HOMEWORK 1:
EXAMINING OBJECT CODE

OVERVIEW

In this lab you will examine the object code created by the compiler for a program which uses some 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 and key data characteristics within a function
  - Function calling behavior
- 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)

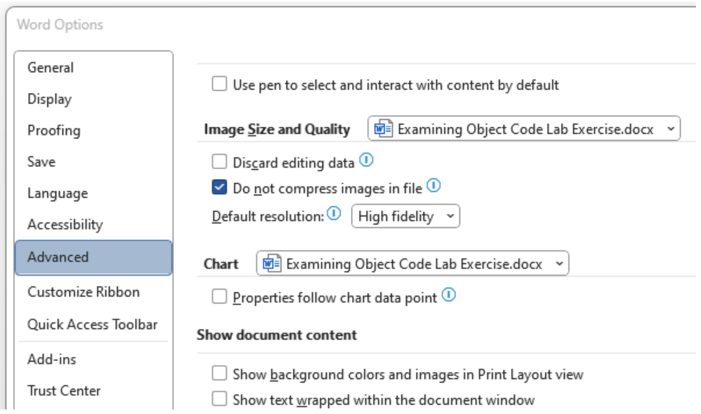
PROCEDURE

Get code and sheet for responses:

- Obtain the program files from GitHub at **agdean/ESO-25/HW/HW1/**.
- Copy **this Google Sheet** (https://docs.google.com/spreadsheets/d/1jyu1JrNzAnzVnaUsmd5sSGEx0Uqh-L97IAdoKR29xN0/edit?usp=sharing) and enter your answers in it. Note that some cells have response validation to prevent common errors.

You will submit **three items** through Moodle for this assignment:

- Save your completed Google Sheet as a tsv file file on your PC: File-> Download -> Tab-Separated Values (.tsv). Rename the tsv file to username-HW1.tsv (e.g. **jdoe-HW1.tsv**) and then submit it to Moodle.
- Submit a PDF single report document(e.g. **jdoe-HW1.pdf**) with images (scans, diagrams, screenshots) to Moodle. 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 pen to select and interact with content by default

Image Size and Quality Examining Object Code Lab Exercise.docx

☐ Discard editing data
☑ Do not compress images in file
Default resolution: High fidelity

Chart Examining Object Code Lab Exercise.docx

☐ Properties follow chart data point

Show document content

☐ Show background colors and images in Print Layout view
☐ Show text wrapped within the document window

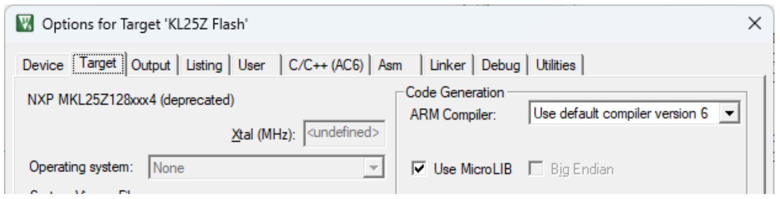
- Submit the **BasicUI.axf** file which you built and analyzed.

GETTING STARTED

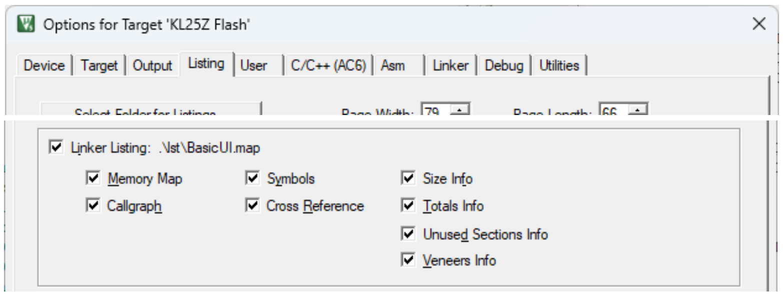
GS-1. Enter your NCSU email address in the spreadsheet.

