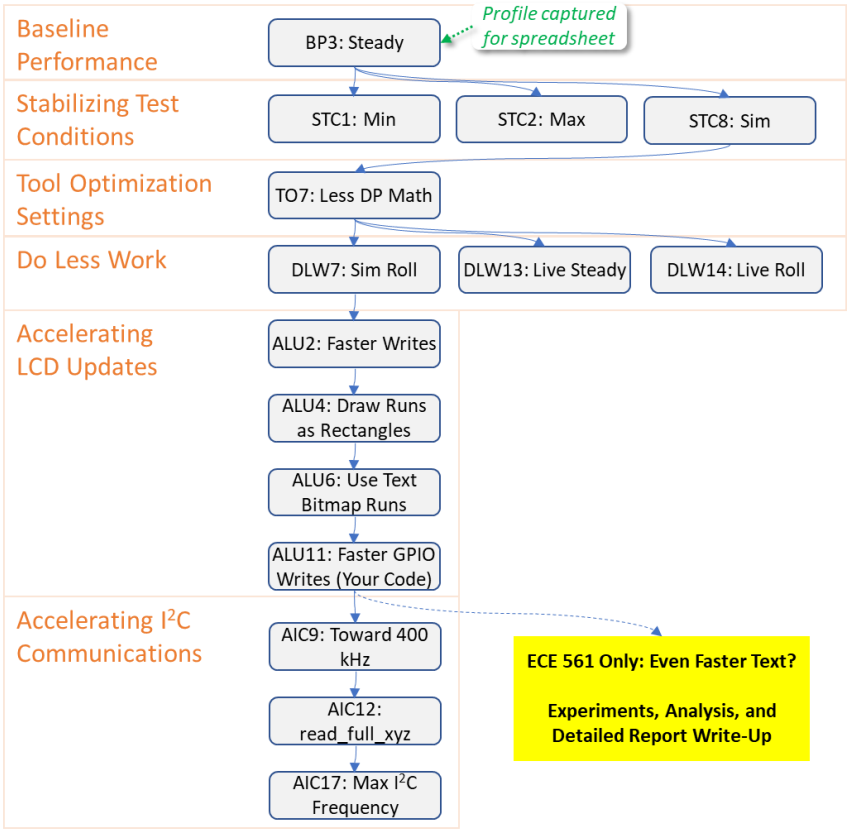
HOMEWORK 2: "OPTIMIZING" EXECUTION SPEED

CONTENTS

OVERVIEW

In this exercise you will evaluate and improve ("optimize") the code for a program which measures X, Y and Z accelerations using the FRDM-KL25Z's I²C-connected three-axis accelerometer (MMA8451Q) and uses them to calculate the board's orientation (roll and pitch) and then display them. You will need a FRDM-KL25Z board and an expansion shield with working LCD and touchscreen.



This diagram shows the general flow of the assignment, with gray nodes representing different code/tool configurations where you will gather profile data. The yellow node represents an in-depth investigation ECE 561 students will complete.

The program uses the LCD to show a pendulum and numerical roll and pitch values. For reference, the FRDM-KL25Z is pointing up when its USB connectors are pointing up. (Imagine hanging the board from its USB cable…) The program uses interrupts and a main loop but does not use an RTOS. The main() function does the following:

- Initialize hardware and software
- Repeat this loop forever
  - Initialize profiling data and screen contents
  - Enable profiling
  - Repeat this loop NUM_TESTS times
    - Read accelerometer x,y,z data
    - Convert to roll and pitch
    - Update text with roll value
    - Update text with pitch value
    - Update graphical arrow indicator to point up, based on roll value
  - Disable profiling
  - Process profiling data and display on LCD. When user presses LCD, advance through each screen of data and repeat test loop.

The program has built-in profiling support. The profiler samples the program counter periodically, as determined by PROFILER_SAMPLE_FREQ_HZ in profile.h.

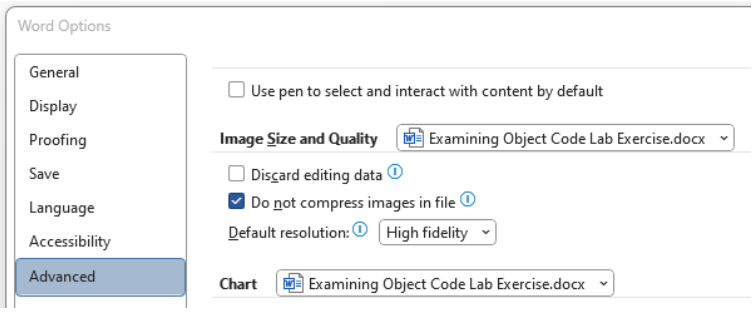
GENERAL PROCEDURE

- You are to work individually on his assignment.
- ECE 461 students may complete "ECE 561 Only" sections for extra credit.

Copy **this Google Sheet** (https://docs.google.com/spreadsheets/d/1yEL9CdVc3TUubRCNDoD77V1Wv6FxftZN63oKduXupYg/edit?usp=sharing agdean-HW2) and enter your spreadsheet answers in that copy.

You will submit three items for this assignment through Moodle:

- A tsv-format file based on the Google Sheet. 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>-HW2.tsv (e.g. jdoe-HW2.tsv) and then submit it to Moodle.
- Submit a PDF single report document with your text and 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.

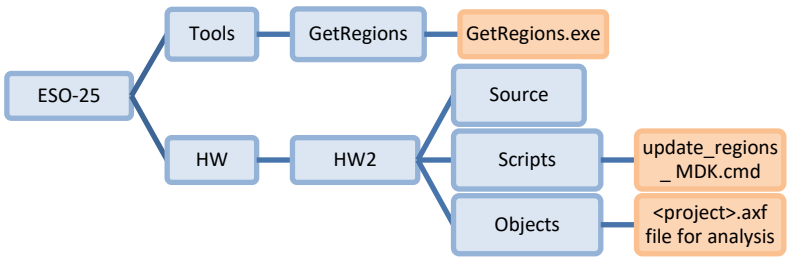


- Submit a zip archive of your HW2 code
  - If you **are not** completing the **Even Faster Text** section, submit a zip archive of your HW2 after completing the **Push Data Rate Further** section. This must include your improved version of the macro **GPIO_Write.**
  - If you **are** completing the **Even Faster Text** section, submit a zip archive of your HW2 code after completing the **Performance Bounding Experiments** section. This must include your improved version of the macro **GPIO_Write** and **your glyph timing experiments code.**

GETTING STARTED

SOFTWARE TOOLS AND COMPONENTS

- Make sure that you are using the Arm Compiler 6 with MicroLIB
  - Options for Target -> Target -> ARM Compiler: Select **Use default compiler version 6**.
  - Options for Target -> Target -> ARM Compiler: Check **Use Microlib**.
- Get the contents of these directory trees from GitHub. Be sure to get all the files and subdirectories (recursively):
  - HW2 project at **agdean/ESO-25/HW/HW2/**
  - Profiler tools at **agdean/ESO-25/Tools/GetRegions/**.
- Keep these points in mind:
  - Spaces in a full path name may keep the tools from working, so avoid them.
  - Be sure to place the subdirectory tree **Tools** in the directory **ESO-25**, because the profiling support script **update_regions_ MDK.cmd** searches for GetRegions.exe starting several directories up (where it expects ESO-25 to be) and searching its subdirectories recursively.



