TEXAS INSTRUMENTS

Application Report
ZHCA648 – Jan 2016

一种提高存储于 DFLASH 数据稳定性的方法

Jack Tan, Sundy Xu High Voltage Power Solutions

摘要

UCD3138 系列产品有 4 类存储器，Program Flash，Data Flash，RAM，ROM。与控制相关的系统参数，比如，环路参数，参考电压，保护阈值等需要在线更新的参数，一般都是存储在 Data Flash。通常在生产过程中，这部分参数需要校验，校验后得到的参数最终保存到 Data Flash，这样方便操作。在保存数据之前，Data Flash 的数据要先被擦除，这就存在 Data Flash 的数据被擦除后没有数据的可能性。

本文主要提供一种提高系统数据稳定性的方案，并且以 PFC 代码为例，给出了相应的代码示例。同时也提供了相应的实验验证结果。

目录

1 引言 2
2 系统参数的处理过程 2
3 在 PLASH 中增加系统参数变量 3
  3.1 修改 cmd 文件 3
  3.2 定义存放系统参数的结构体变量 4
  3.3 定义变量存储于相应的数据区 5
  3.4 启动检测 Data flash 的数据 5
  3.5 添加 Pmbus 通信代码 6
  3.6 Data Flash 更新数据的操作时序 9
4. 实验验证 10
  4.1 通过 MAP 文件查看系统参数变量 10
  4.2 通过 GUI 发送命令更新系统参数 10
  4.3 不烧录 Data Flash 的数据 11
5 结论 13
参考文献： 13

1 引言

UCD3138 采用的内核为 ARM7，该核是冯诺依曼架构，所有存储器数据访问都是通过一条总线，所以读取任何存储器的数据的速度都是相同的。TI 的 UCD3138 的存储器分为 Program Flash，Data Flash，RAM，ROM 与外设寄存器。其中，Program Flash 通常用来存储程序代码也就是可执行的指令数据；与控制相关的系统参数，比如，环路参数，参考电压，保护阈值等需要在线更新的参数，一般都是存储在 Data Flash。通常在生产过程中，这部分参数需要校验，校验后得到的参数最终保存到 Data Flash，这样方便操作。全局变量，局部变量，堆栈等都是保存在 RAM 存储器；ROM 是 UCD3138 的启动代码与 Pmbus boot code，这部分数据是固化在芯片内部，用户是不能够修改。

本文主要介绍如何提高系统参数存储过程中的鲁棒性。

2 系统参数的处理过程

如上面所述，TI 的 EVM 的代码都是将系统相关参数存放在 Data Flash，这样，在工厂生产时，得到正确的检验参数后，会将其保存到 Data Flash 中，而先前的参数会被清除。在 UCD3138 再次上电后，使用最新存储在 Data Flash 的参数，这样就实现了参数的更新。

UCD3138 的 Data Flash 被分成了 A 与 B 两个区域，当芯片启动后，在 main 函数的启始位置会检测 Data Flash 数据的完整性，如果某一区域的数据的 checksum 不正确，那么该区域的数据则不会使用，并且会被擦除。如果 checksum 正确，则会使用该参数。为了便于通过 GUI 更新参数，demo 软件中在 RAM 与 Data Flash 中分别定义了一个元素相同的结构体变量。如下图所示，当 checksum 正确后，首先，Data Flash 的变量值会被拷贝到 RAM 中的变量。然后，将 RAM 中的变量经过一定的数学转换变成寄存器对应的值，同时，这部分 RAM 的变量值也会被送到 Pmbus 主机（TI 的 GUI 也是一个主机）监控与显示。当用户想通过 GUI 更新系统参数时，这部分目标参数会被写入到 RAM 的变量中，然后则会将其转换成寄存器对应的值。当调试的参数可以满足系统的需求时，就可以通过 GUI 下发一个“store ram to flash”的命令，UCD3138 接收到该命令后，RAM 中变量的值就会写到 Data Flash 中。所有这部分对 GUI 通信数据的处理都是 UCD3138 的软件完成。

