1.1.1 DDR Inline ECC 在 Jacinto7 SoC 中的应用

王力（Neo Wang） ***Central FAE***

摘要

Jacinto™ 7 TDA4 系列处理器是 TI 公司基于 Keystone 架构推出的最新一代汽车处理器，主要致力于辅助驾驶系统（ADAS）和自动驾驶（AD）领域芯片解决方案。TDA4 系列汽车处理器基于片上多核异构芯片架构，16nm 系统工艺不仅为芯片带来了高系统集成度，而且大幅降低了多功能高级汽车平台设计所需的外设复杂度以及成本。性能上，在提供高达 ASIL-D 的系统功能安全等级支持的同时，以领先于市场的性能/功耗比为深度学习/视觉加速提供了卓越的处理能力。

ECC 校验作为功能安全保证最重要的组成部分之一，广泛的应用于车载处理器的片内接口、IP 总线以及存储介质中，用于提高系统的功能安全以及稳定性，其中用于执行程序、存放缓存以及交换数据的系统 DDR 的 ECC 校验则尤为重要。传统芯片解决方案必须额外外挂一片 DDR 存储器用来存放系统 DDR 计算 ECC 所得到的汉明码，故必须要求 SoC 具有两路以上的 DDR 通道，而且带来额外的成本负担。而 Jacinto™ 7 TDA4 系列处理器在芯片设计上采用 inline ECC，能够在仅挂载一片系统 DDR 的前提下，将 ECC 计算得到的汉明码和原数据在 DDR 上以交错间隔的方式存储，这样数据都在同一片存储设备上，不仅大大简化了数据处理流程，而且大幅缩减了系统设计所带来的成本负担。

本文对 TDA4 系列处理器中的 DDR inline ECC 进行原理介绍，并针对不同应用场景提供了两种不同的 DDR inline 的使能方法以及测试对比，进一步提高了 TDA4 汽车处理器的功能安全保证。

目录

图

表

1. 系统介绍

双倍数据速率同步动态随机存取内存（DDR SDRAM）技术由于其高密度，架构简单以及低延迟低功耗等优势，目前已经几乎成为所有应用场景下的主内存，特别是在汽车电子领域，对 DDR 的高效性，安全性则有着更近一步的要求，其在 TDA4 系列 SoC 中大致内存系统框图如图 1 所示。

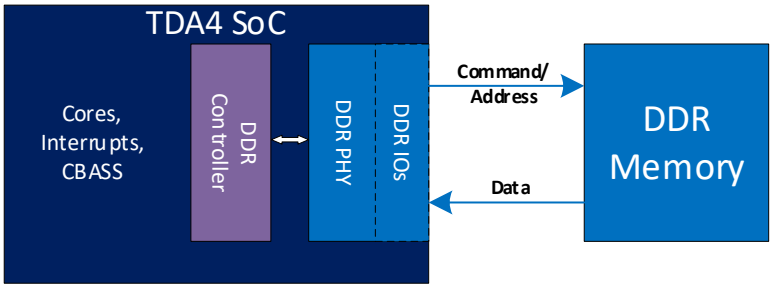

图 1 TDA4 中内存子系统框图

在汽车处理器应用中，DDR 承载着绝大多数应用的运行，但是 DDR 内存系统就像任何电子系统一样，不可避免的会受到系统软件或硬件设计上的缺陷，甚至纹波噪声等外界因素的影响，导致数据发生错误/偏移，进而导致系统出错甚至奔溃，这在要求高功能安全的汽车应用中是不可接受的。

为了检测并解决在运行时遇到的这些内存错误，内存子系统必须具有内存 RAS（可靠性、可用性和诊断性）功能，从而能在内存发生错误时进行检测并修正。其中 ECC 作为 RAS 中较常用的一种方案，可以对 DDR 数据进行计算并得到汉明码，在下一次读取时再次进行实时计算并与汉明码进行对比，从而实现单比特纠错以及双比特检错（SECDED）。

1.1 传统 DDR ECC 实现方案弊端

传统 DDR ECC 方案通常基于标准 DDR4/DDR5 来实现并采用 side-band ECC 方案，如图 2 所示为 side-band ECC 方案的系统框图，其 DDR ECC 数据不能和原 DDR 数据同时存储，所以一般会通过多个 DDR 信道连接多片 DDR 存储设备来分别作为系统运行内存以及 ECC 汉明码存储介质。

