HOMEWORK 1: SOLUTIONS
EXAMINING OBJECT CODE

OVERVIEW

In this lab you will examine the object code created by the compiler for an RTOS-based program which uses many of the components of the KL25Z Freedom board and expansion shield**. You will not run the code, so you do not need an MCU board or expansion shield to complete this lab.**

Objectives:

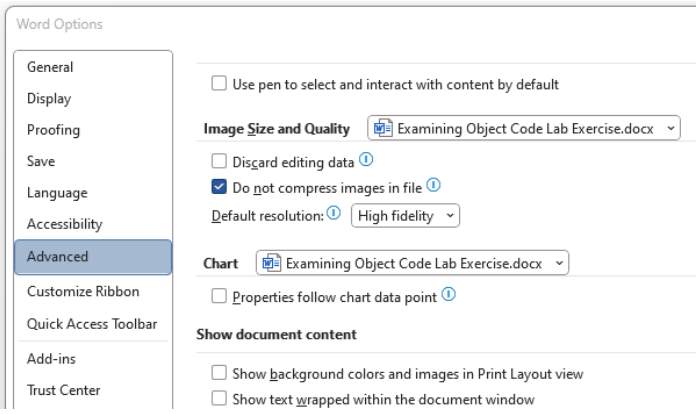
- Learn how to evaluate key program characteristics through analysis
  - Control flow within function
  - Function call graph
  - Decompiler
- Use build tool output and additional visualization tools to simplify analysis
  - IDE, build tools and their output
    - Linker call graph (hypertext)
    - Simulator in µVision with disassembly window
  - Software reverse engineering tool **Ghidra** for analysis and visualization
    - Disassembly, call behavior, control flow, decompilation, etc. from executable program file (AXF)

The revision history appears on the last page.

PROCEDURE

Obtain the project files from GitHub at **agdean/ESO-24/HW1/**. You will submit three items for this lab:

- Use the Google Form at https://forms.gle/t8WDscJu4xxQKfyu7 to submit answers for most of the numbered questions in this document.
- Use Moodle to submit a PDF single report document (PDF or MS Word) with images (scans, diagrams, screenshots). Please try to retain the quality of screenshot image quality in your reports so you don't lose points. If using MS Word, please select File->Options->Advanced and configure Image Size and Quality as shown: do not compress image info in file, and set Default Resolution to High Fidelity. Other word processing programs likely have similar settings available.

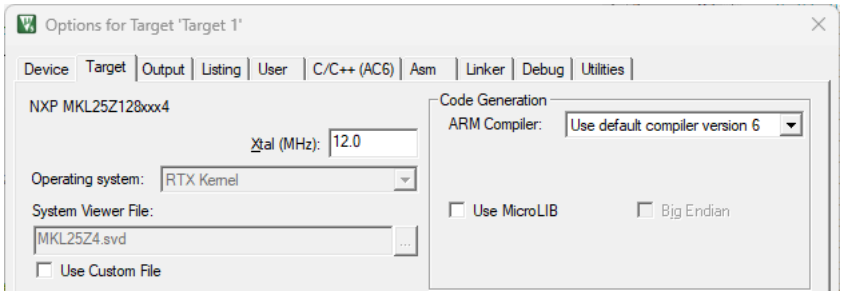


- Use Moodle to submit the Test_Basic.axf file which you built and analyzed.

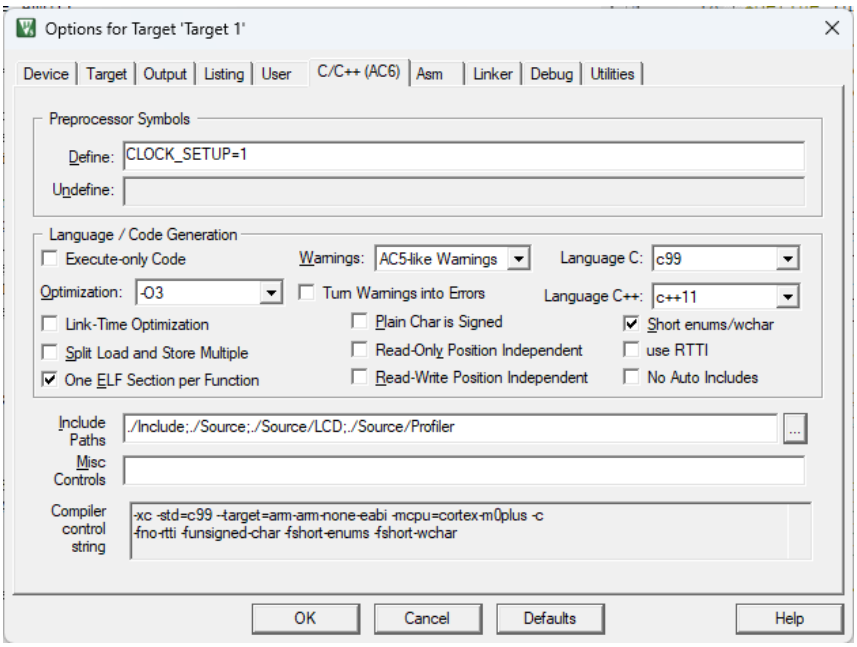
GETTING STARTED

MDK-ARM

- If you don't already have it installed, download MDK (Microvision Developers Kit) from https://www2.keil.com/mdk5 and install it, selecting the MDK-Community edition. Follow the instructions from Lab 0 to come up to speed.
- Open the project (HW1\Test_Basic.uvprojx).
- Select Project->Options for Target.
- Choose the **Target** tab. In the Code Generation section, verify the ARM Compiler selected is "Use default compiler version 6" and that "Use Microlib" is not checked as shown below.



- Choose the **Listing** tab to configure the linker to generate a map file. Verify the Linker Listing section has all boxes checked.
- On the **C/C++ (AC6)** tab, make sure the options are set as shown here.



- Click OK to apply all these changes.
- Build the program.