PROGRAM AND TOOL CONFIGURATION SETTINGS

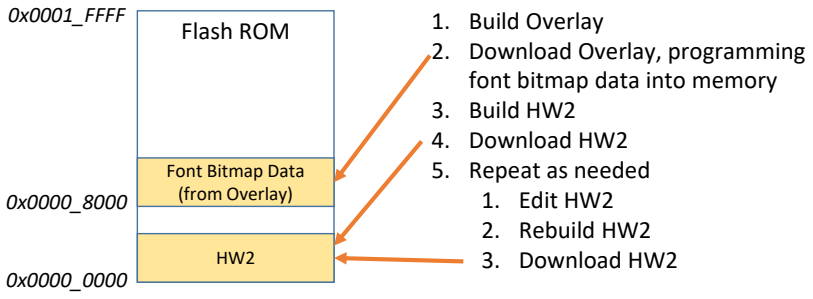
The program has been set up to make it easy to control certain optimizations and parameters. The table below shows these features and their locations.

Option	Location	Description	Initial Value
-O3, -Ofast	Target Options -> C/C++	Compiler option to optimize for time as much as possible.	(not used)
-ffp-mode=fast	Target Options -> C/C++, Misc Controls	Compiler option to generate faster floating point-code	(not used)
Use Microlib	Target Options, Target	Linker option to use smaller C run-time library.	Checked (used)
NUM_TESTS	config.h	Number of accelerometer readings made before reporting profile.	200
LAZY_TEXT_UPDATE	config.h	Update roll and pitch text only if they have changed enough since last display update.	0
LAZY_GRAPHICS_UPDATE	config.h	Update graphical roll indicator only if roll has changed enough since last display update.	0
UPDATE_TEXT	config.h	1 enables text updates (regular or lazy), 0 disables all	1
UPDATE_GRAPHICS	config.h	1 enables graphics updates (regular or lazy), 0 disables all.	1
SIM_ACCEL_XYZ_DATA	config.h	Overwrite X, Y, Z acceleration data with fixed sequence of values.	0
SIM_ACCEL_STEP	config.h	Simulation step value.	0
LCD_BUS_DEFAULTS_TO_DATA	config.h	Set default state of LCD bus to data, not control.	0
DRAW_LINE_RUNS_AS_RECTANGLES	config.h	Draws a run of adjacent pixels in a line as a rectangle.	0
USE_TEXT_BITMAP_RUNS	config.h	Speed up text bitmap rendering based on runs of consecutive pixels.	0
I2C_ICR_VALUE	config.h	Controls clock pre-scaling for I2C communications timing parameters.	0x20
READ_FULL_XYZ	config.h	Read full 16-bit acceleration data for each accelerometer.	0

MEMORY OVERLAY

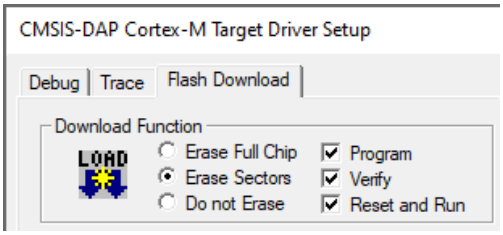
The free version of MDK-ARM limits object image size to 32 kB, even though the MCU has 128 kB of flash ROM. To keep from reaching 32 kB, some large read-only data (font bitmaps) has been moved out of the application project into a separate project (Tools\TestCode\Overlay).

As seen in this diagram, in steps 1 and 2 you'll build and download the Overlay project to your KL25Z first to install the Overlay ROM data. In steps 3 and 4, you'll build your application project (HW2) and download it *without erasing the overlay ROM data.* As you edit



the HW2 project, you'll build it and download it again, as shown in steps 5.1, 5.2 and 5.3. You do not need to repeat steps 1 and 2 (unless you erase the full chip).

How does this work? The MCU's Flash ROM has 128 sectors (1 kB each) which can be individually erased and programmed. The final project's download settings are configured so that only the sectors which are to be used are erased and programmed, rather than all sectors (the full chip). This means that the Overlay data will remain in ROM as long as you use a Flash Download setting of **Erase Sectors** instead of **Erase Full Chip** as shown in the figure and don't reprogram the sectors holding the data.



Using the overlay places the data for three fonts in overlay ROM (starting at address 0x00008000). It also puts a trivial, expendable program into ROM (starting at 0x00000000) which will be overwritten safely by your application program.

INITIAL PERFORMANCE ANALYSIS

BASELINE PERFORMANCE

- Build the program three times (by pressing F7 three times) to ensure the region table is accurate.
- Scroll to the top of the Build Output window, as shown below. The first line indicates the details (e.g. version, update, build numbers) of the compiler used.

1. Enter your compiler version details (e.g. V6.19).

```
Build Output
Build started: Project: HW2
*** Using Compiler 'V6.19', folder: 'C:\Keil_v5\ARM\ARMCLANG\Bin'
Build target 'Target 1'
```

- Connect the FRDM-KL25Z to the PC (if not already connected).
- Download the code and run the program. When it finishes executing, the LCD should display profile information.
    - You can touch the LCD to advance through each page of the profile data and then repeat the test.
    - Long function names are truncated to 14 characters to fit on the LCD.
        - You can look up the function in the array RegionTable (in region.c) to get more of the name (up to 24 characters).
        - If the name is still truncated, open the map file (Listings*.map), search for Image Symbol Table, and then search for the function name.
        - If multiple function names truncate to the same string, run the program from the debugger and stop when the first profile screen is displayed. Use the RegionCount array to find the region with the matching sample count as the function with the truncated name. Then use that region's number to identify the function.

2. What is the value of Lost Samples (listed on the second line)?

- The number of lost samples should be 0. If it is anything else, you will need to troubleshoot the problem. The most likely reason is using a different version of the compiler. The profiler works with AC V6.22 (and should work with later versions), which is available at https://developer.arm.com/downloads/-/arm-compiler-for-embedded. Then restart this assignment. If there are still lost samples at this point, contact the instructor.

