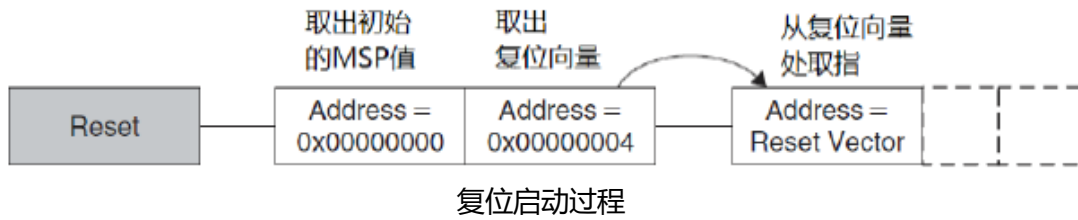
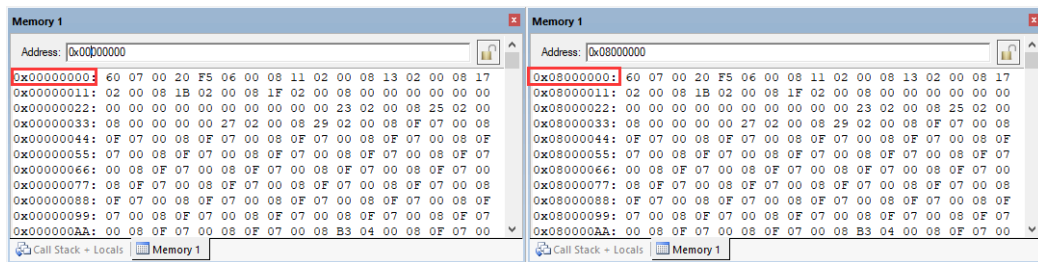


STM32 实验

启动模式：从 Flash 首地址 0x08000000 启动：

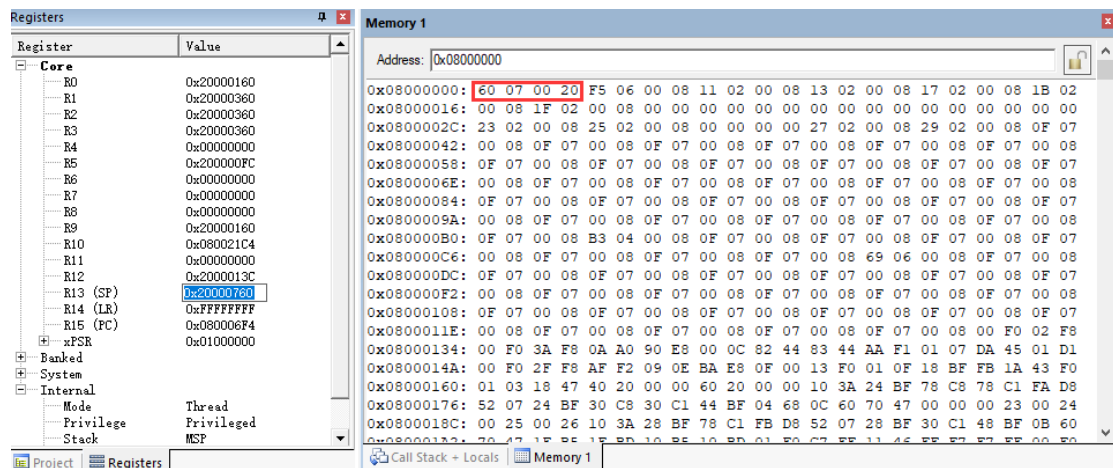


《Cortex-M3 权威指南》指出，在离开复位状态后，CM3 先从地址 0x00000000 处取出第一个 32 数据为 SP 指针的初始值。虽然从 Flash 首地址 0x08000000 启动，但 Flash 可被映射到启动空间(0x00000000)，仍然能够在它原有的地址(0x0800 0000)访问它，即 flash 的内容可以在两个地址区域访问，0x0000 0000 或 0x0800 0000。由下图可见，0x00000000 和 0x08000000 地址起始数据相同。



0x00000000 和 0x08000000 地址数据对比

第一个 32 位数据为 SP 指针的初始值，如下图所示，sp=0x20000760。



堆栈指针 (sp) 初始化赋予值为 0x20000760

SP 指针计算过程：

```
linking...
Program Size: Code=8308 RO-data=336 RW-data=52 ZI-data=1836
FromELF: creating hex file...
"..\\OBJ\\TIMER.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed: 00:00:08
```

工程编译链接后统计程序大小

图中几个参数分别代表：Code：代码的大小；RO：常量所占空间；RW：程序中已经初始化的变量所占空间；ZI：未初始化的静态和全局变量以及堆栈所占的空间。

上述参数和芯片 Flash 以及 SRAM 的对应关系是：Flash 占用大小=Code+RO+RW；SRAM 占用大小=RW+ZI。

RW 参数同时参与了 Flash 和 SRAM 占用量的计算。这是因为 Flash 部分的属性是只读的掉电数据不丢失，而 SRAM 虽然是读写但里面数据掉电丢失，所以只能把已经初始化的值保存到 flash 里，上电后再拷贝到 SRAM 中进行读写操作，即两部分都需要留出 RW 变量所占用的空间。

$$\text{SRAM} = 52 + 1836 = 1888 \text{ (0x760)}$$

Execution Region RW_IRAM1 (Exec base: 0x20000000, Load base: 0x080021c4, Size: 0x0000760, Max: 0x00010000, ABSOLUTE)

Exec Addr	Load Addr	Size	Type	Attr	Idx	E Section Name	Object
0x20000000	0x080021c4	0x00000014	Data	RW	166	.data	system_stm32f10x.o
0x20000014	0x080021d8	0x00000004	Data	RW	221	.data	delay.o
0x20000018	0x080021dc	0x00000006	Data	RW	253	.data	usart.o
0x2000001e	0x080021e2	0x00000014	Data	RW	316	.data	stm32f10x_rcc.o
0x20000032	0x080021f6	0x00000002	PAD				
0x20000034	-	0x000000c8	Zero	RW	252	.bss	usart.o
0x200000fc	-	0x00000060	Zero	RW	390	.bss	c_w.l(libspace.o)
0x2000015c	0x080021f6	0x00000004	PAD				
0x20000160	-	0x00000200	Zero	RW	285	HEAP	startup_stm32f10x_hd.o
0x20000360	-	0x00000400	Zero	RW	284	STACK	startup_stm32f10x_hd.o

