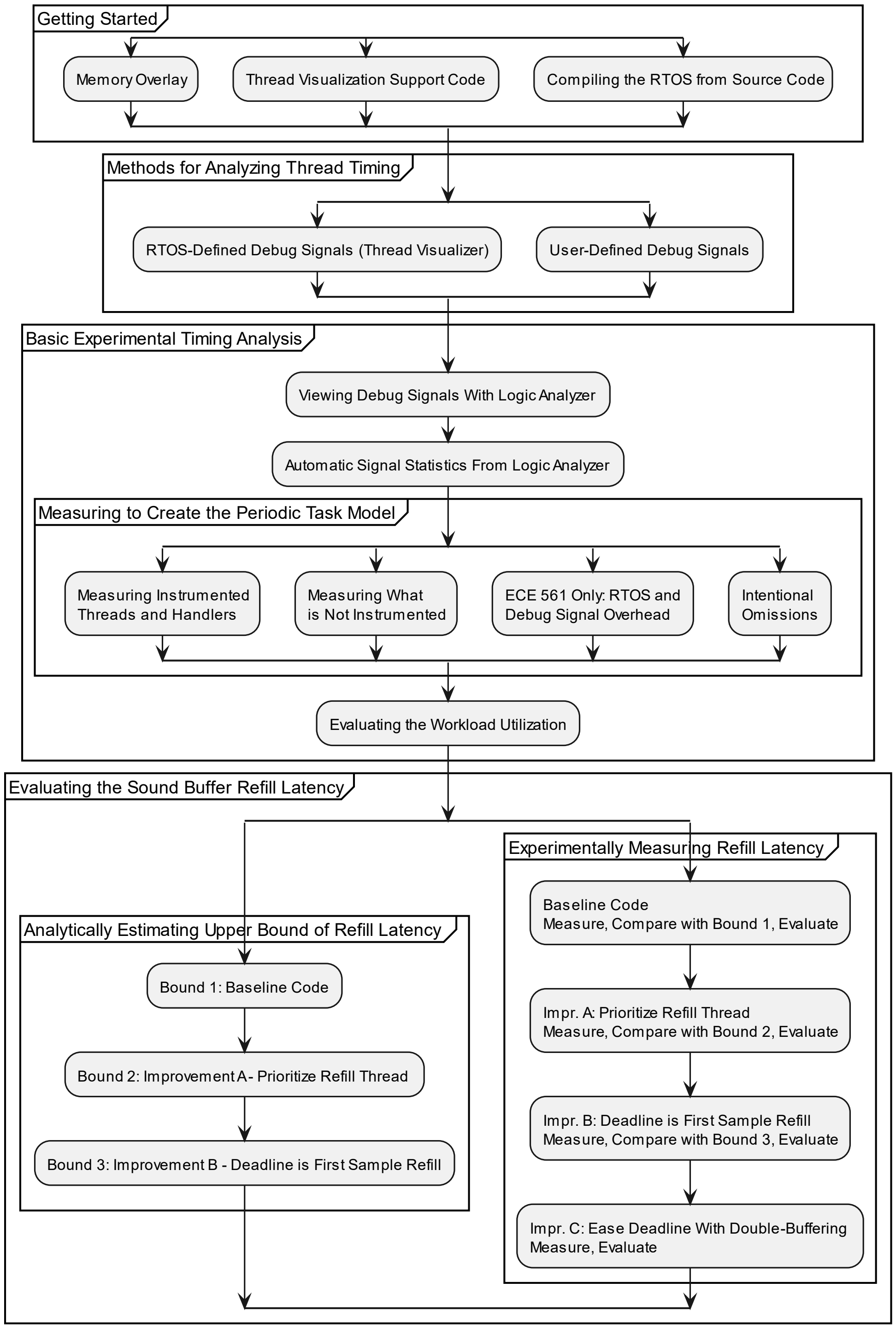
HW3: ANALYZING AND "OPTIMIZING" RESPONSIVENESS (V1.02)