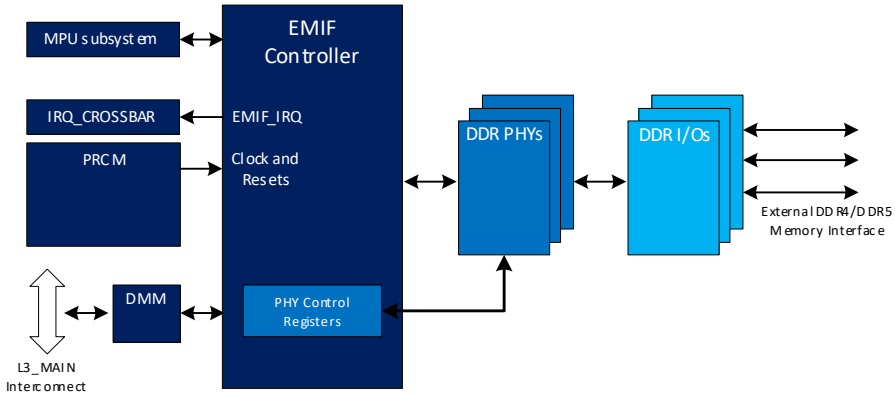

图 2 传统 Side-band ECC 系统框图

在系统启动初期，在完成对 DDR 的初始化之后，系统会对指定进行 ECC 保护的 DDR 区域进行 pattern 填充，并基于此 pattern 进行 ECC 计算得到汉明码，并分别将汉明码以及原数据存储到两片不同的 DDR 存储设备中。以 32bit DDR 的 EMIF 接口为例，对于每 32bit 数据宽度，需要增加 4bit 数据作为 ECC 存储。

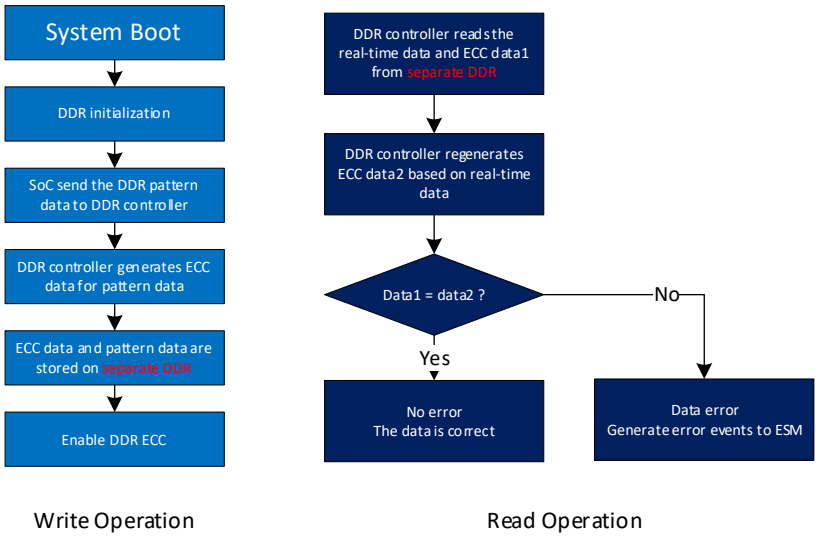

图 3 Side-band DDR ECC 读写流程

在完成 DDR 和 ECC 的初始化以及使能之后，若系统没有重新复写过 DDR 内容，系统运行期间进行读操作时，会读取实时的 DDR 数据以及上一次计算得到的 ECC data1 数值，并基于此实时的 DDR 数据进行计算得到 ECC data2。若两次计算得到的 data1 和 data2 相同，则视为数据没有发生变化，不产生 ECC 错误；若两次计算得到的数据不同，则认为数据发生变化，产生 ECC 错误事件输出到 TDA4VM88 的 ESM 模块中，从而对 ECC 错误进行下一步的处理。如图 3 所示，为传统 Side-band DDR ECC 工作时读写操作流程。

若在此期间系统有复写过 DDR 数据，则 DDR 控制器会首先自动根据复写后的数据产生 ECC data1 并将其更新到 ECC 存储的 DDR 中，并重复上述步骤进行校验。

传统的 DDR ECC 必须要挂载两片以上的 DDR，这就必须要求 SoC 芯片必须支持两路以上的 EMIF 接口，并且额外的 DDR 也会带来硬件设计以及软件开发上的成本负担。

1.2 TDA4 Inline ECC 介绍

Jacinto™ 7 系列 SoC 支持 LPDDR4 内存，LPDDR4 具有更高的电源效率以及更低的功耗，所以更适用于嵌入式处理器平台。如图 4 所示为 TDA4VM 中 DDR 子系统的结构框图，其中 TI inline ECC 模块内部具有一个 cache 缓存，用来把 DDR 数据以及 ECC 数据写到 DDR 存储器中之前进行对齐。

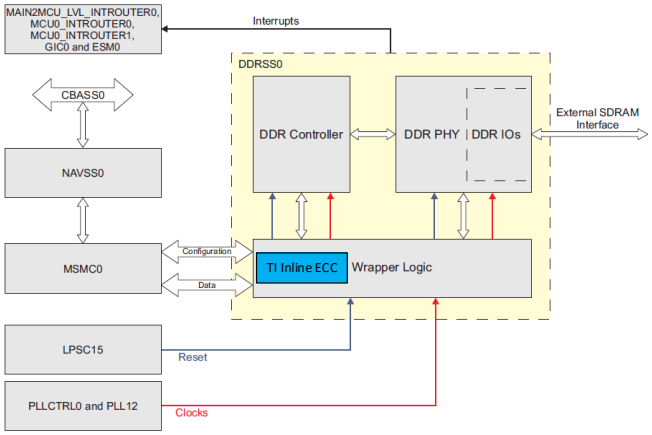

图 4 TDA4VM DDR 子系统框图

同样的，对 DDR 进行 Inline ECC 保护时，也需要在 SOC 内部 MSMC2DDR 中对 Inline ECC 进行使能并填充 pattern，以及基于此 pattern 计算默认汉明码。在初始化完毕后进行读操作时，也会实时计算汉明码并与默认汉明码进行比较，从而判断是否触发 ECC error。其读写流程如图 5 所示。

