南京凌鸥创芯电子有限公司

LKS DSP 应用说明

© 2020，版权归凌鸥创芯所有
机密文件，未经许可不得扩散

目 录

1 概述 1

©2020 版权归凌鸥创芯所有 机密文件未经许可不得扩散

表格目录

未找到图形项目表。

©2020 版权归凌鸥创芯所有 机密文件未经许可不得扩散

图片目录

未找到图形项目表。

©2020 版权归凌鸥创芯所有 机密文件未经许可不得扩散

1 概述

1.1 DSP 运行的硬件资源

DSP 有独立的程序空间和数据空间，即 DSP Code mem 和 Data mem。在 DSP 暂停时可以通过软件直接寻址访问。

表 1-1 DSP 地址空间

模块	空间大小	空间区域	实际存储体大小
code_mem	2kB	0x4001_4000 ~ 0x4001_47FF	512 x 16bit
data_mem	2kB	0x4001_4800 ~ 0x4001_4FFF	64 x 32bit
reg	2kB	0x4001_5000 ~ 0x4001_57FF	
Reserved	2kB	0x4001_5800 ~ 0x4001_5FFF	

此外，DSP 内部有自己独立的核心寄存器和计算通路，可以独立于 ARM Cortex-M0 进行乘累加 MAC，饱和 SAT，以及除法 DIV、开方 SQRT、三角函数等一系列操作。

由于以上资源，DSP 可以完全独立于 CPU 进行程序运行，而且 DSP 从 DSP Code mem 取程序指令，数据存取通过 DSP Data mem。

©2020 版权归凌鸥创芯所有 机密文件未经许可不得扩散

2 DSP 模拟器

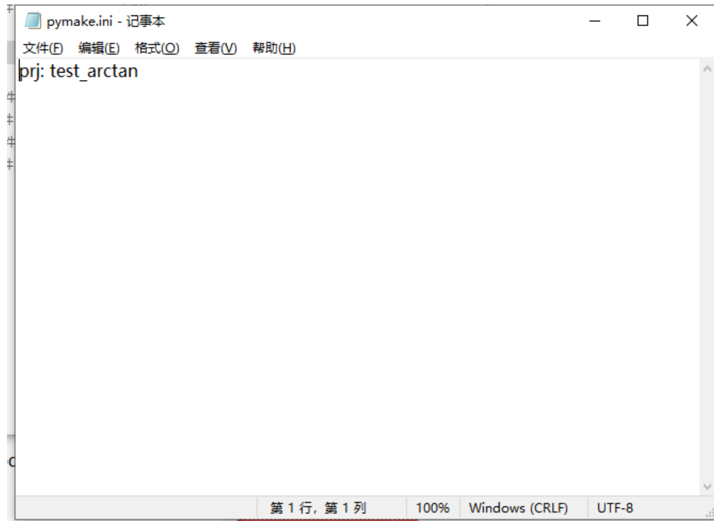
2.1 LKS081DSP_Emulator.exe

DSP 模拟器 LKS081DSP_Emulator.exe，是用 python 编写的，在 windows 下可以直接双击运行。DSP 模拟器是 DSP 行为的模拟，且每周期行为完全一致。

2.2 依赖文件

2.2.1 pymake.ini

内容如下图所示，主要用于指定 DSP 模拟器读入和编译的工程名，同一目录下可以存在多个 DSP 工程。需要保证同名的 .code 和 .data 文件存在在同一目录下



每个 DSP 工程主要是指 .code 文件和 .data 文件。

2.2.2 .code 文件

DSP 程序文件，这个文件是人工编写的 DSP 汇编程序，具体汇编指令和语法格式需要参考 <LKS32MC08x user manaul.pdf>

以下以 test_arctan 为例进行简要说明

```
#Load two halfwords into R3 and R4 from 0x4 address in DSP Data mem
LDRDHI    R3 R4 0x4
# R5=arctan(R3/R4)
ARCTAN    R5 R3 R4

# insert dummy inst to wait for arctan finish
ADD       R0 R0 R0
ADD       R0 R0 R0
```

```
ADD     R0 R0 R0
ADD     R0 R0 R0
ADD     R0 R0 R0
ADD     R0 R0 R0
ADD     R0 R0 R0
ADD     R0 R0 R0
ADD     R0 R0 R0
# arctan is stored at 0x5 in dsp's data mem
STRWI   R5 0x5
# module is stored at 0x6 in dsp's data mem
STRWI   R6 0x6
IRQ

ADD     R0 R0 R0
ADD     R0 R0 R0
ADD     R0 R0 R0
ADD     R0 R0 R0
```

