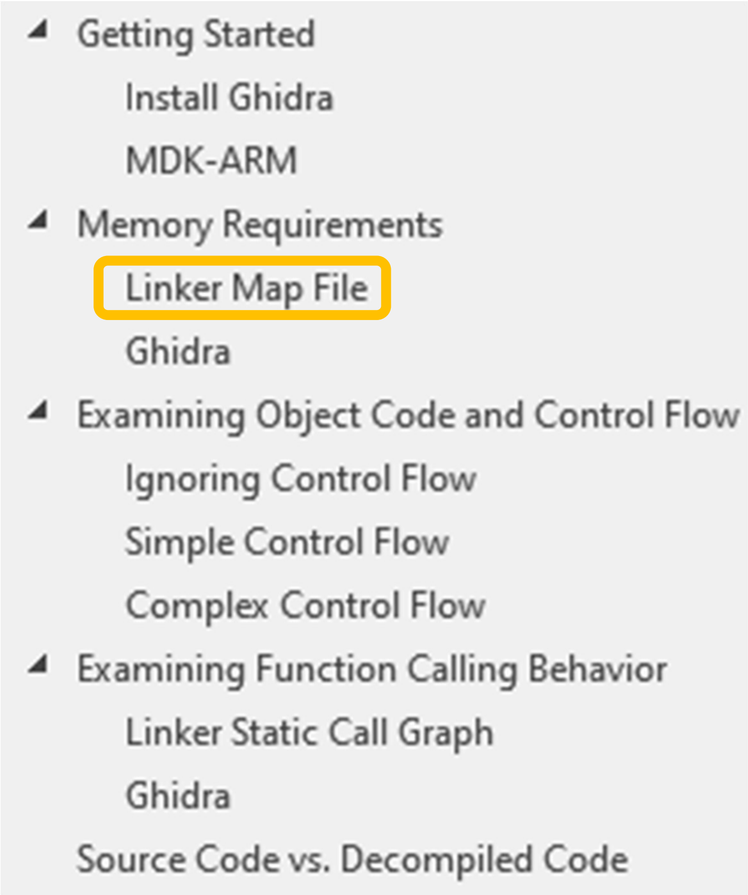
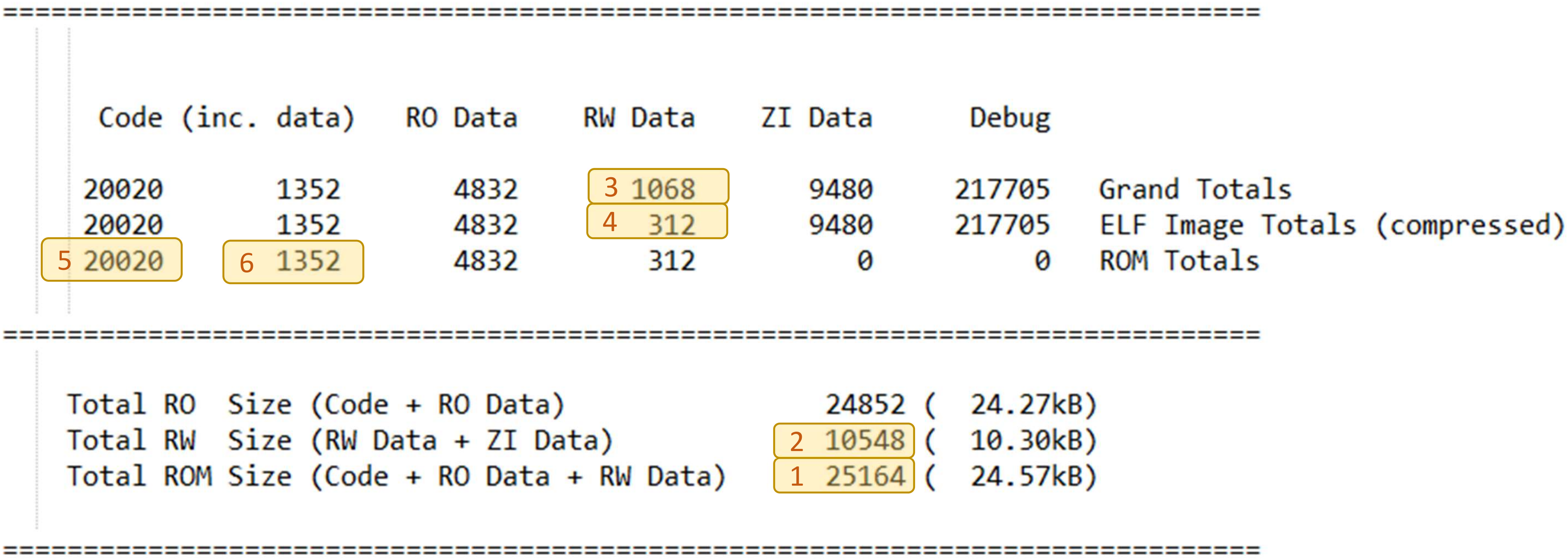


- Getting Started
  - Install Ghidra
  - MDK-ARM
- Memory Requirements
  - Linker Map File
  - Ghidra
- Examining Object Code and Control Flow
  - Ignoring Control Flow
  - Simple Control Flow
  - Complex Control Flow
- Examining Function Calling Behavior
  - Linker Static Call Graph
  - Ghidra
- Source Code vs. Decompiled Code

Scroll down to the last two sections in the map file to determine the following values in bytes (not kB).

1. What is the **Total ROM Size** in bytes?
2. What is the **Total RW Size** in bytes? This is the total RAM required.
3. What is the size of **RW Data** in **RAM** in bytes? Note this is *uncompressed* RW Data (in **Grand Totals**).
4. What is the size of **RW Data** in **ROM** in bytes? Note this is *compressed* RW Data (in **ELF Image Totals (compressed)**).
5. How much **ROM** space in bytes is used for code?
6. How much of that **ROM** space is used for included data (e.g. literal pools, listed as "**.inc data**")?



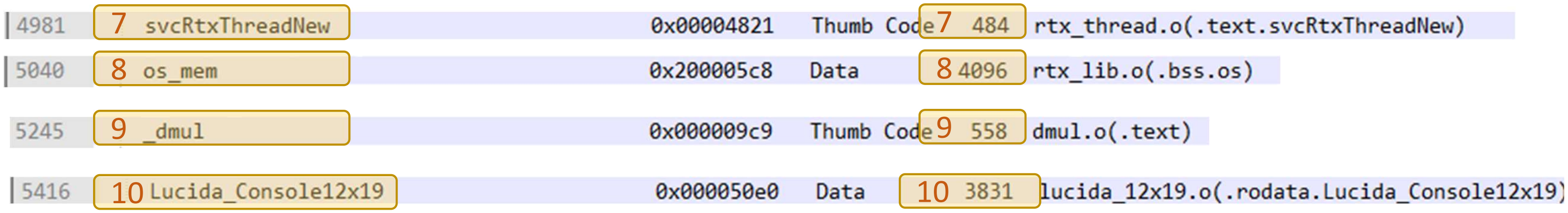
```
==============================================================================

      Code (inc. data)   RO Data    RW Data    ZI Data      Debug

     20020       1352       4832     3 1068       9480     217705   Grand Totals
     20020       1352       4832     4  312       9480     217705   ELF Image Totals (compressed)
   5 20020     6 1352       4832        312          0          0   ROM Totals

==============================================================================

    Total RO  Size (Code + RO Data)                 24852 (  24.27kB)
    Total RW  Size (RW Data + ZI Data)            2 10548 (  10.30kB)
    Total ROM Size (Code + RO Data + RW Data)     1 25164 (  24.57kB)

==============================================================================
```

Examine the **Local Symbols** section.

7. Which is the largest symbol of type “Thumb Code”, and how large is it in bytes?
8. Which is the largest symbol of type “Data”, and how large is it in bytes?

Examine the **Global Symbols** section.

9. Which is the largest symbol of type “Thumb Code”, and how large is it in bytes?
10. Which is the largest symbol of type “Data”, and how large is it in bytes?

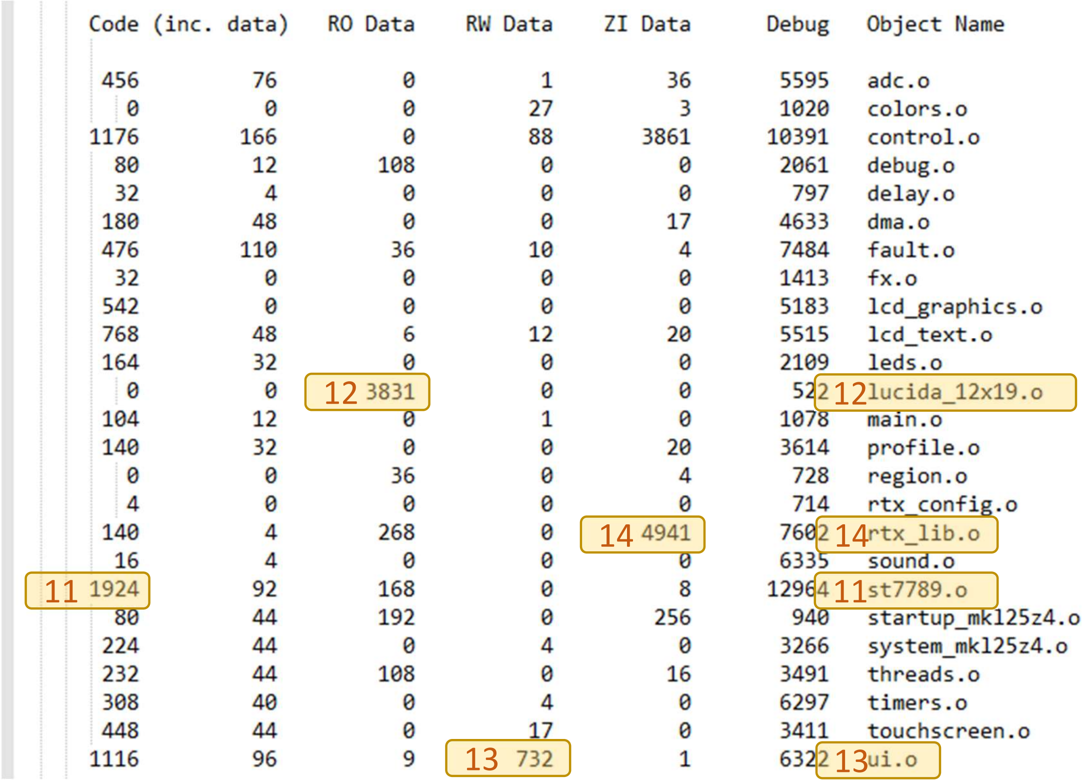


```
4981  7  svcRtxThreadNew        0x00004821   Thumb Code 7  484   rtx_thread.o(.text.svcRtxThreadNew)
5040  8  os_mem                 0x200005c8   Data       8  4096  rtx_lib.o(.bss.os)
5245  9  __dmul                 0x000009c9   Thumb Code 9  558   dmul.o(.text)
5416  10 Lucida_Console12x19    0x000050e0   Data       10 3831  lucida_12x19.o(.rodata.Lucida_Console12x19)
```

Examine the **first** part of the **Image component sizes** section to determine the memory requirements of the **program objects**.

11. Which object member has the largest component of type “Code (inc. Data)” (first column), and how large is that component?
12. Which object member has the largest component of type “RO Data”, and how large is that component?

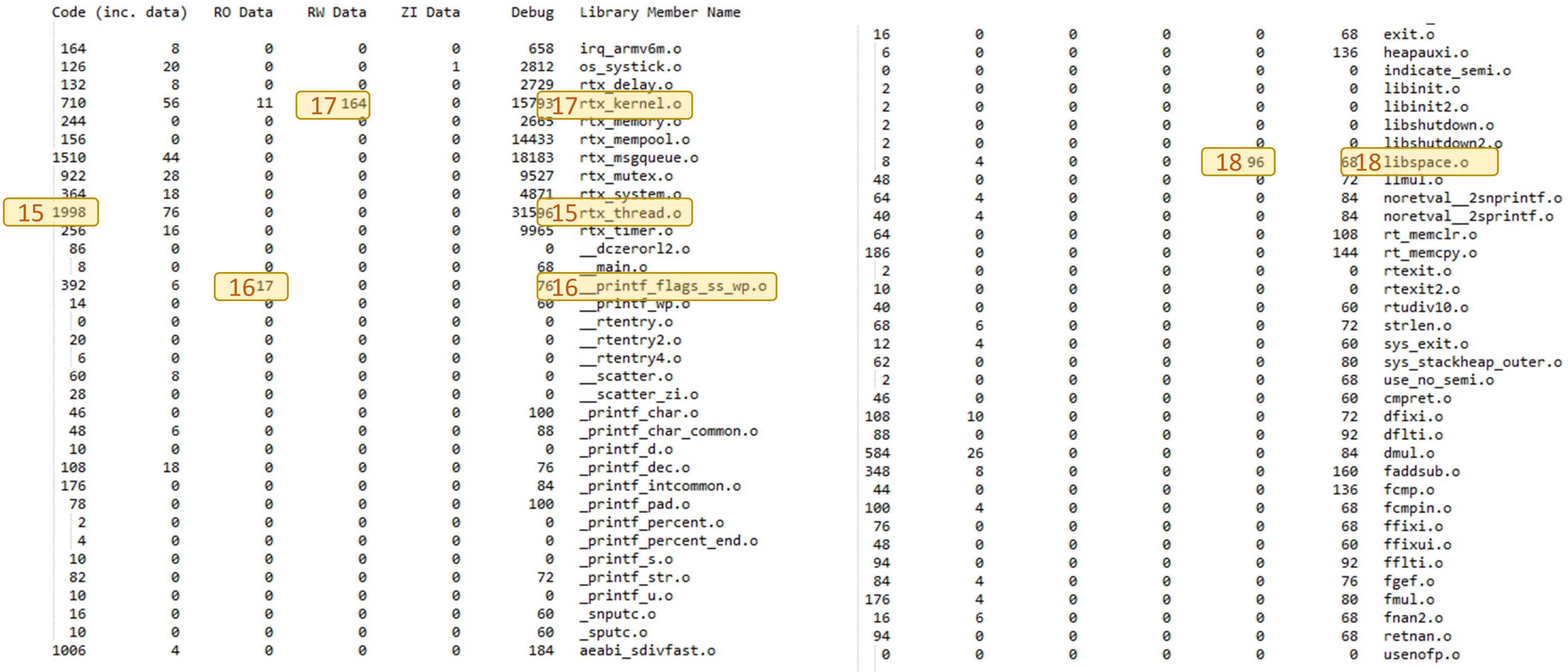
13. Which object member has the largest component of type “RW Data”, and how large is that component?
14. Which object member has the largest component of type “ZI Data”, and how large is that component?



```
Code (inc. data)   RO Data    RW Data    ZI Data      Debug   Object Name

 456         76          0          1         36       5595   adc.o
   0          0          0         27          3       1020   colors.o
1176        166          0         88       3861      10391   control.o
  80         12        108          0          0       2061   debug.o
  32          4          0          0          0        797   delay.o
 180         48          0          0         17       4633   dma.o
 476        110         36         10          4       7484   fault.o
  32          0          0          0          0       1413   fx.o
 542          0          0          0          0       5183   lcd_graphics.o
 768         48          6         12         20       5515   lcd_text.o
 164         32          0          0          0       2109   leds.o
   0          0       3831          0          0       5222   lucida_12x19.o
 104         12          0          1          0       1078   main.o
 140         32          0          0         20       3614   profile.o
   0          0         36          0          4        728   region.o
   4          0          0          0          0        714   rtx_config.o
 140          4        268          0       4941      76022   rtx_lib.o
  16          4          0          0          0       6335   sound.o
1924         92        168          0          8     129644   st7789.o
  80         44        192          0        256        940   startup_mkl25z4.o
 224         44          0          4          0       3266   system_mkl25z4.o
 232         44        108          0         16       3491   threads.o
 308         40          0          4          0       6297   timers.o
 448         44          0         17          0       3411   touchscreen.o
1116         96          9        732          1       6322   ui.o
```

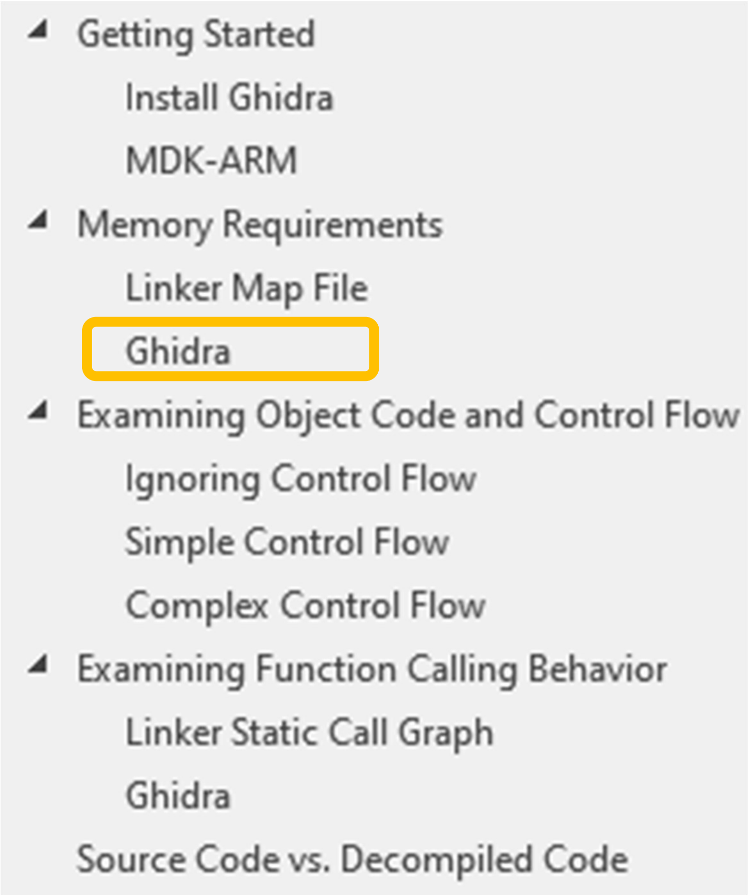
Examine the **second** part of the **Image component sizes** section (with the last column **Library Member Name**) to determine the memory requirements of the **library member objects.**