map 文件中的堆和栈

由上述可得该工程占用 SRAM 的地址空间为 0x20000000-0x20000760，其中 0x20000760 为栈顶指针(sp)，栈向下生长，空间大小为 1k，栈的地址空间为 0x20000360-0x20000760；堆的地址空间为 0x20000160-0x20000360，空间大小为 0.5K，堆为向上生长。

```
Stack_Size EQU 0x00000400

Stack_Mem AREA STACK, NOINIT, READWRITE, ALIGN=3
SPACE Stack_Size
__initial_sp

; <h> Heap Configuration
; <o> Heap Size (in Bytes) <0x0-0xFFFFFFFF:8>
; </h>

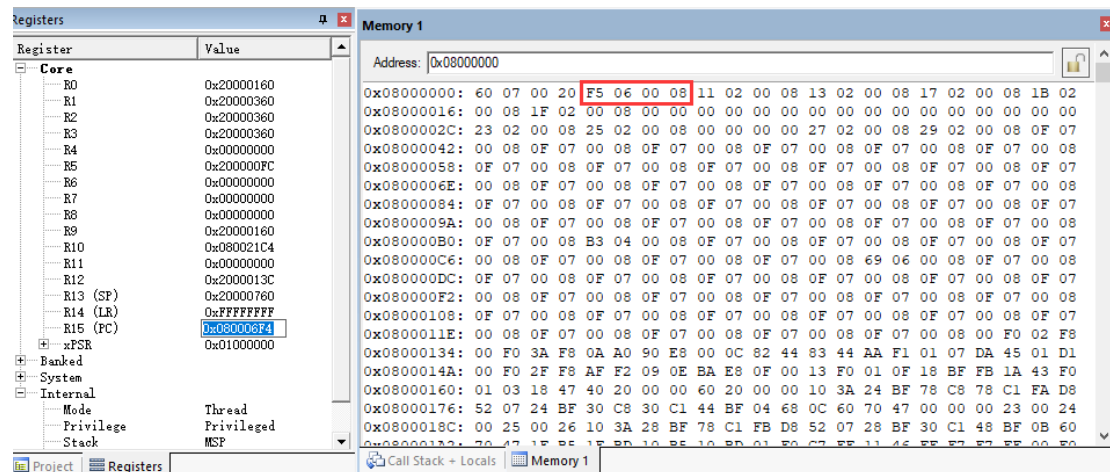
Heap_Size EQU 0x00000200

_heap_base AREA HEAP, NOINIT, READWRITE, ALIGN=3
Heap_Mem SPACE Heap_Size
__heap_limit
```

startup_stm32f10x_hd.s 启动文件中定义的堆栈大小

在取出第一个 32 位数据初始化 SP 指针后，再取出第二个 32 位数据初始化 PC 指针，然后跳转到该数据所对应到的地址（Reset_Handler 函数）处取值。

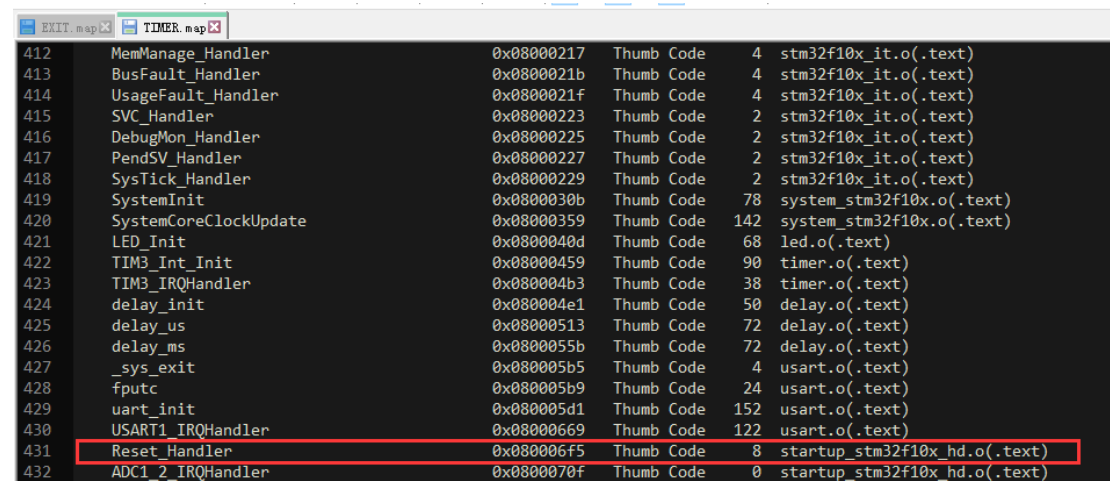
第二个 32 位数据为 PC 指针的初始值，如下图所示，PC=0x080006F4。



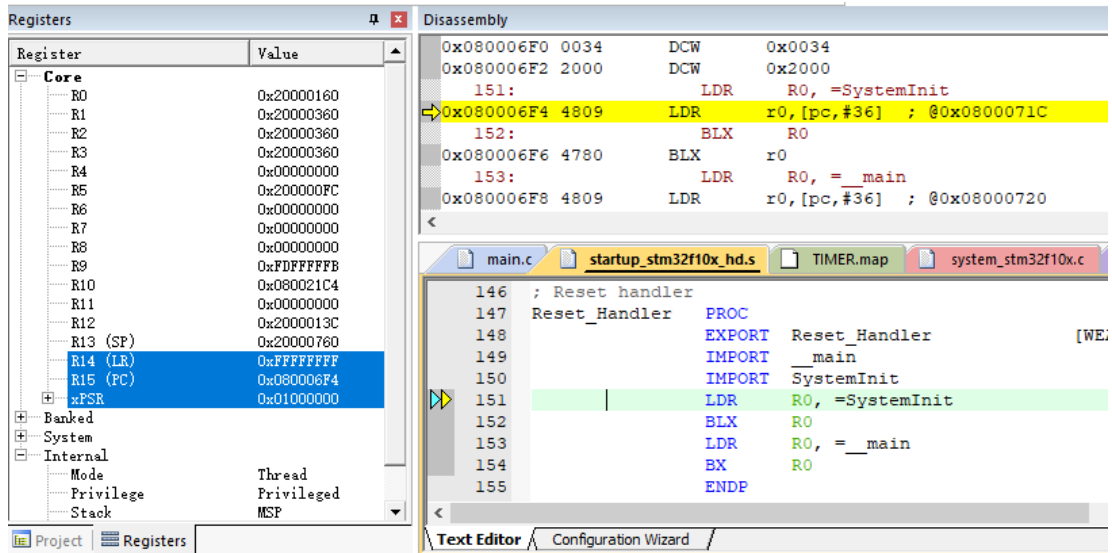
Reset_Handler 函数入口地址为

此处第二个 32 位数据为 0x080006F5，与 PC 值 0x080006F4 并不严格相符，相差 LSB 的 1。PC 寄存器 LSB 为 1 表示使用使用 Thumb-2 指令集，LSB 为 0 时表示试图进入 ARM 指令集模式，但是 M3 内核不支持 ARM 指令集，则会产生错误中断。将地址写入 PC 寄存器时会地址自动对齐，所以 PC 的值为 0x080006F4。