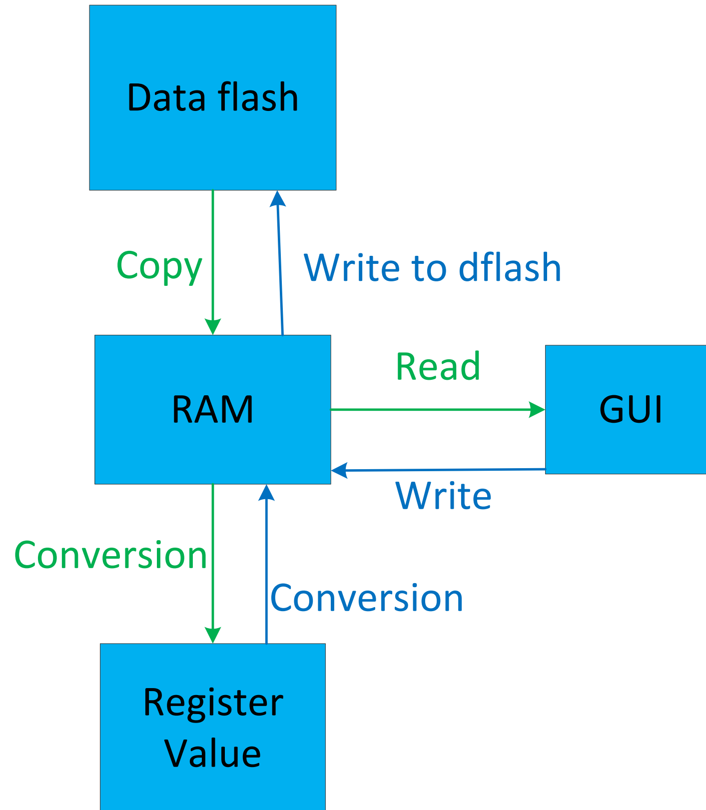


图 1. 系统参数的流程图

3 在 PLASH 中增加系统参数变量

由上述分析可知，Data Flash 的 A 与 B 区的参数都有可能会被更新，在更新之前这部分数据都会被擦除。而且如果启动过程中，为了确保所使用的参数正确，都会先检查数据的完整性。如果任意 A 与 B 区的数据不完整，则会被擦除。所以为了使参数保存更为稳定，这里推荐保存一份默认参数（也即没有校正之前的数据）在 Program Flash，这样可以有效确保在 Data Flash 数据不正确的情况下，电源仍然可以正常启动。如下是以 PFC 程序为例，添加该方法的软件实现步骤：

3.1 修改 cmd 文件

定义 A 与 B 区在 DATA FLASH，将.const 区域存放在 PFLASH

```
SECTIONS
{
    … …
    .const           : {} > (PFLASH align(16)) /* Constant data          */
    … …
    /*-----------------------------------------------------------------*/
    /* D-Flash    2K     0x18800 - 0x18FFF                              */
```

```
    /*-----------------------------------------------------------------------*/

    .CONFIG_A    : {} > (DFLASH align(32))
    .CONFIG_B    : {} > (DFLASH align(32))
    … …
    … …
}
```

3.2 定义存放系统参数的结构体变量

注意：该结构体变量的大小一定要是 **word（4 个字节）的整数倍**。比如，一个结构大小占有 10 个字节，一定得补充 2 个 reserved bytes。否则将会导致更新 data flash 的数据不成功。

```
//for current loop coefficients

typedef struct
{
	struct PFC_SETPOINT_STRUCTPFC_SETPOINT;	// PFC setpoint configuration
	struct PFC_CAL_STRUCT		PFC_CAL;		// PFC calibration
	struct PI_GAINS_STRUCT		PI_GAINS;		// voltage loop gains
	union COEFCONFIG_REG		COEFCONFIG;		// Coefficient Configuration Register
	union FILTERKPCOEF0_REG	FILTERKPCOEF0;	// Filter KP Coefficient 0 Register
	union FILTERKPCOEF1_REG	FILTERKPCOEF1;	// Filter KP Coefficient 1 Register
	union FILTERKICOEF0_REG	FILTERKICOEF0;	// Filter KI Coefficient 0 Register
	union FILTERKICOEF1_REG	FILTERKICOEF1;	// Filter KI Coefficient 1 Register
	union FILTERKDCOEF0_REG	FILTERKDCOEF0;	// Filter KD Coefficient 0 Register
	union FILTERKDCOEF1_REG	FILTERKDCOEF1;	// Filter KD Coefficient 1 Register
	union FILTERKDALPHA_REG	FILTERKDALPHA;	// Filter KD Alpha Register
	union FILTERNL0_REG		FILTERNL0;		// Filter Non-Linear Limit 0 Register
	union FILTERNL1_REG		FILTERNL1;		// Filter Non-Linear Limit 1 Register
	union FILTERNL2_REG		FILTERNL2;		// Filter Non-Linear Limit 2 Register
	union FILTER_MISC_REG		FILTERMISC;		// Nonlinear mode,AFE gain,oversample
}PFC_CONFIG_STRUCT;
```