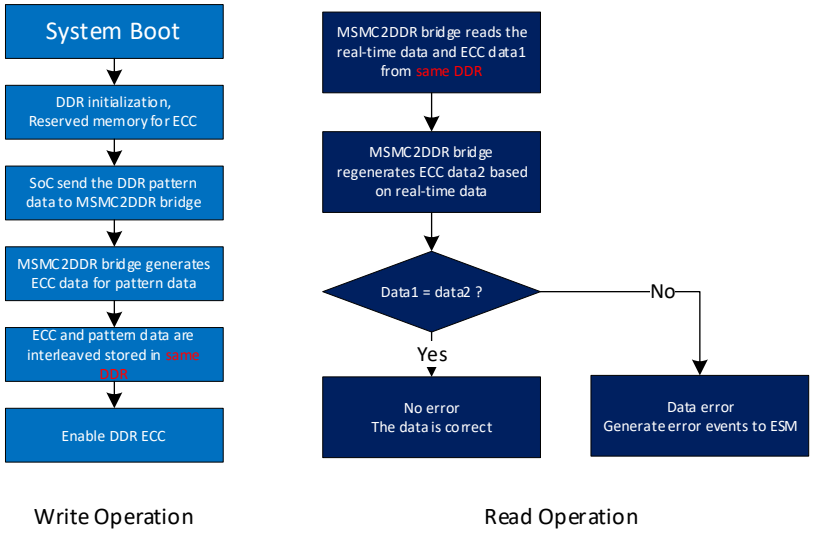

图 5 DDR Inline ECC 读写流程

值得注意的是，Inline ECC 不需要额外的 DDR 信道来存储汉明码数据，而是将汉明码数据和原 DDR 实际数据存储在同一片 DDR 中，其中每 8bit DDR 数据计算得到 1bit DDR ECC 汉明码，两者以交错的形式存储在 DDR 存储器中。并且此存储以及映射关系由 Inline ECC 硬件完成，从系统以及软件的角度来看，其呈现的 DDR 内存中的 8/9 数据单位与下一个 8/9 数据单位为连续字节地址，但实际上中间硬件中由 1/9 的汉明码数据，只是软件无法访问而已。虽然软件无法访问，但是此物理存储是实际存在的，所以 DDR 在初始化时，需要预留出预计 DDR ECC 区域大小 1/9 的内存大小，作为 DDR ECC 计算结果的存储空间。其存储/映射关系如图 6 所示。

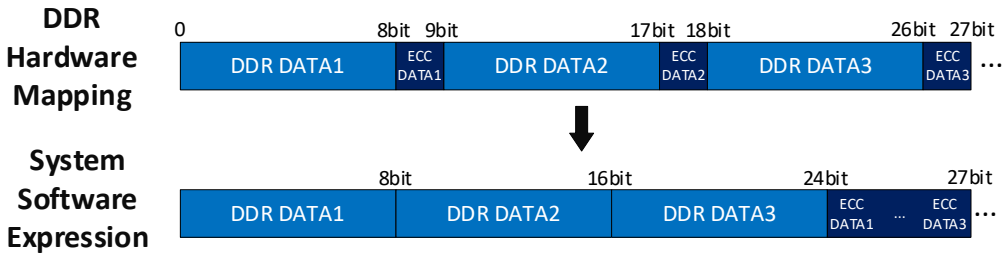

图 6 DDR Inline ECC 存储映射

TDA4 Inline ECC 可以在一路 DDR 信道以及一片 DDR 存储设备的前提下，实现 DDR ECC 的单比特纠错和双比特检错，大幅缩减了开发以及设计成本。

2. TDA4 Inline ECC 解决方案

DDR ECC 需要在系统开始运行之前对 DDR 进行初始化以及 ECC 使能，才能实现对 DDR 运行时的数据进行检错纠错。但是由于 DDR 在首次上电初始化时内存单元可能存在随机取值，所以需要首先对 DDR 需要做 ECC 保护的区域进行 pattern 填充，并基于此 pattern 值进行 ECC 汉明码计算，并使能 DDR ECC。

TDA4 作为多核异构高算力的处理器芯片，其内部各个核心会根据指定的启动顺序进行启动，一般的，将 TDA4 内部处理单元分为高算力单元 A 核以及实时核 R 核，分别会运行高层操作系统（例如 Linux，QNX）以及实

时操作系统（例如 TI-RTOS，AUTOSAR）。系统可以在 R5F 核心中使用 SPL 来为 A72 核加载 u-boot 以及操作系统，也可以使用快起方案，通过 R5F 运行 SBL 对 DDR 进行初始化以及做 DDR 的 ECC 使能。其中 SPL 属于 Linux boot loader，其由 u-boot 代码通过 32 位编译器编译得到；而 SBL 属于 TI RTOS boot loader，其由 TI PDK 驱动代码编译而来。本章会基于 SDK7.1 软件版本以及 TI TDA4 EVM 硬件环境对这两种启动流程分别介绍 TDA4 Inline ECC 的使能方法。