1. Find and copy the "*** Using Compiler …." line from the build output window. For example: *** Using Compiler 'V6.13.1', folder: 'C:\Keil_v5\ARM\ARMCLANG\Bin'

GHIDRA

Follow the Ghidra installation instructions on the course's References page under Software/Tools/Ghidra.

Start Ghidra and create a new project (as described on the course's References page under Software/Tools/Ghidra) to examine the axf file for this lab.

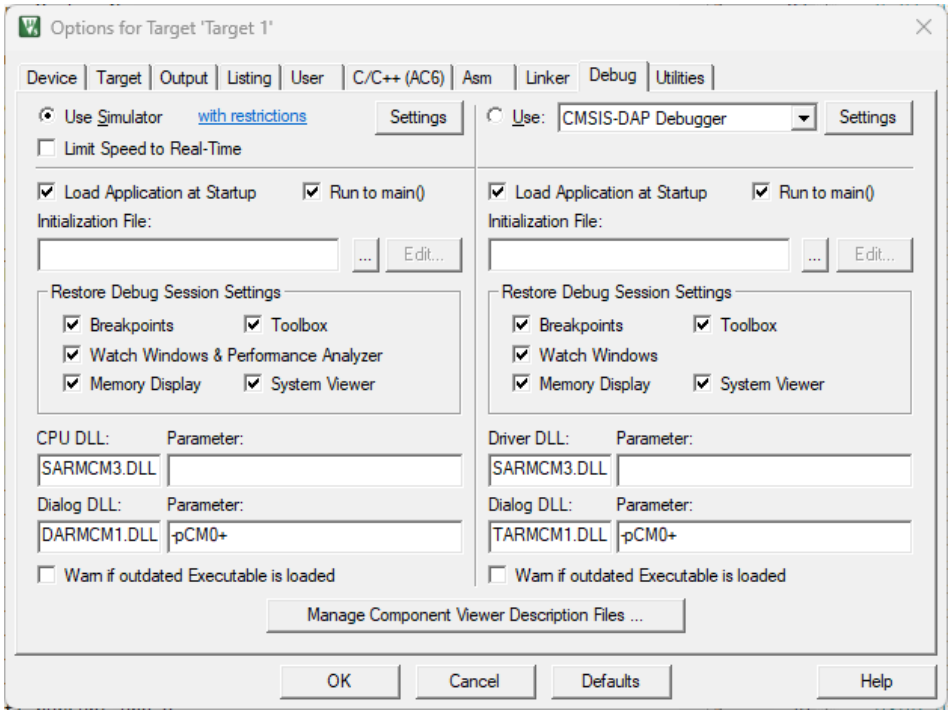
2. In Ghidra, open the Help menu, select "About Ghidra" and copy the version number, build information and Java version. For example, "Version 10.2.2 Build Public 2022-Nov-15 1249 EST Java Version 17.0.5"

EXAMINING OBJECT CODE AND CONTROL FLOW

Object code is confusing enough by itself. The branch instructions that change the control flow make it even worse. Here we'll start simple and work our way into more complex code.

IGNORING CONTROL FLOW

First we'll use the debugger's simulator to examine the mixed code listing (object code with source code interleaved). Configure MDK-ARM's target options to use the Debug Simulator rather the actual KL25Z MCU, as shown below.



Start a debug session (using Control-F5) to examine the disassembled code. **Do not start the program running**.

Open the file timers.c in the source code window. Place the cursor at the beginning of the **`Configure_TPM2_for_DMA`** function, which should be at line 3. This will bring up the corresponding object code in the disassembly window. If that window is not visible, open it with View->Disassembly Window.

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

```
   125: {
   126:         // disable TPM
0x00001818 4905      LDR      r1,[pc,#20]  ; @0x00001830
0x0000181A 2200      MOVS     r2,#0x00
   127:         TPM2->SC = 0;
   128:         //load the counter and mod
   129:         TPM2->MOD = TPM_MOD_MOD(period_us*48);
0x0000181C 600A      STR      r2,[r1,#0x00]
0x0000181E 2230      MOVS     r2,#0x30
   130:         //set TPM to count up and divide by 1 prescaler and clock mode
   131:         TPM2->SC = (TPM_SC_DMA_MASK | TPM_SC_PS(0));
   132:
0x00001820 4342      MULS     r2,r0,r2
0x00001822 B290      UXTH     r0,r2
0x00001824 6088      STR      r0,[r1,#0x08]
0x00001826 2001      MOVS     r0,#0x01
0x00001828 0200      LSLS     r0,r0,#8
   133: #if 0 // if using interrupt for debugging
   134:         // Enable TPM interrupts for debugging
   135:         TPM2->SC |= TPM_SC_TOIE_MASK;
   136:         // Configure NVIC
   137:         NVIC_SetPriority(TPM2_IRQn, 128); // 0, 64, 128 or 192
   138:         NVIC_ClearPendingIRQ(TPM2_IRQn);
   139:         NVIC_EnableIRQ(TPM2_IRQn);
   140: #endif
   141: }
   142:
   143: void TPM2_Start(void) {
   144:         // Enable counter
0x0000182A 6008      STR      r0,[r1,#0x00]
   145:         TPM2->SC |= TPM_SC_CMOD(1);
   146: }
   147:
   148:
   149:
   150: void TPM2_IRQHandler() {
   151:         //clear pending IRQ flag
   152:         TPM2->SC |= TPM_SC_TOF_MASK;
   153: }
0x0000182C 4770      BX       lr
0x0000182E 46C0      MOV      r8,r8
0x00001830 A000      DCW      0xA000
0x00001832 4003      DCW      0x4003
```

Ignore – disassembler is confused

Ignore – disassembler is confused

Ignore – disassembler is confused

1. How is the parameter period_us passed to this function? Through register r0
2. Which is the first instruction of the function **`Configure_TPM2_for_DMA`**? Include address, machine code, opcode and operand.

```
0x00001818 4905      LDR      r1,[pc,#20]
```