3.3 定义变量存储于相应的数据区

在 A 与 B 的区域及 program flash 分别定义相同结构体类型的变量，并将默认值赋给 A 区与存储在 Program flash 的变量。

- 定义存储在 data flash 的变量

```
#pragma DATA_SECTION(pfc_config_constants_a, ".CONFIG_A");
volatile const PFC_CONFIG_STRUCT pfc_config_constants_a =
PFC_CONFIG_DEFAULT;

#pragma DATA_SECTION(pfc_config_checksum_a, ". CONFIG_A");
volatile const Uint32 pfc_config_checksum_a = 0x87654321;

#pragma DATA_SECTION(pfc_config_constants_b, ".CONFIG_B");
volatile const PFC_CONFIG_STRUCT pfc_config_constants_b;

#pragma DATA_SECTION(pfc_config_checksum_b, ".CONFIG_B");
volatile const Uint32 pfc_config_checksum_b;
```

- 定义存储在 program flash 的变量

```
const PFC_CONFIG_STRUCT pfc_config_hardcoded = PFC_CONFIG_DEFAULT;
```

编译器会将 `const` 变量自动存放在`.const section`.

3.4 启动检测 Data flash 的数据

每当芯片启动时，检测 Data Flash 的数据。大概软件流程如下：

- 如果 A 区数据正确，则使用该区域的数据；
- 如果 B 区数据正确，则使用该区域的数据；
- 如果 A 与 B 的参数都不正确，则使用 program flash 中的变量数据

如果想要了解更多软件细节，可以到**www.ti.com**下载PFC的代码。

```
Uint8 pmbus_write_restore_default_all(void)//load configuration from data
flash to registers
{
	Uint32		checksum;

	// Pointers to structures in Data Flash
	volatile const  PFC_CONFIG_STRUCT*	pfc_config_ptr;
	volatile const  EMETER_CONFIG*			emeter_config_ptr;


	// 如果A区数据的完整性
	checksum = calculate_dflash_checksum((Uint8*)&pfc_config_constants_a,
(Uint8*)&pfc_config_checksum_a);
	// A zero checksum only occurs when the segment is all zeroes, which is
not valid.
```

```
        // If the calculated checksum is nonzero and matches the checksum in the
DFlash,
        // that segment is good, so use it.
        if(   (pfc_config_checksum_a == 0x87654321)   // Hardcoded exception for
parms written directly to data flash
                // (GUI download tool does not calculate checksum)
                ||((checksum != 0) && (checksum == pfc_config_checksum_a)) )
        // Checksum is valid and matches.
        {
                // Checksum A Good: Use default values from DFlash A
                pfc_config_ptr = &pfc_config_constants_a;
        }
        else
        {
        // 如果B区数据的完整性
        // Look in Data Flash B for valid values
                checksum =
calculate_dflash_checksum((Uint8*)&pfc_config_constants_b,
(Uint8*)&pfc_config_checksum_b);
                // A zero checksum only occurs when the segment is all zeroes,
which is not valid.
                // If the calculated checksum is nonzero and matches the checksum
in the DFlash,
                // that segment is good, so use it.
                if ((checksum != 0) && (checksum == pfc_config_checksum_b))
                {
                        // Checksum B Good: Use default values from DFlash B
                        pfc_config_ptr = &pfc_config_constants_b;
                }
                else// No valid values found in Data Flash.  Use hardcoded values
from PFlash instead.
                {
                        // Use hardcoded values from Program Flash
                        pfc_config_ptr = &pfc_config_hardcoded;
                }
        }
        memcpy((void *)&pfc_config_in_ram, (void *)pfc_config_ptr,
sizeof(pfc_config_constants_a));

        copy_configuration_to_registers(&Filter1Regs);

        return PMBUS_SUCCESS;
}
```