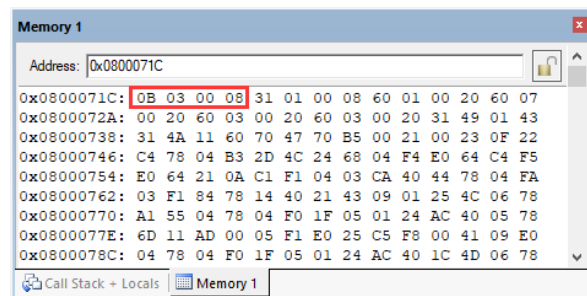
关于 Reset_Handler 入口地址可在 map 文件中查看。



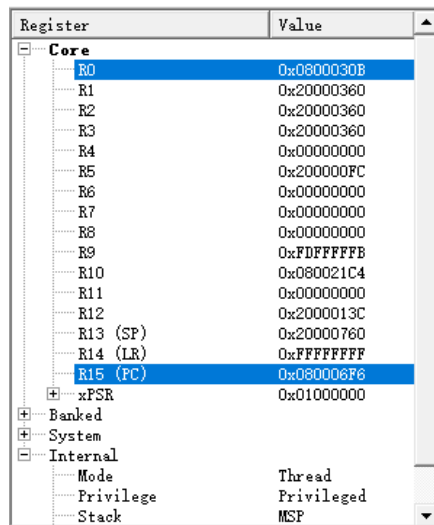
map 文件中 Reset_Handler 入口地址



PC 为 0x080006F4 时，在反汇编窗口和程序运行窗口可以看到，PC 已经指向下一条汇编 LDR r0,[pc,#36] (原汇编代码为 LDR R0,=SystemInit，表明将 SystemInit 函数的入口地址传给 R0)，实际上 SystemInit 函数的入口地址与 PC 当前值相差 36，即 $0x080006F4 + 0x4 + 0x24(\#36) = 0x0800071C$ (此处 PC 值+4 是因为 CM3 内部使用了指令流水线，读 PC 时返回的值是当前指令的地址+4)。将 0x0800071C 地址处的值 (0x0800030B) 存入 R0 寄存器，执行 LDR 指令后，R0 寄存器的值为 0x0800030B



0x0800071C 地址



R0 寄存器

下一条指令 BLX R0，表示跳转到 R0 给出的地址 (0x0800030B)，根据 R0 的 LSB 切换处理器状态，并转移前的下条指令地址 (0x080006F8+0x1) 保存到 LR (0x080006F9) 中。跳转后 PC 值为 0x0800030A (0x0800030B-0x1，地址自动对齐，LSB 表示处理器状态。)

Register	Value
Core	
R0	0x0800030B
R1	0x20000360
R2	0x20000360
R3	0x20000360
R4	0x00000000
R5	0x200000FC
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0xFDFFFFFFF
R10	0x080021C4
R11	0x00000000
R12	0x2000013C
R13 (SP)	0x20000760
R14 (LR)	0x080006F9
R15 (PC)	0x0800030A
+ xPSR	0x01000000
+ Banked	
+ System	
- Internal	
Mode	Thread
Privilege	Privileged