2.1. Enable ECC in R5F SPL

首先由于 Inline ECC 使能后会在同一片 DDR 中占用受保护区域 1/9 大小的内存用于存放 ECC 数据，所以需要在 u-boot 中提前预留出这块地址大小内存，避免被与 Linux 系统内存产生冲突。如下所示，为了计算方便以及冗余考虑，这里通过调整 DDR 的 bank 大小，将默认的 4G DDR 大小缩减到了 3.5G，预留了 512MB 大小内存供 DDR ECC 数据使用。

```
diff --git a/board/ti/j721e/evm.c b/board/ti/j721e/evm.c
--- a/board/ti/j721e/evm.c
+++ b/board/ti/j721e/evm.c
@@ -73,8 +73,8 @@ int dram_init_banksize(void)
 #ifdef CONFIG_PHYS_64BIT
                /* Bank 1 declares the memory available in the DDR high region */
                gd->bd->bi_dram[1].start = CONFIG_SYS_SDRAM_BASE1;
-               gd->bd->bi_dram[1].size = 0x80000000;
-               gd->ram_size = 0x100000000;
+               gd->bd->bi_dram[1].size = 0x40000000;
+               gd->ram_size = 0xE0000000;
 #endif
```

然后在代码“~/ti-processor-sdk-linux-j7200-evm-07_01_01_04/board-support/u-boot-2020.01+gitAUTOINC+3c9ebdb87d-g3c9ebdb87d/drivers/ram/k3-j721e/ k3-j721e-ddrss.c”中通过控制 DDR 中的 ECC Engine 来实现对 DDR ECC 的使能。其使能流程以及对应操作的寄存器如图 7 所示。所有 API 都是通过控制对 DDR ECC Engine 中的寄存器进行操作实现，其中在使用 clear_mem_sect()对 DDR ECC 区域做 pattern 填充时，使用的是逐 8 比特进行赋值。

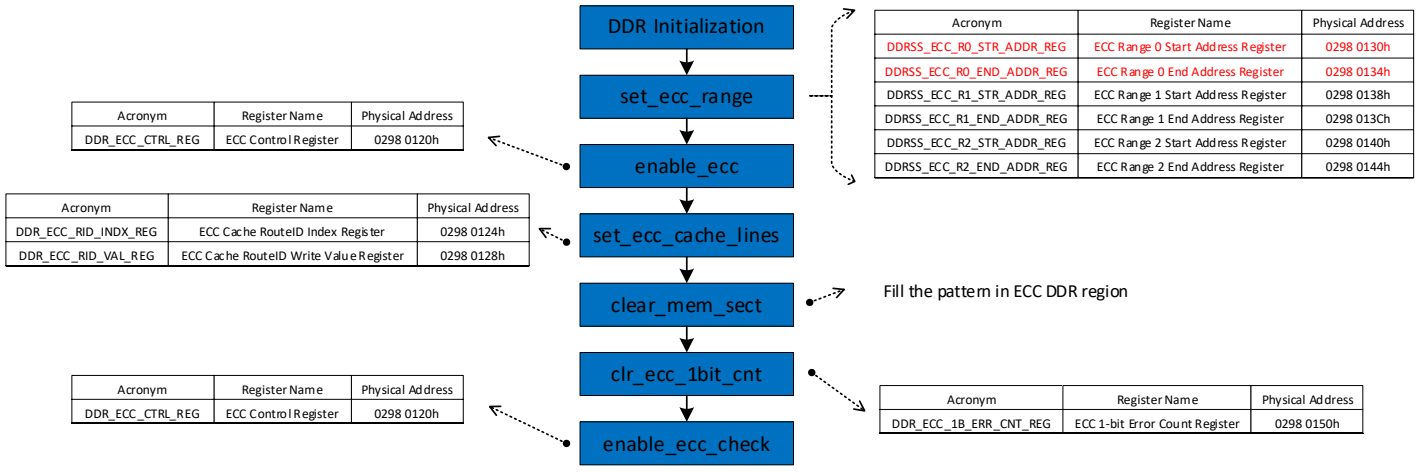

图 7 R5F SPL 中使能 DDR Inline ECC 流程

在本示例代码中，默认 DDR ECC 保护区域大小为 2G，区域为 0x8000 0000 ~ 0xFFFF FFFF。经过上述修改后，按照下述流程进行编译并将二进制文件更新到系统中。

```
# cd ~/ti-processor-sdk-linux-j7200-evm-07_01_01_04
# make u-boot
# cp ~/ti-processor-sdk-linux-j7200-evm-07_01_01_04/board-support/u-boot_build/a72/tispl.bin  /media/<user>/BOOT
# cp ~/ti-processor-sdk-linux-j7200-evm-07_01_01_04/board-support/u-boot_build/r5/tiboot3.bin /media/<user>/BOOT
```