3. The register r1 is loaded with a value which is then used by several store register instructions (str). What is the hexadecimal value loaded into r1, and what does it represent? The value loaded is the word 0x4003a000 (4 bytes s stored starting at 0x18300). It is the address of the TPM2 peripheral's status and control (SC) register.

```
0x00001830 A000      DCW      0xA000
0x00001832 4003      DCW      0x4003
```

Chapter 31 Timer/PWM Module (TPM)

TPM memory map (continued)

Absolute address (hex)	Register name	Width (in bits)	Access	Reset value	Section/ page
4003_A000	Status and Control (TPM2_SC)	32	R/W	0000_0000h	31.3.1/552
4003_A004	Counter (TPM2_CNT)	32	R/W	0000_0000h	31.3.2/553
4003_A008	Modulo (TPM2_MOD)	32	R/W	0000_FFFFh	31.3.3/554
4003_A00C	Channel (n) Status and Control (TPM2_C0SC)	32	R/W	0000_0000h	31.3.4/555

4. What does #0x30 represent in the instruction **`movs r2, #0x30`**? 48, which is the conversion factor from clock cycles (at 48 MHz) to microseconds.

5. **561 Only:** What is the instruction **`uxth`** … doing, and why? Unsigned extension of lower halfword in r2 into word register r0. So r0 gets the lower halfword of r2, but zeroes in the upper halfword. r0 is then stored at [r1,#0x08] = 0x4003a008, which is the register TPM2_MOD. This code writes to TPM2->MOD:

```
//load the counter and mod
TPM2->MOD = TPM_MOD_MOD(period_us*48);
```

Right-clicking on TPM_MOD_MOD and selecting "Go to definition" brings up this code:

MKL25Z4.H

```
6146 /* MOD Bit Fields */
6147 #define TPM_MOD_MOD_MASK                0xFFFFu
6148 #define TPM_MOD_MOD_SHIFT               0
6149 #define TPM_MOD_MOD_WIDTH               16
6150 #define TPM_MOD_MOD(x)                  (((uint32_t)(((uint32_t)(x))<<TPM_MOD_MOD_SHIFT))&TPM_MOD_MOD_MASK)
```

The argument period_us*48 is ANDed with TPM_MOD_MOD_MASK, which is defined as 0xFFFF, which is 0x0000FFFF. That is why the assembly code zeroes out the upper halfword of the argument.

6. Which instruction performs the return-from-subroutine? Include address, machine code, opcode and operand.

```
0x0000182C 4770       BX        lr
```

SIMPLE CONTROL FLOW

In the MDK debugger source code window, open the file **OL_HBLED.c** and click on the function **`Set_OL_HBLED_Pulse_Width`**. This will bring up the object code in the disassembly window. If that window is not visible, open it with View->Disassembly Window.

```
    24: void Set_OL_HBLED_Pulse_Width(uint16_t pw) {
0x0000346C B580       PUSH      {r7,lr}
0x0000346E 4602       MOV       r2,r0
    25:           if (pw > LIM_DUTY_CYCLE)
    26:               pw = LIM_DUTY_CYCLE;
0x00003470 280F       CMP       r0,#0x0F
0x00003472 D300       BCC       0x00003476
0x00003474 220F       MOVS      r2,#0x0F
0x00003476 4802       LDR       r0,[pc,#8]  ; @0x00003480
0x00003478 2104       MOVS      r1,#0x04
    27:           PWM_Set_Value(TPM_HBLED, PWM_HBLED_CHANNEL, pw);
0x0000347A F7FFFF21   BL.W      0x000032C0 PWM_Set_Value
    28: }
    29:
0x0000347E BD80       POP       {r7,pc}
0x00003480 8000       DCW       0x8000
0x00003482 4003       DCW       0x4003
```

1. Identify each basic block in the function, listing them in order of increasing address. Assume that a subroutine call (bl, blx) ends a basic block. Use this format to describe each basic block:
   < *first-instr-start-adx last-instr-start-adx last-instr-changes-control-flow successor_adx* >
   - Possible values of *last-instr-changes-control-flow* are no, yes-branch, yes-call and yes-return
   - There may be from zero to multiple *successor_adx* entries for a basic block.
   - Separate each basic block description with a space or new-line

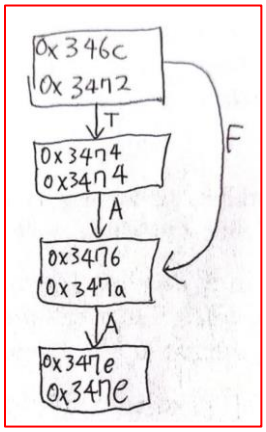
   Example:
   <0x1234 0x1240 yes-branch 0x1242 0x128a> <0x1242 0x1248 no 0x124a> <0x124a 0x124c yes-return>

   <0x1234 0x1240 yes-branch 0x1242 0x128a> describes a basic block starting at 0x1234 and ending with the instruction starting at 0x1240. The basic block ends in a branch which leads to the basic block starting at 0x128a if taken. If the branch is not taken, the next basic block to execute starts at 0x1242.

<0x1242 0x1248 no 0x124a> describes a basic block starting at 0x1242 and ending with the instruction starting at 0x1248. The basic block ends without a branch. The next basic block to execute starts at 0x124a.
The regular expression used for validation is:
(\s*<0x[0-9a-fA-F]+ 0x[0-9a-fA-F]+ (no|yes-branch|yes-call|yes-return)(0x[0-9a-fA-F]+)*>\s*)+