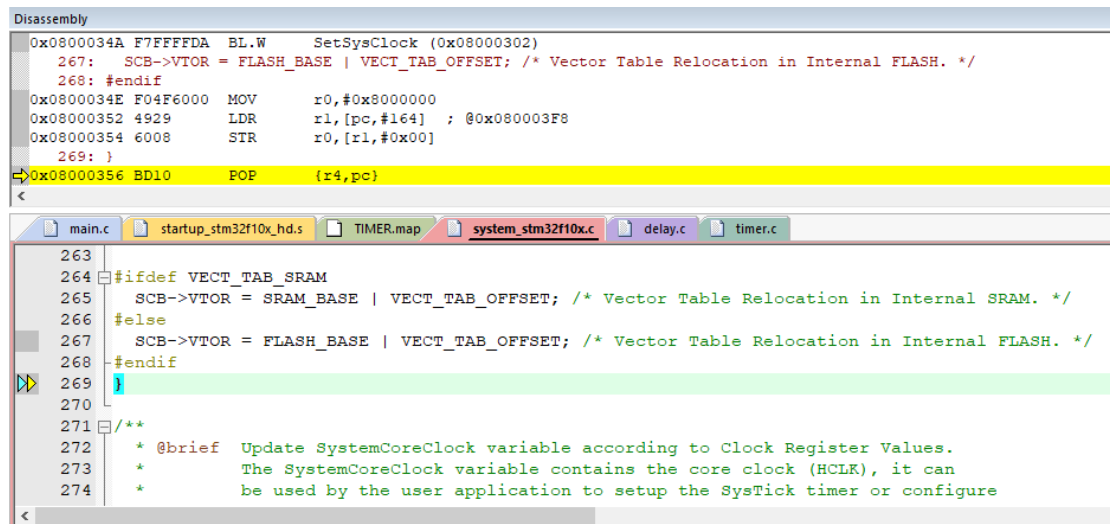
LR 和 PC

```

213: {
214:  /* Reset the RCC clock configuration to the default reset state(for debug purpose) */
215:  /* Set HSION bit */
216:  RCC->CR |= (uint32_t)0x00000001;
217:
218:  /* Reset SW, HPRE, PPRE1, PPRE2, ADCPRE and MCO bits */
219:  #ifndef STM32F10X_CL
220:    RCC->CFGR &= (uint32_t)0xF8FF0000;
221:  #else
222:    /* Reset D1CFGR and D2CFGR bits */
223:    RCC->D1CFGR &= (uint32_t)0xF8000000;
224:    RCC->D2CFGR &= (uint32_t)0xF8000000;
225:  #endif
226:
227:  /* Enable the debugger */
228:  /* Set DBP bit */
229:  RCC->CR |= (uint32_t)0x00000001;
230:
231:  /* Set IWDGCR */
232:  IWDG->CR = 0x00000000;
233:
234:  /* Set IWDGPR */
235:  IWDG->PR = 0x00000000;
236:
237:  /* Set IWDGSR */
238:  IWDG->SR = 0x00000000;
239:
240:  /* Set IWDGKR */
241:  IWDG->KR = 0x00000000;
242:
243:  /* Set IWDGRR */
244:  IWDG->RR = 0x00000000;
245:
246:  /* Set IWDGPR */
247:  IWDG->PR = 0x00000000;
248:
249:  /* Set IWDGSR */
250:  IWDG->SR = 0x00000000;
251:
252:  /* Set IWDGKR */
253:  IWDG->KR = 0x00000000;
254:
255:  /* Set IWDGRR */
256:  IWDG->RR = 0x00000000;
257:
258:  /* Set IWDGPR */
259:  IWDG->PR = 0x00000000;
260:
261:  /* Set IWDGSR */
262:  IWDG->SR = 0x00000000;
263:
264:  /* Set IWDGKR */
265:  IWDG->KR = 0x00000000;
266:
267:  /* Set IWDGRR */
268:  IWDG->RR = 0x00000000;
269:
270:  /* Set IWDGPR */
271:  IWDG->PR = 0x00000000;
272:
273:  /* Set IWDGSR */
274:  IWDG->SR = 0x00000000;
275:
276:  /* Set IWDGKR */
277:  IWDG->KR = 0x00000000;
278:
279:  /* Set IWDGRR */
280:  IWDG->RR = 0x00000000;
281:
282:  /* Set IWDGPR */
283:  IWDG->PR = 0x00000000;
284:
285:  /* Set IWDGSR */
286:  IWDG->SR = 0x00000000;
287:
288:  /* Set IWDGKR */
289:  IWDG->KR = 0x00000000;
290:
291:  /* Set IWDGRR */
292:  IWDG->RR = 0x00000000;
293:
294:  /* Set IWDGPR */
295:  IWDG->PR = 0x00000000;
296:
297:  /* Set IWDGSR */
298:  IWDG->SR = 0x00000000;
299:
300:  /* Set IWDGKR */
301:  IWDG->KR = 0x00000000;
302:
303:  /* Set IWDGRR */
304:  IWDG->RR = 0x00000000;
305:
306:  /* Set IWDGPR */
307:  IWDG->PR = 0x00000000;
308:
309:  /* Set IWDGSR */
310:  IWDG->SR = 0x00000000;
311:
312:  /* Set IWDGKR */
313:  IWDG->KR = 0x00000000;
314:
315:  /* Set IWDGRR */
316:  IWDG->RR = 0x00000000;
317:
318:  /* Set IWDGPR */
319:  IWDG->PR = 0x00000000;
320:
321:  /* Set IWDGSR */
322:  IWDG->SR = 0x00000000;
323:
324:  /* Set IWDGKR */
325:  IWDG->KR = 0x00000000;
326:
327:  /* Set IWDGRR */
328:  IWDG->RR = 0x00000000;
329:
330:  /* Set IWDGPR */
331:  IWDG->PR = 0x00000000;
332:
333:  /* Set IWDGSR */
334:  IWDG->SR = 0x00000000;
335:
336:  /* Set IWDGKR */
337:  IWDG->KR = 0x00000000;
338:
339:  /* Set IWDGRR */
340:  IWDG->RR = 0x00000000;
341:
342:  /* Set IWDGPR */
343:  IWDG->PR = 0x00000000;
344:
345:  /* Set IWDGSR */
346:  IWDG->SR = 0x00000000;
347:
348:  /* Set IWDGKR */
349:  IWDG->KR = 0x00000000;
350:
351:  /* Set IWDGRR */
352:  IWDG->RR = 0x00000000;
353:
354:  /* Set IWDGPR */
355:  IWDG->PR = 0x00000000;
356:
357:  /* Set IWDGSR */
358:  IWDG->SR = 0x00000000;
359:
360:  /* Set IWDGKR */
361:  IWDG->KR = 0x00000000;
362:
363:  /* Set IWDGRR */
364:  IWDG->RR = 0x00000000;
365:
366:  /* Set IWDGPR */
367:  IWDG->PR = 0x00000000;
368:
369:  /* Set IWDGSR */
370:  IWDG->SR = 0x00000000;
371:
372:  /* Set IWDGKR */
373:  IWDG->KR = 0x00000000;
374:
375:  /* Set IWDGRR */
376:  IWDG->RR = 0x00000000;
377:
378:  /* Set IWDGPR */
379:  IWDG->PR = 0x00000000;
380:
381:  /* Set IWDGSR */
382:  IWDG->SR = 0x00000000;
383:
384:  /* Set IWDGKR */
385:  IWDG->KR = 0x00000000;
386:
387:  /* Set IWDGRR */
388:  IWDG->RR = 0x00000000;
389:
390:  /* Set IWDGPR */
391:  IWDG->PR = 0x00000000;
392:
393:  /* Set IWDGSR */
394:  IWDG->SR = 0x00000000;
395:
396:  /* Set IWDGKR */
397:  IWDG->KR = 0x00000000;
398:
399:  /* Set IWDGRR */
400:  IWDG->RR = 0x00000000;
401:
402:  /* Set IWDGPR */
403:  IWDG->PR = 0x00000000;
404:
405:  /* Set IWDGSR */
406:  IWDG->SR = 0x00000000;
407:
408:  /* Set IWDGKR */
409:  IWDG->KR = 0x00000000;
410:
411:  /* Set IWDGRR */
412:  IWDG->RR = 0x00000000;
413:
414:  /* Set IWDGPR */
415:  IWDG->PR = 0x00000000;
416:
417:  /* Set IWDGSR */
418:  IWDG->SR = 0x00000000;
419:
420:  /* Set IWDGKR */
421:  IWDG->KR = 0x00000000;
422:
423:  /* Set IWDGRR */
424:  IWDG->RR = 0x00000000;
425:
426:  /* Set IWDGPR */
427:  IWDG->PR = 0x00000000;
428:
429:  /* Set IWDGSR */
430:  IWDG->SR = 0x00000000;
431:
432:  /* Set IWDGKR */
433:  IWDG->KR = 0x00000000;
434:
435:  /* Set IWDGRR */
436:  IWDG->RR = 0x00000000;
437:
438:  /* Set IWDGPR */
439:  IWDG->PR = 0x00000000;
440:
441:  /* Set IWDGSR */
442:  IWDG->SR = 0x00000000;
443:
444:  /* Set IWDGKR */
445:  IWDG->KR = 0x00000000;
446:
447:  /* Set IWDGRR */
448:  IWDG->RR = 0x00000000;
449:
450:  /* Set IWDGPR */
451:  IWDG->PR = 0x00000000;
452:
453:  /* Set IWDGSR */
454:  IWDG->SR = 0x00000000;
455:
456:  /* Set IWDGKR */
457:  IWDG->KR = 0x00000000;
458:
459:  /* Set IWDGRR */
460:  IWDG->RR = 0x00000000;
461:
462:  /* Set IWDGPR */
463:  IWDG->PR = 0x00000000;
464:
465:  /* Set IWDGSR */
466:  IWDG->SR = 0x00000000;
467:
468:  /* Set IWDGKR */
469:  IWDG->KR = 0x00000000;
470:
471:  /* Set IWDGRR */
472:  IWDG->RR = 0x00000000;
473:
474:  /* Set IWDGPR */
475:  IWDG->PR = 0x00000000;
476:
477:  /* Set IWDGSR */
478:  IWDG->SR = 0x00000000;
479:
480:  /* Set IWDGKR */
481:  IWDG->KR = 0x00000000;
482:
483:  /* Set IWDGRR */
484:  IWDG->RR = 0x00000000;
485:
486:  /* Set IWDGPR */
487:  IWDG->PR = 0x00000000;