2.2. Enable ECC in R5F SBL

为了实现系统快启以及更早的对 DDR 做 ECC 保护，可以在 MCU1_0 对系统以及 DDR 做初始化的时候进行 ECC 操作，首先同样的，需要参照 2.1 中所示，在 HLOS 上将 DDR ECC 区域大小预留出来。

此外，由于默认 SBL 代码没有使能 DDR ECC，所以需要按照下列改动在 SBL 中使能 ECC 校验。

```
diff --git a/packages/ti/boot/sbl/board/k3/sbl_main.c b/packages/ti/boot/sbl/board/k3/sbl_main.c
index ebcb272..dab0168 100644
--- a/packages/ti/boot/sbl/board/k3/sbl_main.c
+++ b/packages/ti/boot/sbl/board/k3/sbl_main.c
@@ -514,7 +520,7 @@ int main()
 #if defined(SBL_ENABLE_DDR) && defined(SBL_ENABLE_PLL) && defined(SBL_ENABLE_CLOCKS)  && !defined(SBL_SKIP_SYSFW_INIT)
     SBL_log(SBL_LOG_MAX, "Initlialzing DDR ...");
     SBL_ADD_PROFILE_POINT;
-    Board_init(BOARD_INIT_DDR);
+    Board_init(BOARD_INIT_DDR | BOARD_INIT_DDR_ECC);
     SBL_log(SBL_LOG_MAX, "done.\n");
 #endif
```

接着需要在 SBL 中增加 DDR ECC 的函数实现，本小节提供两种方法实现 DDR ECC，一种使用 memory copy 的方式对 ECC pattern 进行填充，另一种使用 DDR controller 中的 BIST Engine 对 ECC pattern 进行填充，两种方法各有优劣。

2.2.1 ECC Engine 实现 DDR ECC Pattern Load

在 SBL 的代码中，已经对 DDR ECC Engine 的寄存器做了封装，类似于在 2.1 小节中通过 R5F SPL 使能 DDR ECC 的方法，通过对“SDK_INSTALL/pdk/package/ti/board/src/j721e_evm/board_ddr.c”中 emif_ConfigureECC 函数进行配置并调用，可实现对 DDR ECC Engine 的配置，如下所示。

```
static Board_STATUS emif_ConfigureECC(void)
{
    ..........
    emifCfg.bEnableMemoryECC = true;
    emifCfg.pMemEccCfg.startAddr[0] = BOARD_DDR_START_ADDR-BOARD_DDR_START_ADDR;
    emifCfg.pMemEccCfg.endAddr[0] = BOARD_DDR_ECC_END_ADDR-BOARD_DDR_START_ADDR;
    .........
      /* Prime memory with known pattern */
      for (memPtr = BOARD_DDR_START_ADDR; memPtr < BOARD_DDR_ECC_END_ADDR; memPtr += 4)
      {
         *((volatile uint32_t *) memPtr) = memPtr;
      }
      /* Make sure the write is complete by writeback */
      CacheP_wbInv((const void *)BOARD_DDR_START_ADDR, BOARD_DDR_ECC_END_ADDR-BOARD_DDR_START_ADDR);
      /* Clears ECC errors */
      CSL_emifClearAllECCErrors((CSL_emif_sscfgRegs *)CSL_COMPUTE_CLUSTER0_SS_CFG_BASE);
    ..........
    return status;
}
```

在本示例中利用 SBL 使能 DDR ECC 时，可以看出使用的逐 32bit 进行 pattern 填充，即利用本 DDR 寄存器地址作为 pattern 数值填充。如下所示，做 DDR ECC 保护的区域可在“SDK_INSTALL/pdk/package/ti/board/src/j721e_evm/include/board_cfg.h”中进行设定。然后在检查完 cache 一致性确保数据写完毕之后，清除 ECC error 位，最后使能 DDR ECC。

2.2.2 BIST Engine 实现 DDR ECC Pattern Load

进一步的，SBL 依然可以通过配置 DDR controller 中的 BIST Engine 来实现，BIST Engine 在 DDR Controller 中的构成如图 8 所示，BIST Engine 模块可以支持 MOVI3N 以及 MOVI 算法以及外部软件编程，并通过外部软件控制 DDR controller 的初始化以及实现对 ECC lane 做 ECC 的相关配置。