3.5 添加 Pmbus 通信代码

添加代码，实现可能与上位机通信，来控 data flash 的数据更新。

1. 在pmbus_write_message函数添加命令，可以用于实现将参数更新到data flash

```
// look at command byte from a write perspective
int32 pmbus_write_message(void)
{
```

```
        switch (pmbus_buffer[0])
        {
        … …
        case PMBUS_CMD_STORE_DEFAULT_ALL:
                return pmbus_write_store_default_all();
        … …
        }

}
```

2. 将RAM中的变量值，更新到Data Flash。这一步骤主要有以下几个过程：

- 步骤1：检测Data flash中的哪一个区域可以用于存储数据
- 步骤2：如果A区可用于存储数据，则将RAM的数据存储在该区，直接进入步骤5，否则进入下一步骤
- 步骤3：如果B区可用于存储数据，则将RAM的数据存储在该区，直接进入步骤5，否则进入下一步骤
- 步骤4：如果A与B都不能用于存储数据，直接擦除B区的数据。中断此次数据存储
- 步骤5：如果存储于data flash的数据失败，则中断本次数据存储，并且擦除本次已存储的数据

更多详细代码，可参考UCD3138 PFC demo 程序。

```
//============================================================================
// pmbus_write_store_default_all()
//      Store all PMBus-configurable variables from RAM to Data Flash.
//============================================================================
Uint8 pmbus_write_store_default_all(void)
{
        … …
        … …
        //检测A 区是否可用于存储数据
        if ((pfc_config_checksum_a ==
0xFFFFFFFF)&&(pfc_config_constants_a.FILTERKDCOEF0.bit.KD_COEF_0 == (signed
short)0xFFFF))      // Test Dflash A
        {
                // DFlash A is blank.
                //Store new values in DFlash A and erase DFlash B when done.
                dest_address_pfc_config    = &pfc_config_constants_a;
                dest_checksum           = &pfc_config_checksum_a;

                opposite_bank_start          = &pfc_config_constants_b;
        }
        //检测B 区是否可用于存储数据
        else if ((pfc_config_checksum_b ==
0xFFFFFFFF)&&(pfc_config_constants_b.FILTERKDCOEF0.bit.KD_COEF_0 == (signed
short)0xFFFF))      // Test DFlash B
        {
                // DFlash B is blank.
                // Store new values in DFlash B and erase DFlash A when done.
                dest_address_pfc_config    = &pfc_config_constants_b;
                dest_checksum           = &pfc_config_checksum_b;

                opposite_bank_start          = &pfc_config_constants_a;
        }
        //如果A与B区都不适合存储数据，擦除任意区域的数据
```

```
        else
        {
                if (pfc_config_checksum_b == 0xFFFFFFFF)
                {
                        dest_address_pfc_config    = &pfc_config_constants_b;

                }
                else
                {
                        dest_address_pfc_config    = &pfc_config_constants_a;
                }
                goto flash_write_failed;  // Fail: Destination bank not erased.
        }

        // 将RAM的数据更新到Data flash
        status = update_data_flash((void*)dest_address_pfc_config,
&pfc_config_in_ram, sizeof(pfc_config_in_ram));
        if (status != FLASH_SUCCESS)
        {
                goto flash_write_failed;  // Clean up after flash write failure
        }

        // 计算checksum，并将其存储到Data Flash中
        // Calculate checksum for selected Data Flash segment and write to Data
Flash
        checksum = calculate_dflash_checksum((Uint8*)dest_address_pfc_config,
(Uint8*)dest_checksum);
        status = update_data_flash((void*)dest_checksum, &checksum,
sizeof(checksum));
        if (status != FLASH_SUCCESS)
        {
                goto flash_write_failed;  // Clean up after flash write failure
        }

        {

                // ----- Bank written successfully.  Erase opposite bank. -----
                start_erase_task((void*)opposite_bank_start, bytes_to_erase);
                // If everything works, return success.
                return PMBUS_SUCCESS;
        }

flash_write_failed:        // <--- Destination for several goto's above.
        {
                // ----- This bank write failed.  Erase present bank and report
the failure.
                start_erase_task((void*)dest_address_pfc_config, bytes_to_erase);
                return (PMBUS_MEMORY_FAULT);            // Flash write failed
        }
}
```