3. Enter the most important parts of the profile data, which we will call the **profile subset**. This consists of the total sample count, and the name and sample count for the top four functions. For example:

# Total Samples	Top function Name	# Samples for Top Function	2nd function name	# Samples for 2nd Function	3rd function name	# Samples for 3rd Function	4th function name	# Samples for 4th Function
1000	F1	500	F2	300	F3	100	F4	50

4. What is the average time per loop (total samples * 1 ms/sample)/(NUM_TESTS) **in microseconds**?

STABILIZING TEST CONDITIONS

EVALUATING INPUT DATA IMPACT ON RUN TIME

The code's execution time depends on data derived from input conditions such as board orientation and movement. Drawing the arrow on the LCD depends on the angle of the lines. A horizontal or vertical line is just one rectangle, so it can be drawn quickly. A diagonal line is multiple squares, each drawn as a rectangle (1 pixel by 1 pixel). Any other kind of line is multiple rectangles.

- Run the program several times and observe that the sample count changes based on board orientation and motion.

1. Try to **minimize** the total sample count. Enter the profile data subset in the spreadsheet. We will call this the **Min.** Profile**.**
2. Try to **maximize** the total sample count. Enter the profile data subset in the spreadsheet. We will call this the **Max.** Profile**.**
3. What is the relative difference between the minimum and maximum total sample counts (i.e. 100%*(Max Total Sample Count – Min Total Sample Count)/Max Total Sample Count)?
4. In your report, describe how the Max. profile compares with the Min. profile. Do the same top functions appear in the same order, and with the same relative fraction of sample counts? If not, discuss the differences.

- Let's try to be extremely consistent with test conditions to determine repeatability. Place the board on a stable, steady object such as a table or desk. Run the program ten times and record each total sample count. You do not need to submit these sample counts.

5. What are the minimum and maximum total observed sample counts?

SIMULATING INPUT DATA

The accelerometer is so sensitive that it is essentially impossible to duplicate the physical conditions exactly. This will affect the execution time (and therefore the profile measured) and may mislead us. So, let's simulate the input data. You might be tempted to remove the code which reads the accelerometer, since that data isn't used. However, that would change the program's execution time by eliminating the I2C communications. To keep consistent timing, the code reads the accelerometer but then overwrites the measured accelerations with simulated data.

- Change the definition of SIM_ACCEL_XYZ_DATA from 0 to 1.

- Rebuild the program (three times) and download it. Then run the program ten times and observe the extremes of the sample count. Either keep the board steady for the whole test run or move it around. Either way, the sample count should be consistent and steady.

6. Over the ten runs, what was the minimum observed total sample count?
7. Over the ten runs, what was the maximum observed total sample count?

- Run the program again ("11th time").

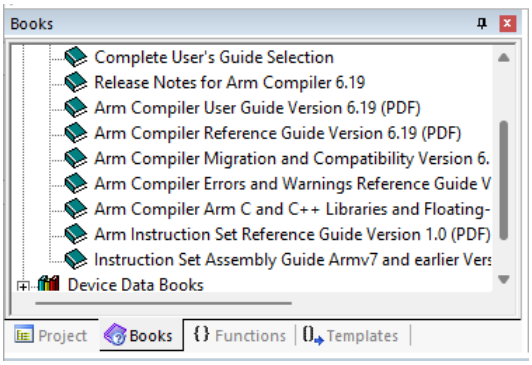
8. Enter the profile data subset. We will call this the **Sim.** profile.
9. In your report, explain how the **Sim**. profile compares with the **Min**. and **Max**. profiles. Do the same top functions appear in the same order, and with the same relative fraction of sample counts?

You will continue to use simulated input data for these experiments (keep **SIM_ACCEL_XYZ_DATA** as 1) unless or until instructed otherwise.

10. What is the average time per loop (total samples * 1 ms/sample)/(NUM_TESTS) **in microseconds**?

TOOL OPTIMIZATION SETTINGS

Let's try out some optimization settings before diving into the code. The class slides and videos use the previous version of the Arm C compiler (version 5). We are using the Arm Compiler for Embedded (version 6), which is a customized version of the llvm toolchain's clang compiler. Much of the compiler-specific syntax has changed, such as command line options, pragmas, intrinsics, keywords, etc. The project's **Books** tab links to various reference guides. Some will be especially useful: Arm Compiler User Guide, Arm Compiler Reference Guide, Arm Compiler Migration and Compatibility.



CHECK CURRENT OPTIMIZATION SETTINGS

- Open the Options for Target dialog (alt-F7) and select the C/C++ tab.

1. What is the currently selected optimization level?

- Change the optimization level to **-Ofast**. Rebuild the program three times and download it. Run the program.

2. What is the total number of samples?

- Change the optimization level to **-O3**. Rebuild the program three times and download it. Run the program.

3. What is the total number of samples?

- Open the target options (Alt-F7) and select the C/C++ tab. Add **-ffp-mode=fast** to the **Misc Controls** box and press OK.
- Build the program (three times) and download it. Run the program.

4. What is the total number of samples?

ANY DOUBLE-PRECISION MATH?

Is there any double-precision floating-point math being performed? It is slow and probably not needed.

5. Examine the most recent profile on the LCD. Which (if any) are the first four functions in the profile seem to be performing double-precision floating-point math, and how many samples does each have?

Some compilers can provide diagnostic messages (when asked) to help us understand what is being done with the source code. AC6 is based on clang, which has documentation here on how to request diagnostic warnings: https://clang.llvm.org/docs/DiagnosticsReference.html. If you search for double, float, and promotion you will find various warnings available.

- Let's ask the compiler to warn us when type promotion leads to double-precision floating-point math. Open **Options for Target,** select **C/C++** and in **Misc Controls** add **-Wdouble-promotion**, then select OK. Rebuild the program once and observe the Build Output window.

6. How many warnings are there about implicit conversion increasing floating-point precision?

- Each warning message shows exactly which source code is responsible. You can double-click on the first line of the warning to go directly to the relevant source code.
- The Build Output window is also saved to the <project_name>.build_log.htm file in the Objects directory. Opening this file with a browser improves visibility, but doesn't offer links to the source code.
- Use the methods described in the lecture slides and videos to edit the source code to eliminate **as many** of these warnings as possible. You will not be able to remove two warnings (caused when float parameters are passed to sprint). Build the program (three times) and download it. Run the program.

7. Enter the profile data subset.
8. Do any of the top four functions in the profile seem to be performing double-precision floating-point math, and how many samples do they each have?