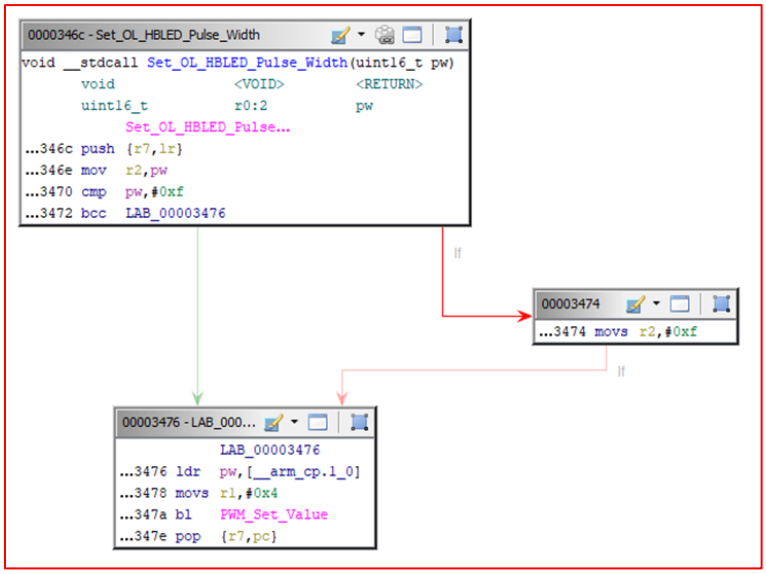
<0x346c 0x3472 yes-branch 0x3474 0x3476>
<0x3474 0x3474 no 0x3476>
<0x3476 0x347a yes-call>
<0x347e 0x347e yes-return>

2. Draw the control flow graph, labeling each basic block with its number and starting address and marking the control-flow edges as T, F, or A (for true, false and always). You can draw by hand or use a program (e.g. gvedit in the graphviz package (https://graphviz.gitlab.io/download/)). Submit this diagram in your report.



In Ghidra's Function window pane, select the function **`Set_OL_HBLED_Pulse_Width`**. This will bring up the disassembly listing in the Listing pane. Select the "Display Function Graph" icon  in the toolbar near the top of the CodeBrowser window. This will open a window with the control flow graph (CFG) of that function.

3. Capture a screenshot of the function's CFG and include it in your report.



4. **561 Only:** How are the two CFGs different? Why do you think Ghidra generates the CFG differently?

The Ghidra CFG has three basic blocks instead of four. It doesn't use a subroutine call as a reason to end a basic block.

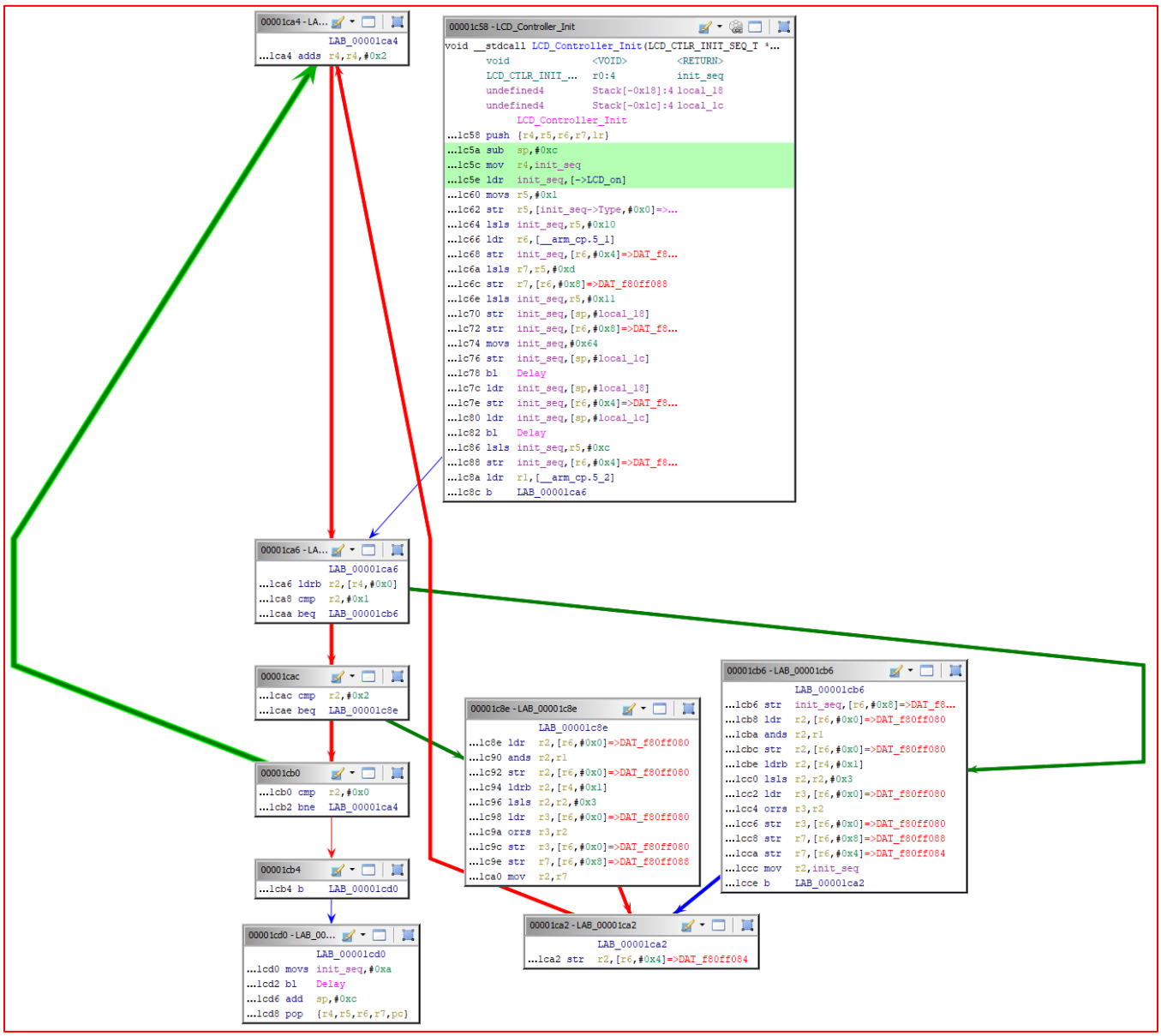
COMPLEX CONTROL FLOW

In Ghidra's Function window pane, select the function **`LCD_Controller_Init`**. This will bring up the disassembly listing in the Listing pane. Select the "Display Function Graph" icon  in the toolbar near the top of the CodeBrowser window. This will open a Function Graph window with the control flow graph (CFG) of that function. You can grab and move basic blocks to improve the layout manually. Ghidra offer many different methods to automatically lay out the control flow graph. The default is called "Nested Code Layout."