Getting Started
Memory Overlay
Thread Visualization Support Code
Compiling the RTOS from Source Code
Methods for Analyzing Thread Timing
RTOS-Defined Debug Signals (Thread Visualizer)
User-Defined Debug Signals
Basic Experimental Timing Analysis
Viewing Debug Signals With Logic Analyzer
Automatic Signal Statistics From Logic Analyzer
Measuring to Create the Periodic Task Model
Measuring Instrumented Threads and Handlers
Measuring What is Not Instrumented
ECE 561 Only: RTOS and Debug Signal Overhead
Intentional Omissions
Evaluating the Workload Utilization
Evaluating the Sound Buffer Refill Latency
Experimentally Measuring Refill Latency
Baseline Code Measure, Compare with Bound 1, Evaluate
Impr. A: Prioritize Refill Thread Measure, Compare with Bound 2, Evaluate
Impr. B: Deadline is First Sample Refill Measure, Compare with Bound 3, Evaluate
Impr. C: Ease Deadline With Double-Buffering Measure, Evaluate
Analytically Estimating Upper Bound of Refill Latency
Bound 1: Baseline Code
Bound 2: Improvement A - Prioritize Refill Thread
Bound 3: Improvement B - Deadline is First Sample Refill

CONTENTS

Overview 3
Formatting 3
Getting Started 4
Memory Overlay 4
Thread Visualization Support Code 5
Compiling the RTOS from Source Code 5
Methods for Analyzing Thread Timing 6
RTOS-Defined Debug Signals (Thread Visualizer) 7
User-Defined Debug Signals 8
Basic Experimental Timing Analysis 9
Viewing Debug Signals With Logic Analyzer 9
Automatic Signal Statistics From Logic Analyzer 9
Measuring to Create the Periodic Task Model 10
Measuring Instrumented Threads and Handlers 11
Measuring What is Not Instrumented 11
SysTick_IRQHandler 11
ECE 561 Only: RTOS and Debug Signal Overhead 13
Intentional Omissions 14
Evaluating the Workload Utilization 14
Evaluating the Sound Buffer Refill Latency 14
Analytically Estimating Upper Bound of Refill Latency 15
Bound 1: Baseline Code 17
Bound 2: Improvement A – Prioritize Refill Thread 17
Bound 3: Improvement B – Deadline is First Sample Refill 17
Experimentally Measuring Refill Latency 18
Baseline Code 19
First Try 19
Simplify Measurements With Automation 19
Second Try 21
Comparing the Model With Reality 21
Examining Refill Delays with Scope 21
Improvement A: Prioritize Refill Thread 22
Experimental Measurement 22
Comparison with Bound 2 23
Improvement B: Deadline is First Sample Refill 23
Experimental Measurement 23
Comparison with Bound 3 23
Taking a Closer Look 23
Improvement C: Ease the Deadline With Double-Buffering 24
Experimental Measurement 24
Taking a Closer Look 25
Revision History 25
V1.01 25
V1.02 25

OVERVIEW

In this exercise you will evaluate the timing characteristics of an RTOS-based system and then improve the quality of audio generation through scheduling and other changes. You need the following to complete this exercise:

- FRDM-KL25Z with Expansion Shield
- One Analog Discovery 2/3 or equivalent device, and Waveforms installed on a PC.
- One jumper wire (socket-to-socket) to connect the DAC output to the AD2 1+ input. These are available from the instructor at class and office hours.

The program is a ~~Pong-style game~~ **physics simulation** with sound effects. It uses the RTXv5 real-time operating system through the CMSIS-RTOS2 programming interface. The RTOS has its own threads and handlers, and the program adds these:

- Thread_Read_TS
- Thread_Read_Accelerometer
- Thread_Update_Screen
- Thread_Sound_Manager
- Thread_Refill_SoundBuffer
- DMA0_IRQHandler

Pressing the left half of the screen will switch the game between two modes. **Players: 0** indicates automatic paddle positioning, while by **Players: 1** lets you tilt the board to the left or right to ~~play the game~~ provide input data for the simulation. Pressing the right half of the screen will toggle the sound between on and off.