2.2.3 .data 文件

DSP Data mem 的初始化文件。在本例中，在 0x4 地址放置了两个半字。分别为 0x3000 和 0x4000，通过 LDRDHI 指令 Load 到 R3 和 R4。

```
0x00000000
0x00000000
0x00000000
0x30004000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000   # 8
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
```

©2020 版权归凌鸥创芯所有 机密文件未经许可不得扩散

0x00000000
0x00000000
0x00000000
0x00000000

2.3 输出

模拟器运行结果如下图，R5=arctan(R3/R4)，R6=sqrt(R3*R3+R4*R4)。可以看到模拟器运行完会输出最终的 DSP 核心寄存器的值。

```
E:\Linko_repo\repo_root\APP_NOTE\08-APP_NOTE\DSP应用\dist\dist\LKS081DSP_Emulator.exe
LKS081 DSP Emulator v1.1
Released on 2020-3-9
Author: zhangwl@linkosemi.com
Copyright. 2020 LINKO Semiconductor Co.,Ltd. All Rights Reserved.
Current project is <test_arctan>.
Reading in <test_arctan.code>...
<test_arctan.code> contains 18 asm instructions
Reading in <test_arctan.data>...
<test_arctan.data> contains 26 words

2021-11-20 22:47:30
<test_arctan> Emulation is starting ...
Info: at Line 18: IRQ
IRQ generated and DSP halted when PC = 13.
<test_arctan> Emulation Finished
3999us elapsed on Emulator.
18Cycles elapsed on DSP.

Final GPR snapshot is as below:
PC = 0x0000
R0 = 0x00000000
R1 = 0x00000000
R2 = 0x00000000
R3 = 0x00003000
R4 = 0x00004000
R5 = 0x000025c0
R6 = 0x00005000
R7 = 0x00000000
请按任意键继续. . .
```

同时，程序运行结束后会在 output 文件夹下输出一些运行日志
<proj>_gpr_trace 是 DSP 程序运行过程各个周期的 GPR 值
<proj>_inst_trace 是 DSP 程序运行过程中各个周期执行的指令
以上两个文件，主要是用于追踪 DSP 程序运行过程情况。
<proj>_inst_statistics 是指令条数统计，统计程序中每种类型的指令执行过多少次

<proj>_code.txt 是 DSP 汇编程序被模拟器处理后生成的二进制文件，用户一般用不到。
<proj>_code.hex 是 DSP 汇编程序被模拟器翻译后生成的十六进制文件，本质上与二进制文件内容相同，只是进制不同。如下图所示，这个文件需要在 Keil 工程中通过软件初始化到 DSP Code ram 中。