15. Which library member has the largest component of type “Code (inc. Data)” (first column), and how large is that component?
16. Which library member has the largest component of type “RO Data”, and how large is that component?
17. Which library member has the largest component of type “RW Data”, and how large is that component?
18. Which library member has the largest component of type “ZI Data”, and how large is that component?



```
Code (inc. data)   RO Data    RW Data    ZI Data      Debug   Library Member Name

 164          8          0          0          0        658   irq_armv6m.o
 126         20          0          0          1       2812   os_systick.o
 132          8          0          0          0       2729   rtx_delay.o
 710         56         11     17 164          0      15793 17 rtx_kernel.o
 244          0          0          0          0       2665   rtx_memory.o
 156          0          0          0          0      14433   rtx_mempool.o
1510         44          0          0          0      18183   rtx_msgqueue.o
 922         28          0          0          0       9527   rtx_mutex.o
 364         18          0          0          0       4871   rtx_system.o
15 1998      76          0          0          0      31596 15 rtx_thread.o
 256         16          0          0          0       9965   rtx_timer.o
  86          0          0          0          0          0   __dczerorl2.o
   8          0          0          0          0         68   __main.o
 392          6      16 17          0          0         76 16 __printf_flags_ss_wp.o
  14          0          0          0          0         60   __printf_wp.o
   0          0          0          0          0          0   __rtentry.o
  20          0          0          0          0          0   __rtentry2.o
   6          0          0          0          0          0   __rtentry4.o
  60          8          0          0          0          0   __scatter.o
  28          0          0          0          0          0   __scatter_zi.o
  46          0          0          0          0        100   _printf_char.o
  48          6          0          0          0         88   _printf_char_common.o
  10          0          0          0          0          0   _printf_d.o
 108         18          0          0          0         76   _printf_dec.o
 176          0          0          0          0         84   _printf_intcommon.o
  78          0          0          0          0        100   _printf_pad.o
   2          0          0          0          0          0   _printf_percent.o
   4          0          0          0          0          0   _printf_percent_end.o
  10          0          0          0          0          0   _printf_s.o
  82          0          0          0          0         72   _printf_str.o
  10          0          0          0          0          0   _printf_u.o
  16          0          0          0          0         60   _snputc.o
  10          0          0          0          0         60   _sputc.o
1006          4          0          0          0        184   aeabi_sdivfast.o
```

```
  16          0          0          0          0         68   exit.o
   6          0          0          0          0        136   heapauxi.o
   0          0          0          0          0          0   indicate_semi.o
   2          0          0          0          0          0   libinit.o
   2          0          0          0          0          0   libinit2.o
   2          0          0          0          0          0   libshutdown.o
   2          0          0          0          0          0   libshutdown2.o
   8          4          0          0      18 96       68 18 libspace.o
  48          0          0          0          0         72   llmul.o
  64          4          0          0          0         84   noretval__2snprintf.o
  40          4          0          0          0         84   noretval__2sprintf.o
  64          0          0          0          0        108   rt_memclr.o
 186          0          0          0          0        144   rt_memcpy.o
   2          0          0          0          0          0   rtexit.o
  10          0          0          0          0          0   rtexit2.o
  40          0          0          0          0         60   rtudiv10.o
  68          6          0          0          0         72   strlen.o
  12          4          0          0          0         60   sys_exit.o
  62          0          0          0          0         80   sys_stackheap_outer.o
   2          0          0          0          0         68   use_no_semi.o
  46          0          0          0          0         60   cmpret.o
 108         10          0          0          0         72   dfixi.o
  88          0          0          0          0         92   dflti.o
 584         26          0          0          0         84   dmul.o
 348          8          0          0          0        160   faddsub.o
  44          0          0          0          0        136   fcmp.o
 100          4          0          0          0         68   fcmpin.o
  76          0          0          0          0         68   ffixi.o
  48          0          0          0          0         60   ffixui.o
  94          0          0          0          0         92   fflti.o
  84          4          0          0          0         76   fgef.o
 176          4          0          0          0         80   fmul.o
  16          6          0          0          0         68   fnan2.o
  94          0          0          0          0         68   retnan.o
   0          0          0          0          0          0   usenofp.o
```



◢ Getting Started

- Install Ghidra
- MDK-ARM

◢ Memory Requirements

- Linker Map File
- Ghidra

◢ Examining Object Code and Control Flow

- Ignoring Control Flow
- Simple Control Flow
- Complex Control Flow

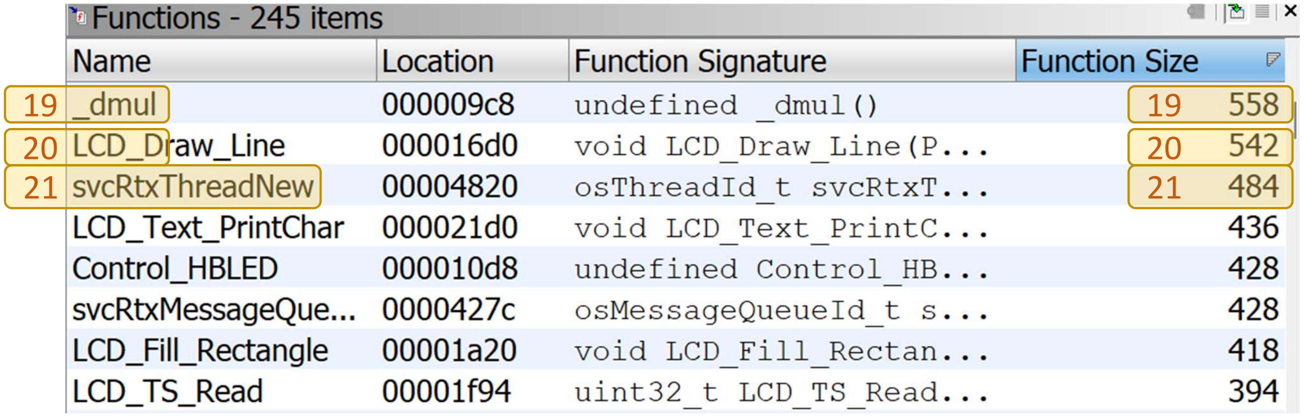
◢ Examining Function Calling Behavior

- Linker Static Call Graph
- Ghidra

Source Code vs. Decompiled Code

In Ghidra, open the Functions window (Window -> Functions). Click on the Function Size to sort by size.

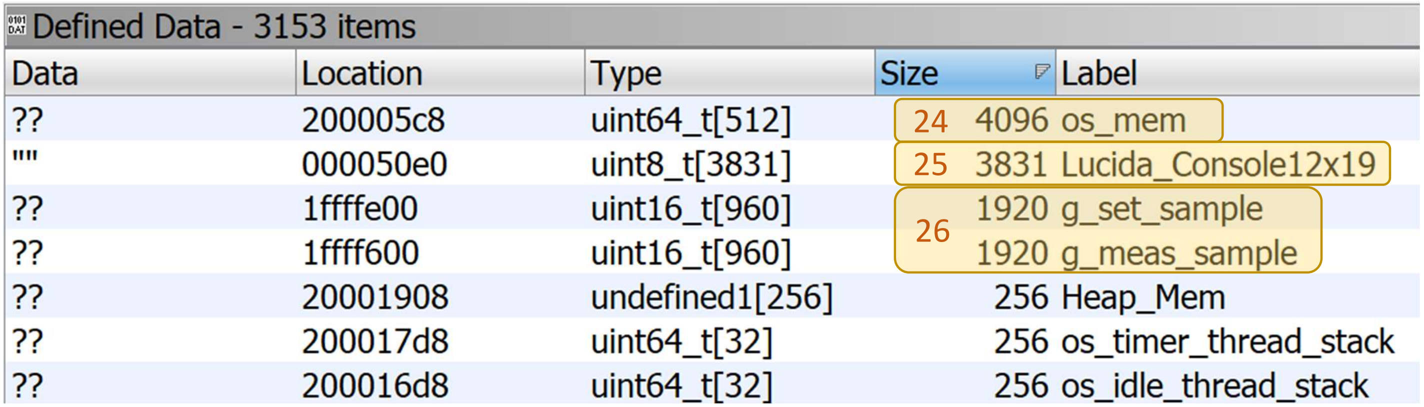
19. What are the name and size of the largest function?
20. What are the name and size of the second-largest function?
21. What are the name and size of the third-largest function?
22. Do this information match what you found above in the map file?
23. Does Ghidra differentiate between local and global symbols?



Functions - 245 items

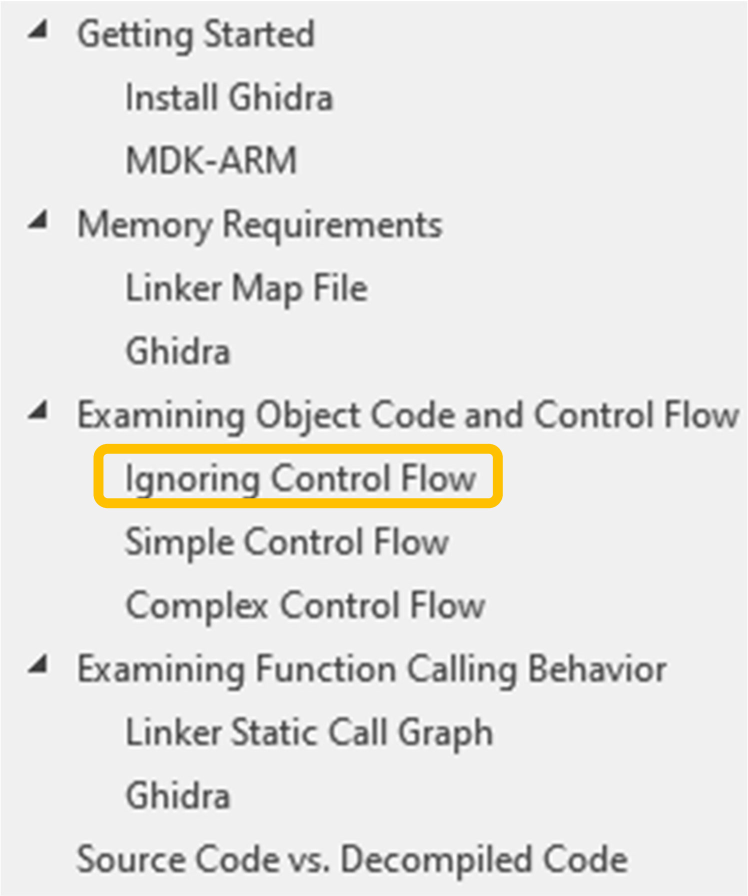
Name	Location	Function Signature	Function Size
_dmul	000009c8	undefined _dmul()	558
LCD_Draw_Line	000016d0	void LCD_Draw_Line(P...	542
svcRtxThreadNew	00004820	osThreadId_t svcRtxT...	484
LCD_Text_PrintChar	000021d0	void LCD_Text_PrintC...	436
Control_HBLED	000010d8	undefined Control_HB...	428
svcRtxMessageQue...	0000427c	osMessageQueueId_t s...	428
LCD_Fill_Rectangle	00001a20	void LCD_Fill_Rectan...	418
LCD_TS_Read	00001f94	uint32_t LCD_TS_Read...	394

24. What are the name (label) and size in bytes of the largest data item in memory (not in _elf*, .symtab, .strtab, .debug_frame, .comment, etc.)?
25. What are the name (label) and size in bytes of the second-largest data item in memory (not in _elf*, .symtab, .strtab, .debug_frame, .comment, etc.)?
26. What are the name (label) and size in bytes of the third-largest data item in memory (not in _elf*, .symtab, .strtab, .debug_frame, .comment, etc.)?
27. Does this information match what you found above in the map file? If not, try to explain the difference.



Defined Data - 3153 items

Data	Location	Type	Size	Label
??	200005c8	uint64_t[512]	4096	os_mem
""	000050e0	uint8_t[3831]	3831	Lucida_Console12x19
??	1ffffe00	uint16_t[960]	1920	g_set_sample
??	1ffff600	uint16_t[960]	1920	g_meas_sample
??	20001908	undefined1[256]	256	Heap_Mem
??	200017d8	uint64_t[32]	256	os_timer_thread_stack
??	200016d8	uint64_t[32]	256	os_idle_thread_stack



- ◢ Getting Started
  - Install Ghidra
  - MDK-ARM
- ◢ Memory Requirements
  - Linker Map File
  - Ghidra
- ◢ Examining Object Code and Control Flow
  - Ignoring Control Flow
  - Simple Control Flow
  - Complex Control Flow
- ◢ Examining Function Calling Behavior
  - Linker Static Call Graph
  - Ghidra
- Source Code vs. Decompiled Code

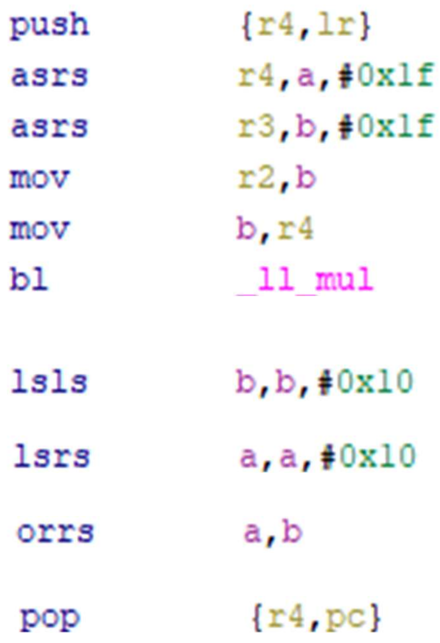
For the following questions, look in the disassembly window to see the object code which implements the function.

1. Which is the first instruction of the function `Multiply_FX`? Include address, machine code, opcode and operand.
2. **ECE 561 Only:** Enter the start address of the function from the disassembly window and the start address for the function listed in the map file. Explain why they don't exactly match.
3. When BL.W __aeabi_lmul executes, which registers hold variable pa?
4. When BL.W __aeabi_lmul executes, which registers hold variable pb?
5. Immediately after BL.W __aeabi_lmul completes, which registers hold variable p?
6. Which instruction performs the return-from-subroutine? Include address, machine code, opcode and operand.