```

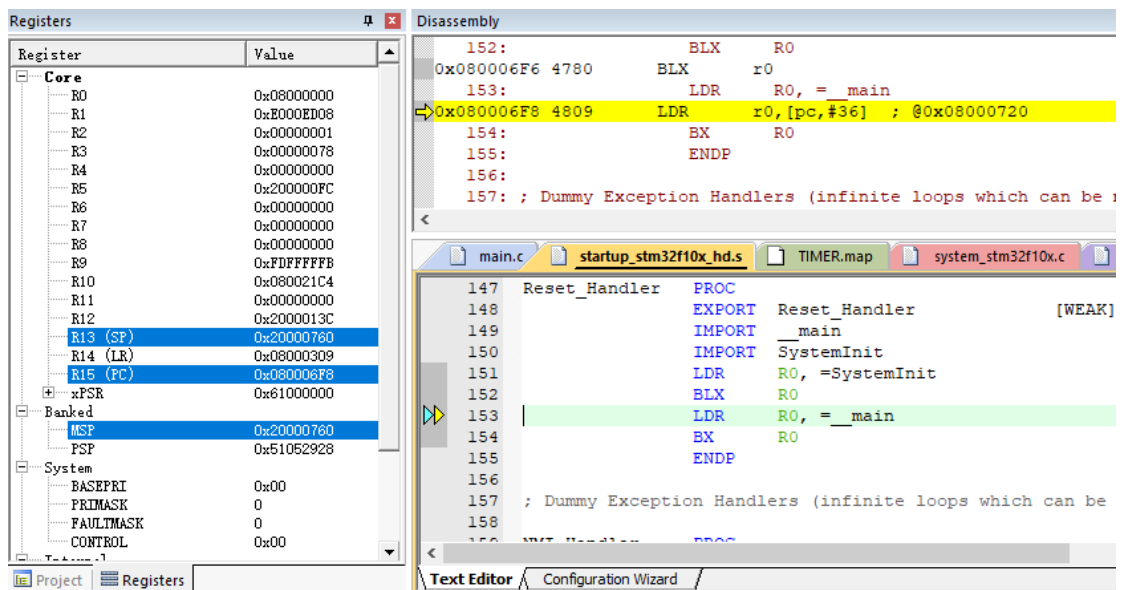
此处进入 `SystemInit` 函数进行一系列系统时钟初始化操作，前期报告有过关于 `SystemInit` 函数功能的详细解释，此处不再赘述。反汇编窗口表示，在执行 `SystemInit` 函数第一条代码之前，对 `R4` 和 `LR` 寄存器进行了压栈操作（`PUSH {r4,lr}`），用于保存返回地址。



```
Disassembly
0x0800034A F7FFFDFA BL.W SetSysClock (0x08000302)
267: SCB->VTOR = FLASH_BASE | VECT_TAB_OFFSET; /* Vector Table Relocation in Internal FLASH. */
268: #endif
0x0800034E F04F6000 MOV r0,#0x80000000
0x08000352 4929 LDR r1,[pc,#164] ; @0x080003F8
0x08000354 6008 STR r0,[r1,#0x00]
269: }
0x08000356 BD10 POP {r4,pc}

main.c startup_stm32f10x_hd.s TIMER.map system_stm32f10x.c delay.c timer.c
263
264 #ifdef VECT_TAB_SRAM
265 SCB->VTOR = SRAM_BASE | VECT_TAB_OFFSET; /* Vector Table Relocation in Internal SRAM. */
266 #else
267 SCB->VTOR = FLASH_BASE | VECT_TAB_OFFSET; /* Vector Table Relocation in Internal FLASH. */
268 #endif
269 }
270
271 /**
272 * @brief Update SystemCoreClock variable according to Clock Register Values.
273 * The SystemCoreClock variable contains the core clock (HCLK), it can
274 * be used by the user application to setup the SysTick timer or configure
```

在运行到 SystemInit 函数最后一行时，可以看到出栈过程（POP {r4,pc}），出栈并没有弹出到 LR 寄存器，而是直接弹出到 PC 寄存器，直接将进入 SystemInit 函数前地址赋给 PC，省去中间弹出到 LR 再将 LR 赋给 PC 的过程。



```
Registers
Register Value
Core
R0 0x08000000
R1 0xE000ED08
R2 0x00000001
R3 0x00000078
R4 0x00000000
R5 0x200000FC
R6 0x00000000
R7 0x00000000
R8 0x00000000
R9 0xFDFDFFFB
R10 0x080021C4
R11 0x00000000
R12 0x2000013C
R13 (SP) 0x20000760
R14 (LR) 0x08000309
R15 (PC) 0x080006F8
xPSR 0x61000000
Banked
MSP 0x20000760
PSP 0x61052928
System
BASEPRI 0x00
PRIMASK 0
FAULTMASK 0
CONTROL 0x00

Disassembly
152: BLX R0
0x080006F6 4780 BLX r0
153: LDR R0, =_main
0x080006F8 4809 LDR r0,[pc,#36] ; @0x08000720
154: BX R0
155: ENDP
156:
157: ; Dummy Exception Handlers (infinite loops which can be

main.c startup_stm32f10x_hd.s TIMER.map system_stm32f10x.c
147 Reset_Handler PROC
148 EXPORT Reset_Handler [WEAK]
149 IMPORT __main
150 IMPORT SystemInit
151 LDR R0, =SystemInit
152 BLX R0
153 LDR R0, =__main
154 BX R0
155 ENDP
156
157 ; Dummy Exception Handlers (infinite loops which can be
158
159 WFI_Handler PROC
```

由上图可知，在 systemInit 函数执行结束后，返回主程序执行的位置。此时 PC 为 0x080006F8（之前保存到 LR 的值 0x080006F9 经自动对齐后的结果），同时 SP 指针回到栈顶 0x20000760。

之后的汇编代码 LDR R0,=__main，此处并不是直接跳转到用户的 main 函数，而是在进入 main 函数之前需先跳转到地址为 0x08000130 处进行关于上面所描述的堆栈空间的开辟，并将部分 flash 的数据拷贝到 sram 中，并初始化 ZI 区域。

在 keil5 软件 debug 过程中需在执行完 LDR R0,=__main 指令后，手动赋予 PC 寄存器 0x08000131，跳转至初始化程序段，否则软件默认跳过初始化过程直接进入用户 main 函数。若此处 PC 被赋值为 0x08000131 外的其他值，则在系统执行数次指令后进入错误

