/$(SOC)/c7x_1/$(TARGET_BUILD)/
 LDIRS += $(PDK_PATH)/packages/ti/drv/udma/lib/$(SOC)/c7x_1/$(TARGET_BUILD)/
@@ -45,6 +49,7 @@ STATIC_LIBS += vx_app_ptk_demo_common
 STATIC_LIBS += vx_kernels_common
 STATIC_LIBS += vx_target_kernels_img_proc_c71
 STATIC_LIBS += vx_app_c7x_voxel2point
+STATIC_LIBS += c7_uart_print

 PTK_LIBS =
 PTK_LIBS += ptk_algos
@@ -76,6 +81,9 @@ ADDITIONAL_STATIC_LIBS += ipc.ae71
 ADDITIONAL_STATIC_LIBS += dmautils.ae71
 ADDITIONAL_STATIC_LIBS += sciclient.ae71
 ADDITIONAL_STATIC_LIBS += udma.ae71
+ADDITIONAL_STATIC_LIBS += ti.drv.uart.ae71
+ADDITIONAL_STATIC_LIBS += ti.board.ae71
+ADDITIONAL_STATIC_LIBS += ti.drv.i2c.ae71

 ifeq ($(RTOS),FREERTOS)
        ADDITIONAL_STATIC_LIBS += ti.kernel.freertos.ae71
```

The above describes all changes at the SDK level. The next step is only required to recompile the C7 firmware and flash it to SD card or EMMC.

5 Summary

This application note aims to provide you with a basic introduction about the UART analysis and the common UART applications. You can refer to this guide to setup your own software designs on your boards depending on the hardware connections. This document also introduces ways for users to perform a better debugging based on UART.

6 References

- Trace Layer
- Kernel UART Driver
- PDK UART User Interface
- Texas Instruments: *TDA4VM Technical Reference Manual*