Ignoring Control Flow

Arguments longer than 1 word are ordered little-endian in registers or memory

Register Contents After Instruction Executes



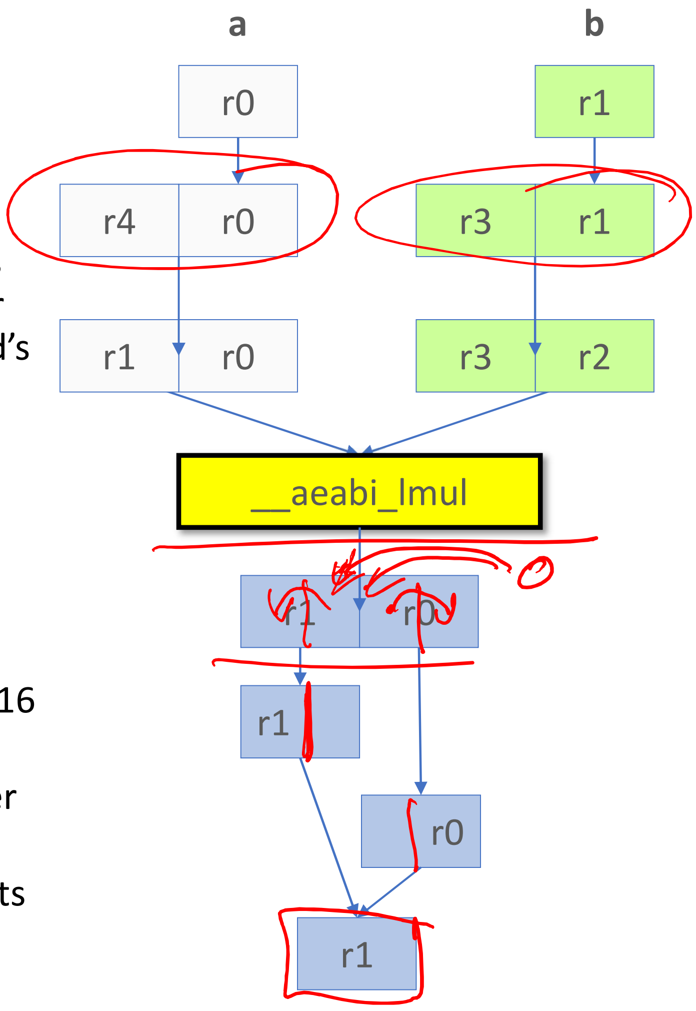
```
push    {r4,lr}
asrs    r4,a,#0x1f
asrs    r3,b,#0x1f
mov     r2,b
mov     b,r4
bl      _ll_mul

lsls    b,b,#0x10

lsrs    a,a,#0x10

orrs    a,b

pop     {r4,pc}
```

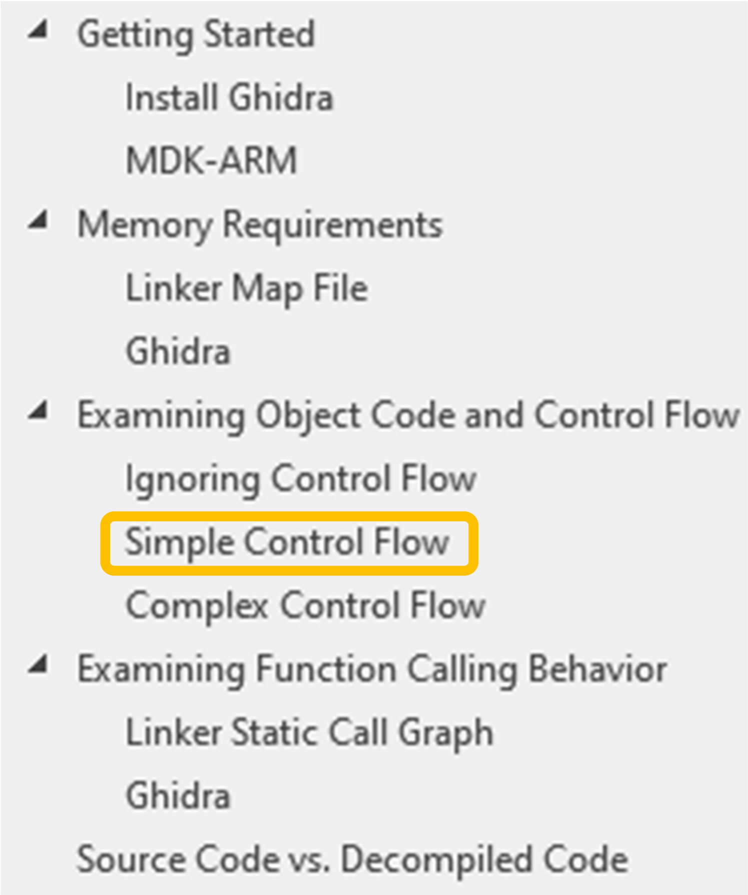
r0 ("a")		r1 ("b")		r2	r3	r4	
a		**b**					
a		**b**				**pa** MSW	sign extend **a** to get **pa** bits 63-32
a		**b**			**pb** MSW	**pa** MSW	sign extend **b** to get **pb** bits 63-32
a		**b**		**b**	**pb** MSW	**pa** MSW	
a		**pa** MSW		**b**	**pb** MSW	**pa** MSW	
p LSW		**p** MSW					r0, r1 as two words
p 3rdMSHW	**p** LSHW	**p** MSHW	**p** 2ndMSHW				r0, r1 as four half-words
p 3rdMSHW	**p** LSHW	**p** 2ndMSHW	0				shift p MSW left to get lower halfword, zero out upper halfword
0	**p** 3rdMSHW	**p** 2ndMSHW	0				shift p LSW right to get upper halfword, zero out lower halfword
p 2ndMSHW	**p** 3rdMSHW	**p** 2ndMSHW	0				OR together results to get middle 32 bits of **p**

Signed 16.16 * 16.16 Explained

```
PUSH    {r4,lr}
ASRS    r4,r0,#31
ASRS    r3,r1,#31



MOV     r2,r1
MOV     r1,r4

BL      __aeabi_lmul

LSLS    r1,r1,#16
LSRS    r0,r0,#16
ORRS    r0,r0,r1

POP     {r4,pc}
```

- Sign-extension to 64 bits
  - a (r0) and b (r1) to 64 bits pa (r4:r0) and pb (r3:r2)
  - ASRS: arithmetic shift right performs sign extension by setting all of upper word's sign bits to match lower word's sign
- Move pa and pb into argument registers (r1:r0 and r3:r2)
- Call __aeabi_lmul for long multiply
- Extract middle 32 bits of result
  - LSLS: logical shift left extracts lower 16 bits of r1
  - LSRS: logical shift right extracts upper 16 bits of r0
  - ORRS: merges together middle 32 bits

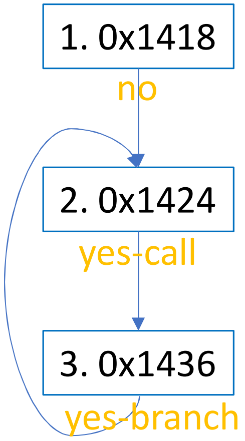


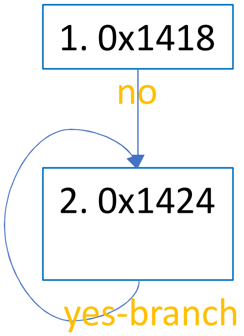


- Getting Started
  - Install Ghidra
  - MDK-ARM
- Memory Requirements
  - Linker Map File
  - Ghidra
- Examining Object Code and Control Flow
  - Ignoring Control Flow
  - Simple Control Flow
  - Complex Control Flow
- Examining Function Calling Behavior
  - Linker Static Call Graph
  - Ghidra
- Source Code vs. Decompiled Code

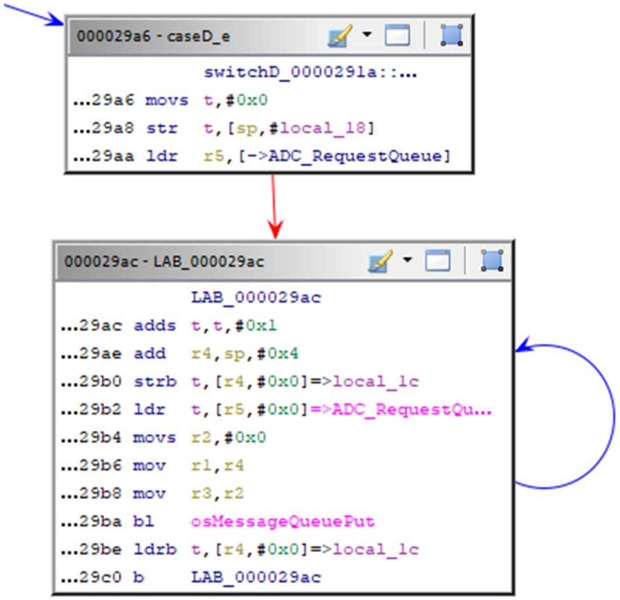
Simple Control Flow #7 (v1.0): CFG of Fault_Fill_Queue

- 2 or 3 basic blocks ok since bl osMessageQueuePut may be interpreted as ending basic block
- Note that there should not be a basic block for the subroutine osMessageQueuePut. The call instruction (BL) is in the 2nd basic block.





BBs	no	yes-branch	yes-call	yes-return
3	1	1	1	0
2	1	1	0	0

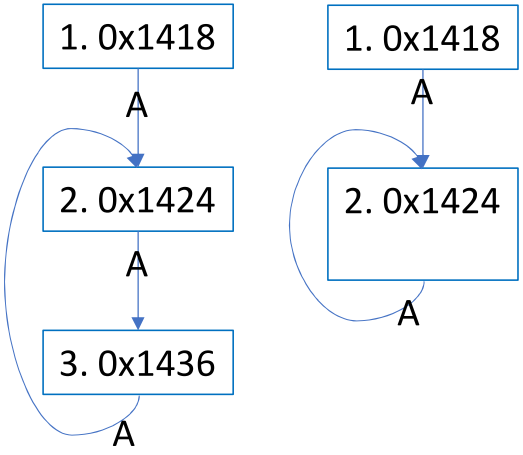


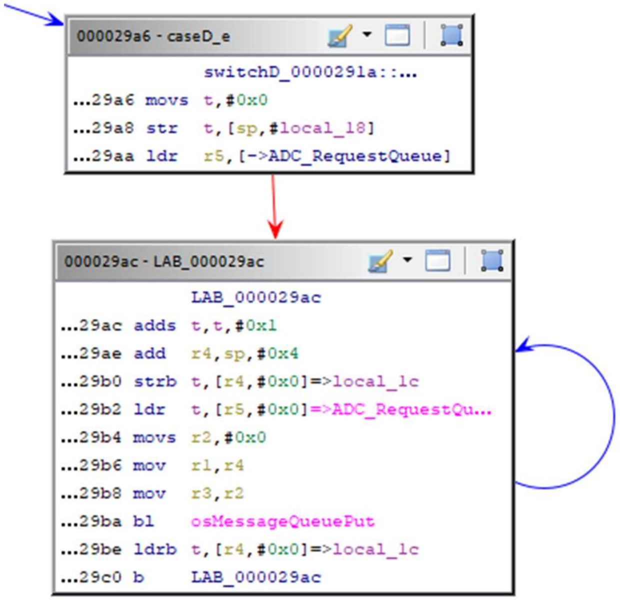
```
             switchD_0000291a::...
...29a6 movs t,#0x0
...29a8 str  t,[sp,#local_18]
...29aa ldr  r5,[->ADC_RequestQueue]
```

```
             LAB_000029ac
...29ac adds t,t,#0x1
...29ae add  r4,sp,#0x4
...29b0 strb t,[r4,#0x0]=>local_1c
...29b2 ldr  t,[r5,#0x0]=>ADC_RequestQu...
...29b4 movs r2,#0x0
...29b6 mov  r1,r4
...29b8 mov  r3,r2
...29ba bl   osMessageQueuePut
...29be ldrb t,[r4,#0x0]=>local_1c
...29c0 b    LAB_000029ac
```

Simple Control Flow #8 (v1.0): CFG of Fault_Fill_Queue

- 2 or 3 basic blocks ok since bl osMessageQueuePut may be interpreted as ending basic block
- Addresses should match basic blocks listed in #7, not necessarily what's in this diagram.
- Labeled edges connecting basic blocks as shown
- Note that there should not be a basic block for the subroutine osMessageQueuePut. The call instruction (BL) is in the 2nd basic block.





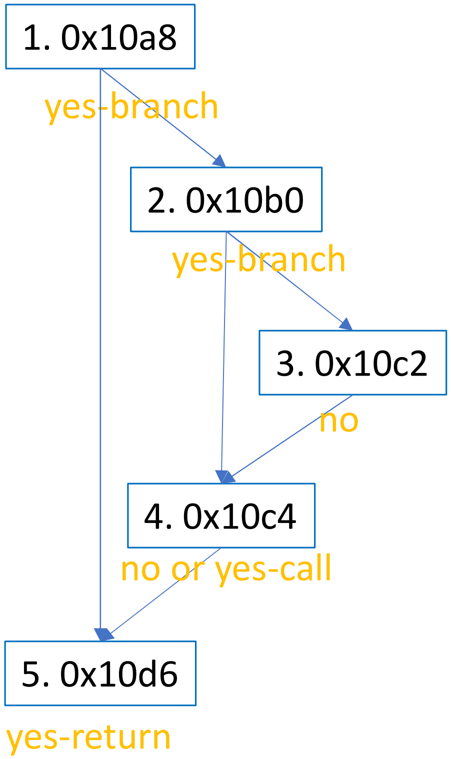
```
000029a6 - caseD_e
                switchD_0000291a::...
...29a6 movs  t,#0x0
...29a8 str   t,[sp,#local_18]
...29aa ldr   r5,[->ADC_RequestQueue]
```

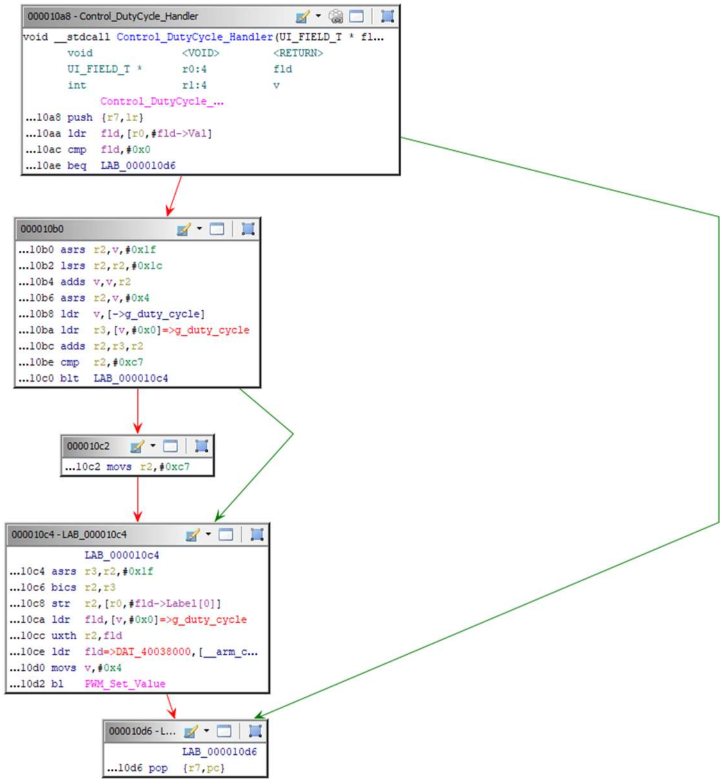
```
000029ac - LAB_000029ac
                LAB_000029ac
...29ac adds  t,t,#0x1
...29ae add   r4,sp,#0x4
...29b0 strb  t,[r4,#0x0]=>local_1c
...29b2 ldr   t,[r5,#0x0]=>ADC_RequestQu...
...29b4 movs  r2,#0x0
...29b6 mov   r1,r4
...29b8 mov   r3,r2
...29ba bl    osMessageQueuePut
...29be ldrb  t,[r4,#0x0]=>local_1c
...29c0 b     LAB_000029ac
```

Simple Control Flow #7 (v1.1+): CFG of Control_DutyCycle_Handler

BBs	no	yes-branch	yes-call	yes-return
5	2	2	0	1
5	1	2	1	1

- Note that there should not be a basic block for the subroutine PWM_Set_Value. The call instruction (BL) is in the 4th basic block.





```
void __stdcall Control_DutyCycle_Handler(UI_FIELD_T * fl...
       void              <VOID>        <RETURN>
       UI_FIELD_T *      r0:4          fld
       int               r1:4          v
            Control_DutyCycle_...
...10a8 push {r7,lr}
...10aa ldr  fld,[r0,#fld->Val]
...10ac cmp  fld,#0x0
...10ae beq  LAB_000010d6
```

```
...10b0 asrs r2,v,#0x1f
...10b2 lsrs r2,r2,#0x1c
...10b4 adds v,v,r2
...10b6 asrs r2,v,#0x4
...10b8 ldr  v,[->g_duty_cycle]
...10ba ldr  r3,[v,#0x0]=>g_duty_cycle
...10bc adds r2,r3,r2
...10be cmp  r2,#0xc7
...10c0 blt  LAB_000010c4
```

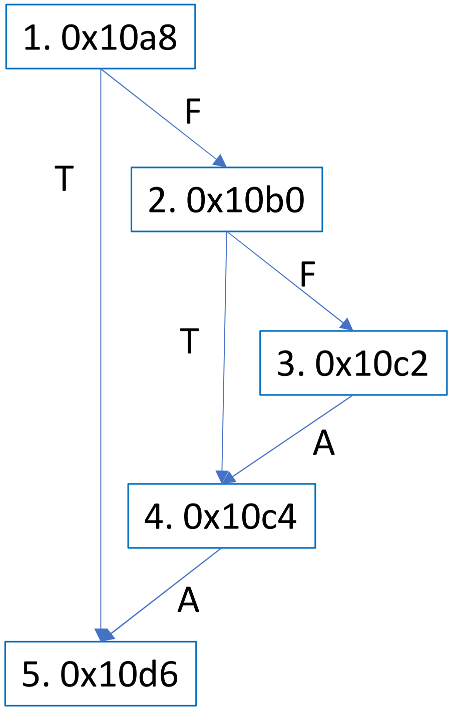
```
...10c2 movs r2,#0xc7
```

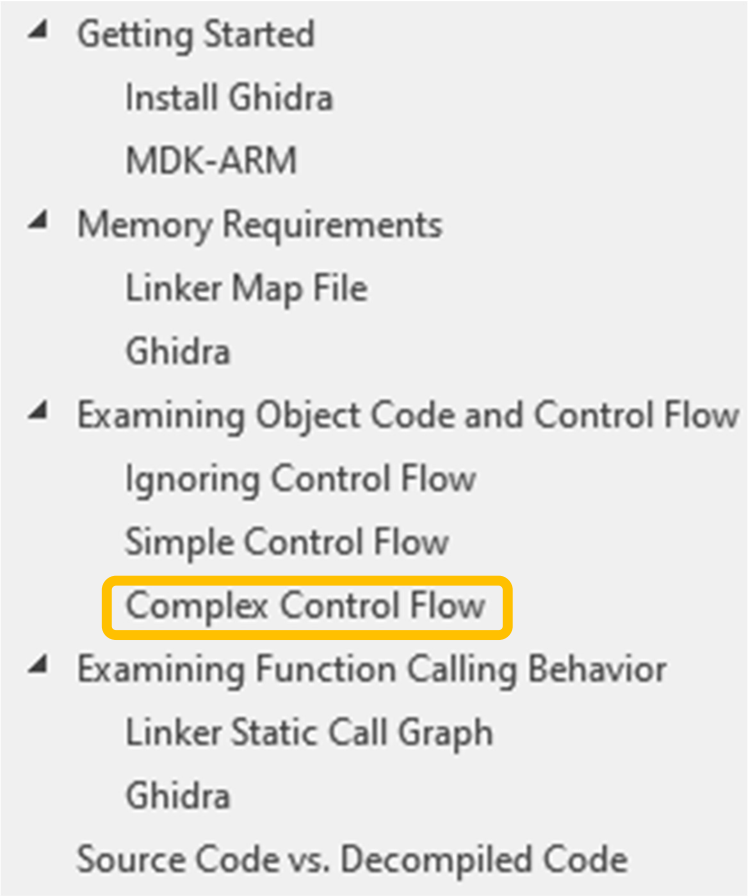
```
             LAB_000010c4
...10c4 asrs r3,r2,#0x1f
...10c6 bics r2,r3
...10c8 str  r2,[r0,#fld->Label[0]]
...10ca ldr  fld,[v,#0x0]=>g_duty_cycle
...10cc uxth r2,fld
...10ce ldr  fld=>DAT_40038000,[__arm_c...
...10d0 movs v,#0x4
...10d2 bl   PWM_Set_Value
```

```
             LAB_000010d6
...10d6 pop  {r7,pc}
```

Simple Control Flow #8 (v1.1+): CFG of Control_DutyCycle_Handler

- 5 basic blocks, labeled as shown
    - Addresses should match basic blocks listed in #7, not necessarily what's in this diagram.
- Labeled edges connecting basic blocks as shown
- Note that there should not be a basic block for the subroutine PWM_Set_Value. The call instruction (BL) is in the 4th basic block.