中断函数，之后只能通过复位进行初始化。

地址为 0x08000130 的初始化过程：

```
0x08000130 F000F802 BL.W    __scatterload (0x08000138)
/* ( _main 入口地址为 0x08000130 ) BL.W 跳转到 0x08000138,并且把转移前的下条指令地址保存到 LR，该代码段表示负责把 RW/RO 输出段从装载域地址复制到运行域地址，并完成了 ZI 运行域的初始化工作。*/
```

```
0x08000134 F000F83A BL.W    __rt_entry (0x080001AC)
```

```
0x08000138 A00A      ADR      r0,{pc}+0x2C ; @0x08000164
```

```
/*ADR 是为了取出附近某条指令或者变量的地址，而 LDR 则是取出一个通用的 32 位整数，ADR 得到了优化，它的代码效率比 LDR 要高，( RO = 0x08000164 )
```

```
0x0800013A E8900C00 LDM      r0,{r10-r11}
```

```
/*从一片连续的地址空间(R0:0x08000164)中加载两个字到 R10(0x00002040)和 R11(0x00002060)*/
```

```
0x0800013E 4482      ADD      r10,r10,r0
```

```
//R10 = R10 + R0 = 0x00002040 + 0x08000164 = 0x080021A4
```

```
0x08000140 4483      ADD      r11,r11,r0
```

```
//R11 = R11 + R0 = 0x00002060 + 0x08000164 = 0x080021C4
```

```
0x08000142 F1AA0701 SUB      r7,r10,#0x01
```

```
//R7 = R10 - 0x1
```

```
0x08000146 45DA      CMP      r10,r11
```

```
//两次大段传输，判断 map 文件的两个数值是否相等，相等则 flash 到 sram 传输完毕
```

673	Region\$\$Table\$\$Base	0x080021a4	Number	0	anon\$\$obj.o(Relocation\$Table)
674	Region\$\$Table\$\$Limit	0x080021c4	Number	0	anon\$\$obj.o(Relocation\$Table)

```
//比较 R10 和 R11
```

```
/* cmp 指令可以直接影响 CPSR 寄存器的 Z 标识位 ( 条件位 ): 比较结果为 0 时，Z 位置 1，比较结果为非 0 时，Z 位为 0；*/
```

```
0x08000148 D101      BNE      0x0800014E
```

```
//cmp 的结果为 1，或者 CPSR 的 Z 标识位为 0 时，程序跳转到 0x0800014E 后的标签处
```

```
//R10=R11 时代表 flash 到 SRAM 传输完毕
```

```
/*flash 传输到 sram 的过程，先取出 flash 中需要传输的数据的起始地址和 SRAM 的起始地址和数据的大小*/
```

```
0x0800014E F2AF0E09 ADR.W    lr,{pc}-0x07 ; @0x08000147
```

```
// LR = 0x08000147，记录返回到 CMP R10 R11 指令的地址。
```

```
0x08000152 E8BA000F LDM      r10!,{r0-r3} //r10 自增 0x10
```

```
//将 R10 对应地址存放的 4 个字 copy 到 R0~R3 中
```

```
/* 然后执行__scatterload_null 代码 将 R10 对应地址存放的 4 个字 copy 到 R0~R3 中，可以看出：
```

```
R0:0x080021C4 表示的是 map 文件中表示 flash 移植 sram 段首个地址 ( system_stm32f10x.o 加载域起始地址 )
```

R1:0x20000000 为 system_stm32f10x.o 运行域地址 (即拷贝到 SRAM 中的位置)

R2:0x00000034 为 copy 的大小 , 该部分包含了从 system_stm32f10x.o 加载域起始地址到 PAD 处

R3:0x0800016C 是 _scatterload_copy 函数的起始地址 , 后续使用指令 BX R3 跳转到 _scatterload_copy 来复制代码。*/

```

0x08000156 F0130F01  TST      r3,#0x01
/测试指令 测试 R3 和 0x01 更新 CPSR 寄存器的 Z 标识位( 条件位 ),比较结果为 0 时 ,
Z 位置 1 ( 此时 Z 位 1 ) */
0x0800015A BF18      IT      NE
//Z 为 0 时
0x0800015C 1AFB      SUBNE   r3,r7,r3
0x0800015E F0430301  ORR     r3,r3,#0x01
//R3 = R3 + 1 = 0x0800 016D( _scatterload_copy 函数地址)
0x08000162 4718      BX      r3
//跳转到 _scatterload_copy 函数

/*****跳转到 _scatterload_copy 函数 , 进行传输操作*****/
0x0800016C 3A10      SUBS    r2,r2,#0x10
//R2 自减 0x10 , 并且根据结果更新标志 ( 有 " S " 后缀 )
0x0800016E BF24      ITT     CS
//C 标志为 1 进行下面的
0x08000170 C878      LDMCS   r0!,{r3-r6}
/*R0 开始连续的数据传输至 R3 R4 R5 R6(0x080021C4 地址开始将后面 4 个字传到 R3-
R6) , R0 自增 0x10*/
0x08000172 C178      STMCS   r1!,{r3-r6}
//将上面已经保存到 R3 R4 R5 R6 的数据传到 R1(0x20000000),R1 自+0x10
0x08000174 D8FA      BHI     __scatterload_copy (0x0800016C)
//R2(需要 copy 的大小)每次循环自减 0x10 , 计算需要传输的次数
0x08000176 0752      LSLS    r2,r2,#29
//循环传输结束
0x08000178 BF24      ITT     CS
//判断是否 标志位 C== 1
0x0800017A C830      LDMCS   r0!,{r4-r5}
0x0800017C C130      STMCS   r1!,{r4-r5}
//由于 C≠1 , 上述两行代码不执行
0x0800017E BF44      ITT     MI
/*在上述执行 _scatterload_copy 函数过程中 , 循环 copy4 次 , 返回 _scatterload 函数后
判断 R10 和 R11 , 后再次进行后续 copy 赋值 , 针对该工程存在两次 flash 到 sram 赋值的

```


过程*/

第一次 copy 传输内容：

Exec Addr	Load Addr	Size	Type	Attr	Idx	E Section Name	Object
0x20000000	0x080021c4	0x00000014	Data	RW	166	.data	system_stm32f10x.o
0x20000014	0x080021d8	0x00000004	Data	RW	221	.data	delay.o
0x20000018	0x080021dc	0x00000006	Data	RW	253	.data	usart.o
0x2000001e	0x080021e2	0x00000014	Data	RW	316	.data	stm32f10x_rcc.o
0x20000032	0x080021f6	0x00000002	PAD				