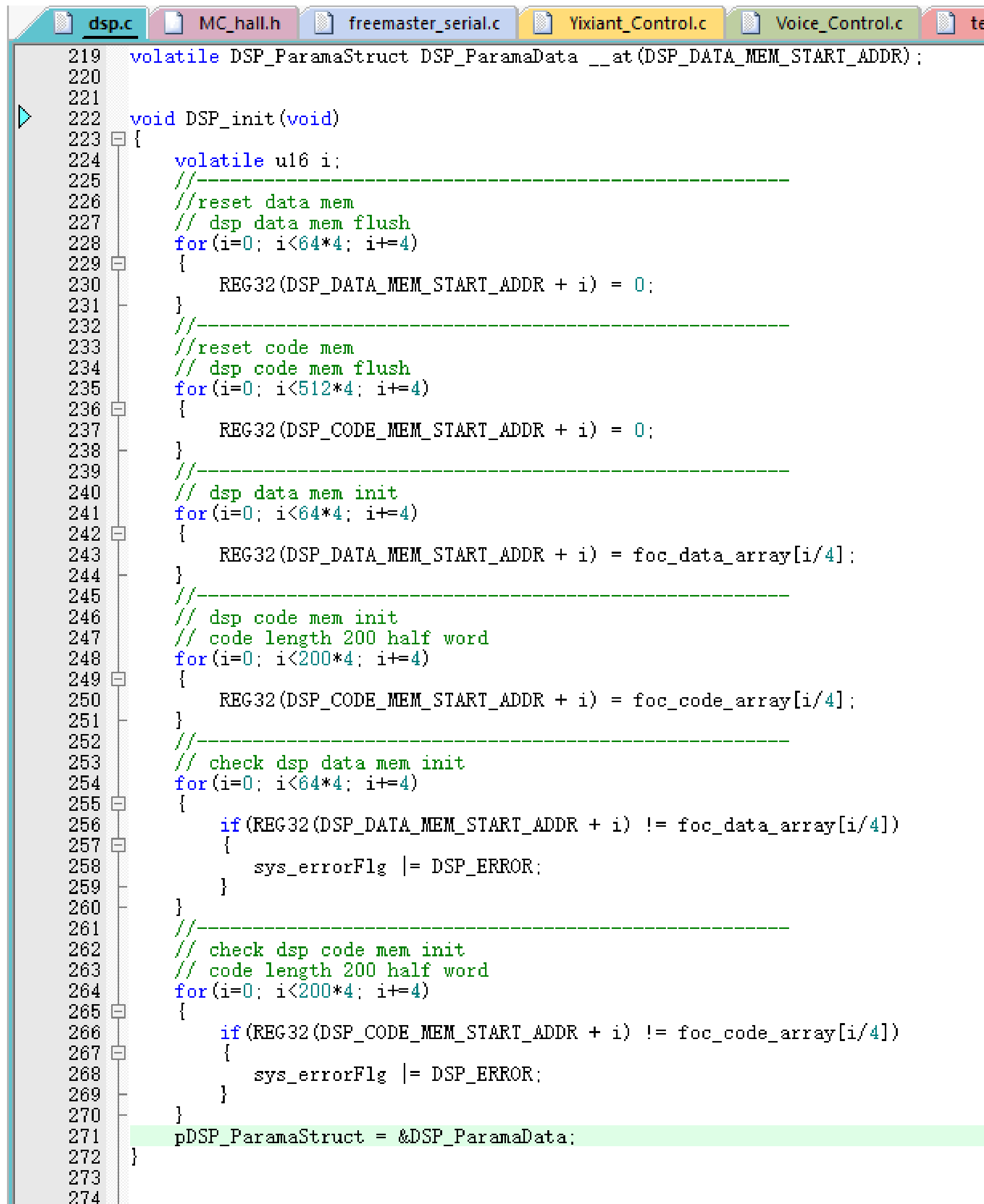
```
0x0000f11c,
0x0000792b,
0x00000000,
0x00000000,
0x00000000,
0x00000000,
0x00000000,
0x00000000,
0x00000000,
0x00000000,
0x00000000,
0x0000b145,
0x0000b186,
0x0000e000,
0x00000000,
0x00000000,
0x00000000,
0x00000000,
```