MDK-ARM

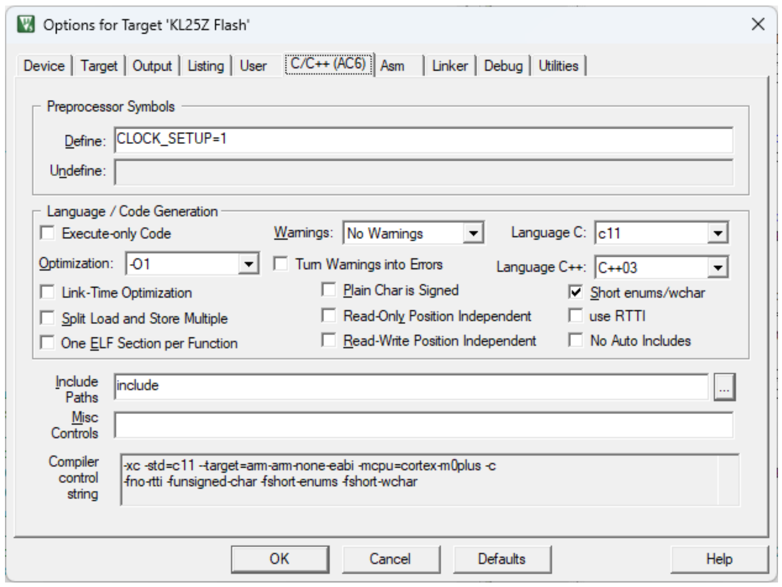
- If you don't already have it installed, download MDK (Microvision Developers Kit) from https://www2.keil.com/mdk5 and install it, selecting the MDK-Community edition. Follow the instructions from Lab 0 to come up to speed.
- Open the project (HW1\Basic_UI.uvprojx).
- Select Project->Options for Target.
- 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 checked as shown.


Options for Target 'KL25Z Flash'
NXP MKL25Z128xxx4 (deprecated)
Xtal (MHz): <undefined>
Operating system: None
Code Generation
ARM Compiler: Use default compiler version 6
☑ Use MicroLIB ☐ Big Endian

- Choose the **Listing** tab to configure the linker to generate a map file. Verify the Linker Listing section has all boxes checked.


Options for Target 'KL25Z Flash'
☑ Linker Listing: .\lst\BasicUI.map
☑ Memory Map ☑ Symbols ☑ Size Info
☑ Callgraph ☑ Cross Reference ☑ Totals Info
☑ Unused Sections Info
☑ Veneers Info

- 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.


Options for Target 'KL25Z Flash'
Preprocessor Symbols
Define: CLOCK_SETUP=1
Undefine:
Language / Code Generation
☐ Execute-only Code Warnings: No Warnings Language C: c11
Optimization: -O1 ☐ Turn Warnings into Errors Language C++: C++03
☐ Link-Time Optimization ☐ Plain Char is Signed ☑ Short enums/wchar
☐ Split Load and Store Multiple ☐ Read-Only Position Independent ☐ use RTTI
☐ One ELF Section per Function ☐ Read-Write Position Independent ☐ No Auto Includes
Include Paths: include
Misc Controls:
Compiler control string: -xc -std=c11 --target=arm-arm-none-eabi -mcpu=cortex-m0plus -c -fno-rtti -funsigned-char -fshort-enums -fshort-wchar

GS-2. Find and copy the "*** Using Compiler ...." line from the build output window. For example: *** Using Compiler 'V6.22', 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.