第二次传输 copy 传输内容

Exec Addr	Load Addr	Size	Type	Attr	Idx	E Section Name	Object
0x20000000	0x080021c4	0x00000014	Data	RW	166	.data	system_stm32f10x.o
0x20000014	0x080021d8	0x00000004	Data	RW	221	.data	delay.o
0x20000018	0x080021dc	0x00000006	Data	RW	253	.data	usart.o
0x2000001e	0x080021e2	0x00000014	Data	RW	316	.data	stm32f10x_rcc.o
0x20000032	0x080021f6	0x00000002	PAD				
0x20000034	-	0x000000c8	Zero	RW	252	.bss	usart.o
0x200000fc	-	0x00000060	Zero	RW	390	.bss	c_w.l(libspace.o)
0x2000015c	0x080021f6	0x00000004	PAD				
0x20000160	-	0x00000200	Zero	RW	285	HEAP	startup_stm32f10x_hd.o
0x20000360	-	0x00000400	Zero	RW	284	STACK	startup_stm32f10x_hd.o

/******R10=R11 时代表 flash 到 SRAM 传输完毕******/

0x0800014A F000F82F BL.W __rt_entry (0x080001AC)

0x080001AC F001FFC7 BL.W __user_setup_stackheap (0x0800213E)

/*传输完毕后跳转到__user_setup_stackheap (堆栈开辟函数), 并且把转移前的下条指令地址保存到 LR*/

/*在 flash 传输到 sram 后, 之后进行 ZI 区域初始化和开辟堆栈, 在这个过程中有多次寄存器之间数据的互相传输, 此部分的详细解析省略, 只阐述大致流程。*/

0x0800213E 4675 MOV r5,lr

0x08002140 F000F82C BL.W __user_libspace (0x0800219C)

0x0800219C 4800 LDR r0,[pc,#0] ; @0x080021A0

0x0800219E 4770 BX lr

.....

0x08002156 F7FEFADB BL.W __user_initial_stackheap (0x08000710)

//由启动文件创建, 开辟堆和栈

/*LDR R0, = Heap_Mem

LDR R1, =(Stack_Mem + Stack_Size)

LDR R2, =(Heap_Mem + Heap_Size)

LDR R3, = Stack_Mem

BX LR*/

0x080001B2 F7FFFFFF BL.W __rt_lib_init (0x080001A4)

.....

0x080001B6 F000F809 BL.W main (0x080001CC)

//执行上述指令之后可跳转到用户 main 函数

上述过程为进入用户 main 函数之前的流程，整理如下，主要过程为：

1. 0x08000130 F000F802 BL.W __scatterload (0x08000138)
//执行__scatterload 函数 Initialization, Zero Initialization regions to 0, load region
//relocation to execution addresses
2. 0x08000174 D8FA BHI __scatterload_copy (0x0800016C)
//执行__scatterload_copy 函数，循环 copy4 次，返回__scatterload 函数后判断 R10
//和 R11 后再次进行 copy 赋值
3. 0x0800014A F000F82F BL.W __rt_entry (0x080001AC)
0x080001AC F001FFC7 BL.W __user_setup_stackheap (0x0800213E)
//传输完毕后跳转到__user_setup_stackheap (堆栈开辟函数)，并且把转移前的下条指
//令地址保存到 LR
0x0800213E 4675 MOV r5,lr
4. 0x08002140 F000F82C BL.W __user_libspace (0x0800219C)
0x0800219C 4800 LDR r0,[pc,#0] ; @0x080021A0
0x0800219E 4770 BX lr (红色字体为连续过程)
.....
5. 0x08002156 F7FEFADB BL.W __user_initial_stackheap (0x08000710)
//由启动文件创建，开辟堆和栈
.....
6. 0x080001B2 F7FFFFFF BL.W __rt_lib_init (0x080001A4)
.....
7. 0x080001B6 F000F809 BL.W main (0x080001CC)
.....

关于中断向量表的问题，Cortex-M3 内核是固定了中断向量表的位置而中断函数入口地址是可变化的。

358	__Vectors_Size	0x00000130	Number	0	startup_stm32f10x_hd.o	ABSOLUTE
359	__Vectors	0x08000000	Data	4	startup_stm32f10x_hd.o(RESET)	
360	__Vectors_End	0x08000130	Data	0	startup_stm32f10x_hd.o(RESET)	

map 文件中关于中断向量表的定义

在 map 文件 Global Symbols 中上图所示位置处，0x08000000 起始位置定义了对中断向量表。中断向量表地址范围为 0x08000000- 0x08000130， $0x130 / 4 = 0x4c$ 转换成十进制为 76，对应启动文件中定义的 76 个中断向量。

769	Execution Region RW_IRAM1 (Exec base: 0x20000000, Load base: 0x080021c4, Size: 0x00000760, Max: 0x00010000, ABSOLUTE)							
770								
771								
772	Exec Addr	Load Addr	Size	Type	Attr	Idx	E Section Name	Object
773								
774	0x20000000	0x080021c4	0x00000014	Data	RW	166	.data	system_stm32f10x.o
775	0x20000014	0x080021d8	0x00000004	Data	RW	221	.data	delay.o
776	0x20000018	0x080021dc	0x00000006	Data	RW	253	.data	usart.o
777	0x2000001e	0x080021e2	0x00000014	Data	RW	316	.data	stm32f10x_rcc.o
778	0x20000032	0x080021f6	0x00000002	PAD				
779	0x20000034	-	0x000000c8	Zero	RW	252	.bss	usart.o
780	0x200000fc	-	0x00000060	Zero	RW	390	.bss	c_w.l(libspace.o)
781	0x2000015c	0x080021f6	0x00000004	PAD				
782	0x20000160	-	0x00000200	Zero	RW	285	HEAP	startup_stm32f10x_hd.o
783	0x20000360	-	0x00000400	Zero	RW	284	STACK	startup_stm32f10x_hd.o
784								

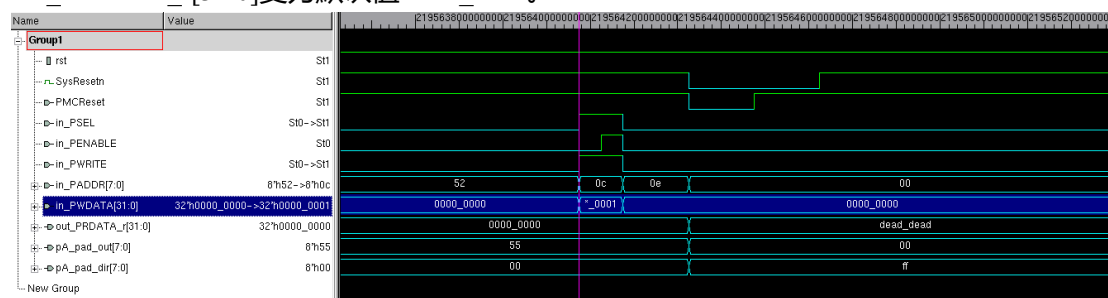
Map 文件中 Flash 数据拷贝至 SRAM

上图表明 Flash 内的 RW 数据将会拷贝到 SRAM 处（这一操作将会在正式运行用户代码之前完成），这也对应了开头所讲的 Flash 占用大小=Code+RO+RW；SRAM 占用大小=RW+ZI。RW 数据同时占用了 Flash 和 SRAM 的空间。