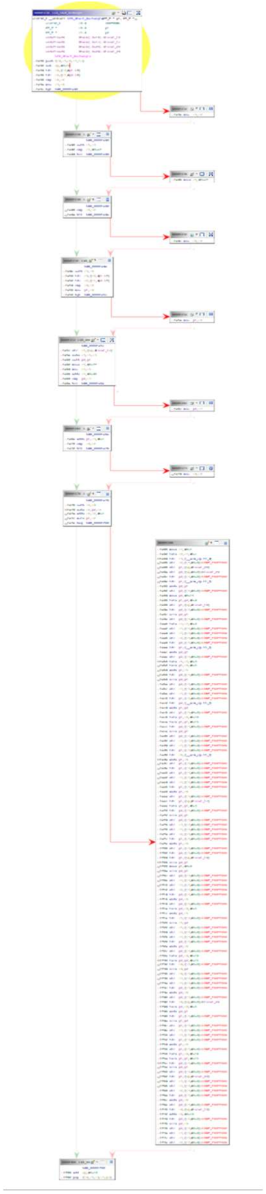




- Getting Started
  - Install Ghidra
  - MDK-ARM
- Memory Requirements
  - Linker Map File
  - Ghidra
- Examining Object Code and Control Flow
  - Ignoring Control Flow
  - Simple Control Flow
  - Complex Control Flow
- Examining Function Calling Behavior
  - Linker Static Call Graph
  - Ghidra
- Source Code vs. Decompiled Code

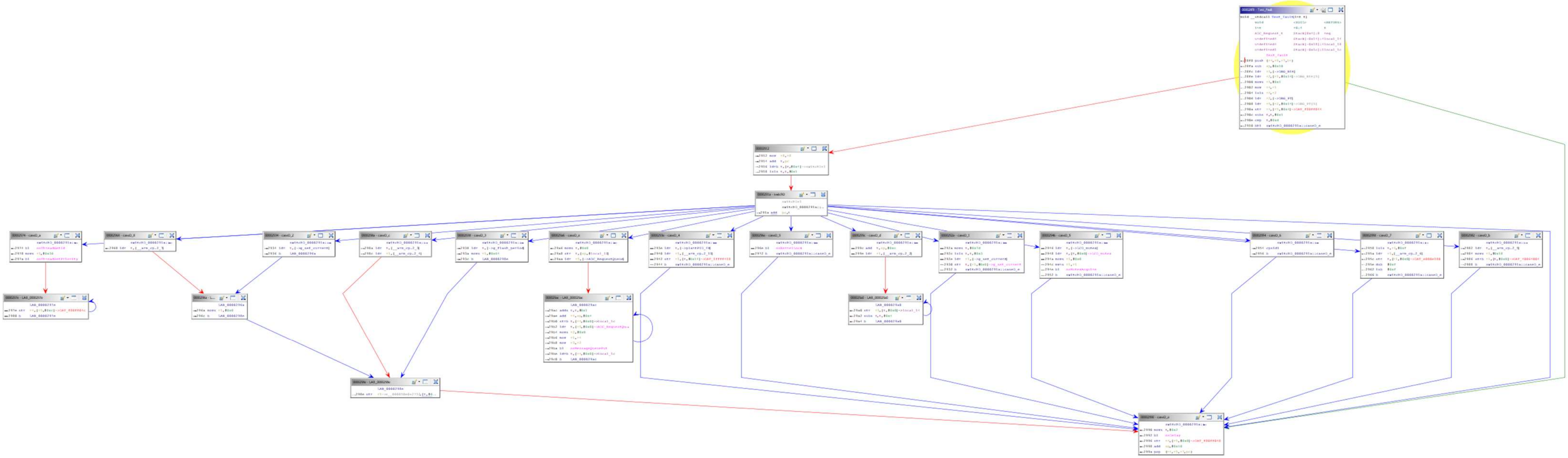
Complex Control Flow #9

- Should show the right function (LCD_Start_Rectangle) using this layout
- Text doesn't have to be legible, but basic blocks and control flow edges must be visible

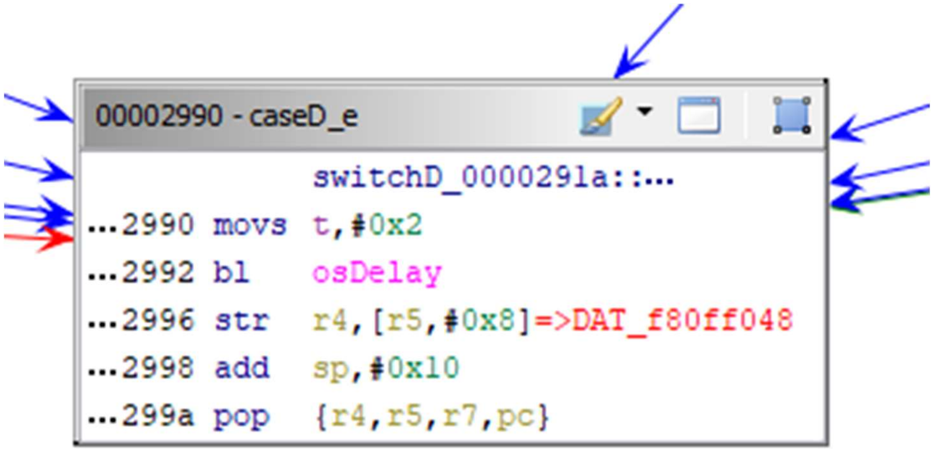


Complex Control Flow #11

- Must be from Ghidra for Test_Fault
- May be a different layout from this one
- Must show basic blocks and edges



Complex Control Flow #12

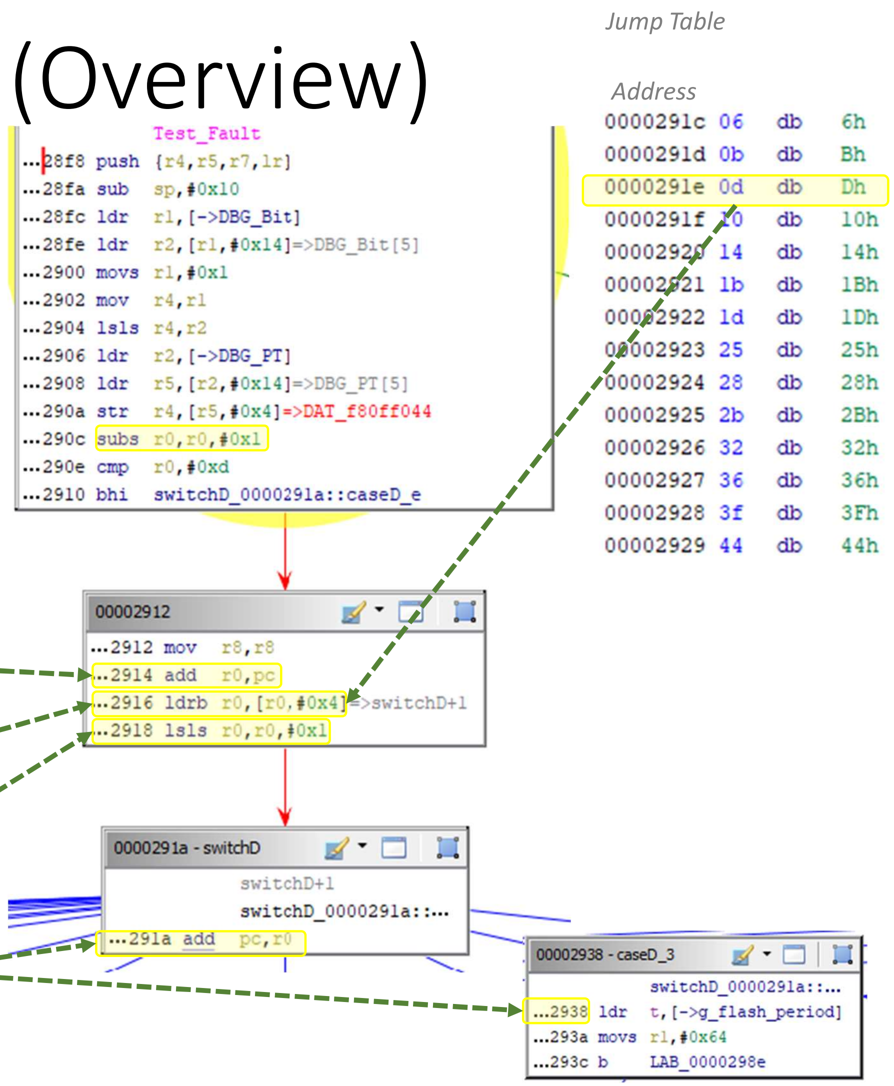


```
                 switchD_0000291a::...
...2990 movs  t,#0x2
...2992 bl    osDelay
...2996 str   r4,[r5,#0x8]=>DAT_f80ff048
...2998 add   sp,#0x10
...299a pop   {r4,r5,r7,pc}
```

Complex Control Flow #15 (Overview)

- Test case number t (in r0) is decremented by one, since table has covers cases 1 through 14. Example: t = 3, so r0 <- 3-1 = 2
- r0 is compared to the largest test case offset 0xd (14–1=13), and the result determines if bhi branches.
  - r0 > 13?
    - branch to caseD_e, which is code following last case statement, since there is no default case
  - r0 <= 13? Ex: r0 = 2 <= 13
    - *Note: pc is address of cur. instruction + 4. Arm Arch. Ref. Manual, DDI 0419C*

      > Read the PC value, that is, the address of the current instruction + 4.
    - Add pc to r0, so r0 points to …. Ex: r0 <- 2 + (0x2914 + 4) = 0x2918 + 2 = 0x291a
    - Load register r0 with byte from entry in jump table (in memory at address r0+4). Ex: r0 <- memory[0x291a+4] = memory[0x291e] = 0xd = 13
    - Shift r0 left by one bit (multiply by two) to convert offset to bytes. Ex: r0 <- 26
    - Add r0 to pc, causing the program to jump to that case's code. Ex: pc <- (0x291a + 4) + 26 = 0x291e + 26 = 0x2938



Jump Table

```
Address
0000291c 06   db   6h
0000291d 0b   db   Bh
0000291e 0d   db   Dh
0000291f 10   db   10h
00002920 14   db   14h
00002921 1b   db   1Bh
00002922 1d   db   1Dh
00002923 25   db   25h
00002924 28   db   28h
00002925 2b   db   2Bh
00002926 32   db   32h
00002927 36   db   36h
00002928 3f   db   3Fh
00002929 44   db   44h
```

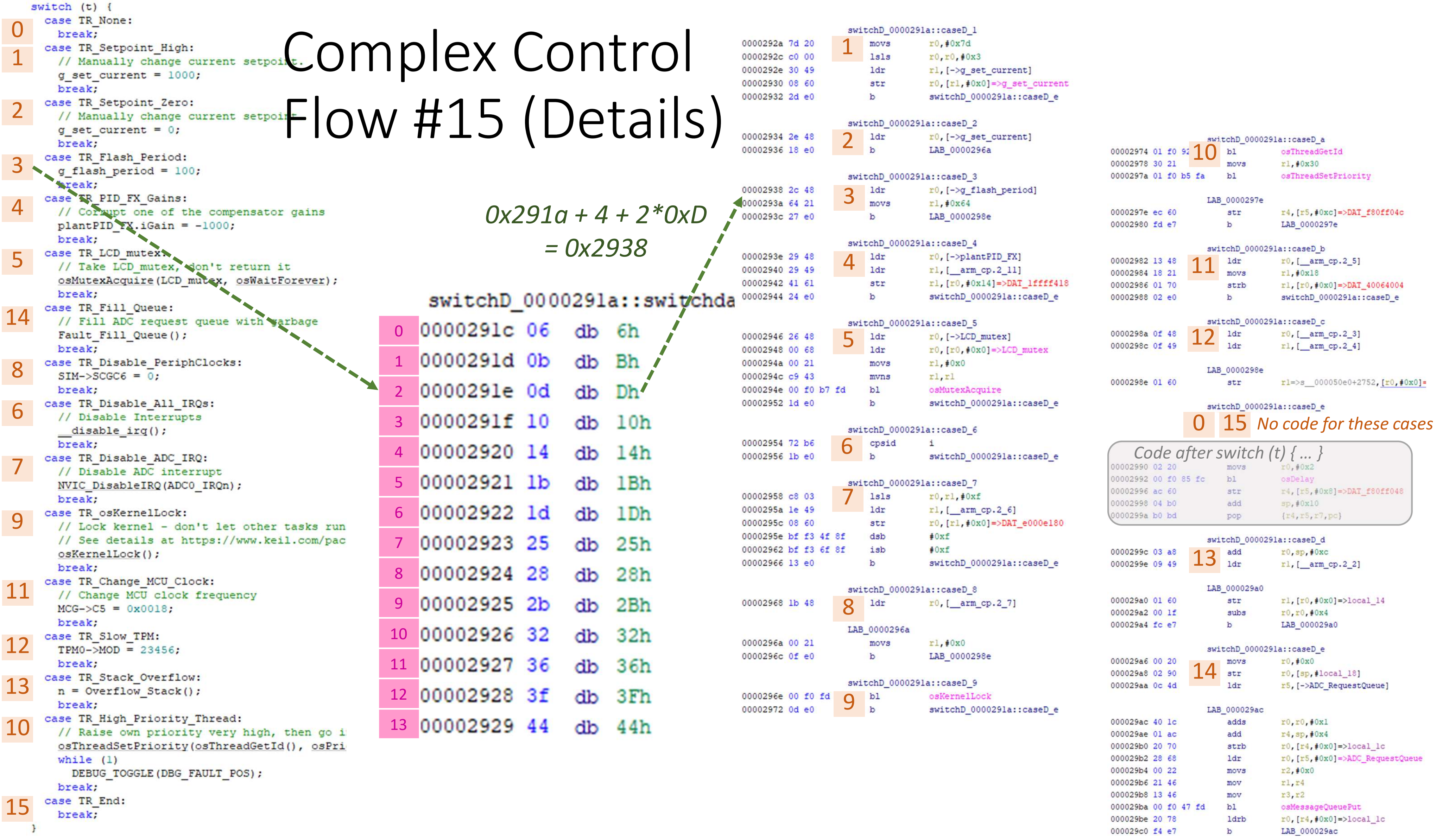
```
              Test_Fault
...28f8 push {r4,r5,r7,lr}
...28fa sub  sp,#0x10
...28fc ldr  r1,[->DBG_Bit]
...28fe ldr  r2,[r1,#0x14]=>DBG_Bit[5]
...2900 movs r1,#0x1
...2902 mov  r4,r1
...2904 lsls r4,r2
...2906 ldr  r2,[->DBG_PT]
...2908 ldr  r5,[r2,#0x14]=>DBG_PT[5]
...290a str  r4,[r5,#0x4]=>DAT_f80ff044
...290c subs r0,r0,#0x1
...290e cmp  r0,#0xd
...2910 bhi  switchD_0000291a::caseD_e
```

```
00002912
...2912 mov  r8,r8
...2914 add  r0,pc
...2916 ldrb r0,[r0,#0x4]=>switchD+1
...2918 lsls r0,r0,#0x1
```

```
0000291a - switchD
                switchD+1
                switchD_0000291a::...
...291a add     pc,r0
```

```
00002938 - caseD_3
                switchD_0000291a::...
...2938 ldr     t,[->g_flash_period]
...293a movs    r1,#0x64
...293c b       LAB_0000298e
```