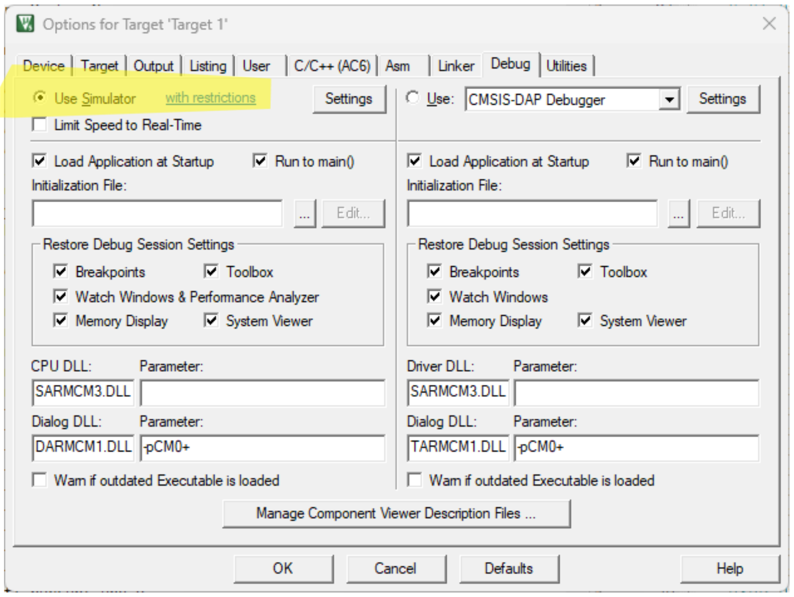
GS-3. In Ghidra, open the Help menu, select "About Ghidra" and copy the version number, build information and Java version. For example, "Version 10.2.2 Build Public 2022-Nov-15 1249 EST Java Version 17.0.5"

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 here:



Options for Target 'Target 1'

Device | Target | Output | Listing | User | C/C++ (AC6) | Asm | Linker | Debug | Utilities

Simulator side	Debugger side
(●) Use Simulator with restrictions	( ) Use: CMSIS-DAP Debugger
☐ Limit Speed to Real-Time	
☑ Load Application at Startup ☑ Run to main()	☑ Load Application at Startup ☑ Run to main()
Initialization File:	Initialization File:
Restore Debug Session Settings: ☑ Breakpoints ☑ Toolbox ☑ Watch Windows & Performance Analyzer ☑ Memory Display ☑ System Viewer	Restore Debug Session Settings: ☑ Breakpoints ☑ Toolbox ☑ Watch Windows ☑ Memory Display ☑ System Viewer
CPU DLL: SARMCM3.DLL Parameter:	Driver DLL: SARMCM3.DLL Parameter:
Dialog DLL: DARMCM1.DLL Parameter: -pCM0+	Dialog DLL: TARMCM1.DLL Parameter: -pCM0+
☐ Warn if outdated Executable is loaded	☐ Warn if outdated Executable is loaded

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

Open the file UART.c in the source code window. Place the cursor at the beginning of the `Init_UART2` function, which should be at line 29. 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.

XOC-1. Which register is used to pass the parameter baud_rate to this function? Hint: find the first C statement which references baud_rate, and then examine the object code (in the Disassembly window) to find the register.

XOC-2. Which is the first instruction of the function `Init_UART2`? Copy the line of text (not a screenshot) from the Disassembly window, including address, machine code, opcode and operand.

Examine the source and object code used to enable clock gating for UART2 and Port D. Look in the Disassembly window for the first instruction of the form "LDR rx, [pc, #offset] ; @...address...."

XOC-3. Copy the line of text (not a screenshot) from the Disassembly window, including address, machine code, opcode, operands and comment.

XOC-4. Into which register is the memory value loaded?

XOC-5. What is the address (@...address...) of the memory value?

XOC-6. What is the memory value located at that address? Scroll down in the Disassembly window to reach that address. Note that the LDR instruction loads a word (32 bit, four bytes) into a register. The disassembly listing does not represent the data in that size, so you will need to put together the halfwords in the listing to create a word, following Arm's endianness conventions.

XOC-7. That memory value should be the address of a register within the System Integration Module peripheral. Which register has that address (e.g. SIM_.....)? Refer to the KL25Z Subfamily Reference Manual's chapter on the System Integration Module.

XOC-8. Which instruction performs the return-from-subroutine? As before, copy the line of text, including address, machine code, opcode, operands and any comment.

SIMPLE CONTROL FLOW

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

XOC-9. Identify each basic block in the function, listing them in order of increasing address. There are **six basic blocks** in this function. 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 zero or more *successor_adx* entries for a basic block.
- Enter each basic block description on a separate row in the spreadsheet.

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*)+

XOC-10. 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. PlantUML activity diagrams through the PlantUML Server), or 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 **`Calibrate_ADC`**. 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.