©2020 版权归凌鸥创芯所有 机密文件未经许可不得扩散

3 Keil 集成

3.1.1 DSP 初始化

dsp.c 中通过 DSP_init 函数对 DSP 的 code mem 和 data mem 进行初始化，并检查。



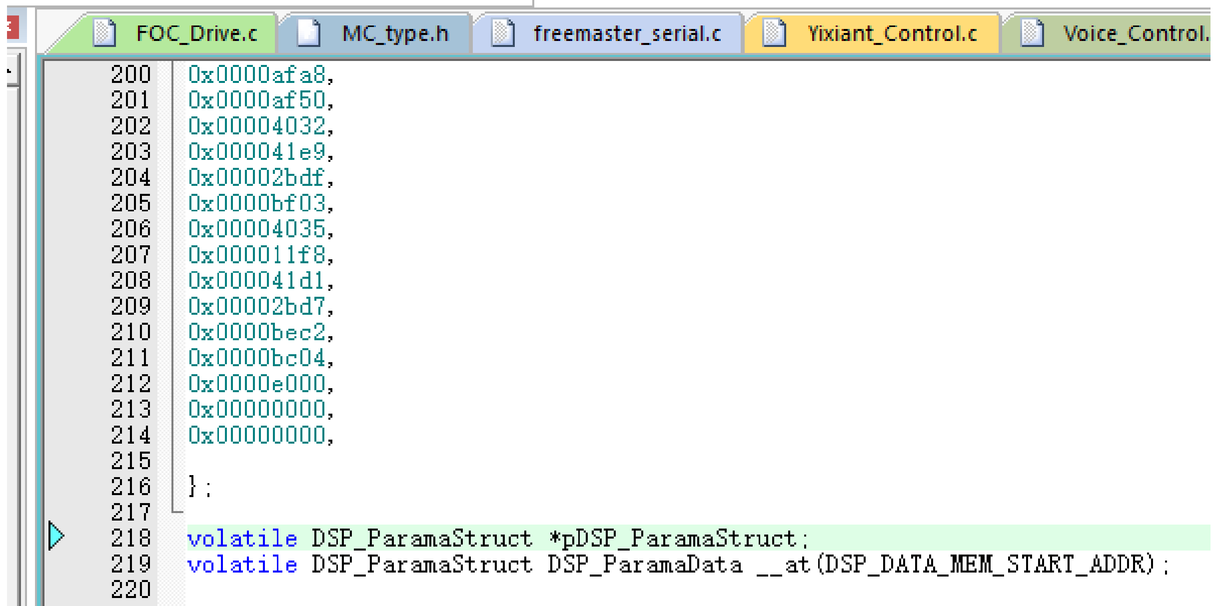
```
dsp.c  MC_hall.h  freemaster_serial.c  Yixiant_Control.c  Voice_Control.c  te
219 volatile DSP_ParamaStruct DSP_ParamaData __at(DSP_DATA_MEM_START_ADDR);
220
221
222 void DSP_init(void)
223 {
224     volatile u16 i;
225     //------------------------------------------------------
226     //reset data mem
227     // dsp data mem flush
228     for(i=0; i<64*4; i+=4)
229     {
230         REG32(DSP_DATA_MEM_START_ADDR + i) = 0;
231     }
232     //------------------------------------------------------
233     //reset code mem
234     // dsp code mem flush
235     for(i=0; i<512*4; i+=4)
236     {
237         REG32(DSP_CODE_MEM_START_ADDR + i) = 0;
238     }
239     //------------------------------------------------------
240     // dsp data mem init
241     for(i=0; i<64*4; i+=4)
242     {
243         REG32(DSP_DATA_MEM_START_ADDR + i) = foc_data_array[i/4];
244     }
245     //------------------------------------------------------
246     // dsp code mem init
247     // code length 200 half word
248     for(i=0; i<200*4; i+=4)
249     {
250         REG32(DSP_CODE_MEM_START_ADDR + i) = foc_code_array[i/4];
251     }
252     //------------------------------------------------------
253     // check dsp data mem init
254     for(i=0; i<64*4; i+=4)
255     {
256         if(REG32(DSP_DATA_MEM_START_ADDR + i) != foc_data_array[i/4])
257         {
258             sys_errorFlg |= DSP_ERROR;
259         }
260     }
261     //------------------------------------------------------
262     // check dsp code mem init
263     // code length 200 half word
264     for(i=0; i<200*4; i+=4)
265     {
266         if(REG32(DSP_CODE_MEM_START_ADDR + i) != foc_code_array[i/4])
267         {
268             sys_errorFlg |= DSP_ERROR;
269         }
270     }
271     pDSP_ParamaStruct = &DSP_ParamaData;
272 }
273
274
```

3.1.2 软件与 DSP 的数据交互

此外，程序维护了一个 DSP 数据结构，用于软件和 DSP 程序之间的数据传递，即 DSP_ParamaData。可以注意到，这个结构体就定义在 DSP Data mem 地址，因此，软件读写这个结构体，就是在读写 DSP Data mem 中的内容。

DSP Data mem 仅在 DSP 暂停状态下可以被 ARM 软件访问，在 DSP 运行阶段是无法被 ARM 软件访问的，DSP 执行到 IRQ 指令时，会产生中断，并暂停 DSP 程序运行，通知 ARM 软件进行数据交互。