SEP8000 PMU 测试

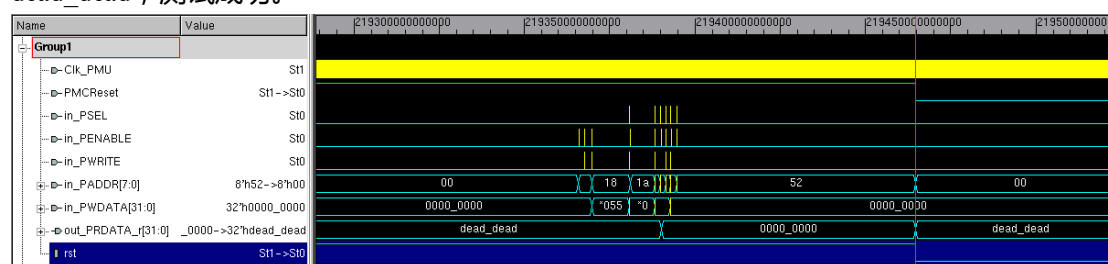
软件复位测试：

根据 8100 用户手册，配置软件复位寄存器 RCTR 寄存器第 0 位以实现软件复位，通过 APB 总线读数据信号线 out_PRDATA_r[31:0]可以看到在软件复位信号有效后，out_PRDATA_r[31:0]变为默认值 dead_dead。



硬件复位测试：

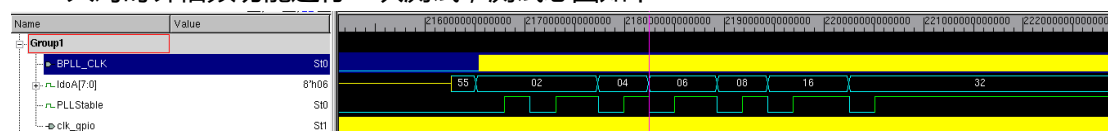
在 test.v 文件中输出 rst 置 0 信号，模拟外部硬件复位按键，通过查看 APB 总线来观察是否可以实现正常复位。如下图所示，在 rst 信号置 0 时，out_PRDATA_r[31:0]变为默认值 dead_dead，测试成功。



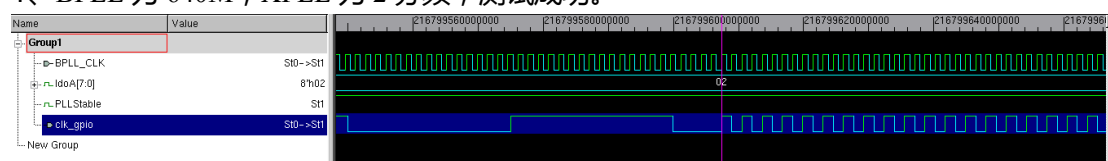
PLL 测试，测试总图：

通过配置 PMCR 寄存器、BPLL_CFG2 寄存器、XPLL_CFG1 寄存器，观察时钟倍频后的输出信号 BPLL_CLK 是否正常。通过 SEP8000 时钟网络可得，XPLL 时钟信号可通过 GPIO 外设模块的时钟信号 clk_gpio 观测。

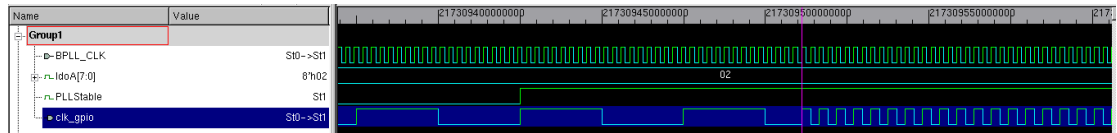
共对时钟倍频功能进行 7 次测试，测试总图如下：



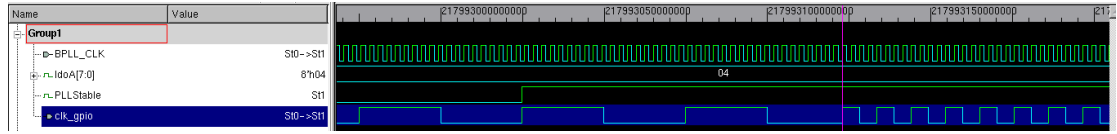
1、BPLL 为 640M，XPLL 为 2 分频，测试成功。



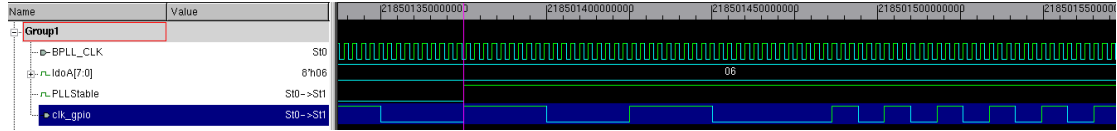
2、BPLL 为 380M，XPLL 为 2 分频，测试成功。



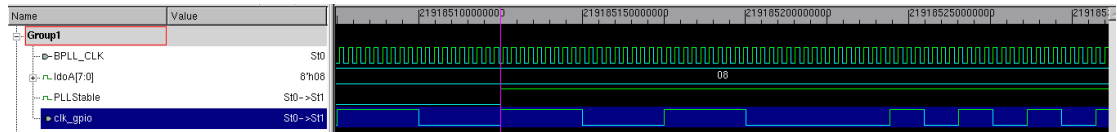
3、BPLL 为 380M，XPLL 为 4 分频，测试成功。



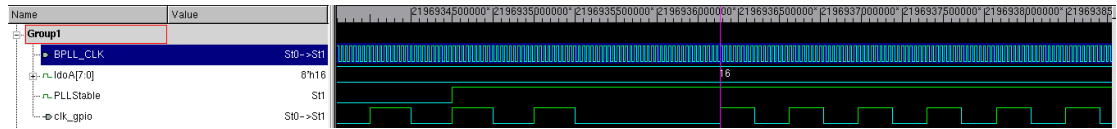
4、BPLL 为 380M，XPLL 为 6 分频，测试成功。



5、BPLL 为 380M，XPLL 为 8 分频，测试成功。



6、BPLL 为 380M，XPLL 为 16 分频，测试成功。



7、BPLL 为 380M，XPLL 为 32 分频，测试成功。