Complex Control Flow #15 (Details)



```
    switch (t) {
0     case TR_None:
        break;
1     case TR_Setpoint_High:
        // Manually change current setpoint.
        g_set_current = 1000;
        break;
2     case TR_Setpoint_Zero:
        // Manually change current setpoint
        g_set_current = 0;
        break;
3     case TR_Flash_Period:
        g_flash_period = 100;
        break;
4     case TR_PID_FX_Gains:
        // Corrupt one of the compensator gains
        plantPID_FX.iGain = -1000;
        break;
5     case TR_LCD_mutex:
        // Take LCD_mutex, don't return it
        osMutexAcquire(LCD_mutex, osWaitForever);
        break;
14    case TR_Fill_Queue:
        // Fill ADC request queue with garbage
        Fault_Fill_Queue();
        break;
8     case TR_Disable_PeriphClocks:
        SIM->SCGC6 = 0;
        break;
6     case TR_Disable_All_IRQs:
        // Disable Interrupts
        __disable_irq();
        break;
7     case TR_Disable_ADC_IRQ:
        // Disable ADC interrupt
        NVIC_DisableIRQ(ADC0_IRQn);
        break;
9     case TR_osKernelLock:
        // Lock kernel - don't let other tasks run
        // See details at https://www.keil.com/pac
        osKernelLock();
        break;
11    case TR_Change_MCU_Clock:
        // Change MCU clock frequency
        MCG->C5 = 0x0018;
        break;
12    case TR_Slow_TPM:
        TPM0->MOD = 23456;
        break;
13    case TR_Stack_Overflow:
        n = Overflow_Stack();
        break;
10    case TR_High_Priority_Thread:
        // Raise own priority very high, then go i
        osThreadSetPriority(osThreadGetId(), osPri
        while (1)
          DEBUG_TOGGLE(DBG_FAULT_POS);
        break;
15    case TR_End:
        break;
    }
```

*0x291a + 4 + 2*0xD*
= 0x2938

```
     switchD_0000291a::switchda
0    0000291c 06  db  6h
1    0000291d 0b  db  Bh
2    0000291e 0d  db  Dh
3    0000291f 10  db  10h
4    00002920 14  db  14h
5    00002921 1b  db  1Bh
6    00002922 1d  db  1Dh
7    00002923 25  db  25h
8    00002924 28  db  28h
9    00002925 2b  db  2Bh
10   00002926 32  db  32h
11   00002927 36  db  36h
12   00002928 3f  db  3Fh
13   00002929 44  db  44h
```

```
                       switchD_0000291a::caseD_1
0000292a 7d 20      1  movs     r0,#0x7d
0000292c c0 00         lsls     r0,r0,#0x3
0000292e 30 49         ldr      r1,[->g_set_current]
00002930 08 60         str      r0,[r1,#0x0]=>g_set_current
00002932 2d e0         b        switchD_0000291a::caseD_e

                       switchD_0000291a::caseD_2
00002934 2e 48      2  ldr      r0,[->g_set_current]
00002936 18 e0         b        LAB_0000296a

                       switchD_0000291a::caseD_3
00002938 2c 48      3  ldr      r0,[->g_flash_period]
0000293a 64 21         movs     r1,#0x64
0000293c 27 e0         b        LAB_0000298e

                       switchD_0000291a::caseD_4
0000293e 29 48      4  ldr      r0,[->plantPID_FX]
00002940 29 49         ldr      r1,[__arm_cp.2_11]
00002942 41 61         str      r1,[r0,#0x14]=>DAT_1ffff418
00002944 24 e0         b        switchD_0000291a::caseD_e

                       switchD_0000291a::caseD_5
00002946 26 48      5  ldr      r0,[->LCD_mutex]
00002948 00 68         ldr      r0,[r0,#0x0]=>LCD_mutex
0000294a 00 21         movs     r1,#0x0
0000294c c9 43         mvns     r1,r1
0000294e 00 f0 b7 fd   bl       osMutexAcquire
00002952 1d e0         b        switchD_0000291a::caseD_e

                       switchD_0000291a::caseD_6
00002954 72 b6      6  cpsid    i
00002956 1b e0         b        switchD_0000291a::caseD_e

                       switchD_0000291a::caseD_7
00002958 c8 03      7  lsls     r0,r1,#0xf
0000295a 1e 49         ldr      r1,[__arm_cp.2_6]
0000295c 08 60         str      r0,[r1,#0x0]=>DAT_e000e180
0000295e bf f3 4f 8f   dsb      #0xf
00002962 bf f3 6f 8f   isb      #0xf
00002966 13 e0         b        switchD_0000291a::caseD_e

                       switchD_0000291a::caseD_8
00002968 1b 48      8  ldr      r0,[__arm_cp.2_7]

                       LAB_0000296a
0000296a 00 21         movs     r1,#0x0
0000296c 0f e0         b        LAB_0000298e

                       switchD_0000291a::caseD_9
0000296e 00 f0 fd   9  bl       osKernelLock
00002972 0d e0         b        switchD_0000291a::caseD_e
```

```
                       switchD_0000291a::caseD_a
00002974 01 f0 92  10  bl       osThreadGetId
00002978 30 21         movs     r1,#0x30
0000297a 01 f0 b5 fa   bl       osThreadSetPriority

                       LAB_0000297e
0000297e ec 60         str      r4,[r5,#0xc]=>DAT_f80ff04c
00002980 fd e7         b        LAB_0000297e

                       switchD_0000291a::caseD_b
00002982 13 48     11  ldr      r0,[__arm_cp.2_5]
00002984 18 21         movs     r1,#0x18
00002986 01 70         strb     r1,[r0,#0x0]=>DAT_40064004
00002988 02 e0         b        switchD_0000291a::caseD_e

                       switchD_0000291a::caseD_c
0000298a 0f 48     12  ldr      r0,[__arm_cp.2_3]
0000298c 0f 49         ldr      r1,[__arm_cp.2_4]

                       LAB_0000298e
0000298e 01 60         str      r1=>s__000050e0+2752,[r0,#0x0]=

                       switchD_0000291a::caseD_e
```

0 15 *No code for these cases*

Code after switch (t) { … }

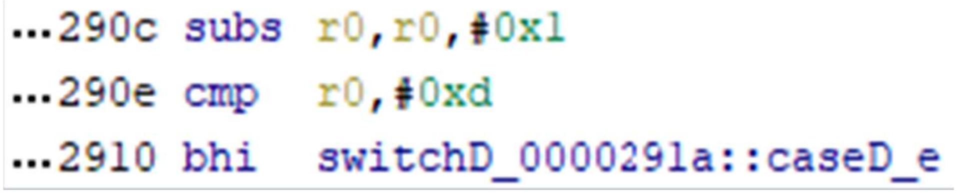
```
00002990 02 20         movs     r0,#0x2
00002992 00 f0 85 fc   bl       osDelay
00002996 ac 60         str      r4,[r5,#0x8]=>DAT_f80ff048
00002998 04 b0         add      sp,#0x10
0000299a b0 bd         pop      {r4,r5,r7,pc}
```

```
                       switchD_0000291a::caseD_d
0000299c 03 a8     13  add      r0,sp,#0xc
0000299e 09 49         ldr      r1,[__arm_cp.2_2]

                       LAB_000029a0
000029a0 01 60         str      r1,[r0,#0x0]=>local_14
000029a2 00 1f         subs     r0,r0,#0x4
000029a4 fc e7         b        LAB_000029a0

                       switchD_0000291a::caseD_e
000029a6 00 20     14  movs     r0,#0x0
000029a8 02 90         str      r0,[sp,#local_18]
000029aa 0c 4d         ldr      r5,[->ADC_RequestQueue]

                       LAB_000029ac
000029ac 40 1c         adds     r0,r0,#0x1
000029ae 01 ac         add      r4,sp,#0x4
000029b0 20 70         strb     r0,[r4,#0x0]=>local_1c
000029b2 28 68         ldr      r0,[r5,#0x0]=>ADC_RequestQueue
000029b4 00 22         movs     r2,#0x0
000029b6 21 46         mov      r1,r4
000029b8 13 46         mov      r3,r2
000029ba 00 f0 47 fd   bl       osMessageQueuePut
000029be 20 78         ldrb     r0,[r4,#0x0]=>local_1c
000029c0 f4 e7         b        LAB_000029ac
```

Complex Control Flow #17

ECE 561 only: If the number of test cases don’t match, try to explain why.

- Test cases TR_None (0) and TR_End (15) both do nothing, so there is no case code generated for them.
- Before the jump table, those cases are detected with one test
- They are both handled by this code:
  - Subtract 1 from r0
    - 0 -> 0xffffffff
    - 15 -> 14 (0x0000000e)
  - Compare with 0xd (13)
  - bhi: Branch if unsigned greater than is true
  - Both are higher than 0xd, so branch to caseD_e, which is code which follows the switch statement

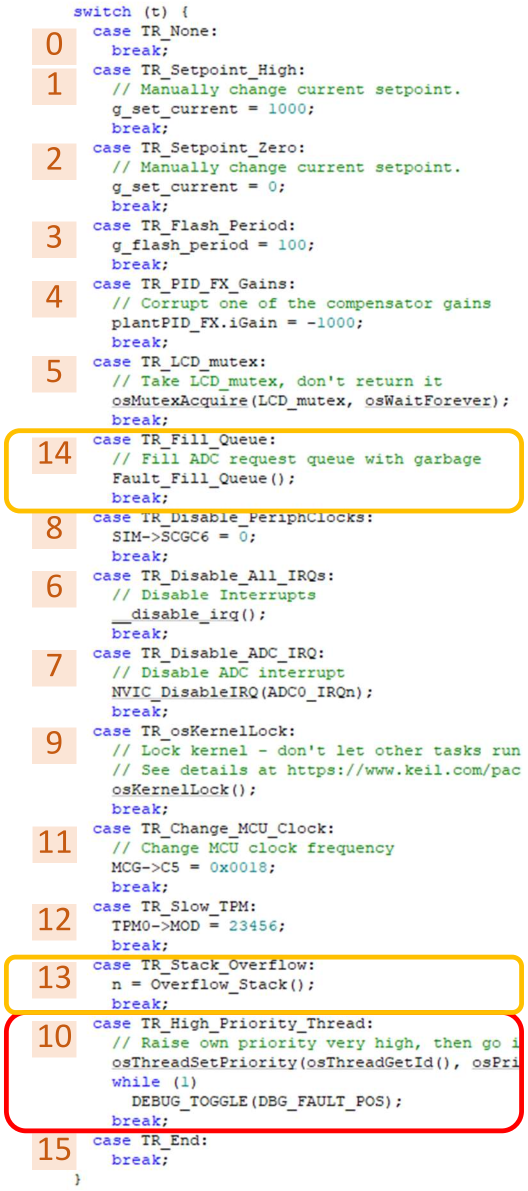


```
...290c subs  r0,r0,#0x1
...290e cmp   r0,#0xd
...2910 bhi   switchD_0000291a::caseD_e
```

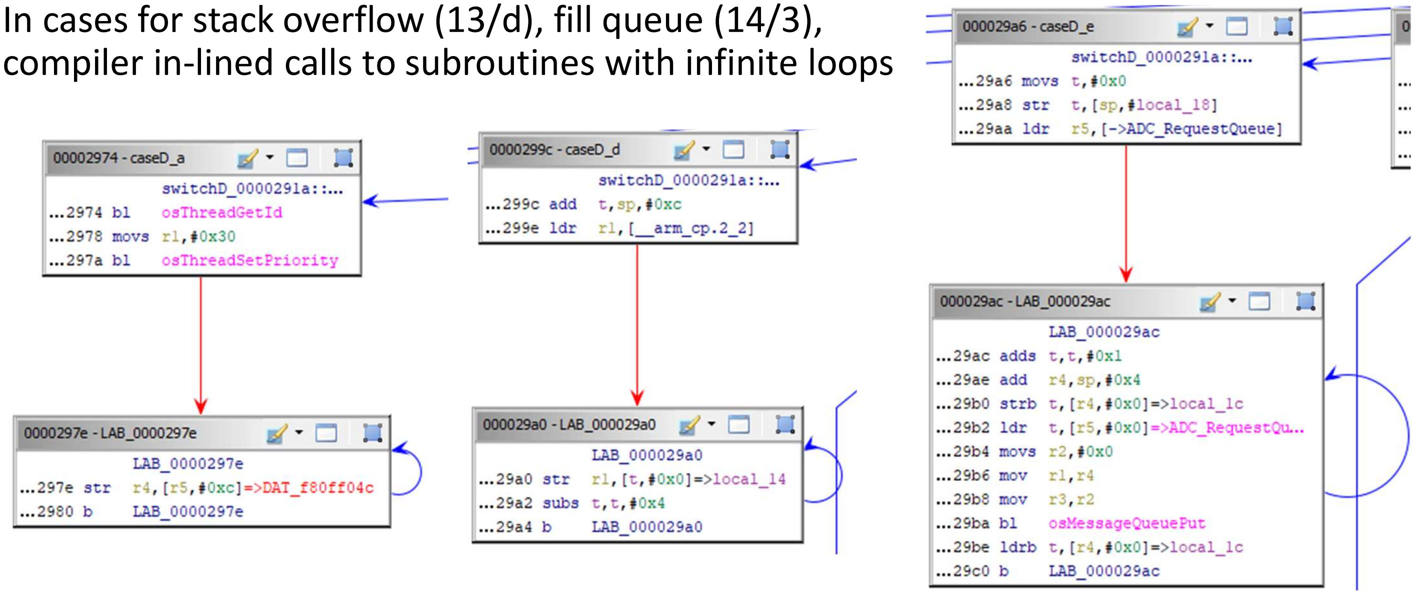
Complex Control Flow #20.

ECE 561 only: If the number of test cases ending in infinite loops don’t match, try to explain what happened.

- In source code
  - one case ends in explicit infinite loop: high priority thread (10)
  - Two cases call subroutine with infinite loop: stack overflow (13), fill queue (14)
- CFG shows three cases ending in infinite loops, partially matching source code.
  - In cases for stack overflow (13/d), fill queue (14/3), compiler in-lined calls to subroutines with infinite loops



```
switch (t) {
  case TR_None:
    break;
  case TR_Setpoint_High:
    // Manually change current setpoint.
    g_set_current = 1000;
    break;
  case TR_Setpoint_Zero:
    // Manually change current setpoint.
    g_set_current = 0;
    break;
  case TR_Flash_Period:
    g_flash_period = 100;
    break;
  case TR_PID_FX_Gains:
    // Corrupt one of the compensator gains
    plantPID_FX.iGain = -1000;
    break;
  case TR_LCD_mutex:
    // Take LCD_mutex, don't return it
    osMutexAcquire(LCD_mutex, osWaitForever);
    break;
  case TR_Fill_Queue:
    // Fill ADC request queue with garbage
    Fault_Fill_Queue();
    break;
  case TR_Disable_PeriphClocks:
    SIM->SCGC6 = 0;
    break;
  case TR_Disable_All_IRQs:
    // Disable Interrupts
    __disable_irq();
    break;
  case TR_Disable_ADC_IRQ:
    // Disable ADC interrupt
    NVIC_DisableIRQ(ADC0_IRQn);
    break;
  case TR_osKernelLock:
    // Lock kernel - don't let other tasks run
    // See details at https://www.keil.com/pac
    osKernelLock();
    break;
  case TR_Change_MCU_Clock:
    // Change MCU clock frequency
    MCG->C5 = 0x0018;
    break;
  case TR_Slow_TPM:
    TPM0->MOD = 23456;
    break;
  case TR_Stack_Overflow:
    n = Overflow_Stack();
    break;
  case TR_High_Priority_Thread:
    // Raise own priority very high, then go i
    osThreadSetPriority(osThreadGetId(), osPri
    while (1)
      DEBUG_TOGGLE(DBG_FAULT_POS);
    break;
  case TR_End:
    break;
}
```



```
00002974 - caseD_a
                  switchD_0000291a::...
...2974 bl        osThreadGetId
...2978 movs      r1,#0x30
...297a bl        osThreadSetPriority

0000297e - LAB_0000297e
                  LAB_0000297e
...297e str       r4,[r5,#0xc]=>DAT_f80ff04c
...2980 b         LAB_0000297e
```

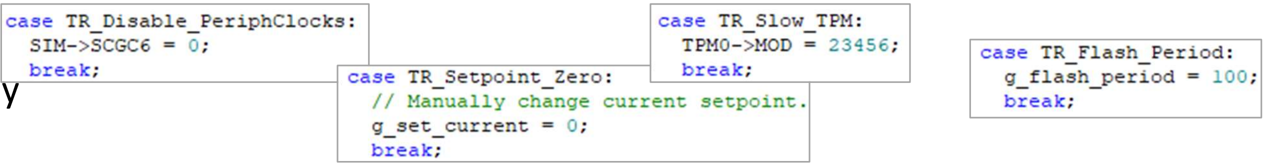
```
0000299c - caseD_d
                  switchD_0000291a::...
...299c add       t,sp,#0xc
...299e ldr       r1,[__arm_cp.2_2]

000029a0 - LAB_000029a0
                  LAB_000029a0
...29a0 str       r1,[t,#0x0]=>local_14
...29a2 subs      t,t,#0x4
...29a4 b         LAB_000029a0
```

```
000029a6 - caseD_e
                  switchD_0000291a::...
...29a6 movs      t,#0x0
...29a8 str       t,[sp,#local_18]
...29aa ldr       r5,[->ADC_RequestQueue]

000029ac - LAB_000029ac
                  LAB_000029ac
...29ac adds      t,t,#0x1
...29ae add       r4,sp,#0x4
...29b0 strb      t,[r4,#0x0]=>local_1c
...29b2 ldr       t,[r5,#0x0]=>ADC_RequestQu...
...29b4 movs      r2,#0x0
...29b6 mov       r1,r4
...29b8 mov       r3,r2
...29ba bl        osMessageQueuePut
...29be ldrb      t,[r4,#0x0]=>local_1c
...29c0 b         LAB_000029ac
```

Complex Control Flow #21 (1 of 2)

The control flow for four different cases merges into a single basic block before reaching the basic block with the call to osDelay and the pop instruction. Try to explain why the control flow merges for those cases.