3.6 Data Flash 更新数据的操作时序

有些程序在更新 data flash 的数据时会导致芯片复位，从而会使得数据更新失败。其主要原因是没有按照操作时序与要求。

在向 Data Flash 写数据或擦除时，要遵循以下几个原则：

- CPU 处于特权模式，通常使用软件中断可以实现
- 定义在 Data flash 数据大小需要是字(4 个字节)的整数倍
- 执行擦出或写的时序：
  - 等待 Data Flash 上一次写或擦除完成

```
while(DecRegs.DFLASHCTRL.bit.BUSY != 0)
{
        ; //do nothing while it programs
}
```

  - 写解锁密钥

```
DecRegs.FLASHILOCK.all = 0x42DC157E; //unlock flash write
```

  - 清 RONLY 位，以使 Data Flash 可写。RONLY 位在 MFBALR2 寄存中，由于该寄存器只支持 Word 写，所以字节写或半字写（LDRB, LDRH 指令）可能会导致出错，从而使得芯片复位。建议先将 MFBALR2 的值存到 ram 中，修改完后，再更新到 MFBALR2 寄存器中，这样就不会出错。

```
// copy the value to ram
mfbalr2shadow.all = DecRegs.MFBALR2.all;
// clear read-only bit
mfbalr2shadow.bit.RONLY = 0;
// update it
DecRegs.MFBALR2.all = mfbalr2shadow.all;
```

  - 执行写操作或擦除，注意执行擦除时，同样得遵循与清 RONLY 位相同的操作。
    - 写操作，可以使用指针方式

```
// Write the temp word to DFlash.
*dest_ptr = temp_word;
```

    - 页擦除

```
// copy the value to ram
dflashctrl_shadow.all = DecRegs.DFLASHCTRL.all;
// Erase one segment
dflashctrl_shadow.bit.PAGE_ERASE = 1;
// Segment number
dflashctrl_shadow.bit.PAGE_SEL = arg1;
// Update to hardware register
DecRegs.DFLASHCTRL.all = dflashctrl_shadow.all;
```

    - 整块擦除

```
// copy the value to ram
dflashctrl_shadow.all = DecRegs.DFLASHCTRL.all;
// Mass erase
dflashctrl_shadow.bit.MASS_ERASE = 1;
// Update to hardware register
```

```
DecRegs.DFLASHCTRL.all = dflashctrl_shadow.all;
```

- 置位 RONLY bit，使 Data Flash 处于只读状态。同样得遵循与清 RONLY 位相同的操作。

```
// copy the value to ram
mfbalr2shadow.all = DecRegs.MFBALR2.all;
// clear read-only bit
mfbalr2shadow.bit.RONLY = 1;
// update it
DecRegs.MFBALR2.all = mfbalr2shadow.all;
```

- 等待 Data Flash 擦出与写操作完成。

```
while(DecRegs.DFLASHCTRL.bit.BUSY != 0)
{
        ; //do nothing while it programs
}
```