Examine each of the different layout methods available using the "Relayout Graph" button marked above in magenta. Initially the icon is  , but it may change (for example to ) for some layout methods. However, it should remain in the same position on the toolbar.

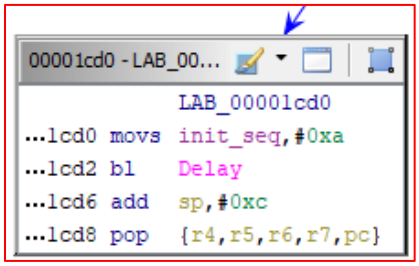
1. Which layout method produces the CFG (without manual modifications) which is easiest for you to understand?
   This is entirely up to you. (Example below is VHMCLP).

2. Capture screenshot of the function's CFG using that layout and include it in your report.



One of the basic blocks in the CFG returns from the subroutine by using a pop instruction which loads the PC.

3. Include a screenshot of that basic block in your report.

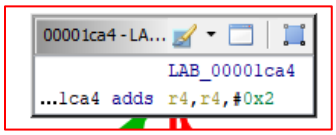


4. Which C source code statements does that basic block's instructions implement (including the subroutine return **}**)? Examine the C source code (in ST7789.c) using the MDK debugger's source code window. Enter the C source code statements.

```
235     Delay(10);
236 }
```

The function uses a loop to write a series of initialization command and data bytes to the LCD controller.

5. Which register is used to hold the variable **i**? r4
6. Find the basic block which increments variable **i** within the loop and take a screenshot and include it in your report



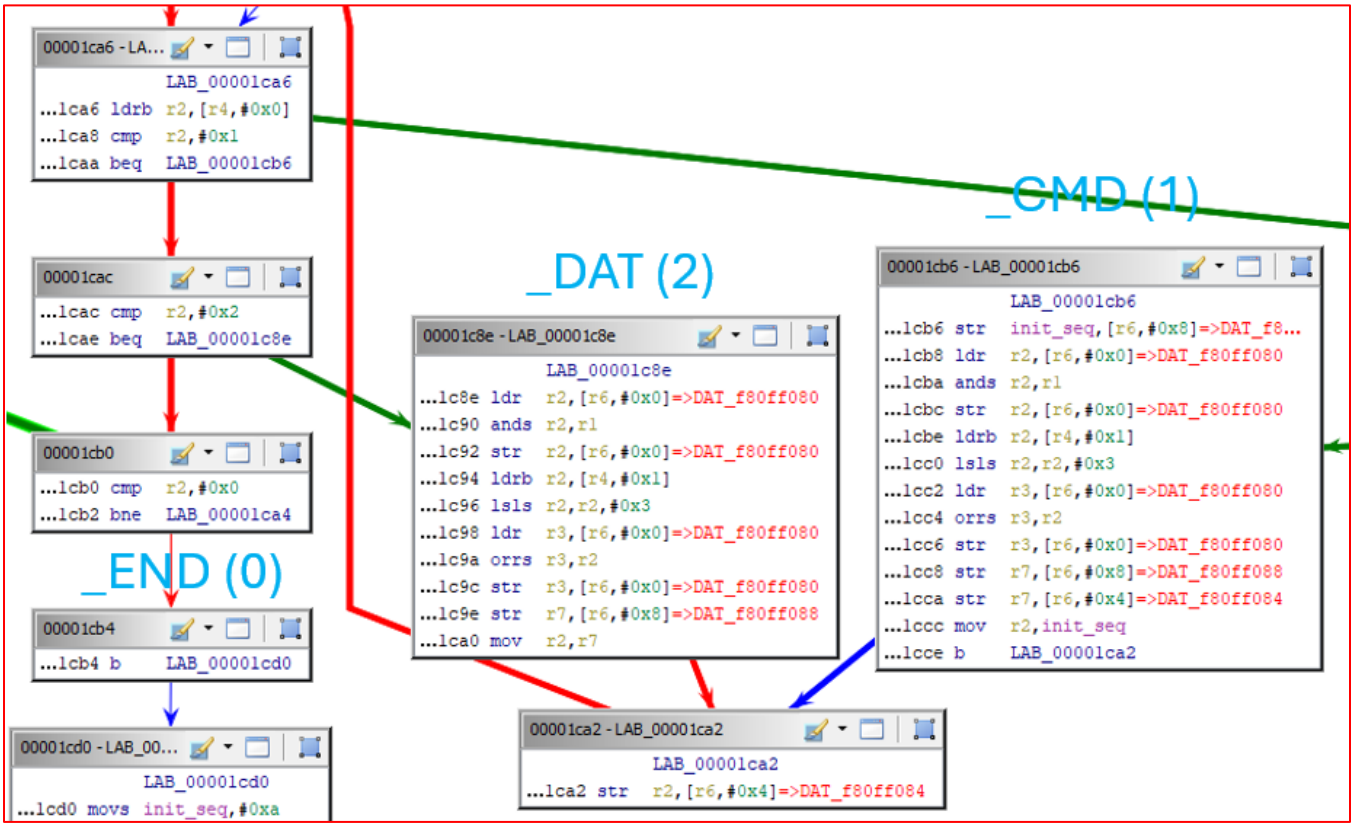
7. **561 Only:** Why is variable **i** incremented by 2 instead of 1? Hint: look at the definition of the **LCD_CTLR_INIT_SEQ_T** data type.

```
typedef struct {
  uint8_t Type;  // 0: end, 1: command, 2: data
  uint8_t Value;
} LCD_CTLR_INIT_SEQ_T; // sequence of commands and
```

```
while (!done) {
  switch (init_seq[i].Type) {
    case LCD_CTRL_INIT_SEQ_CMD:
      LCD_24S_Write_Command(init_seq[i].Value);
      break;
    case LCD_CTRL_INIT_SEQ_DAT:
      LCD_24S_Write_Data(init_seq[i].Value);
      break;
    case LCD_CTRL_INIT_SEQ_END:
      done = 1;
      break;
    default:
      break;
  }
  i++;
}
```

The variable **i** is only used to index the array **init_seq**. Each element in that array is of type LCD_CTLR_INIT_SEQ_T. That is a struct with two uint8_t fields (each one byte long), so the struct is two bytes long.
Before optimization, in each iteration the code would need increment i by one in and then multiply it by 2 to get the address of element[i]. Incrementing i by two simplifies the code by turning these two operations into one.