- These four cases merge:



```
case TR_Disable_PeriphClocks:
  SIM->SCGC6 = 0;
  break;
```

```
case TR_Setpoint_Zero:
  // Manually change current setpoint.
  g_set_current = 0;
  break;
```

```
case TR_Slow_TPM:
  TPM0->MOD = 23456;
  break;
```

```
case TR_Flash_Period:
  g_flash_period = 100;
  break;
```

- All four cases are very similar: they just write a constant value to certain location in memory. To do this they need to do the following:
  - Put the destination address in a pointer register (e.g. r0)
  - Put the constant value in a register (e.g. r1)
  - Store the value to memory STR r0, [r1]

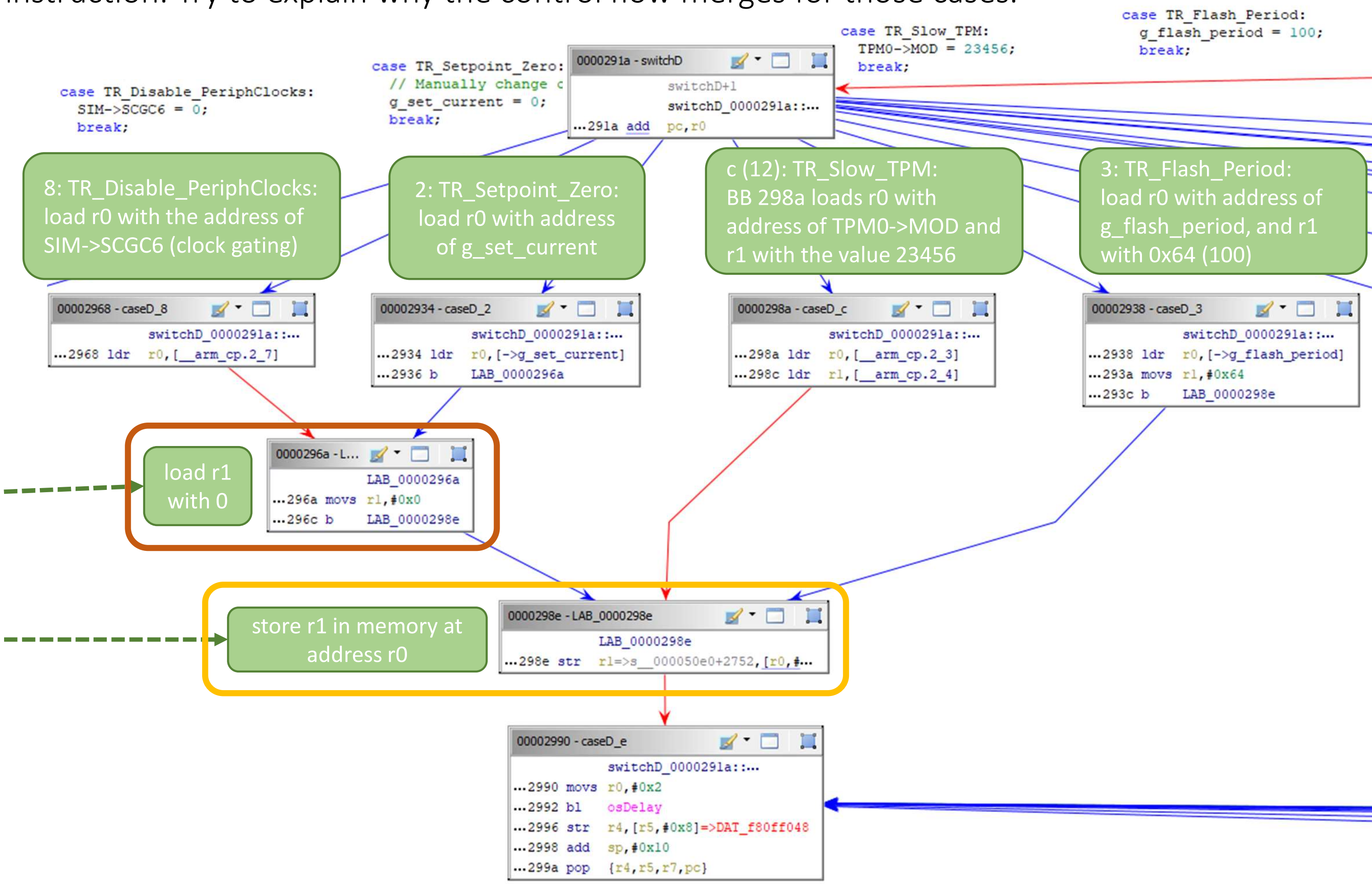
r0 ← &(SIM->SCGC6)	r0 ← &(g_set_current)	r0 ← &(TPM0->MOD)	r0 ← &(g_flash_period)
r1 ← 0	r1 ← 0	r1 ← 23456	r1 ← 100
mem[r1] ← r0	mem[r1] ← r0	mem[r1] ← r0	mem[r1] ← r0

- All four cases can use the same store instruction: STR r0, [r1]
- For two cases, the value written to memory will be 0, so the same instruction can move 0 into r0.
- The remaining operations in each case different and get their own code.

Complex Control Flow #21 (2 of 2)

The control flow for four different cases merges into a single basic block before reaching the basic block with the call to osDelay and the pop instruction. Try to explain why the control flow merges for those cases.

- Each of the four cases gets its own code to load non-common values
- For cases 8 and 2, the value written to memory will be 0. The common code to load r0 with 0 is placed in BB 296a
- An instruction to store r1 to memory at [r0] is placed in BB 298e for use by all four cases.



```
case TR_Disable_PeriphClocks:
  SIM->SCGC6 = 0;
  break;
```

```
case TR_Setpoint_Zero:
  // Manually change c
  g_set_current = 0;
  break;
```

```
case TR_Slow_TPM:
  TPM0->MOD = 23456;
  break;
```

```
case TR_Flash_Period:
  g_flash_period = 100;
  break;
```

```
0000291a - switchD
                  switchD+1
                  switchD_0000291a::...
...291a add       pc,r0
```

8: TR_Disable_PeriphClocks: load r0 with the address of SIM->SCGC6 (clock gating)

2: TR_Setpoint_Zero: load r0 with address of g_set_current

c (12): TR_Slow_TPM: BB 298a loads r0 with address of TPM0->MOD and r1 with the value 23456

3: TR_Flash_Period: load r0 with address of g_flash_period, and r1 with 0x64 (100)

```
00002968 - caseD_8
                  switchD_0000291a::...
...2968 ldr       r0,[__arm_cp.2_7]
```

```
00002934 - caseD_2
                  switchD_0000291a::...
...2934 ldr       r0,[->g_set_current]
...2936 b         LAB_0000296a
```

```
0000298a - caseD_c
                  switchD_0000291a::...
...298a ldr       r0,[__arm_cp.2_3]
...298c ldr       r1,[__arm_cp.2_4]
```

```
00002938 - caseD_3
                  switchD_0000291a::...
...2938 ldr       r0,[->g_flash_period]
...293a movs      r1,#0x64
...293c b         LAB_0000298e
```

load r1 with 0

```
0000296a - L...
                  LAB_0000296a
...296a movs      r1,#0x0
...296c b         LAB_0000298e
```

store r1 in memory at address r0

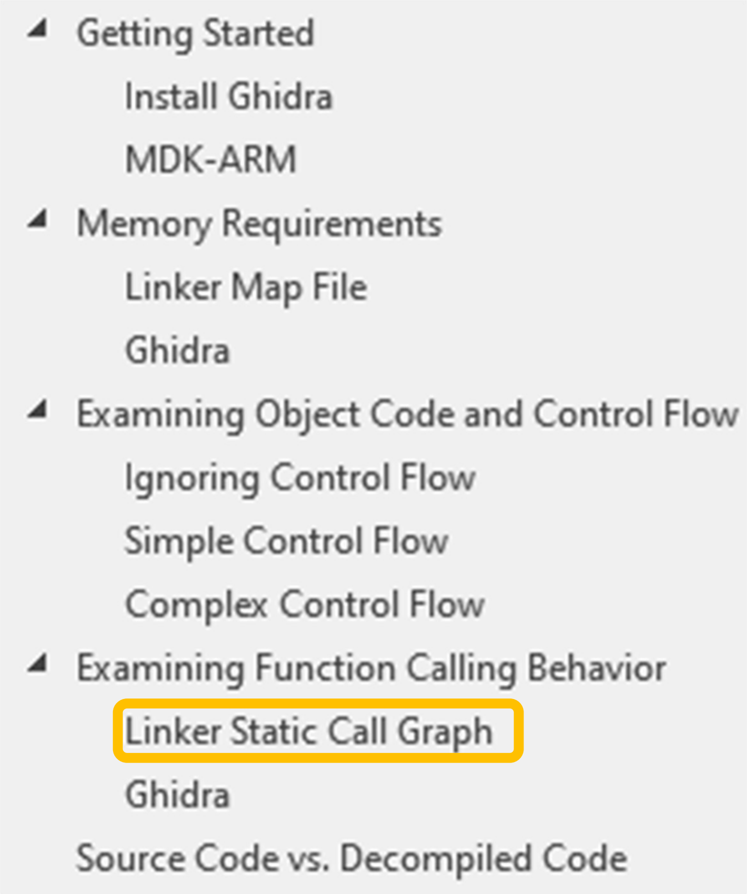
```
0000298e - LAB_0000298e
                  LAB_0000298e
...298e str       r1=>s__000050e0+2752,[r0,#...
```

```
00002990 - caseD_e
                  switchD_0000291a::...
...2990 movs      r0,#0x2
...2992 bl        osDelay
...2996 str       r4,[r5,#0x8]=>DAT_f80ff048
...2998 add       sp,#0x10
...299a pop       {r4,r5,r7,pc}
```

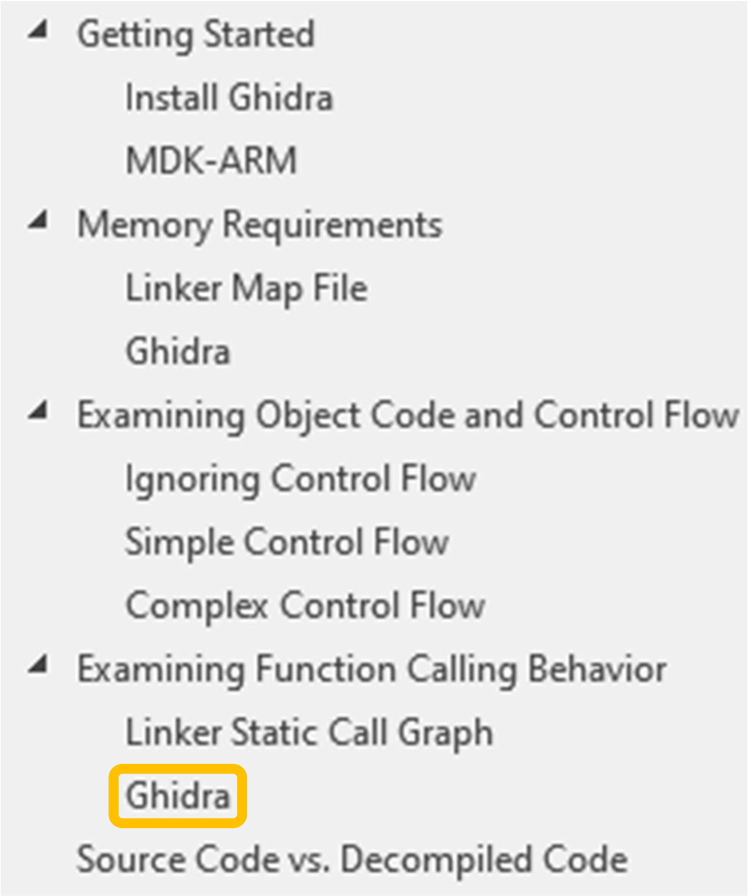
Complex Control Flow #22

Do all the cases end in an infinite loop or a control flow edge to another basic block? If not, which one doesn't, and why not?

- With my code, all cases end in an infinite loop or a control flow edge to another block.
- It's possible that if the compiler didn't inline a call to a function with an infinite loop (stack overflow (13/d), fill queue (14/e)), then the basic block would just end in a subroutine call (BL, BLX instruction) with no successor BB.



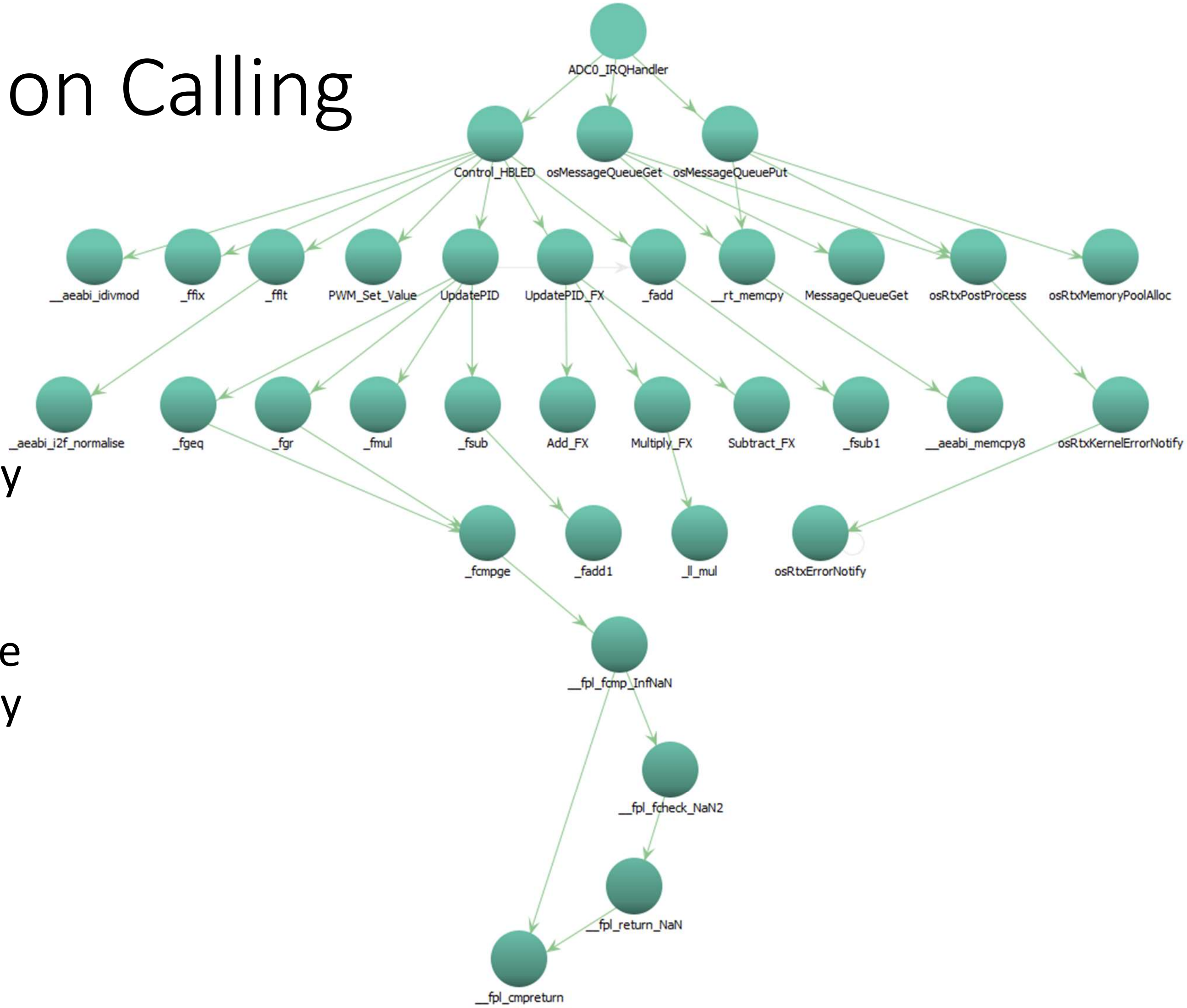
- Getting Started
  - Install Ghidra
  - MDK-ARM
- Memory Requirements
  - Linker Map File
  - Ghidra
- Examining Object Code and Control Flow
  - Ignoring Control Flow
  - Simple Control Flow
  - Complex Control Flow
- Examining Function Calling Behavior
  - Linker Static Call Graph
  - Ghidra
- Source Code vs. Decompiled Code



- Getting Started
  - Install Ghidra
  - MDK-ARM
- Memory Requirements
  - Linker Map File
  - Ghidra
- Examining Object Code and Control Flow
  - Ignoring Control Flow
  - Simple Control Flow
  - Complex Control Flow
- Examining Function Calling Behavior
  - Linker Static Call Graph
  - Ghidra
- Source Code vs. Decompiled Code

Examining Function Calling Behavior #10

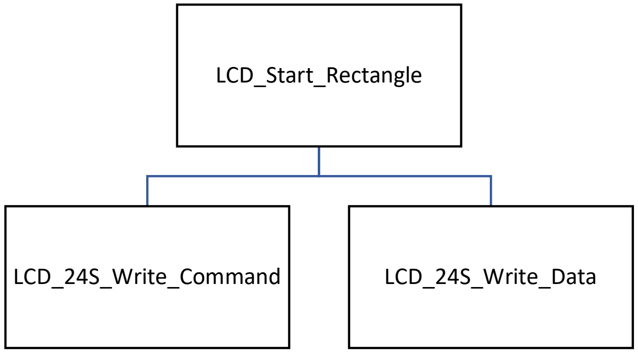
- 20%: Root node must be ADC0_IRQHandler
- 30%: All direct calls must be included (scale proportionally for missing or extra function nodes)
- 50%: All indirect calls must be included (scale proportionally for missing or extra function nodes)



Examining Function Calling Behavior #11

Use Ghidra to create a function call graph for LCD_Start_Rectangle. Compare it with what you would expect from the source code (located in ST7789.c). What are the differences? How do you explain them?