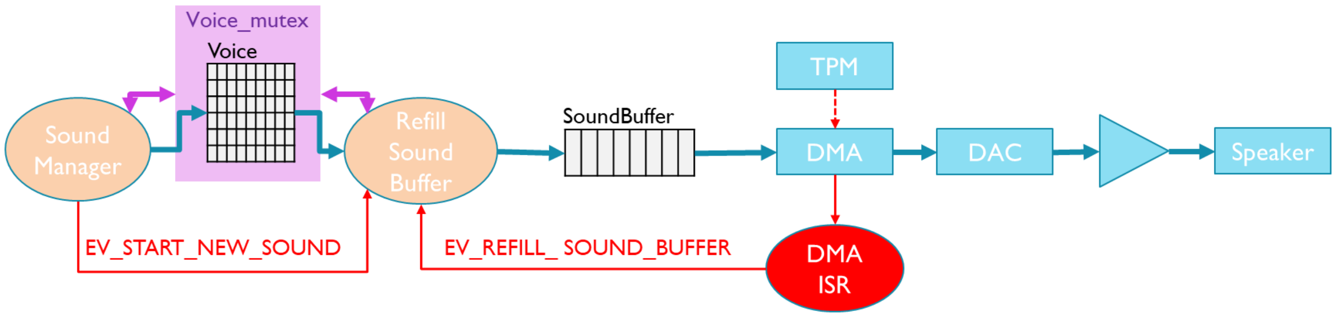


Figure 1. Sound generation architecture with default single buffer.

Figure 1 shows the most time-critical processing in the program: updating the audio output at a 20 kHz sample rate. Every 50 µs the direct memory access (DMA) peripheral transfers a sample from a sound buffer array to the DAC. When the buffer's last sample is transferred, the DMA controller requests an interrupt. The DMA interrupt service routine tells the RTOS that the thread Refill Sound Buffer should run. That thread will compute a batch of sound samples and refill the buffer. If this thread doesn't stay ahead of all upcoming DMA transfers, old data will be reused corrupting the audio output.

In this assignment, **thread** and **task** are used and are generally equivalent. A **job** is a specific release of a thread (or task).

FORMATTING

➢ An arrow indicates that this is a step you must perform. It may have sub-steps.

0. A numbered item needs to be answered in the Google Form or the PDF report submitted to Moodle. Be sure to use the correct item number.

You are to work **individually** on this assignment. Submit your work and responses through Moodle as follows:

- Spreadsheet file:
    - Open this spreadsheet: https://docs.google.com/spreadsheets/d/1FTr0_fAnXu6IN9_vtBtdS2oZb88JbHYucKGmnBmb2VM/edit?usp=sharing
    - Copy the spreadsheet: **File -> Make a Copy**
    - Download your completed spreadsheet as a tsv (tab-separated values) text file: **File -> Download -> Tab-Separated Values (.tsv).**
    - Rename the .tsv file with your email username (part before @ncsu.edu): e.g. agdean-HW3.tsv
    - Upload this renamed .tsv file to Moodle.
- PDF report (based on the provided report template) with screenshots for questions 4, 31, 35, 46, and 57.
- Build log file (Objects/*.build_log.htm)

GETTING STARTED

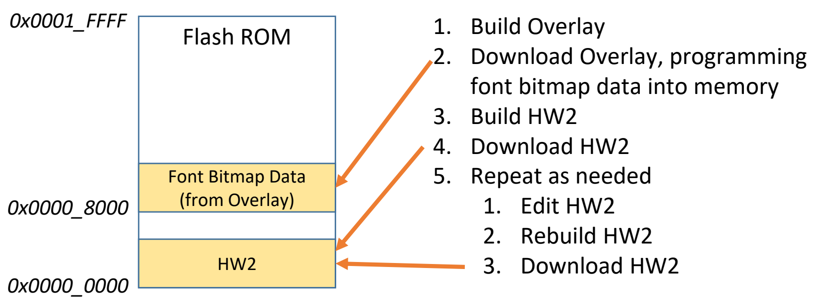
Set up the code as follows:

- Obtain the project code for the assignment from the remote repository on GitHub. Here are two ways to do this:
    - Download a ZIP archive
        - Use a web browser to go to the class GitHub repository.
        - Press the green <> Code button on that page and then select Download ZIP.
        - Unzip that zip archive into a directory on your PC with NO SPACES in the path.
        - Use File Explorer to navigate within the new directory to ESO-25/HW/HW3/
        - Open the HW3.uvprojx file to start MDK-ARM.
    - Use git
        - Copy your local repository directory ESO-25 (and all its contents, recursively) into a backup folder (e.g. ESO-25-Copy) to save your changes from previous work. Use File Explorer or a command-line tool (e.g. xcopy) to make the copy.
        - Run **git pull origin** in the command to update your local repository (ESO-25. Note that this will overwrite any changes you made to this copy.
        - Access the project code on your PC at ESO-25\HW\HW3\.
- Confirm that you can build the program without errors. Note that we are **not using the profiler** here, so you only need to build the program **once** after a code change, not three times.

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).

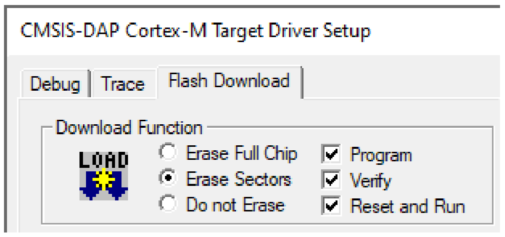
As seen in this diagram, in steps 1 and 2 you'll build and download the Overlay project to your



1. Build Overlay
2. Download Overlay, programming font bitmap data into memory
3. Build HW2
4. Download HW2
5. Repeat as needed
    1. Edit HW2
    2. Rebuild HW2
    3. Download HW2

KL25Z first to install the Overlay ROM data. In steps 3 and 4, you'll build your application project (HW3, not HW2) and download it *without erasing the overlay ROM data.* As you edit the HW3 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.



Download Function
- Erase Full Chip
- [✓ Erase Sectors]
- Do not Erase
- ✓ Program
- ✓ Verify
- ✓ Reset and Run

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.

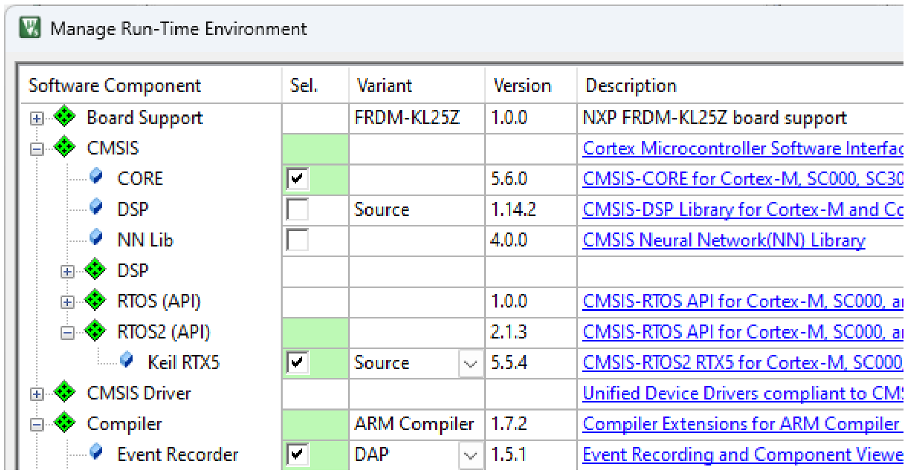
THREAD VISUALIZATION SUPPORT CODE

The thread visualizer code in ThreadVis\new_events.[ch] lets you use a logic analyzer to see when different threads are executing in a system built on the RTX5 kernel. We will modify the RTOS to run special *hook* functions when certain events occur: thread created, blocked, switched in (resumed), or preempted. These hook functions are defined in ThreadVis\new_events.[ch] and contain instrumentation code to assign and manipulate debug signals. We will make the RTOS call these hook functions by recompiling the RTOS with **event recorder** support enabled, described next.