4. 实验验证

4.1 通过 MAP 文件查看系统参数变量

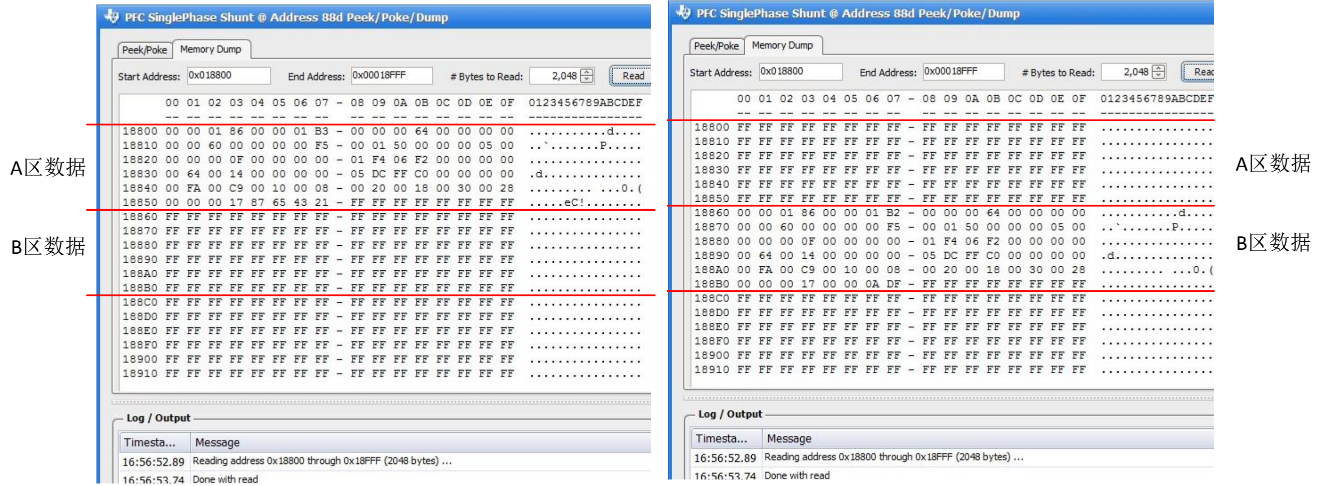
当程序添加完成后，打开地址映像文件（.map）。如下图所示，可以发现生成了 3 组元素相同的变量，其中变量_pfc_config_constants_a 与_pfc_config_constants_b 分别为 0x18800 与 0x18860，都位于 data flash 中；_pfc_config_hardcoded 的起始地址为 0x345c 位于 Program Flash 区域。同时还有一个变量_pfc_config_in_ram，地址为 0x1921c。当 Data Flash 的数据正确时，该变量就装载 Data Flash 的数据；否则，pfc_config_hardcoded 的数据会被拷贝 pfc_config_in_ram 变量中。

```
362 00018854  _pfc_config_checksum_a
363 000188b4  _pfc_config_checksum_b
364 00018800  _pfc_config_constants_a
365 00018860  _pfc_config_constants_b
366 0000345c  _pfc_config_hardcoded
367 0001921c  _pfc_config_in_ram
```

图 2. 地址映像（.MAP）文件

4.2 通过 GUI 发送命令更新系统参数

当确定参数后，可以通过 GUI 下发命令将 RAM 的数据存储到 Data Flash 中（Fusion online GUI 有一’Store RAM to Flash’按钮）。打开 Memory peek/poke 工具，查看更新前与更新后，Data Flash 的区别。下图示出了更新前后，DATA FLASH 的数据对比。



更新前，A区域有参数，而
B区域的参数全为0xFF

更新后，B区域有参数，而
A区域的参数全为0xFF

图 3. Data Flash 数据更新前后的对比

4.3 不烧录 Data Flash 的数据

在烧写程序时，只将 Data Flash 的数据擦除，并不写入新数据。在起机后，读取 ram 的数据与存储在 program flash 数据相同。

```
PFC SinglePhase Shunt @ Address 88d Peek/Poke/Dump

Peek/Poke | Memory Dump
Start Address: 0x00018800   End Address: 0x00018FFF   # Bytes to Read: 2,048   Read   Copy to Clipboard

      00 01 02 03 04 05 06 07 - 08 09 0A 0B 0C 0D 0E 0F  0123456789ABCDEF
      -- -- -- -- -- -- -- --   -- -- -- -- -- -- -- --  ----------------
18800 FF FF FF FF FF FF FF FF - FF FF FF FF FF FF FF FF  ................
18810 FF FF FF FF FF FF FF FF - FF FF FF FF FF FF FF FF  ................
18820 FF FF FF FF FF FF FF FF - FF FF FF FF FF FF FF FF  ................
18830 FF FF FF FF FF FF FF FF - FF FF FF FF FF FF FF FF  ................
18840 FF FF FF FF FF FF FF FF - FF FF FF FF FF FF FF FF  ................
18850 FF FF FF FF FF FF FF FF - FF FF FF FF FF FF FF FF  ................
18860 FF FF FF FF FF FF FF FF - FF FF FF FF FF FF FF FF  ................
18870 FF FF FF FF FF FF FF FF - FF FF FF FF FF FF FF FF  ................
18880 FF FF FF FF FF FF FF FF - FF FF FF FF FF FF FF FF  ................
18890 FF FF FF FF FF FF FF FF - FF FF FF FF FF FF FF FF  ................
188A0 FF FF FF FF FF FF FF FF - FF FF FF FF FF FF FF FF  ................
188B0 FF FF FF FF FF FF FF FF - FF FF FF FF FF FF FF FF  ................
188C0 FF FF FF FF FF FF FF FF - FF FF FF FF FF FF FF FF  ................
188D0 FF FF FF FF FF FF FF FF - FF FF FF FF FF FF FF FF  ................
188E0 FF FF FF FF FF FF FF FF - FF FF FF FF FF FF FF FF  ................
188F0 FF FF FF FF FF FF FF FF - FF FF FF FF FF FF FF FF  ................
18900 FF FF FF FF FF FF FF FF - FF FF FF FF FF FF FF FF  ................
18910 FF FF FF FF FF FF FF FF - FF FF FF FF FF FF FF FF  ................

Log / Output
Timesta...     Message
18:02:20.572   Reading address 0x18800 through 0x18FFF (2048 bytes) ...
18:02:21.274   Done with read

Clear Log   Copy Log to Clipboard   OK
```