WHAT'S THE COMPILER DOING?

- Let's ask the compiler tell us about the optimizations it performed. According to the manual (**Arm Compiler for Embedded Reference Guide**) we can use the **-Rpass** option to get remarks from the optimization passes.
- Open **Options for Target,** select **C/C++.** In **Misc Controls** add **-Rpass=.*** then select OK. Rebuild the program once. The compiler's remarks will appear in the Build Output window. Open the build log file with a web browser for easier analysis.

9. How many function calls are inlined? Use the browser's search function to count the number of times **inlined into** appears in the file.
10. How many loops are unrolled?
11. How many of those loops are completely unrolled?
12. How many times was hoisting used?

DO LESS WORK

LAZY DISPLAY UPDATES

LCD_ functions should dominate the execution time. There are many different ways to try to speed them up, but let's start with something easy: Don't update the roll or pitch information on the display if it hasn't changed since the last time it was drawn.

- Change the definition of LAZY_TEXT_UPDATE from 0 to 1.

1. Examine the code controlled by LAZY_TEXT_UPDATE. Why should this change speed up the program?

- Build the program (three times), download it and run it.

2. What is the total observed sample count?
3. Why do you think this change did (or did not) speed up the program?

- Change the definition of LAZY_GRAPHICS_UPDATE from 0 to 1, and LAZY_TEXT_UPDATE back to 0.

4. Examine the code controlled by LAZY_GRAPHICS_UPDATE. Why should this change speed up the program?

- Build the program (three times), download it and run it.

5. What is the total observed sample count?
6. Why do you think this change did (or did not) speed up the program?

- Change the definition of LAZY_TEXT_UPDATE to 1.
- Change the definition of LAZY_GRAPHICS_UPDATE to 1.
- Build the program (three times), download it and run it.

7. Enter the profile data subset. We'll call this the **sim. roll** profile.

LAZY DISPLAY UPDATES WITH LIVE DATA

Let's see about how much lazy updates help when we have live data from a steadier board.

- Change the definition of SIM_ACCEL_XYZ_DATA from 1 to 0.
- Change the definition of LAZY_TEXT_UPDATE to 0.
- Change the definition of LAZY_GRAPHICS_UPDATE to 0.
- Build the program (three times), download it and run it three times with the board motionless.

8. What is the **average** total observed sample count?

- Change the definition of LAZY_TEXT_UPDATE from 0 to 1.
- Build the program (three times), download it and run it three times with the board motionless.

9. What is the **average** total observed sample count?

- Change the definition of LAZY_GRAPHICS_UPDATE from 0 to 1, and LAZY_TEXT_UPDATE back to 0.

10. Examine the code controlled by LAZY_GRAPHICS_UPDATE. Why should this change speed up the program?

- Build the program (three times), download it and run it three times with the board motionless.

11. What is the **average** total observed sample count?

- Change the definition of LAZY_TEXT_UPDATE to 1.
- Change the definition of LAZY_GRAPHICS_UPDATE to 1.
- Build the program (three times), download it and run it three times with the board motionless.

12. What is the **average** total observed sample count?
13. Enter the profile data subset. We'll call this the **live steady** profile.

- Run the program again while gently moving the board.

14. Enter the profile data subset. We'll call this the **live roll** profile.
15. In your report, explain why both the overall performance and profiles (time breakdown among functions) vary so much among these three test cases (sim. roll, live steady, live roll).

ACCELERATING LCD UPDATES

In many cases, the top functions are associated with the LCD, so let's work on that first. Please refer to the slides **Speed – Shield Optimizations** (on the course website in the Speed module) for information on details of the LCD interface.

- Change the definition of SIM_ACCEL_XYZ_DATA to 1 to provide consistent input data.

FASTER DATA WRITES ON LCD BUS

One optimization option speeds up data writes on the LCD bus.

- Change the definition of LCD_BUS_DEFAULTS_TO_DATA from 0 to 1.

1. Examine the code controlled by LCD_BUS_DEFAULTS_TO_DATA. Why should this change speed up the program?

- Build the program (three times), download it and run it.

2. Enter the profile data subset.

DRAW RUNS AS RECTANGLES, ELIMINATING LCD_PLOT_PIXEL

The function LCD_Plot_Pixel should appear in the profile. We learned in class that plotting one pixel at a time is very inefficient because of the set-up overhead.

- Change the definition of DRAW_LINE_RUNS_AS_RECTANGLES from 0 to 1.

3. Examine the code controlled by DRAW_LINE_RUNS_AS_RECTANGLES. Why should this change speed up the program?

- Build the program (three times) and download it. Run the program.

4. Enter the profile data subset.

USE RUNS IN BITMAPS

Some of the LCD accesses come from updating the LCD with text. The function LCD_Text_PrintChar in LCD_Text.c draws a character (or other symbol) on the LCD using the LCD_Start_Rectangle and LCD_Write_Rectangle_Pixel functions. It determines whether a pixel should be the foreground or background color by using the bitmap data for the character.

EXPLANATION OF IMPLEMENTATION

LCD_Text_PrintStr calls LCD_Text_PrintChar for each character in the string. LCD_Text_PrintChar prints each character symbol ("glyph") as a rectangle. It first calls LCD_Start_Rectangle to define the location on the screen with the pixels to be updated. Each pixel is given either the foreground or background color, based on the glyph's bitmap data. This is done by calling LCD_Write_Rectangle_Pixel with the color and the number of pixels of that color to write.

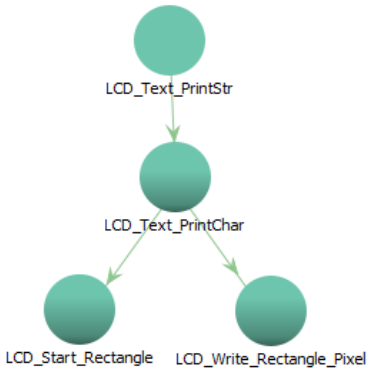
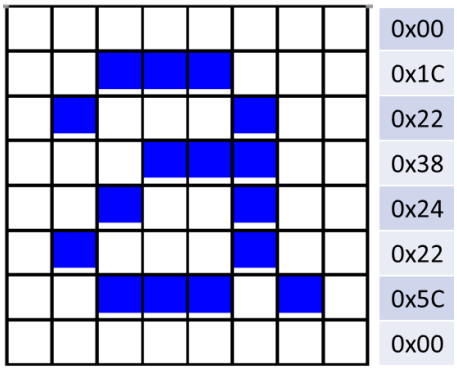


Figure 1. Partial function call graph of LCD_Text_PrintStr

Note that the function LCD_Write_Rectangle_Pixel calls LCD_24S_Write_Data, which calls GPIO_Write, GPIO_ResetBit and GPIO_SetBit. However, the high level of compiler optimization has in-lined those function calls, so they don't appear in this call graph.