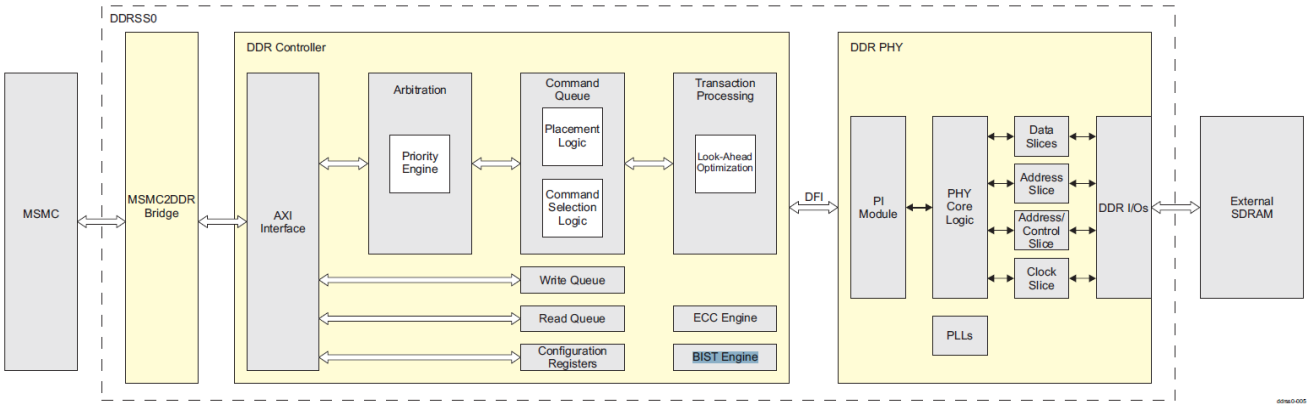

图 8 TDA4VM 中 DDR 子系统分解图

通过 BIST Engine 来预加载 DDR ECC 保护区域 pattern 以及 DDR ECC 使能的流程图以及寄存器说明如图 9 所示，详细寄存器说明请参照 TDA4 设计手册，此流程和 2.1 以及 2.2.1 小节小节通过 ECC Engine 来操作的目的是一样的，区别在于作用形式和控制方法的差异，从而带来效率的差别，具体测试结果详见第三章。

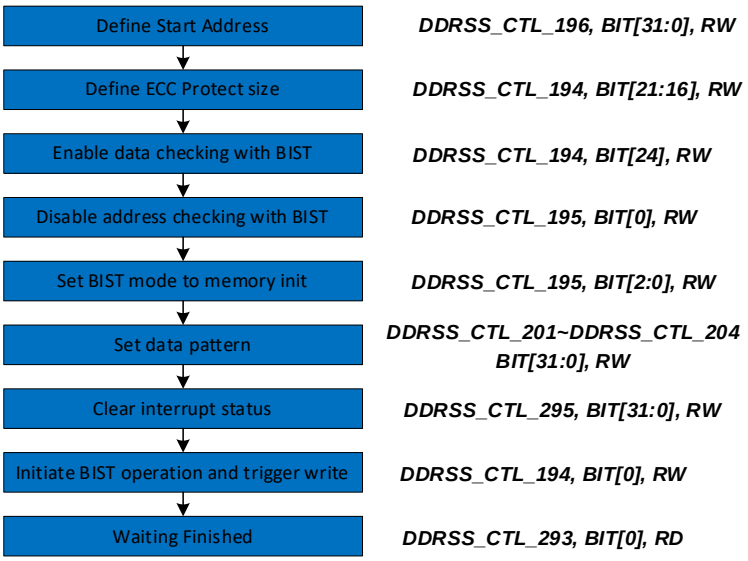

图 9 BIST Engine 实现 DDR inline ECC 流程图

针对上述流程框架设计，本示例代码如下所示，从 DDR 存储器 0x8000 0000 地址开始，使能 DDR ECC 区域大小为 2^{16} bit = 2GB 大小，其中对此区域进行初始化时，每 512MB 内存做一次差别化 pattern 填充，分别填充 DDR_DATA 0x11111111，DDR_DATA1 0xAAAAAAAA，DDR_DATA2 0x55555555 以及 DDR_DATA3 0xFFFFFFFF。

```
/************************macro define************************/
#define DDR_START_ADDR 0x80000000
#define DDR_SIZE 16
#define DDR_DATA 0x11111111
#define DDR_DATA1 0xAAAAAAAA
#define DDR_DATA2 0x55555555
#define DDR_DATA3 0xFFFFFFFF

#define J7ES_DDR_SS_BASE_MCUStart               0x02990000U
#define DDRSS_CTL_194__SFR_OFFS    0x308
#define DDRSS_CTL_195__SFR_OFFS    0x30c
#define DDRSS_CTL_196__SFR_OFFS    0x310
#define DDRSS_CTL_200__SFR_OFFS    0x320
#define DDRSS_CTL_201__SFR_OFFS    0x324
#define DDRSS_CTL_202__SFR_OFFS    0x328
#define DDRSS_CTL_203__SFR_OFFS    0x32c
#define DDRSS_CTL_204__SFR_OFFS    0x330
#define DDRSS_CTL_293__SFR_OFFS    0x494
#define DDRSS_CTL_295__SFR_OFFS    0x49c

Board_STATUS BISTEngine_ECC()
{
  Board_STATUS   status    = BOARD_SOK;

  BOARD_DEBUG_LOG("Start BISTEngine_ECC test!\r\n");
   // Define start address
   HW_WR_REG32_RAW(J7ES_DDR_SS_BASE_MCUStart + DDRSS_CTL_196__SFR_OFFS, (DDR_START_ADDR - 0x80000000));