©2020 版权归凌鸥创芯所有 机密文件未经许可不得扩散



FOC_Drive.c 中，有对 DSP_ParamaData 结构体进行初始化的动作。此外，我们还看到一些相同数据的初始化，但是是赋值给软件的其他变量，这个主要是在开发阶段用于 DSP 程序验证的，即 ARM 软件与 DSP 程序，使用相同的数据进行相同计算，然后看 DSP 计算结果与 ARM 计算结果是否在一定精度误差范围内，如果是，则 DSP 计算是正确的，后续可以将这一部分计算任务完全交给 DSP，将 ARM 软件的比对程序注释掉。

FOC_Drive.c | dsp.c | MC_hall.h | freemaster_serial.c | Yixiant_Control.c | Voice_Co

```
64  extern volatile DSP_ParamaStruct *pDSP_ParamaStruct;
65
66  RC_t BusLimCurr_RC;
67
68  u16 dbg_theatae, dbg_self_add;
69
70
71  void LowPass_Filter_Init(void)
72  {
73      Curr_RC_a.yk_1 = 0;
74      Curr_RC_a.coef = 0x6978; /* 1000Hz */
75      Curr_RC_b.yk_1 = 0;
76      Curr_RC_b.coef = 0x6978; /* 1000Hz */
77
78      BusLimCurr_RC.yk_1 = 0;
79      BusLimCurr_RC.coef = 200;
80
81      Curr_RC_d.yk_1 = 0;
82      Curr_RC_d.coef = 5000;
83      Curr_RC_q.yk_1 = 0;
84      Curr_RC_q.coef = 5000;
85
86      pDSP_ParamaStruct->Curr_RC_a_coef = 0x6978;
87      pDSP_ParamaStruct->Curr_RC_b_coef = 0x6978;
88      pDSP_ParamaStruct->Curr_RC_d_coef = 25000;
89      pDSP_ParamaStruct->Curr_RC_q_coef = 25000;
90
91      pDSP_ParamaStruct->Curr_RC_a_yk_1 = 0;
92      pDSP_ParamaStruct->Curr_RC_b_yk_1 = 0;
93      pDSP_ParamaStruct->Curr_RC_q_yk_1 = 0;
94      pDSP_ParamaStruct->Curr_RC_d_yk_1 = 0;
95
96
97      pDSP_ParamaStruct->Ki_Gain_D = PID_FLUX_KI_DEFAULT;
98      pDSP_ParamaStruct->Ki_Divisor_D = FLUX_KIDIV;
99      pDSP_ParamaStruct->Kp_Divisor_D = FLUX_KPDIV;
100     pDSP_ParamaStruct->Kp_Gain_D = PID_FLUX_KP_DEFAULT;
101     pDSP_ParamaStruct->hLower_Limit_Output_D =  -MAX_VD_LIM;
102     pDSP_ParamaStruct->hUpper_Limit_Output_D = MAX_VD_LIM;
103     pDSP_ParamaStruct->wLower_Limit_Integral_D = -MAX_VD_LIM << FLUX_KIDIV;
104     pDSP_ParamaStruct->wUpper_Limit_Integral_D = MAX_VD_LIM << FLUX_KIDIV;
105
106     pDSP_ParamaStruct->Ki_Gain_Q = PID_TORQUE_KI_DEFAULT;
107     pDSP_ParamaStruct->Ki_Divisor_Q = TF_KIDIV;
108     pDSP_ParamaStruct->Kp_Divisor_Q = TF_KPDIV;
109     pDSP_ParamaStruct->Kp_Gain_Q = PID_TORQUE_KP_DEFAULT;
110     pDSP_ParamaStruct->hLower_Limit_Output_Q =  -MAX_VQ_LIM;
111     pDSP_ParamaStruct->hUpper_Limit_Output_Q = MAX_VQ_LIM;
112     pDSP_ParamaStruct->wLower_Limit_Integral_Q = -MAX_VQ_LIM << TF_KIDIV;
113     pDSP_ParamaStruct->wUpper_Limit_Integral_Q = MAX_VQ_LIM << TF_KIDIV;
114
115 }
116
117
```