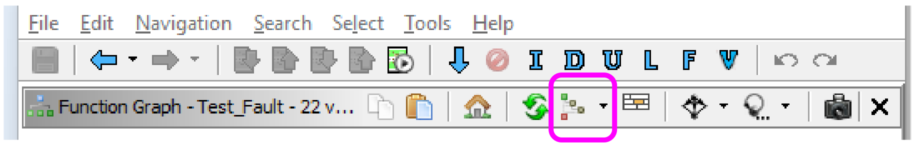
XOC-11. Capture a screenshot of the function's CFG and include it in your report.
XOC-12. 561 Only: If there are any significant differences (e.g. extra/missing/different basic blocks, extra/missing/different control flow edges) between Ghidra's CFG and the one you drew, please summarize them briefly.

COMPLEX CONTROL FLOW

In Ghidra's Function window pane, select the function **`main`**. 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."

XOC-13. Capture a screenshot of the function's CFG using the **Nested Code Layout** and include it in your report.


File Edit Navigation Search Select Tools Help
Function Graph - Test_Fault - 22 v...

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

XOC-14. Which layout method produces the CFG (without manual modifications) which is easiest for you to understand?
XOC-15. Capture a screenshot of the function's CFG using that layout and include it in your report.

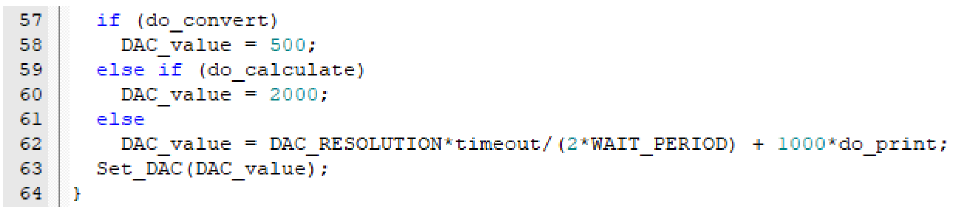
The object code in the CFG nodes provides some help for matching up object code with source code. For example, symbol names (e.g. function names used for subroutine calls) are colored in magenta. Use this information to match up basic blocks in the CFG with the corresponding C source code statements.

XOC-16. Think of the control flow defined by main's source code (initialization code, then loop with three conditionals in series). If the CFG does not match it well (consider XOC-12 for examples of significant differences), summarize the differences.

Use Ghidra to examine the object code for the function **`LPTMR0_IRQHandler`**.

XOC-17. Capture a screenshot of the function's CFG using that layout and include it in your report. You may need to move the basic blocks for clarity.
XOC-18. 561 Only: There are four basic blocks which will always execute once each time **`LPTMR0_IRQHandler`** is called. Identify these basic blocks by examining the CFG. Enter the starting address of the first instruction of each of these basic blocks. We will refer to these basic blocks as B1, B2, B3 and B4.

In the last basic block (B4), the function calls **`Set_DAC`** with an argument in r0 and before returning from the interrupt handler function. We will investigate the control and data flow in the function leading up to the **`Set_DAC`** function call. The source code shows three different statements assigning a value to DAC_value:



```
  if (do_convert)
    DAC_value = 500;
  else if (do_calculate)
    DAC_value = 2000;
  else
    DAC_value = DAC_RESOLUTION*timeout/(2*WAIT_PERIOD) + 1000*do_print;
  Set_DAC(DAC_value);
}
```

XOC-19. ECE 561 Only: Take a screenshot of CFG from B3 to B4, with all intervening basic blocks (there should be 6 basic blocks in total). Make sure the resolution is good enough to read the instructions. You may need to move the basic blocks for clarity. Include it in your report.
XOC-20. 561 Only: The disassembled code listing does not contain the values 500 and 2000 (or their hex equivalents, 0x1f4 and 0x7d0). Find the code which generates these values, and explain the instruction sequences the compiler generates to create them.

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. .\lst\....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 **obj** (or **Objects)** directory.

We will consider the function **main** starting with the source code.

XFC-1. Examine the source code for **main** in main.c. Which function(s) can **main** call directly (i.e. without intervening nested calls)? List each function once, omitting duplicate calls by **main**.