LCD_Text_PrintChar (original)

```
set up
LCD_Start_Rectangle
for each row
  do {
    read next bitmap_byte // <= 8 cols of pixel data
    for each column in bitmap_byte
      color = (bitmap_byte & 1)? fg : bg
      write pixel with color
      bitmap_byte >>= 1
  } while more columns in glyph
  if spacing needed
    write pixels with bg
```



The pseudocode above summarizes the function LCD_Text_PrintChar(), which is described below in more detail:

- Execute set up code for initialization, check inputs for errors, and calculate glyph and screen locations.
- Call LCD_Start_Rectangle to tell the LCD controller to start drawing a rectangle at the glyph's location on screen (starting and ending columns, starting and ending rows). However, LCD_Start_Rectangle does not send any pixel data to the LCD.
- The outer loop steps through each row of the glyph (character symbol), while the inner loop steps through each column of the glyph data, checking the corresponding pixel of bitmap data and calling LCD_Write_Rectangle_Pixel with the foreground or background color to write one pixel.
- Additional background pixels may be written when needed for spacing.

As shown in the diagram above, the bitmap data for each glyph is encoded as an array of bytes, starting with the left-most pixel in the least-significant bit of the byte. Glyphs wider than eight columns use multiple bytes per row to hold bitmap data. Each glyph may have its own width, which is stored along with the other glyph data.

The LCD_Text_PrintChar function is shown below (with simplifications for readability).

```
void LCD_Text_PrintChar(PT_T * pos, char ch) {
  if (ch > font_header->LastChar)  // error: character not represented in font
    ch = '?';
  glyph_index_entry = ch - font_header->FirstChar;
  glyph_width = glyph_index[glyph_index_entry].Width;
  offset = glyph_index[glyph_index_entry].Offset;
  glyph_data = &(font[offset]);
  end_pos.X = pos->X + CHAR_WIDTH - 1 + CHAR_TRACKING;
  end pos.Y = pos->Y + CHAR HEIGHT - 1;
  LCD Start Rectangle(pos, &end pos);
  for (row = 0; row < CHAR_HEIGHT; row++) {
    x_bm = 0; // x position within glyph bitmap, can span bytes
    do {
      bitmap_byte = *glyph_data;
        if (bitmap_byte & 0x01) // if pixel is to be set
          pixel_color = &fg;
        else
          pixel_color = &bg;
        LCD_Write_Rectangle_Pixel(pixel_color, 1);
        bitmap_byte >>= 1;
        x_bm++;
      }
      glyph_data++;   // Advance to next byte of glyph data
    } while (x_bm < glyph_width);
    if (x_bm < CHAR_WIDTH+CHAR_TRACKING) {
      // fill in rest of cell with background color for narrow glyphs
      LCD_Write_Rectangle_Pixel(&bg, CHAR_WIDTH + CHAR_TRACKING - x_bm);
    }
  }
```

Listing 1. Simplified excerpt from LCD_Text_PrintChar

Finally, Listing 2 shows an excerpt of the function LCD_Write_Rectangle_Pixel. It takes a color and a pixel count as arguments. It converts that color from a 24-bit value (8:8:8) to a 16-bit value (5:6:5) and then sends **count** copies of that value to the LCD (one byte at a time) over the GPIO port using LCD_24S_Write_Data.

```
/* Write color in the next count pixel locations in the active rectangle.
You must have called LCD_Start_Rectangle before calling this function. */
void LCD_Write_Rectangle_Pixel(COLOR_T * color, unsigned int count) {
  uint8_t b1, b2;

  // 16 bpp, 5-6-5. Assume color channel data is left-aligned
  b1 = (color->R&0xf8) | ((color->G&0xe0)>>5);
  b2 = ((color->G&0x1c)<<3) | ((color->B&0xf8)>>3);
  while (count--) {
    LCD_24S_Write_Data(b1);
    LCD_24S_Write_Data(b2);
  }
}
```

Listing 2. Simplified excerpt from LCD_Write_Rectangle_Pixel.

You can find more information in the **Speed – Shield Optimizations** slides and lecture.

EVALUATING THE OPTIMIZATION

- Change the definition of USE_TEXT_BITMAP_RUNS from 0 to 1. Examine the code controlled by USE_TEXT_BITMAP_RUNS. Pseudocode for the optimized version of LCD_Text_PrintChar is shown here.

5. Why should this version speed up the program?

- Build the program (three times), download it and run it.

6. Enter the profile data subset.

LCD_Text_PrintChar with USE_TEXT_BITMAP_RUNS

```
set up
LCD_Start_Rectangle
for each row
  do {
    read next bitmap_byte // <= 8 cols of pixel data
    if 8-bit background run
      write 8 bg pixels
    else if 8-bit foreground run
      write 8 fg pixels
    else if 7-bit background run
      write 7 bgpixels
    . . .
    else if 4-bit foreground run
      write 4 fg pixels
    for remaining columns in bitmap_byte
      color = (bitmap_byte & 1)? fg : bg
      write pixel with color
      bitmap_byte >>= 1
  } while more columns in glyph
if spacing needed
    write pixels with bg
```

FASTER GPIO WRITES TO LCD

The macro GPIO_Write could be improved. Currently it uses read/modify/write instructions on the port data output register PDOR. You can use the PSOR, PCOR, and PTOR registers to simplify some of the software.

- Use the microVision debugger or Ghidra to examine object code (assembly language) generated for GPIO_Write. Since it is a macro, it is integrated into each function which uses it (for example, LCD_24S_Write_Data, LCD_24S_Write_Command). Those functions may be in-lined into other functions, making code analysis more complex.

7. In your report, show the object code used to implement the GPIO_Write operation once.
8. How many assembly instructions are used to implement the GPIO_Write operation?

Remember the GPIO peripheral hardware can change its outputs with a hardware read/modify/write operation by using registers PSOR, PCOR, or PTOR.

- Use one or more of these to rewrite the GPIO_Write C source code resulting in shorter and faster object code.

9. In your report, provide the source code and object code for your improved version of GPIO_Write.
10. How many assembly instructions are used to implement the GPIO_Write operation?

- Build the program (three times), download and run it.

11. Enter the profile data subset.

ACCELERATING I2C COMMUNICATION

The I^2C communication with the accelerometer takes some of the program's time. For details on I^2C communications, refer to the section "Inter-Integrated Circuit Bus (I^2C)" in Chapter 8 of the textbook **Embedded System Fundamentals**, and Chapter 38 of the **KL25Z Subfamily Reference Manual**.