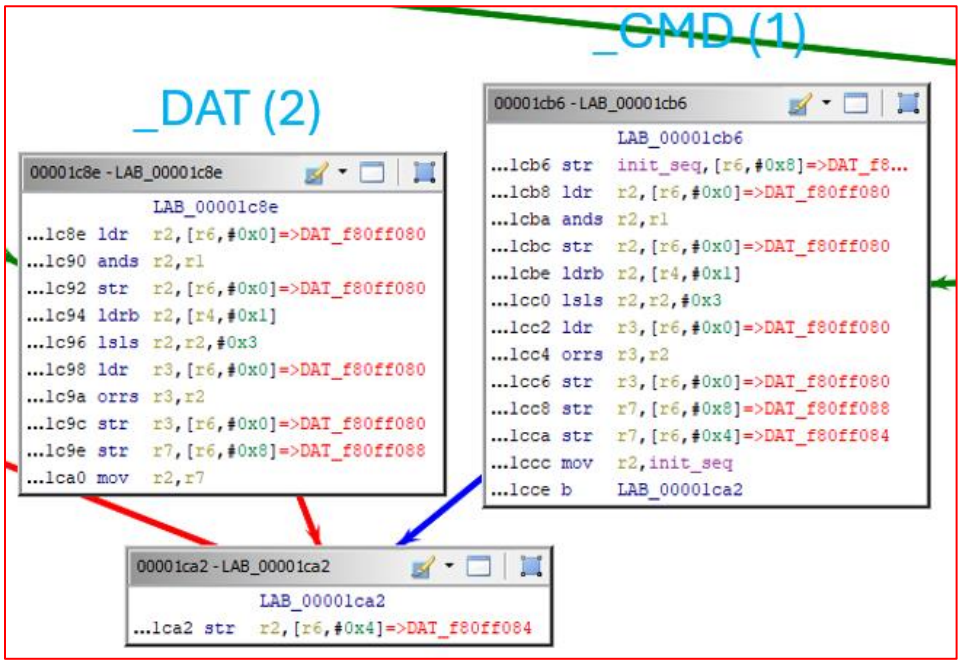
The function uses a **switch** statement to select one of multiple possible **case**s for execution.



8. Which register is used to select the case? r2
9. What is the starting address of the first basic block which implements the **LCD_CTRL_INIT_SEQ_CMD** case? Note that in MDK you can left-click in a symbol name and press F12 to bring up its definition (e.g. numerical value). 0x1cb6
10. What is the starting address of the first basic block which implements the **LCD_CTRL_INIT_SEQ_DAT** case? 0x1c8e
11. What is the starting address of the first basic block which implements the **LCD_CTRL_INIT_SEQ_END** case? 0x1cb4
12. Why is there no code or basic block implementing the **`default`** case? The default case is empty (only has a break statement), so there is no code needed for it.

The control flow from two of the cases (1 and 2) merges into a common basic block (with a store register instruction **`str`**) before repeating the loop.

13. Take a screenshot of the three basic blocks. Make sure the resolution is good enough to read the instructions. You may want to move the basic blocks for clarity. Include it in your report.



14. **561 Only:** What does that common basic block do, and what is at the address targeted by the store register instruction? The common basic block stores r2 to the memory location 0xf80ff084. This is the address of the Fast GPIO C port set output register PSOR.

F80F_F084	Port Set Output Register (FGPIOC_PSOR)	32	W (always reads 0)	0000_0000h	41.3.2/781

15. **561 Only:** Explain why you think the compiler generated the merge.
The compiler removed duplicated code. In the source code, the cases call LCD_24S_Write_Command or LCD_24S_Write_Command, but the compiler has inlined both of these calls. Both of these functions end in similar code (using the macro GPIO_SetBit, which writes to FGPIOC_PSOR) but with different data.

```
/* Write one byte as a command
void LCD_24S_Write_Command(uin
  GPIO_ResetBit(LCD_D_NC_POS);
  GPIO_Write(command);
  GPIO_ResetBit(LCD_NWR_POS);
  GPIO_SetBit(LCD_NWR_POS);
  GPIO_SetBit(LCD_D_NC_POS);
}
```

```
/* Write one byte as data to the
void LCD_24S_Write_Data(uint8_t
//  GPIO_SetBit(LCD_D_NC_POS); /
  GPIO_Write(data);
  GPIO_ResetBit(LCD_NWR_POS);
  GPIO_SetBit(LCD_NWR_POS);
}
```

The compiler sees there is common code in both cases, and so it has them merge just before the common code.

EXAMINING FUNCTION CALLING BEHAVIOR

LINKER STATIC CALL GRAPH

Programs typically contain multiple threads and exception/interrupt handlers. Each can call functions independently of other threads or handlers. Subroutine calls and high-level language features hide or encapsulate information to make the source code easier to understand. (Consider adding two integers vs. adding two floating-point values on a CPU without hardware support for floating-point math). A **static function call graph** helps understand part of a program's structure and possible behavior by showing all **possible** function-calling behavior of the program, even if it is hidden within other

subroutine calls. Which calls occur, how often, and in which order depends on the program's structure and probably input data; this information would be in a **dynamic call graph** for a specific run of the program with specific input data.