   // Define memory size
   HW_WR_FIELD32_RAW(J7ES_DDR_SS_BASE_MCUStart + DDRSS_CTL_194__SFR_OFFS, 63 << 16, 16, DDR_SIZE);
   // Enable data check
  HW_WR_FIELD32_RAW(J7ES_DDR_SS_BASE_MCUStart + DDRSS_CTL_194__SFR_OFFS, 1 << 24, 24, 1);
   // Disable addr check
   HW_WR_FIELD32_RAW(J7ES_DDR_SS_BASE_MCUStart + DDRSS_CTL_195__SFR_OFFS, 1, 0, 0);
   // Set BIST Mode to Mem Init
   HW_WR_FIELD32_RAW(J7ES_DDR_SS_BASE_MCUStart + DDRSS_CTL_200__SFR_OFFS, 7, 0, 4);
   // Set Data Patterns
   HW_WR_REG32_RAW(J7ES_DDR_SS_BASE_MCUStart + DDRSS_CTL_201__SFR_OFFS, DDR_DATA);
   HW_WR_REG32_RAW(J7ES_DDR_SS_BASE_MCUStart + DDRSS_CTL_202__SFR_OFFS, DDR_DATA1);
   HW_WR_REG32_RAW(J7ES_DDR_SS_BASE_MCUStart + DDRSS_CTL_203__SFR_OFFS, DDR_DATA2);
   HW_WR_REG32_RAW(J7ES_DDR_SS_BASE_MCUStart + DDRSS_CTL_204__SFR_OFFS, DDR_DATA3);
   // Clear Interrupt
   HW_WR_FIELD32_RAW(J7ES_DDR_SS_BASE_MCUStart + DDRSS_CTL_295__SFR_OFFS, 1 << 11, 11, 1);
   // Trigger Write
   HW_WR_FIELD32_RAW(J7ES_DDR_SS_BASE_MCUStart + DDRSS_CTL_194__SFR_OFFS, 1, 0, 1);
   // Wait for Finish
  while(HW_RD_FIELD32_RAW(J7ES_DDR_SS_BASE_MCUStart + DDRSS_CTL_293__SFR_OFFS, 1 << 11, 11) !=1)    BOARD_DEBUG_LOG("keep waiting finish!\r\n");
   // Clear Interrupt
  HW_WR_FIELD32_RAW(J7ES_DDR_SS_BASE_MCUStart + DDRSS_CTL_295__SFR_OFFS, 1 << 11, 11, 1);

  BOARD_DEBUG_LOG("DDR Data Initialization and ECC enable Complete!\n");
  return status;
}
```