图 4. Data Flash 中的数据没有数据

```
PFC SinglePhase Shunt @ Address 88d Peek/Poke/Dump

Peek/Poke | Memory Dump
Start Address: 0x345C   End Address: 0x000034AF   # Bytes to Read: 84   Read

     00 01 02 03 04 05 06 07 - 08 09 0A 0B 0C 0D 0E 0F  0123456789ABCDEF
     -- -- -- -- -- -- -- --   -- -- -- -- -- -- -- --  ----------------
3450                         -             00 00 01 86              ....
3460 00 00 01 B3 00 00 00 64 - 00 00 00 00 00 00 60 00  .......d......`.
3470 00 00 00 F5 00 01 50 00 - 00 00 05 00 00 00 00 0F  ......P.........
3480 00 00 00 00 01 F4 06 F2 - 00 00 00 00 00 64 00 14  .............d..
3490 00 00 00 00 05 DC FF C0 - 00 00 00 00 00 FA 00 C9  ................
34A0 00 10 00 08 00 20 00 18 - 00 30 00 28 00 00 00 17  ..... ...0.(....
```

结构体pfc_config_hardcoded的数据

```
PFC SinglePhase Shunt @ Address 88d Peek/Poke/Dump

Peek/Poke | Memory Dump
Start Address: 0x01921C   End Address: 0x0001926F   # Bytes to Read: 84   Read

      00 01 02 03 04 05 06 07 - 08 09 0A 0B 0C 0D 0E 0F  0123456789ABCDEF
      -- -- -- -- -- -- -- --   -- -- -- -- -- -- -- --  ----------------
19210                         -             00 00 01 86              ....
19220 00 00 01 B3 00 00 00 64 - 00 00 00 00 00 00 60 00  .......d......`.
19230 00 00 00 F5 00 01 50 00 - 00 00 05 00 00 00 00 0F  ......P.........
19240 00 00 00 00 01 F4 06 F2 - 00 00 00 00 00 64 00 14  .............d..
19250 00 00 00 00 05 DC FF C0 - 00 00 00 00 00 FA 00 C9  ................
19260 00 10 00 08 00 20 00 18 - 00 30 00 28 00 00 00 17  ..... ...0.(....
```

结构体pfc_config_In_ram的数据

图 5. Program Flash 中的数据被装载至 RAM 的变量

5 结论

本文主要介绍了一种提高数据存储稳定性的方法，将系统参数的数据分别存放在 Data Flash 与 Program Flash。通过本文可以看出，该方法即可以实现系统参数的在线更新，同时在 Data Flash 的数据不正确时，会使用存储在 Program Flash 的数据。

参考文献：

1. SLUSAP2C - UCD3138 Highly Integrated Digital Controller for Isolated Power, Texas Instruments, 2013

2. SPNU151J - ARM Optimizing C/C++ Compiler v5.2 User's Guide, Texas Instruments, 2014

3. SPNU118M - ARM Assembly Language tools v5.2 User's Guide, Texas Instruments, 2014

4. SLUU994 - UCD3138 ARM and Digital System Programmer’s Manual, July 2012

重要声明

德州仪器(TI) 及其下属子公司有权根据 JESD46 最新标准, 对所提供的产品和服务进行更正、修改、增强、改进或其它更改， 并有权根据 JESD48 最新标准中止提供任何产品和服务。客户在下订单前应获取最新的相关信息, 并验证这些信息是否完整且是最新的。所有产品的销售都遵循在订单确认时所提供的TI 销售条款与条件。