在 FOC_Drive.c 中的 FOC_Model 函数中，我们可以看到，在 211 到 215 行，每次启动 DSP 前，我们会更新 DSP Data mem 的内容。然后启动 DSP，并清楚 IRQ 标志。等待 DSP 产生 IRQ 标志并暂停后，说明 DSP 已经完成一段程序的运行，并执行到了 IRQ 指令，此时我们可以读取 DSP Data mem 的内容获取 DSP 上一段程序执行的结果,也可以向 DSP Data mem 写入新的下一段程序需要用到的数据。此时，我们如果不通过 DSP_SC 寄存器复位 DSP 的 PC，则 DSP 程序会继续往下执行。如果复位 DSP PC=0，则 DSP 开始新的一次循环。

DSP 程序中可以出现不止一次 IRQ 指令，即可以通过 IRQ 指令将 DSP 程序分割为多段，多次与 ARM 软件进行交互。

©2020 版权归凌鸥创芯所有 机密文件未经许可不得扩散

FOC_Drive.c | MC_type.h | freemaster_serial.c | Yixiant_Control.c | Voice_Control.c | te

```
208  //    pDSP_ParamaStruct->Stat_Curr_a = hPhaseCurrA;
209  //    pDSP_ParamaStruct->Stat_Curr_b = hPhaseCurrB;
210
211      pDSP_ParamaStruct->Stat_Curr_a = Stat_Curr_a_b_test.qI_Component1;
212      pDSP_ParamaStruct->Stat_Curr_b = Stat_Curr_a_b_test.qI_Component2;
213      pDSP_ParamaStruct->theta = theatae;
214      pDSP_ParamaStruct->Desired_value_D = hDCur_Reference;
215      pDSP_ParamaStruct->Desired_value_Q = hQCur_Reference;
216
217      PUSH_CACHE_INTO_MEM();
218
219      // [1] dsp_paused, write 0 to start dsp
220      // [0] irq write 1 to clear
221      DSP_SC = 0x01;
222
223  #ifdef MCU_RUN_FOC
224      hPhaseCurrA = lowPass_filter(&Curr_RC_a, hPhaseCurrA);
225      hPhaseCurrB = lowPass_filter(&Curr_RC_b, hPhaseCurrB);
226
227      Stat_Curr_a_b.qI_Component1 = hPhaseCurrA;
228      Stat_Curr_a_b.qI_Component2 = hPhaseCurrB;
229
230      Stat_Curr_alfa_beta = Clarke(Stat_Curr_a_b);
231
232      Stat_Curr_q_d = Park(Stat_Curr_alfa_beta, theatae);
233
234      Stat_Curr_q_d.qI_Component1 = lowPass_filter(&Curr_RC_q,
235                                    Stat_Curr_q_d.qI_Component1);
236      Stat_Curr_q_d.qI_Component2 = lowPass_filter(&Curr_RC_d,
237                                    Stat_Curr_q_d.qI_Component2);
238
239
240      if (ABS(hDqCurrentFir) > 500)
241      {
242          CalculateDeadTime(theatae, &Stat_Curr_q_d);
243      }
244      else
245      {
246          Vdtcomp.u = 0;
247          Vdtcomp.v = 0;
248          Vdtcomp.w = 0;
249      }
250  #else
251     while(0x03 != DSP_SC);
252  #endif
253      /*loads the Torque Regulator output reference voltage Vqs*/
254      if (firstOpenMosFlgCnt < 2)
255      {
256          firstOpenMosFlgCnt++;
257          Stat_Volt_q_d.qV_Component2 = 0;
258          Stat_Volt_q_d.qV_Component1 = PID_Torque.wIntegral
259                                        >> PID_Torque.hKi_Divisor;
260          lastPhaseCurrA = Stat_Curr_a_b_test.qI_Component1;
261          lastPhaseCurrB = Stat_Curr_a_b_test.qI_Component2;
262
263          Stat_Curr_a_b.qI_Component1 = 0;
264          Stat_Curr_a_b.qI_Component2 = 0;
```