We would like to use a logic analyzer (e.g. Analog Discovery 2) to monitor the I^2C signals, but this is not easily done on the FRDM-KL25Z board because those signals are not easily accessed. The logical place to reach them is at the end of the tiny SMD pull-up resistors R16 and R18 (size 0402?). So, you will need to imagine the I2C communication without seeing it on a logic analyzer.

CHECK DATA RATE

- The communication speed of I^2C communications is determined by the SCL signal's frequency. Open the latest datasheet for the accelerometer **MMA8451Q-rev10.pdf** (located in the assignment's References directory) to find the maximum frequency of SCL.

1. Look at Table 4 (I^2C Timing Values). What is the maximum SCL clock frequency?

- The I2C peripheral's SCL clock frequency is set by dividing the bus clock signal by the SCL divider. In the **KL25 Subfamily Reference Manual.pdf**, Table 38-41 shows how the ICR value (first column) determines the values of the SCL divider (second column) and other fields. You can calculate the I^2C baud rate using the equation in Section 38.3.2. The bus speed (clock frequency) is 24 MHz. Note that there is a hardware bug in the MCU's I2C peripheral, so the I2C_F MULT field must be kept at 00 (for a **mul** value of 1). Other MULT values may cause communication failure.

2. Examine the function i2c_init (in I2C.c). What numerical value (base 10) is loaded into ICR? The symbol is #defined in config.h.
3. What is the I^2C baud rate resulting from this ICR value?
4. Which SCL divider will get us closest to the maximum SCL frequency for the chip?
5. What SCL frequency do you expect, given that divider value?
6. The SCL divider is selected by the ICR field of the peripheral's F register. What ICR value is needed for that SCL divider?

- Change the #define for I2C_ICR_VALUE in config.h to use this new ICR value.
- We want to ensure I2C communications still work, so disable data simulation by defining SIM_ACCEL_XYZ_DATA as 0 again. Rebuild the program three times and download it. Run it and verify that it still works correctly. The display should be updated as you move the board.

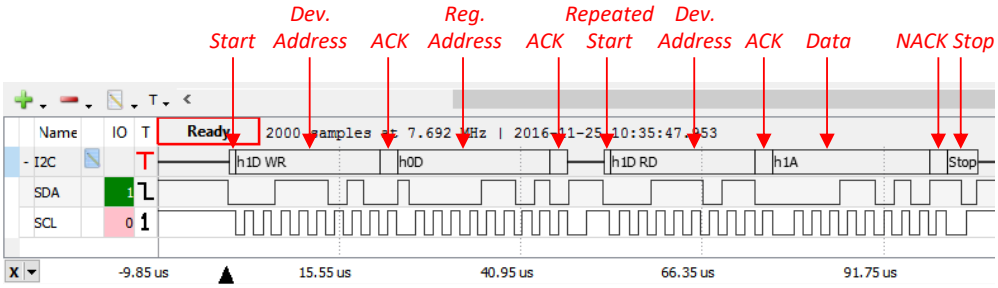
7. Does the program still work correctly?

- Next we'll need to stabilize the input data to continue our timing analysis. Change the definition of SIM_ACCEL_XYZ_DATA to 1. Build the program (three times) and download it. Run the program three times.

8. What are the minimum and maximum total observed sample counts?
9. Enter the profile data subset from the latest profile.

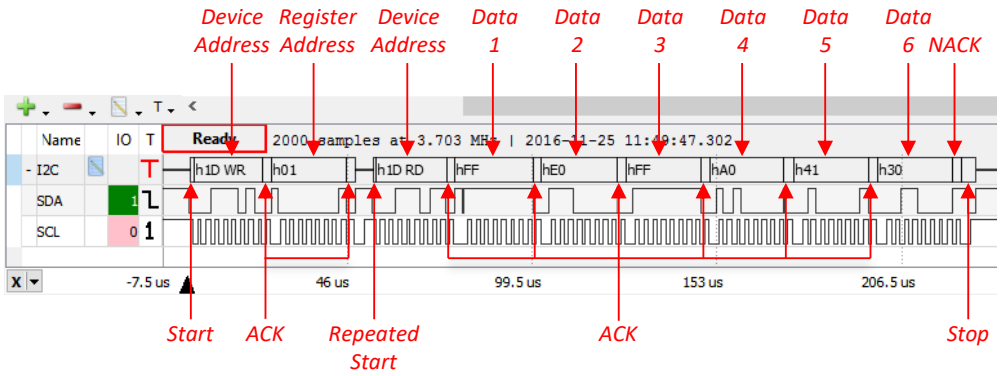
HOW MANY I2C TRANSACTIONS?

Now let's take a different approach to improving communication performance. Consider what is happening during I²C communications. I²C transactions contain a device address, a read/write indicator flag, a register address, and data for that register. The logic analyzer trace below shows how one byte of data is read.



- Open the accelerometer datasheet and look for the table called "Register Address Map". There are two bytes of data for each acceleration component.
- Examine the code in **read_xyz**. It performs three separate read transactions: one for the most-significant byte of the X acceleration, another for the Y and another for the Z.
- Read the description of the data registers on page 22 of the data sheet. You'll see that the data registers for X, Y and Z accelerations (OUT_...) are in sequential order, and also that these registers can be read with an auto-incrementing address.

An I²C transaction which reads multiple bytes of data from register address A will return the data from register addresses A, A+1, A+2, etc. This autoincrement means we can read sequential registers in a single read operation. The logic analyzer trace below shows how six consecutive bytes of data are read.



Examine the function **read_full_xyz**. It uses a single read transaction to get six consecutive bytes (starting at register OUT_X_MSB) holding all the data for each acceleration component: X, Y and Z.

- Change the definition of READ_FULL_XYZ from 0 to 1 to switch to use the single I²C message.
- We want to ensure I2C communications still work, so disable data simulation by defining SIM_ACCEL_XYZ_DATA as 0 again. Rebuild the program three times and download it. Run it and verify that it still works correctly. The display should be updated as you move the board.

10. Does the program still work correctly?

- We'll need to stabilize the input data to continue our timing analysis. Define SIM_ACCEL_XYZ_DATA as 1. Build the program (three times) and download it. Run the program three times.

11. What are the minimum and maximum total observed sample counts?
12. Enter the profile data subset from the latest profile.

PUSH DATA RATE FURTHER

A previous version of the datasheet for the accelerometer (**MMA8451Q-rev7.1.pdf**, in the References folder) suggests a higher I^2C clock speed is possible.