Use a web browser to open the linker's call graph file **BasicUI.htm** in the project's object file directory (**obj, Objects**, or something similar).

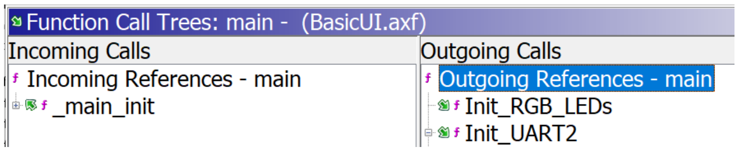
XFC-2. Which function(s) does the linker's call graph file indicate that **main** can call directly (i.e. without intervening nested calls)? These are listed in that function's **[Calls]** section.

XFC-3. Which function calls are "extra", appearing in XFC-2 but not in XFC-1?

XFC-4. Which function calls are "missing", appearing in XFC-1 but not in XFC-2?

GHIDRA STATIC CALL GRAPHS AND TREES

VISUALIZATION


Function Call Trees: main - (BasicUI.axf)

Incoming Calls	Outgoing Calls
Incoming References - main	Outgoing References - main
_main_init	Init_RGB_LEDs
	Init_UART2

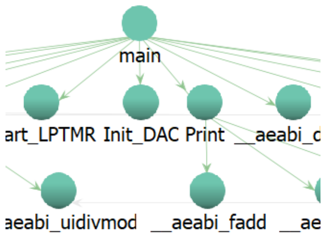
Ghidra can create function call trees and call graphs. A call tree is shown as a hierarchical text listing, showing the functions which can be called by this function (Outgoing Calls) or which can call this function (Incoming Calls). A call graph is a diagram with nodes (vertices) representing functions, 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.

XFC-5. Which function(s) can `main` *call* directly? That is, which functions might be called directly by `main`?

XFC-6. Which function(s) can *call* `main` directly or indirectly? Right-click on "Incoming References" and select "Expand Nodes to Depth Limit" to see all of the function names.

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).
- 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.
- Left-click on a function node to select it: if the function has any outgoing calls a **+** symbol will appear, letting you show the outgoing edges.
- Whether you want indirect callers/callees hidden or shown depends on your analysis goals and the program size.

XFC-7. Use Ghidra to create a function call graph for **`main`**, showing all functions which it can call directly or indirectly. Take a screenshot and include it in your report. You will need to show outgoing edges on multiple nodes to get the full call graph.

EXTRA FUNCTION CALLS

Our program may need to perform double-precision floating-point math division but the CPU lacks that kind of instruction. Instead, the compiler links in a library function **`__aeabi_ddiv`** to perform this operation. That function is part of the Run-Time ABI's standard compiler helper function library (see Chapter 5).

Use Ghidra's **Functions** window pane to select this function.

XFC-8. Which functions can call **`__aeabi_ddiv`** directly?

XFC-9. Which of the extra functions listed in **XFC-3** are NOT part of the standard compiler helper function library?

MISSING FUNCTION CALLS

Consider the function calls in **main** which you listed as missing in XFC-4 above. Use the control flow graph ("Function Graph") of **main** to find basic blocks where the missing function calls should be. Use the Nested Code Layout to help match up source code with the control flow graph.

XFC-10. What code is present at the points where you expect the missing function calls from question **XFC-4**?

SOURCE CODE VS. OBJECT CODE VS. DECOMPILED CODE

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

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

In Ghidra's Functions pane, click on **`Calibrate_ADC`** to select it.

SVD-2. Capture a screenshot of the function's CFG using that layout and include it in your report. You may need to move the basic blocks for clarity.

SVD-3. Does the control flow match between the source code and object code? (They should match, but your compiler may have acted differently from mine).

SVD-4. If they do not match, what is different or missing?

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. FYI, you can delete that information by clearing the Debug Information checkbox in MDK-ARM's Options for Target -> Output tab and rebuilding the project.

SVD-5. Take a screenshot of the decompiled function code and put it in your report.

SVD-6. What is _DAT_4003b000 and how is it used here? Hint: examine the source and object code.

SVD-7. Does the control flow match between the object code and the decompiled code?

SVD-8. If they do not match, what is different or missing?