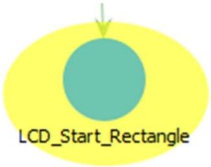
- Differences:
  - The source code has multiple subroutine calls to LCD_24S_Write_Command and LCD_24S_Write_Data


LCD_Start_Rectangle

LCD_24S_Write_Command

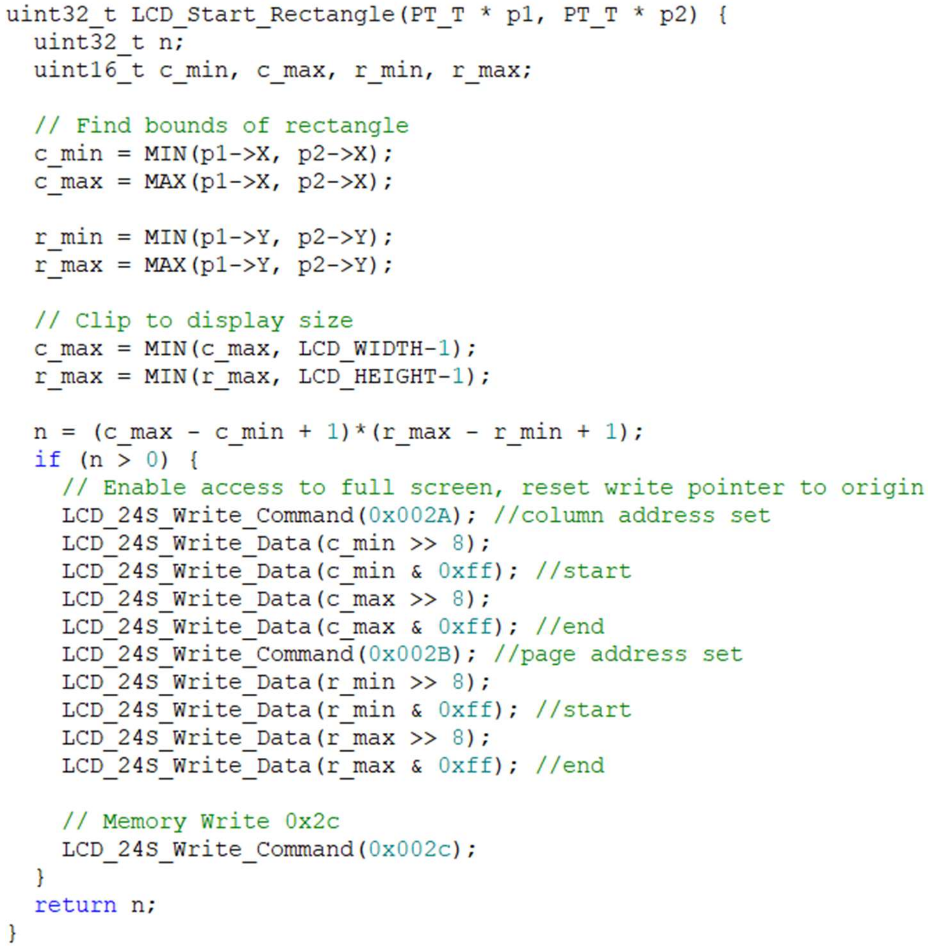
LCD_24S_Write_Data

  - The Ghidra function call graph has no subroutine calls.

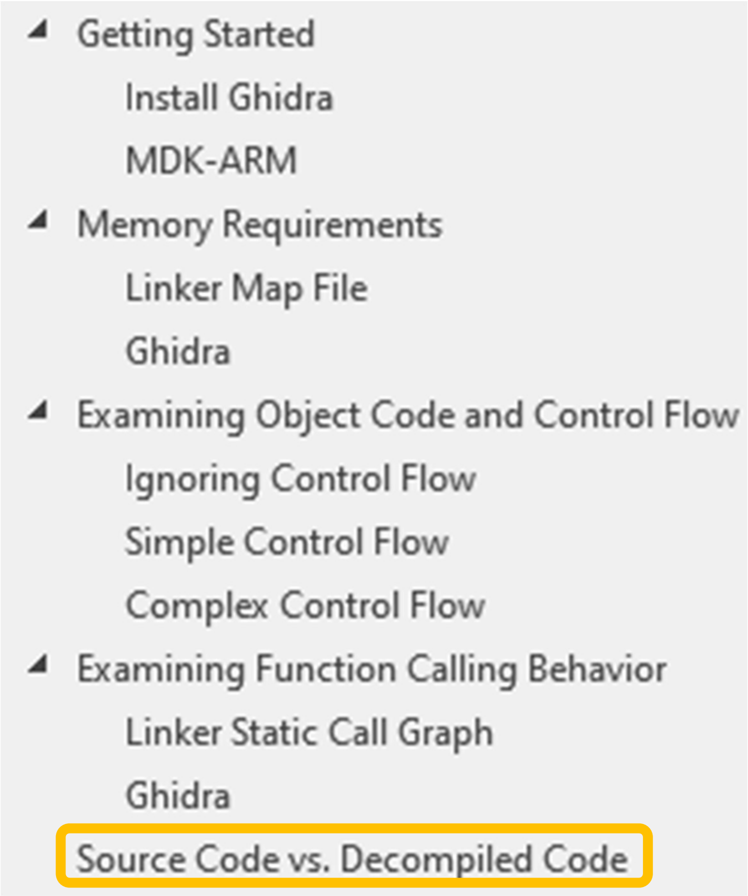


- Explanation: The compiler in-lined the function calls when optimizing the code (to speed it up).

Source Code vs. Decompiled Code #1



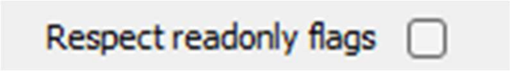
```c
uint32_t LCD_Start_Rectangle(PT_T * p1, PT_T * p2) {
  uint32_t n;
  uint16_t c_min, c_max, r_min, r_max;

  // Find bounds of rectangle
  c_min = MIN(p1->X, p2->X);
  c_max = MAX(p1->X, p2->X);

  r_min = MIN(p1->Y, p2->Y);
  r_max = MAX(p1->Y, p2->Y);

  // Clip to display size
  c_max = MIN(c_max, LCD_WIDTH-1);
  r_max = MIN(r_max, LCD_HEIGHT-1);

  n = (c_max - c_min + 1)*(r_max - r_min + 1);
  if (n > 0) {
    // Enable access to full screen, reset write pointer to origin
    LCD_24S_Write_Command(0x002A); //column address set
    LCD_24S_Write_Data(c_min >> 8);
    LCD_24S_Write_Data(c_min & 0xff); //start
    LCD_24S_Write_Data(c_max >> 8);
    LCD_24S_Write_Data(c_max & 0xff); //end
    LCD_24S_Write_Command(0x002B); //page address set
    LCD_24S_Write_Data(r_min >> 8);
    LCD_24S_Write_Data(r_min & 0xff); //start
    LCD_24S_Write_Data(r_max >> 8);
    LCD_24S_Write_Data(r_max & 0xff); //end

    // Memory Write 0x2c
    LCD_24S_Write_Command(0x002c);
  }
  return n;
}
```



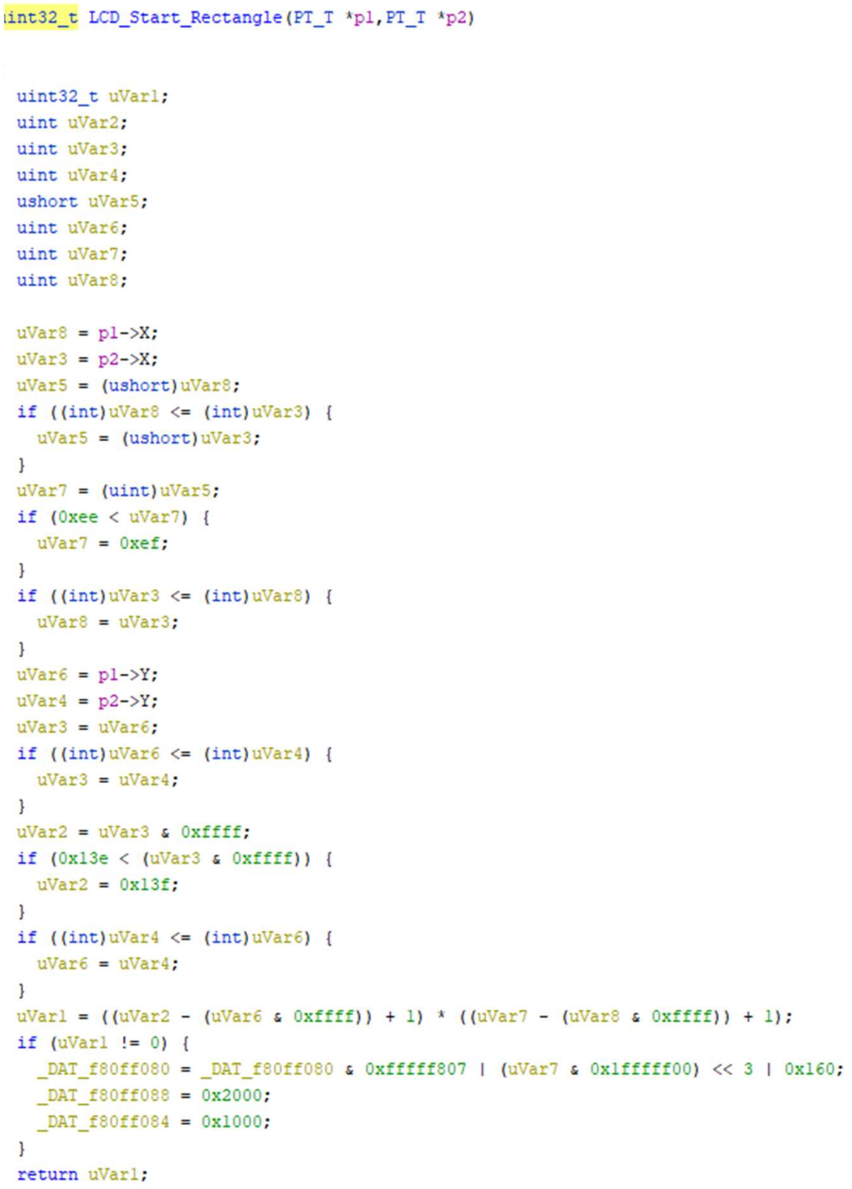
- ◢ Getting Started
  - Install Ghidra
  - MDK-ARM
- ◢ Memory Requirements
  - Linker Map File
  - Ghidra
- ◢ Examining Object Code and Control Flow
  - Ignoring Control Flow
  - Simple Control Flow
  - Complex Control Flow
- ◢ Examining Function Calling Behavior
  - Linker Static Call Graph
  - Ghidra
- Source Code vs. Decompiled Code

Source Code vs. Decompiled Code #2 (option 1)

- Either option 1 or 2 (next page) is fine
- In Code Browser, Edit -> Tool Options-> Decompiler-> Analysis



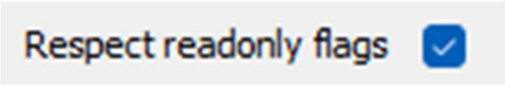
Respect readonly flags □



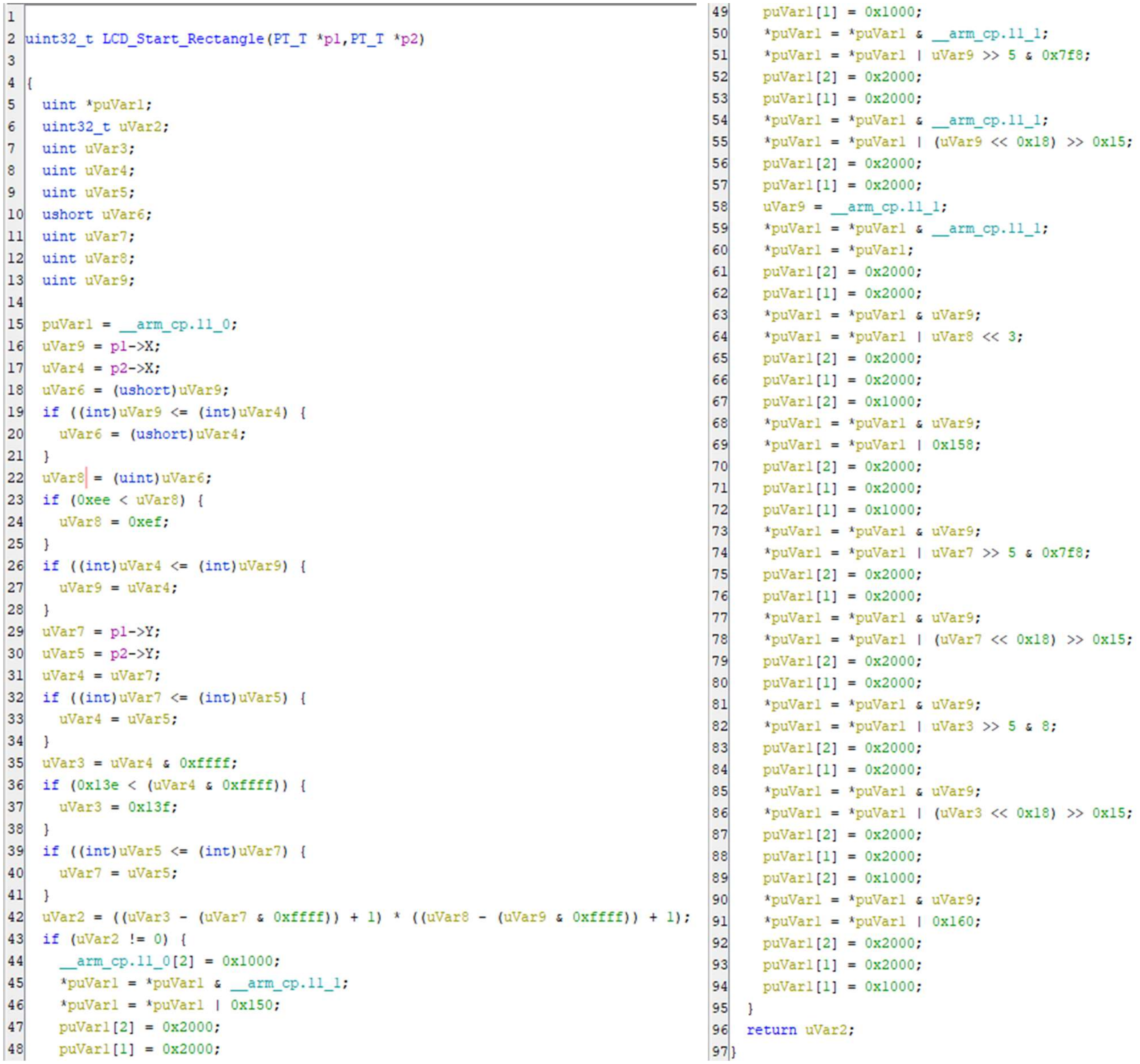
```
uint32_t LCD_Start_Rectangle(PT_T *p1,PT_T *p2)


  uint32_t uVar1;
  uint uVar2;
  uint uVar3;
  uint uVar4;
  ushort uVar5;
  uint uVar6;
  uint uVar7;
  uint uVar8;

  uVar8 = p1->X;
  uVar3 = p2->X;
  uVar5 = (ushort)uVar8;
  if ((int)uVar8 <= (int)uVar3) {
    uVar5 = (ushort)uVar3;
  }
  uVar7 = (uint)uVar5;
  if (0xee < uVar7) {
    uVar7 = 0xef;
  }
  if ((int)uVar3 <= (int)uVar8) {
    uVar8 = uVar3;
  }
  uVar6 = p1->Y;
  uVar4 = p2->Y;
  uVar3 = uVar6;
  if ((int)uVar6 <= (int)uVar4) {
    uVar3 = uVar4;
  }
  uVar2 = uVar3 & 0xffff;
  if (0x13e < (uVar3 & 0xffff)) {
    uVar2 = 0x13f;
  }
  if ((int)uVar4 <= (int)uVar6) {
    uVar6 = uVar4;
  }
  uVar1 = ((uVar2 - (uVar6 & 0xffff)) + 1) * ((uVar7 - (uVar8 & 0xffff)) + 1);
  if (uVar1 != 0) {
    _DAT_f80ff080 = _DAT_f80ff080 & 0xfffff807 | (uVar7 & 0x1fffff00) << 3 | 0x160;
    _DAT_f80ff088 = 0x2000;
    _DAT_f80ff084 = 0x1000;
  }
  return uVar1;
```

Source Code vs. Decompiled Code #2 (option 2)