The linker can generate a static call graph for the program during each build. To do this, make sure the Callgraph box is checked in the **Options for Target -> Listing -> Linker Listing** section. The other boxes may be checked or unchecked, depending on your goals. Note that the call graph will not be created in the linker listing file (e.g. .\Listings\....map). Instead, it will be created in a separate file. Two output formats are available: hypertext (default) and plain-text. The hypertext file **<project_name>.htm** will be created in the **Objects** directory.

We will continue examining the function **`LCD_Controller_Init`**. Use a web browser to open the call graph **htm** file in the project's **Objects** directory.

LCD_Controller_Init (Thumb, 132 bytes, Stack size 32 bytes, st7789.o(.text.LCD_Controller_Init))

[Stack]

- Max Depth = 36
- Call Chain = LCD_Controller_Init ⇒ Delay

[Calls]

- >> Delay

[Called By]

- >> LCD_Init

1. Which function(s) can **`LCD_Controller_Init`** call directly (i.e. without intervening nested calls)? Delay
2. Which function(s) can call **`LCD_Controller_Init`** directly? LCD_Init

Examining the C source code for **`LCD_Controller_Init`** seems to show ten possible function calls. This does not match the linker's call graph information above. Use the MDK debugger's disassembly window to examine the object code for **`LCD_Controller_Init`**.

3. Which function calls actually appear in the object code? Three calls to Delay

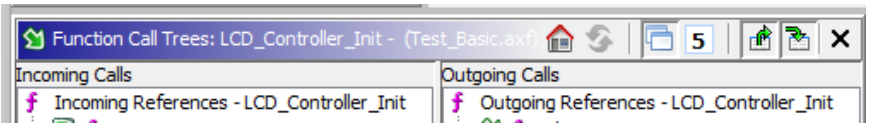
C-language macros are created by **`#define MACRO_NAME (MACRO BODY CODE)`** statements. An early stage of the compiler preprocesses the input files. In this stage it replaces all uses of macro names with the macro body code.

4. Which apparent function calls in **`LCD_Controller_Init`** are actually uses of macros? GPIO_SetBit, GPIO_ResetBit
5. Which function calls are missing because the compiler inlined the function body code? LCD_24S_Write_Data, LCD_24S_Write_Command

GHIDRA STATIC CALL GRAPHS AND TREES

Ghidra can create function call trees and call graphs. Ghidra uses hierarchical text listings to represent call trees. These show the functions which can be called by this function (Outgoing Calls) or which can call this function (Incoming Calls). A call graph is a diagram with nodes (vertices) representing functions, and connections (edges) representing calls.

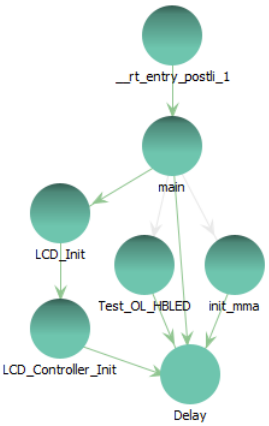
Use Ghidra's **Functions** window pane to select the function of interest. If the **Function Call Trees** window is not open, open it with the menu option Window -> Function Call Trees: …. The call trees will be automatically updated as you select different functions in the Functions window.



1. Which function(s) can **`LCD_Controller_Init`** call directly (i.e. without intervening nested calls)? Delay
2. Which function(s) can call **`LCD_Controller_Init`** directly? LCD_Init
3. Right-click on "Incoming References" and select "Expand Nodes to Depth Limit" to see all the functions which can indirectly call **`LCD_Controller_Init`**. List the function names. __rt_entry_postli_1, main

Our program needs to perform unsigned integer division but the CPU lacks that kind of instruction. Instead, the compiler links in a library function **`__aeabi_uidiv`** which performs an unsigned integer division. Use Ghidra's **Functions** window pane to select this function.

4. Which functions can call **`__aeabi_uidiv`** directly? LCD_Init, OS_Tick_Setup, Thread_Read_TS, LCD_TS_Read, Thread_Sound_Manager

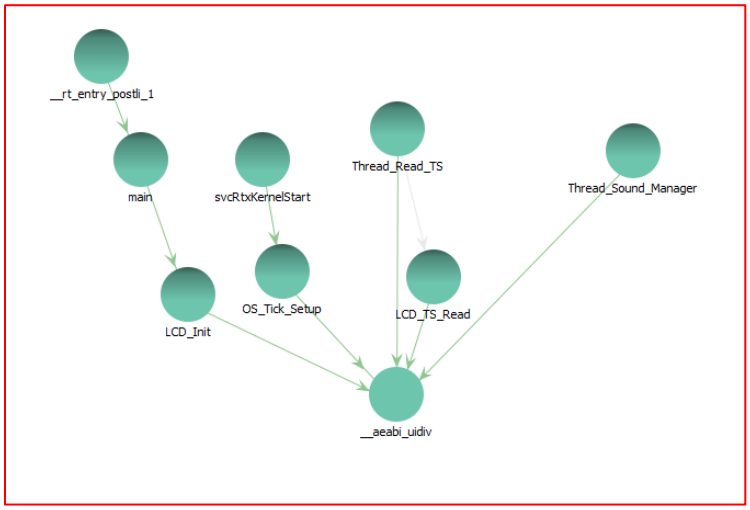


Ghidra can create a function call graph, as shown in the diagram to the right. If the **Function Call Graph** window is not open, open it with the menu option Window -> Function Call Graph: …. If it is open, then the call graph for the function will be automatically updated. By default, the call graph only shows the function and its direct callers and callees (incoming and outgoing calls).

- Left-click on a function node to select it: if the function has any outgoing calls a + symbol will appear.
- Right-click on the node and you will be able to show outgoing edges (callees) from that node or from all the nodes on that level. Or, click on the + symbol to show the outgoing edges (callees) for only that node.

Whether you want indirect callers/callees hidden or shown depends on your analysis goals and the program size.

5. Use Ghidra to create a function call graph for **`__aeabi_uidiv`**, showing functions which can call it directly or indirectly. Take a screenshot and include it in your report.