- Try pushing the clock frequency as high as possible (with the system still working) by reducing ICR and therefore increasing the frequency of SCL. We want to ensure I2C communications still work, so disable data simulation by defining SIM_ACCEL_XYZ_DATA as 0 again. Rebuild the program three times and download it. Run it and verify that it still works correctly. The display should be updated as you move the board.

13. What is the minimum working SCL divider?
14. What is the maximum working SCL frequency?
15. What is the ICR value for this frequency?

- We'll need to stabilize the input data to complete our timing analysis. Define SIM_ACCEL_XYZ_DATA as 1. Build the program (three times) and download it. Run the program three times.

16. What are the minimum and maximum total observed sample counts?
17. Enter the profile data subset from the latest profile.

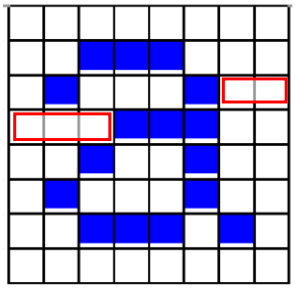
ECE 561 ONLY: EVEN FASTER TEXT?

LIMITATIONS AND OPPORTUNITIES

In the section USE RUNS IN CHARACTER BITMAPS above, you defined USE_TEXT_BITMAP_RUNS as 1 in order to draw characters as a sequence of runs (typically multiple pixels long), rather than as a sequence of pixels. LCD_Write_Rectangle_Pixel is called for each run rather than for each pixel.

In the improved code, LCD_Text_PrintChar identifies same-color pixel runs in the font bitmap data (bitmap_byte) as the program runs in order to draw each run with a single call to LCD_Write_Rectangle_Pixel. This identification takes time each time a glyph is drawn, as it sequentially tests for runs of 8, 7, 6, 5, and finally 4 pixels. Furthermore, it doesn't identify all of the runs.

You are considering whether to change the font bitmap data structures to explicitly identify each run (a color and the number of consecutive pixels) to simplify the code. The diagram shows the concepts for a simple 8x8 bitmap (fg = foreground, bg = background). Now all runs will all be pre-identified, and the new LCD_Text_PrintChar code just needs to call LCD_Write_Rectangle_Pixel for each run. If runs can wrap around the end of the row (e.g. bg 2 at end of 3rd row) and continue on the next row (e.g. bg 3 at start of 4th row), the number of runs falls, so the code should be even faster.



bitmap_byte	Runs per Row	Runs with Wrapping
0x00	bg 8	bg 10
0x1C	bg 2, fg 3, bg 3	fg 3, bg 4
0x22	bg 1, fg 1, bg 3, fg 1, bg 2	fg 1, bg 3, fg 1, bg 5
0x38	bg 3, fg 3, bg 2	fg 3, bg 4
0x24	bg2, fg 1, bg 2, fg 1, bg 2	fg 1, bg 2, fg 1, bg 3
0x22	bg 1, fg 1, bg 3, fg 1, bg 2	fg 1, bg 3, fg 1, bg 4
0x5C	bg 2, fg 3, bg 1, fg 1, bg 1	fg 3, bg 1, fg 1, bg 9
0x00	bg 8	

IS IT WORTH DOING?

Applying this optimization will take some work: changing the code and also changing the bitmap fonts in use. Instead of blindly diving in and optimizing, let's get an idea of whether it's likely to be worth doing. It would help to know the **best possible performance** (smallest execution time) that we could expect from the new code. If that performance isn't much better than the existing code, then there is little reason to spend the time on that optimization.

PERFORMANCE BOUNDING CONCEPTS

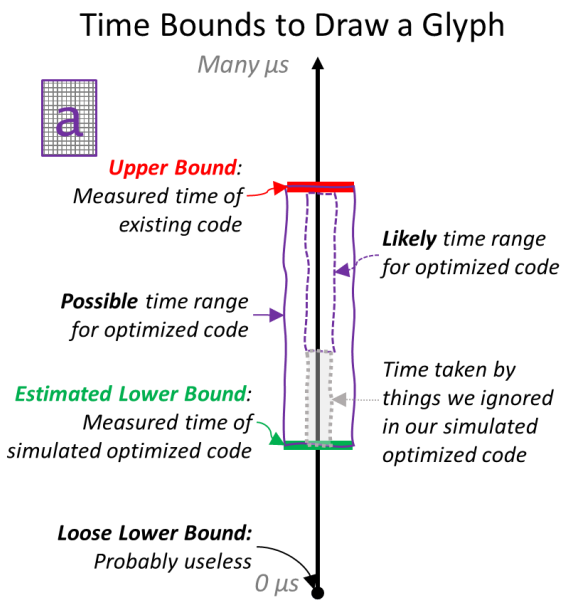
Our starting point is determining how long the existing LCD_Text_PrintChar code takes to draw a glyph.[1] This will be our **upper bound** on execution time for the optimized code. We know that the execution time will be no larger than that bound. After all, if our new "optimized" code is slower than the existing code, we will just use the existing code.

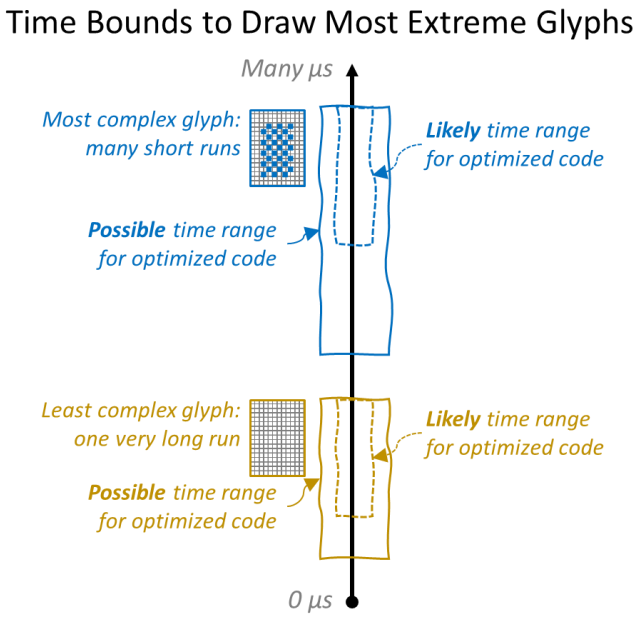
We are very interested in estimating a **lower bound** on the execution time for the optimized LCD_Text_PrintChar code. As that bound gets closer to the real execution time (i.e. becomes **tighter**), the more useful it is. A lower bound of 0 µs is accurate, but not very useful because it is so **loose** (too far from the real value). It is likely to mislead us when we try to decide what to optimize and how[2].