如下图，读取 DSP Data mem 中的内容后，再次填入新的数据到结构体中。

```
FOC_Drive.c | MC_type.h | freemaster_serial.c | Yixiant_Control.c | Voice_Control.c | tempera
433             MCPWM_TH20 = -hTimePhA;
434             MCPWM_TH21 = hTimePhA;
435         }
436         else
437     #endif
438         {
439
440             // clear irq and resume dsp
441
442
443             Stat_Volt_q_d.qV_Component1 = pDSP_ParamaStruct->qPI_out;
444             Stat_Volt_q_d.qV_Component2 = pDSP_ParamaStruct->dPI_out;
445
446             RevPark_Circle_Limitation();
447             Stat_Volt_q_d.qV_Component1 = ((s32)(Stat_Volt_q_d.qV_Component1)
448                                            * MAX_MODULE_VALUE) >> 15;
449             Stat_Volt_q_d.qV_Component2 = ((s32)(Stat_Volt_q_d.qV_Component2)
450                                            * MAX_MODULE_VALUE) >> 15;
451
452             pDSP_ParamaStruct->qPI_out = (s32)Stat_Volt_q_d.qV_Component1;
453             pDSP_ParamaStruct->dPI_out = (s32)Stat_Volt_q_d.qV_Component2;
454             PUSH_CACHE_INTO_MEM()
455
456             DSP_SC = 0x01;
457             /*Performs the Reverse Park transformation,
458               i.e transforms stator voltages Vqs and Vds into Valpha and Vbeta on a
459               stationary reference frame*/
460     #ifdef MCU_RUN_FOC
461             Stat_Volt_alfa_beta = Rev_Park(Stat_Volt_q_d);
462     #else
463             while(0x03 != DSP_SC);
464             Stat_Volt_alfa_beta.qV_Component1 = pDSP_ParamaStruct->vAlpha;
465             Stat_Volt_alfa_beta.qV_Component2 = pDSP_ParamaStruct->vBeta;
466     #endif
467            GPIO1_PDO &= ~BIT15;
468             /*Valpha and Vbeta finally drive the power stage*/
469             SVPWM_2PH(Stat_Volt_alfa_beta);
470
471     //        if(motorTypeflg % 3) // motorTypeflg
472     //        {
473     //            MCPWM_TH00 = -hTimePhA;
474     //            MCPWM_TH01 = hTimePhA;
475
476     //            MCPWM_TH10 = -hTimePhB;
477     //            MCPWM_TH11 = hTimePhB;
478
479     //            MCPWM_TH20 = -hTimePhC;
480     //            MCPWM_TH21 = hTimePhC;
481     //        }
482     //        else
483             {
484                 MCPWM_TH20 = -hTimePhA;
485                 MCPWM_TH21 = hTimePhA;
486
487                 MCPWM_TH10 = -hTimePhB;
488                 MCPWM_TH11 = hTimePhB;
489
```

3.1.3 FAQ

1，DSP 代码始终从 PC 指针=0 开始执行

2，IRQ 指令中断 dsp 代码执行，PC 指针不清 0

3，PAUSED 清 0，DSP 代码将从当前 PC 指针开始继续执行，还是 PC 指针同时清 0，代码从 0 开始重新执行？

地址：0x4001_5000

复位值：0x2

表 15-4 DSP 状态控制寄存器 DSP_SC

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
												RESET_PC	CORDIC_MODE	PAUSED	IRQ
												WO	RW	RW	RWIC
												0	0	1	0

位置	位名称	说明
[31:4]		保留
[3]	RESET_PC	当 DSP 暂停时，写 1 重置 DSP PC 到 0 地址
[2]	CORDIC_MODE	CORDIC mode, 0: arctan, 1: sin/cos
[1]	PAUSED	指示 DSP 处于暂停状态，当 DSP 执行到 IRQ 指令时此 bit 置 1，软件写可以将此 bit 置 1。软件将此 bit 清零可以启动 DSP 运行
[0]	IRQ	DSP 中断标志，写 1 清零

IRQ 仅仅是令 DSP 进入暂停状态，通知 CPU 来获取 DSP 计算好的数据，以及填装下一次 DSP 运算所需要的数据，DSP PC 在遇到 IRQ 指令的时候不会清零。

如果需要清零重新执行，使用 DSP_SC reset DSP PC。如果不 reset_PC，而是将 PAUSED 清零，dsp 是继续执行。