- In Code Browser, Edit -> Tool Options-> Decompiler-> Analysis



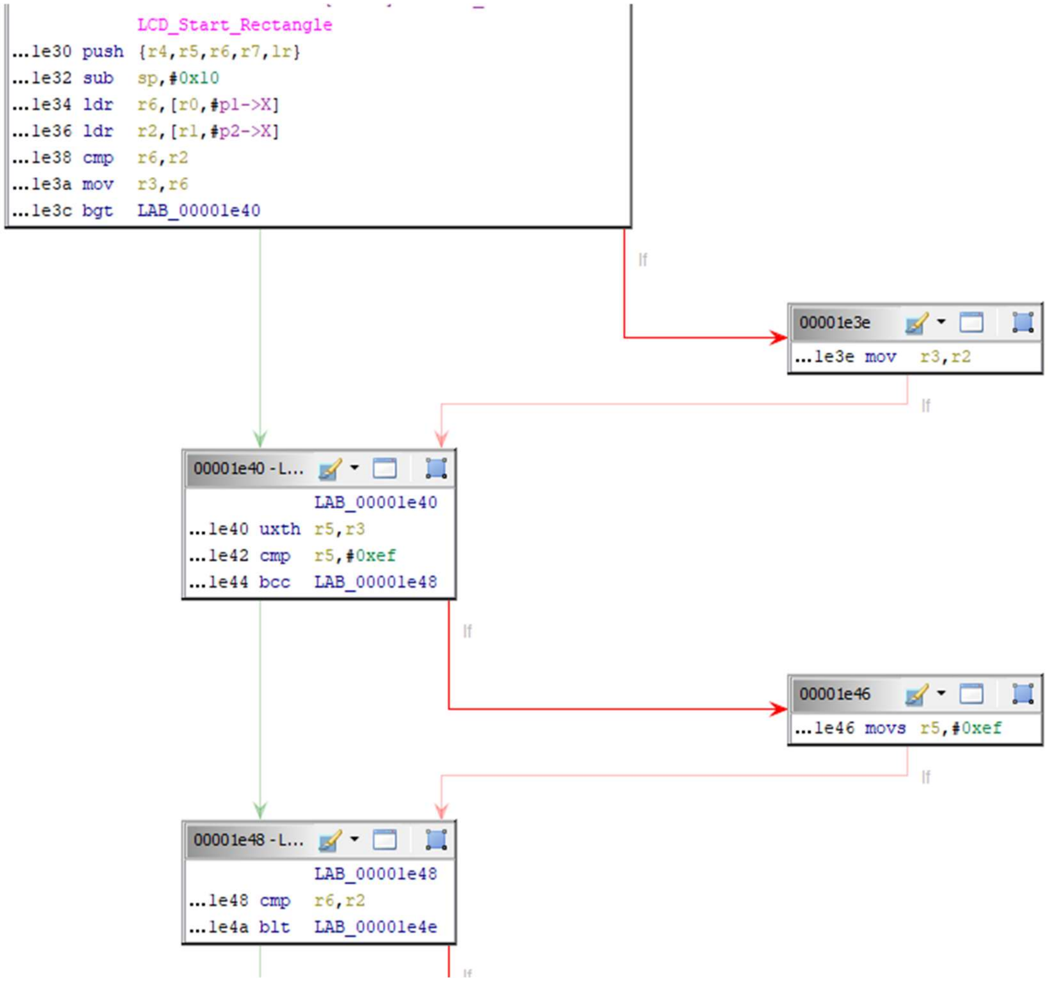
Respect readonly flags ☑



```

uint32_t LCD_Start_Rectangle(PT_T *p1,PT_T *p2)

{
  uint *puVar1;
  uint32_t uVar2;
  uint uVar3;
  uint uVar4;
  uint uVar5;
  ushort uVar6;
  uint uVar7;
  uint uVar8;
  uint uVar9;

  puVar1 = __arm_cp.11_0;
  uVar9 = p1->X;
  uVar4 = p2->X;
  uVar6 = (ushort)uVar9;
  if ((int)uVar9 <= (int)uVar4) {
    uVar6 = (ushort)uVar4;
  }
  uVar8 = (uint)uVar6;
  if (0xee < uVar8) {
    uVar8 = 0xef;
  }
  if ((int)uVar4 <= (int)uVar9) {
    uVar9 = uVar4;
  }
  uVar7 = p1->Y;
  uVar5 = p2->Y;
  uVar4 = uVar7;
  if ((int)uVar7 <= (int)uVar5) {
    uVar4 = uVar5;
  }
  uVar3 = uVar4 & 0xffff;
  if (0x13e < (uVar4 & 0xffff)) {
    uVar3 = 0x13f;
  }
  if ((int)uVar5 <= (int)uVar7) {
    uVar7 = uVar5;
  }
  uVar2 = ((uVar3 - (uVar7 & 0xffff)) + 1) * ((uVar8 - (uVar9 & 0xffff)) + 1);
  if (uVar2 != 0) {
    __arm_cp.11_0[2] = 0x1000;
    *puVar1 = *puVar1 & __arm_cp.11_1;
    *puVar1 = *puVar1 | 0x150;
    puVar1[2] = 0x2000;
    puVar1[1] = 0x2000;
    puVar1[1] = 0x1000;
    *puVar1 = *puVar1 & __arm_cp.11_1;
    *puVar1 = *puVar1 | uVar9 >> 5 & 0x7f8;
    puVar1[2] = 0x2000;
    puVar1[1] = 0x2000;
    *puVar1 = *puVar1 & __arm_cp.11_1;
    *puVar1 = *puVar1 | (uVar9 << 0x18) >> 0x15;
    puVar1[2] = 0x2000;
    puVar1[1] = 0x2000;
    uVar9 = __arm_cp.11_1;
    *puVar1 = *puVar1 & __arm_cp.11_1;
    *puVar1 = *puVar1;
    puVar1[2] = 0x2000;
    puVar1[1] = 0x2000;
    *puVar1 = *puVar1 & uVar9;
    *puVar1 = *puVar1 | uVar8 << 3;
    puVar1[2] = 0x2000;
    puVar1[1] = 0x2000;
    puVar1[2] = 0x1000;
    *puVar1 = *puVar1 & uVar9;
    *puVar1 = *puVar1 | 0x158;
    puVar1[2] = 0x2000;
    puVar1[1] = 0x2000;
    puVar1[1] = 0x1000;
    *puVar1 = *puVar1 & uVar9;
    *puVar1 = *puVar1 | uVar7 >> 5 & 0x7f8;
    puVar1[2] = 0x2000;
    puVar1[1] = 0x2000;
    *puVar1 = *puVar1 & uVar9;
    *puVar1 = *puVar1 | (uVar7 << 0x18) >> 0x15;
    puVar1[2] = 0x2000;
    puVar1[1] = 0x2000;
    *puVar1 = *puVar1 & uVar9;
    *puVar1 = *puVar1 | uVar3 >> 5 & 8;
    puVar1[2] = 0x2000;
    puVar1[1] = 0x2000;
    *puVar1 = *puVar1 & uVar9;
    *puVar1 = *puVar1 | (uVar3 << 0x18) >> 0x15;
    puVar1[2] = 0x2000;
    puVar1[1] = 0x2000;
    puVar1[2] = 0x1000;
    *puVar1 = *puVar1 & uVar9;
    *puVar1 = *puVar1 | 0x160;
    puVar1[2] = 0x2000;
    puVar1[1] = 0x2000;
    puVar1[1] = 0x1000;
  }
  return uVar2;
}
```

Source Code vs. Decompiled Code #3

- Each MIN or MAX macro has become a compare instruction, a conditional branch, and a move instruction.



```
                LCD_Start_Rectangle
...1e30 push  {r4,r5,r6,r7,lr}
...1e32 sub   sp,#0x10
...1e34 ldr   r6,[r0,#p1->X]
...1e36 ldr   r2,[r1,#p2->X]
...1e38 cmp   r6,r2
...1e3a mov   r3,r6
...1e3c bgt   LAB_00001e40
```

```
...1e3e mov   r3,r2
```

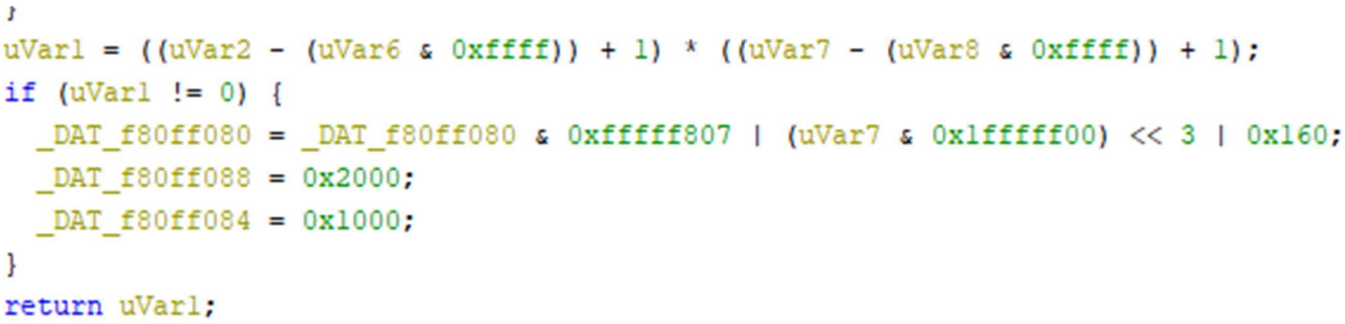
```
                LAB_00001e40
...1e40 uxth  r5,r3
...1e42 cmp   r5,#0xef
...1e44 bcc   LAB_00001e48
```

```
...1e46 movs  r5,#0xef
```

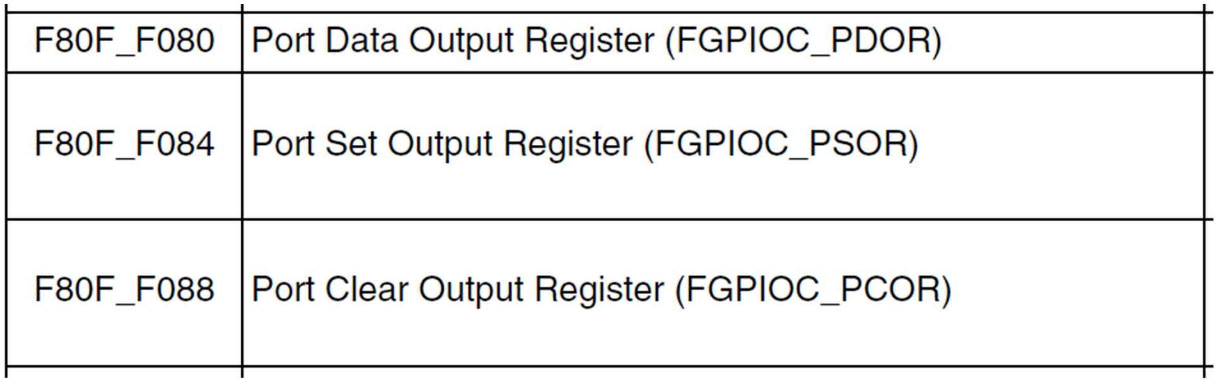
```
                LAB_00001e48
...1e48 cmp   r6,r2
...1e4a blt   LAB_00001e4e
```

Source Code vs. Decompiled Code #4 (option 1)

- These are addresses that correspond to the fast GPIO registers for FGPIOC (Chapter 41 of KL25 SF Reference Manual, Rev. 3)



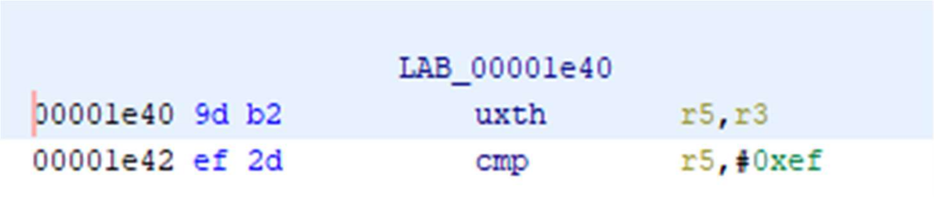
```
;
uVar1 = ((uVar2 - (uVar6 & 0xffff)) + 1) * ((uVar7 - (uVar8 & 0xffff)) + 1);
if (uVar1 != 0) {
  _DAT_f80ff080 = _DAT_f80ff080 & 0xfffff807 | (uVar7 & 0x1fffff00) << 3 | 0x160;
  _DAT_f80ff088 = 0x2000;
  _DAT_f80ff084 = 0x1000;
}
return uVar1;
```



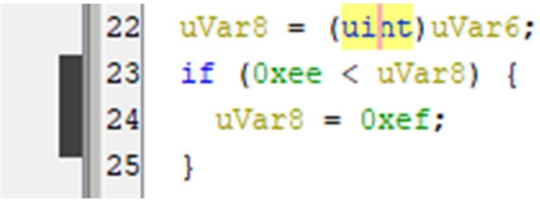
F80F_F080	Port Data Output Register (FGPIOC_PDOR)
F80F_F084	Port Set Output Register (FGPIOC_PSOR)
F80F_F088	Port Clear Output Register (FGPIOC_PCOR)

Source Code vs. Decompiled Code #5, #6

5. One cast does appear, performing an unsigned extension of a halfword from r3 into a full word in r5



```
                          LAB_00001e40
00001e40 9d b2               uxth        r5,r3
00001e42 ef 2d               cmp         r5,#0xef
```



```
uVar8 = (uint)uVar6;
if (0xee < uVar8) {
  uVar8 = 0xef;
}
```

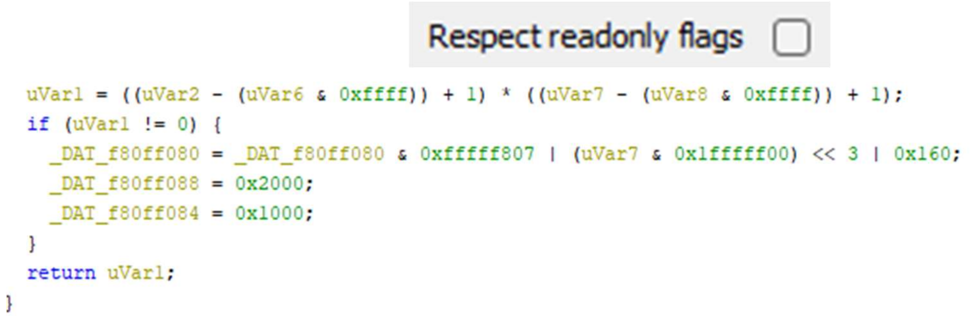
6. Most of the casts don’t appear in the object code – for example a conditional branch is highlighted as implementing the code. The casts are probably in the source code to clarify how the comparisons are performed.

Source Code vs. Decompiled Code #7, #8

7. The compiler inlined the function calls. This makes the program faster.

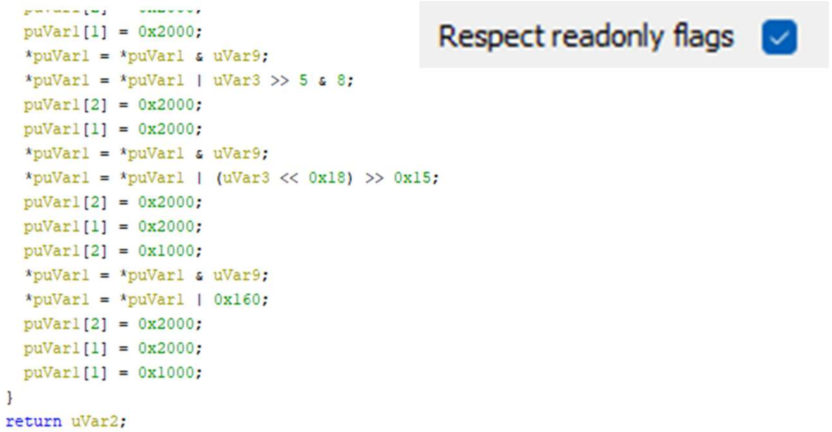
8a. Without **Respect readonly flags** checked, the decompiler omits all but the last writes to _DAT_f80ff080, _DAT_f80ff088, and _DAT_f80ff084 because it didn't know they were control registers (not regular memory), so each write matters.

8b. With **Respect readonly flags** checked, the decompiler includes all writes to those addresses.



Respect readonly flags ☐

```
  uVar1 = ((uVar2 - (uVar6 & 0xffff)) + 1) * ((uVar7 - (uVar8 & 0xffff)) + 1);
  if (uVar1 != 0) {
    _DAT_f80ff080 = _DAT_f80ff080 & 0xfffff807 | (uVar7 & 0x1fffff00) << 3 | 0x160;
    _DAT_f80ff088 = 0x2000;
    _DAT_f80ff084 = 0x1000;
  }
  return uVar1;
}
```



Respect readonly flags ☑

```
  puVar1[2] = 0x2000;
  puVar1[1] = 0x2000;
  *puVar1 = *puVar1 & uVar9;
  *puVar1 = *puVar1 | uVar3 >> 5 & 8;
  puVar1[2] = 0x2000;
  puVar1[1] = 0x2000;
  *puVar1 = *puVar1 & uVar9;
  *puVar1 = *puVar1 | (uVar3 << 0x18) >> 0x15;
  puVar1[2] = 0x2000;
  puVar1[1] = 0x2000;
  puVar1[2] = 0x1000;
  *puVar1 = *puVar1 & uVar9;
  *puVar1 = *puVar1 | 0x160;
  puVar1[2] = 0x2000;
  puVar1[1] = 0x2000;
  puVar1[1] = 0x1000;
}
return uVar2;
```