经过上述修改后，通过下面流程进行编译并将二进制文件更新到系统中。

```
# cd ~/ti-processor-sdk-rtos-j721e-evm-07_01_00_11/pdk_jacinto_07_01_00_45/packages/ti/build

# make pdk_libs

# make BOARD=j721e_evm CORE=mcu1_0 BUILD_PROFILE=release sbl_mmcsd_img

# cp ~/ti-processor-sdk-rtos-j721e-evm-
07_01_00_11/pdk_jacinto_07_01_00_45/packages/ti/boot/sbl/binary/j721e_evm/mmcsd/bin/sbl_mmcsd_img_mcu1_0_release.tiimage
/media/<user>/BOOT/tiboot3.bin
```

3. 测试结果

3.1. SPL 测试结果

通过在使能 DDR ECC 对 2G 大小内存的保护前后，基于 SDK7.1 默认的 AVP4 usecase 作为测试标准，其测试结果如图 10 所示。

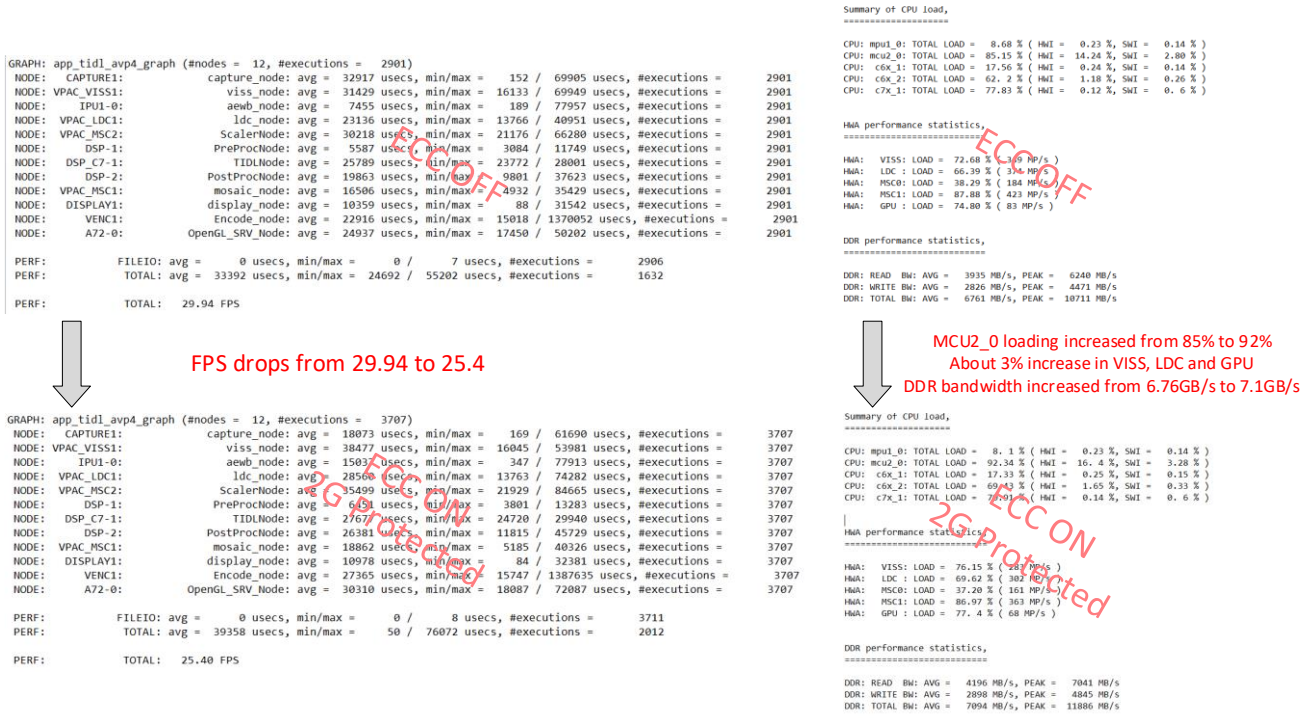

图 10 SPL 中使能 DDR Inline ECC 前后测试结果对比

根据上述测试数据可以看出，使能 DDR Inline ECC 后，会导致 DDR 可用内存缩减，DDR 带宽占用增加，原运行在 DDR ECC 保护区域内存上的应用 loading 增高。

所以综合系统功能安全设计以及性能平衡考虑，建议仅对数据信息敏感的 DDR 区域做 Inline ECC 保护，而非对全局 DDR 进行 ECC 保护。

3.2. SBL 测试结果

第 2.2 小结中 SBL 对 DDR Inline ECC 使能的两种方法，区别在于作用的引擎不同，分别调用的是 ECC Engine 以及 BIST Engine，这两种最大的区别在于 DDR ECC 使能之前，对 DDR ECC 保护区域的 pattern 填充方式不一样，

而不同的填充方式对系统带来最大的影响在于启动时间。分别使用 memory copy 以及 BIST 进行填充时，针对不同 DDR ECC 内存大小，系统的启动时间如表 1 所示：

表 1 SBL 中不同方法填充 DDR ECC Pattern 耗时对比

Cost Time / DDR ECC Size / Method	64MB	256MB	2GB
ECC Engine (Memory Copy Fill)	513ms	1.72s	13s
BIST Engine (BIST Fill)	16.6ms	169ms	711ms

根据上述测试数据可以看出，虽然使用 ECC Engine 对 DDR ECC 进行使能会更加简便，但使用 BIST Engine 对 DDR ECC 区域做初始化 pattern 会大幅减少初始化时间，从而降低启动耗时。所以在启动时间严格的应用设计中，使用 BIST Engine 对 DDR ECC 进行使能会更加高效。

4. 总结

DDR ECC 作为汽车功能安全设计中不可或缺的一环，TDA4 系列处理器提供了更加高效，成本大幅缩减的 DDR Inline ECC 架构，本文针对不同的启动方式以及应用场景提供了不同的使能方式。总的来说，对 DDR Inline ECC 进行使能，能提高系统功能安全，但同时会占用部分 DDR 内存区域以及运行带宽，并略微增加系统启动时间。建议系统设计师对功能安全要求以及性能表现进行综合考虑，仅对数据敏感区域使能 DDR ECC 保护。

5. 参考文献

1. TDA4VM Jacinto™ Processors for ADAS and Autonomous Vehicles Silicon Revisions 1.0 and 1.1 datasheet (Rev. J)
2. DRA829/TDA4VM/AM752x Technical Reference Manual (Rev. B)
3. TI's Inline ECC for DDR
4. Error Correction Code (ECC) in DDR Memories
5. Built-in self-test

重要声明和免责声明

TI“按原样”提供技术和可靠性数据（包括数据表）、设计资源（包括参考设计）、应用或其他设计建议、网络工具、安全信息和其他资源，不保证没有瑕疵且不做出任何明示或暗示的担保，包括但不限于对适销性、某特定用途方面的适用性或不侵犯任何第三方知识产权的暗示担保。

这些资源可供使用 TI 产品进行设计的熟练开发人员使用。您将自行承担以下全部责任：(1) 针对您的应用选择合适的 TI 产品，(2) 设计、验证并测试您的应用，(3) 确保您的应用满足相应标准以及任何其他功能安全、信息安全、监管或其他要求。

这些资源如有变更，恕不另行通知。TI 授权您仅可将这些资源用于研发本资源所述的 TI 产品的应用。严禁对这些资源进行其他复制或展示。您无权使用任何其他 TI 知识产权或任何第三方知识产权。您应全额赔偿因在这些资源的使用中对 TI 及其代表造成的任何索赔、损害、成本、损失和债务，TI 对此概不负责。

TI 提供的产品受 TI 的销售条款或 ti.com 上其他适用条款/TI 产品随附的其他适用条款的约束。TI 提供这些资源并不会扩展或以其他方式更改 TI 针对 TI 产品发布的适用的担保或担保免责声明。

TI 反对并拒绝您可能提出的任何其他或不同的条款。

邮寄地址：Texas Instruments, Post Office Box 655303, Dallas, Texas 75265