So, how can we easily get a lower bound on the optimized code's execution time which is **tight** enough to be useful? One way is by examining the problem, simplifying it by identifying likely extreme performance cases (best-case and worst-case), implementing them in code ("simulated optimized code") and measuring that code's timing behavior.

In the best case, the glyph is a rectangle with only one color (e.g. the space character). The simulated optimized code won't need to identify runs as the code executes, instead immediately filling in one rectangle with a one run of pixels.

We can use the execution time of this code as a **lower bound** on our estimate of the execution time possible by optimizing LCD_Text_PrintChar. This is shown as a **possible** time range in the diagram. This bound is tighter and more useful than the 0 µs bound mentioned earlier.





[1] This is data-dependent. Unless each glyph takes the same time to draw, **which** glyphs are drawn affects the overall execution time for existing code (and optimized code, and therefore the resulting speed-ups). A glyph's drawing time depends on bitmap dimensions (# of pixels to send), number of runs, and other factors. For more accuracy, you would draw a fixed set of glyphs with a frequency distribution close to the program's real data. That is outside the scope of this assignment.

[2] We also want the bound to be **safe**. A **safe lower bound** will never be larger than the actual value. A **safe upper bound** will never be smaller than the actual value.

In order to make this analysis easier, our simulated optimized code ignores some aspects of the work and also targets the best case, so we expect it to be faster than a real, complete implementation. That is reasonable for this initial rough estimation. We don't know the timing impact of these assumptions and omissions, so we expect the **likely time range** for the optimized code to be somewhat worse than the **possible time range** determined by the simulated optimized code.

PERFORMANCE BOUNDING EXPERIMENTS

You will simulate the optimized LCD_Text_PrintChar function by creating a test function which …

- … Calls LCD_Start_Rectangle and then calls the LCD_Write_Rectangle_Pixel once for each run (using parameters for pixel color and count). Note that the program uses a larger bitmap (20x31) than the example above (8x8), so your calls to LCD_Start_Rectangle and LCD_Write_Rectangle_Pixel may need to be modified.
- … Has timing instrumentation code (DEBUG_START(…), DEBUG_STOP(…)) to indicate the character printing time using output port debug bits. Then you can use the Waveforms/Analog Discovery logic analyzer or scope to view and measure that time. Remember that Waveforms can automatically measure pulse widths and accumulate statistics (average, min, max, etc.) over repeated acquisitions.

The number of runs per glyph affects performance, since each run requires a call to LCD_Write_Rectangle_Pixel. Different glyphs are likely to have different performance improvements. You should test multiple glyphs to get a more accurate performance range.

- The easiest bound is the fastest case: a glyph with only one run. This run covers the entire area of the rectangle. The space character is an example.
- A somewhat more realistic case is a glyph with three runs per row. For example, the glyph **1** has at most two runs per row (with run wrapping) or three (without run wrapping).
- You may even want to print all the font's glyphs (for (uint8_t c = ' '; c <= '~'; c++) ….) and estimate the average number of runs per line. Note that our fonts omit glyphs following ~, where there are many extreme cases (e.g ¾ÖhuDžŮ õA).

WHAT ABOUT FONT CONVERSION?

Don't forget that for this optimization to be useful, the font bitmaps need to be converted to explicitly represent those bitmap runs. You can get extra credit by suggesting a way to do this conversion automatically (see report section **e** below). Please refer to the **Speed – Shield Optimizations.pdf** lecture slides for details on font bitmap format (p. 28) and use (pp. 11-14).

REPORT STRUCTURE

1. In your report, present the following information. The report template contains a parallel structure so you can plug in your responses easily:
    a. Performance of Existing Code
        i. Methods used to determine performance.
            1. Which code was run?
            2. Which glyphs did you draw?
            3. How did you measure the time taken?
            4. Where appropriate, how did you determine the average, minimum and maximum times?
            5. Did you use profiling or not, and why?

ii. Performance data (execution time in µs) for existing code when printing …
    1. Most simple glyph (space character)
    2. Most complex glyph (and which is it?)
    3. Glyphs for numbers: all digits, decimal point, comma, space. Include minimum, average and maximum times.
    4. All glyphs in set (from space to ~). Include minimum, average and maximum times.

iii. Analysis of this performance data.
    1. How much variation is there?
    2. Does the data make sense, or are there any surprises?

b. Performance of Test Code

i. Explanation of Simulation
    1. Explain how your test function simulates the optimized code.
    2. What is included?
    3. What is omitted?

ii. Approach to Glyph Group Simulation
    1. What is your approach to simulating each of the four groups of glyphs above (ii.1. to ii.4.)?
    2. How accurate do you think that approach is for each group?

iii. Performance data (execution time in µs) for test code when printing the **simulated** …
    1. Most simple glyph
    2. Most complex glyph
    3. Extra Credit: Glyph with three runs per row
    4. ~~Glyphs for numbers: all digits, decimal point, comma, space. Include minimum, average and maximum times.~~
    5. ~~All glyphs in set (from space to ~). Include minimum, average and maximum times.~~

iv. Analysis of this performance data.
    1. How much variation is there?
    2. Does the data make sense, or are there any surprises?

c. Performance Improvement

i. Performance improvement ratios (speed-ups), where speed-up = t_{old}/t_{new}) for the…
    1. Most simple glyph
    2. Most complex glyph
    3. Extra Credit: Glyph with three runs per row
    4. ~~Glyphs for numbers. Include ratios of minima, averages and maxima.~~
    5. ~~All glyphs in set. Include ratios of minima, averages and maxima.~~

ii. Analysis of this performance data.
    1. How much variation is there?
    2. Does the data make sense, or are there any surprises?

d. Extra Credit: Font Conversion
    1. Suggest a way to do this conversion automatically. Provide pseudocode & diagrams.

e. Summary

i. Consider what the test code **omits**.
    1. What does the test code **omit**?
    2. Is adding the omitted likely to reduce the overall performance improvement much?
    3. Why or why not?

ii. Is it worth implementing?

1. After considering the omitted code and the resulting performance, do you think it is **worthwhile** for someone (e.g. future students) to implement the optimization?

2. Enter the speed-up data from 1.c.i.1 and 1.c.i.2 (speed-up factors for the simplest and most complex glyphs).

REVISION HISTORY

V1.02

Corrected DRAW_RUNS_AS_RECTANGLES to DRAW_LINE_RUNS_AS_RECTANGLES, highlighted in yellow.

V1.03

In **Even Faster Text**, simplified requirements for test code in report (1.b.iii.3-5, 1.c.i.3-5)., and question 2 (speed-up data).