TI 保证其所销售的组件的性能符合产品销售时 TI 半导体产品销售条件与条款的适用规范。仅在 TI 保证的范围内，且 TI 认为 有必要时才会使用测试或其它质量控制技术。除非适用法律做出了硬性规定，否则没有必要对每种组件的所有参数进行测试。

TI 对应用帮助或客户产品设计不承担任何义务。客户应对其使用 TI 组件的产品和应用自行负责。为尽量减小与客户产品和应 用相关的风险，客户应提供充分的设计与操作安全措施。

TI 不对任何 TI 专利权、版权、屏蔽作品权或其它与使用了 TI 组件或服务的组合设备、机器或流程相关的 TI 知识产权中授予 的直接或隐含权限作出任何保证或解释。TI 所发布的与第三方产品或服务有关的信息，不能构成从 TI 获得使用这些产品或服 务的许可、授权、或认可。使用此类信息可能需要获得第三方的专利权或其它知识产权方面的许可，或是 TI 的专利权或其它 知识产权方面的许可。

对于 TI 的产品手册或数据表中 TI 信息的重要部分，仅在没有对内容进行任何篡改且带有相关授权、条件、限制和声明的情况 下才允许进行复制。TI 对此类篡改过的文件不承担任何责任或义务。复制第三方的信息可能需要服从额外的限制条件。

在转售 TI 组件或服务时，如果对该组件或服务参数的陈述与 TI 标明的参数相比存在差异或虚假成分，则会失去相关 TI 组件 或服务的所有明示或暗示授权，且这是不正当的、欺诈性商业行为。TI 对任何此类虚假陈述均不承担任何责任或义务。

客户认可并同意，尽管任何应用相关信息或支持仍可能由 TI 提供，但他们将独力负责满足与其产品及在其应用中使用 TI 产品 相关的所有法律、法规和安全相关要求。客户声明并同意，他们具备制定与实施安全措施所需的全部专业技术和知识，可预见 故障的危险后果、监测故障及其后果、降低有可能造成人身伤害的故障的发生机率并采取适当的补救措施。客户将全额赔偿因 在此类安全关键应用中使用任何 TI 组件而对 TI 及其代理造成的任何损失。

在某些场合中，为了推进安全相关应用有可能对 TI 组件进行特别的促销。TI 的目标是利用此类组件帮助客户设计和创立其特 有的可满足适用的功能安全性标准和要求的终端产品解决方案。尽管如此，此类组件仍然服从这些条款。

TI 组件未获得用于 FDA Class III（或类似的生命攸关医疗设备）的授权许可，除非各方授权官员已经达成了专门管控此类使 用的特别协议。

只有那些 TI 特别注明属于军用等级或"增强型塑料"的 TI 组件才是设计或专门用于军事/航空应用或环境的。购买者认可并同 意，对并非指定面向军事或航空航天用途的 TI 组件进行军事或航空航天方面的应用，其风险由客户单独承担，并且由客户独 力负责满足与此类使用相关的所有法律和法规要求。

TI 已明确指定符合 ISO/TS16949 要求的产品，这些产品主要用于汽车。在任何情况下，因使用非指定产品而无法达到 ISO/TS16949 要求，TI不承担任何责任。

产品		应用	
数字音频	www.ti.com.cn/audio	通信与电信	www.ti.com.cn/telecom
放大器和线性器件	www.ti.com.cn/amplifiers	计算机及周边	www.ti.com.cn/computer
数据转换器	www.ti.com.cn/dataconverters	消费电子	www.ti.com/consumer-apps
DLP® 产品	www.dlp.com	能源	www.ti.com/energy
DSP - 数字信号处理器	www.ti.com.cn/dsp	工业应用	www.ti.com.cn/industrial
时钟和计时器	www.ti.com.cn/clockandtimers	医疗电子	www.ti.com.cn/medical
接口	www.ti.com.cn/interface	安防应用	www.ti.com.cn/security
逻辑	www.ti.com.cn/logic	汽车电子	www.ti.com.cn/automotive
电源管理	www.ti.com.cn/power	视频和影像	www.ti.com.cn/video
微控制器 (MCU)	www.ti.com.cn/microcontrollers		
RFID 系统	www.ti.com.cn/rfidsys		
OMAP应用处理器	www.ti.com/omap		
无线连通性	www.ti.com.cn/wirelessconnectivity	德州仪器在线技术支持社区	www.deyisupport.com

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2016, Texas Instruments Incorporated