SOURCE CODE VS. DECOMPILED CODE

In MDK-ARM, view the source code for the function **`Get_DMA_Transfers_Completed`**, located in DMA.c.

1. Take a screenshot of the function's C source code from MDK-ARM and put it in your report.

```
int32_t Get_DMA_Transfers_Completed(uint32_t ch) {
  // Get progress from byte count register
  int32_t bytes_xferred = Reload_DMA_Byte_Count - (DMA0->DMA[ch].DSR_BCR & DMA_DSR_BCR_BCR_MASK);
  if (bytes_xferred < 0)
    return -1;
  switch ((DMA0->DMA[ch].DCR & DMA_DCR_SSIZE_MASK) >> DMA_DCR_SSIZE_SHIFT)  {
    case 0:
      return bytes_xferred/4;
      break;
    case 1:
      return bytes_xferred;
      break;
    case 2:
      return bytes_xferred/2;
      break;
    default:
      return -1;
      break;
  }

}
```

In Ghidra's Functions pane, click on **`Get_DMA_Transfers_Completed`** to select it. At the bottom of the Functions pane, select the Decompile: tab. This should bring up the decompiled view of function. Ghidra has been able to name the function and the arguments because the axf object file contains some debug symbol (name) information.

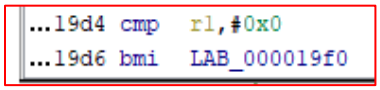
2. Take a screenshot of the decompiled **`Get_DMA_Transfers_Completed`** function code and put it in your report.

```
int32_t Get_DMA_Transfers_Completed(uint32_t ch)


{
  uint uVar1;
  uint uVar2;
  uint uVar3;

  uVar2 = Reload_DMA_Byte_Count - (*(uint *)(&DAT_40008108 + ch * 0x10) & 0xffffff);
  uVar1 = 0xffffffff;
  if (-1 < (int)uVar2) {
    uVar3 = (uint)(*(int *)(&DAT_4000810c + ch * 0x10) << 10) >> 0x1e;
    if (uVar3 == 2) {
      uVar1 = uVar2 >> 1;
    }
    else {
      if (uVar3 == 1) {
        return uVar2;
      }
      if (uVar3 == 0) {
        return uVar2 >> 2;
      }
    }
  }
  return uVar1;
}
```

3. What is uVar1 used for? Return value of -1, or bytes_xferred/2
4. What is uVar2 used for? bytes_xferred
5. What is uVar3 used for? switch statement argument (result of (DMA0->DMA[ch].DCR & DMA_DCR_SSIZE_MASK) >> DMA_DCR_SSIZE_SHIFT)
6. What is DAT_40008108 and how is it used here? Hint: examine the source and object code. This is the address of DMA0 Status Reg./Byte Count Reg. for Channel 0. It is used as base pointer to access channel ch (argument to function).
7. Are the switch cases in the same order in both the source and decompiled code? If not, what is the order for the decompiled code? No, the decompiled code cases are in order 2, 1, 0, but the source code is in order 0, 1, 2.

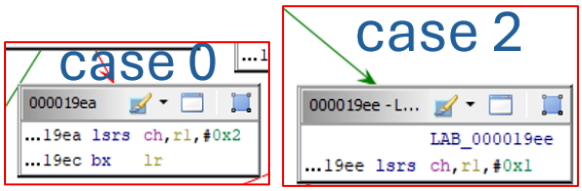
For the next three questions, consider the source code statement **`if (bytes_xferred < 0)`**.

8. How does the decompiled code represent this statement? if (-1 < (int)uVar2)
9. How is the decompiled code different? The terms are reversed, the comparison has changed (was var < 0?, became -1 < var?), type of bytes_xferred has changed from int32_t to uint.
10. **561 Only:** Examine the CFG. Which two assembly code instructions (comparison and conditional branch) implement that statement?



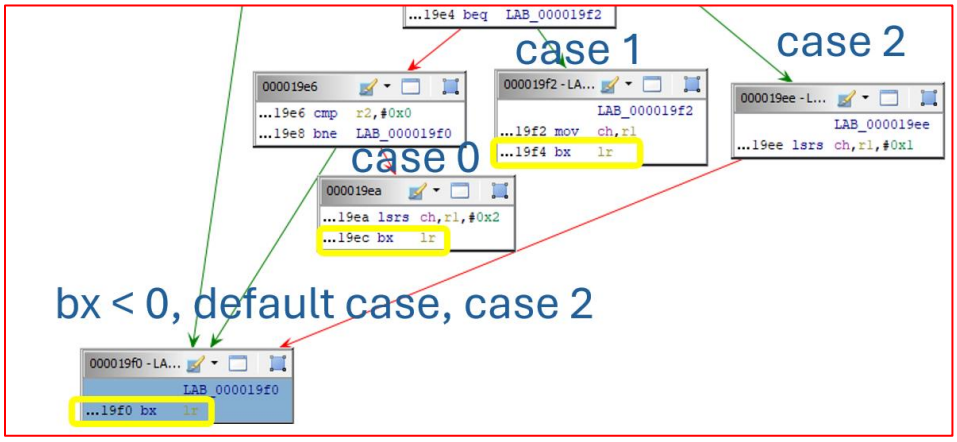
For the next two questions, consider the source code statements **`bytes_xferred/4`** and **`bytes_xferred/2`**.

11. How are the statements implemented in the decompiled code? right shift >> by 2 or 1
12. How are the statements implemented in the assembly code? logical shift right, set condition code flags: lsrs by #2 or #1



13. **561 Only:** Consider the number of return statements in the function. How many are in the source code, and how many are in the decompiled code? If they are not the same, explain what the compiler did. Hint: examine the CFG for subroutine returns.

The source code has five returns (bytes_xferred < 0, cases 0, 1, 2, default). The decompiled code has three.



The compiler generated code to merge three returns (for bx < 0, default case and case 2) into one.

REVISION HISTORY

V1.0

Base version.

V1.1

Changed control.c to OL_HBLED.c.