COMPILING THE RTOS FROM SOURCE CODE

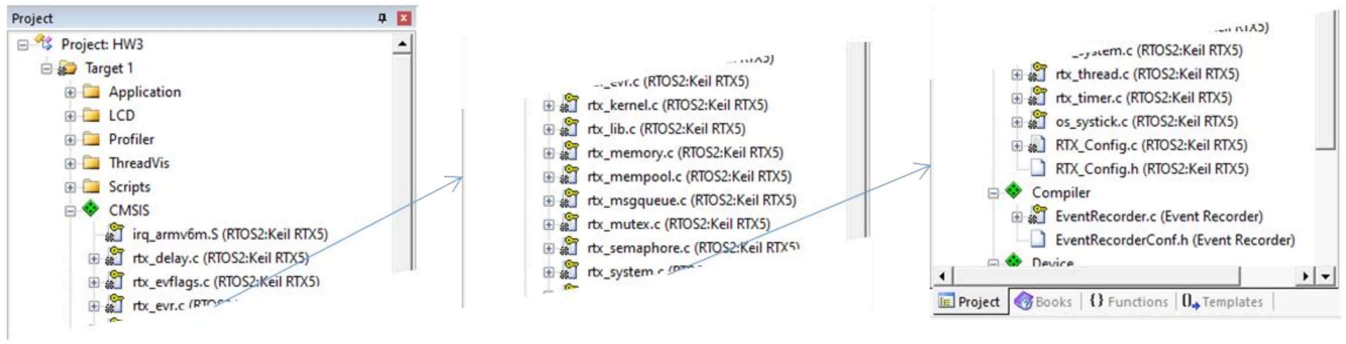
To use this code, we have changed the project to compile the RTX5 RTOS from source code (rather than link with its pre-compiled library), include the event recorder interface, and override certain default event recorder functions with our own. These steps have already been completed for you to create the HW3.uvprojx file. **You do not need to perform them**.

- To build the RTOS source code (instead of linking to the RTOS existing library), you would pick the Manage Run-Time Environment dialog and then CMSIS-> RTOS2 (API) -> Keil RTX5, select Source (rather than Library) and check the box.
- The debug signals rely on the Event Recorder support. To enable the Event Recorder, you would pick the Manage Run-Time Environment dialog and then Compiler->Event Recorder, select DAP and check the box.



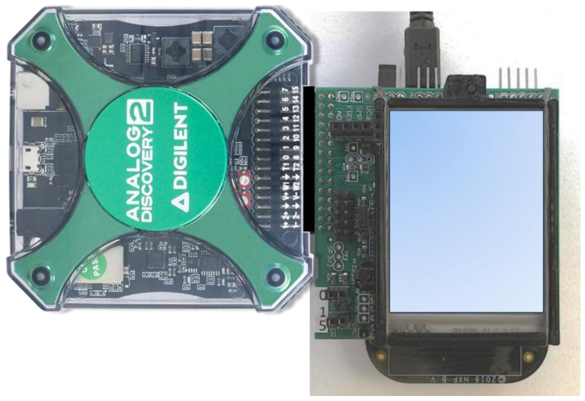
Software Component	Sel.	Variant	Version	Description
Board Support		FRDM-KL25Z	1.0.0	NXP FRDM-KL25Z board support
CMSIS	🟩			Cortex Microcontroller Software Interfac
CORE	✓		5.6.0	CMSIS-CORE for Cortex-M, SC000, SC30
DSP	☐	Source	1.14.2	CMSIS-DSP Library for Cortex-M and Cc
NN Lib	☐		4.0.0	CMSIS Neural Network(NN) Library
DSP				
RTOS (API)			1.0.0	CMSIS-RTOS API for Cortex-M, SC000, ai
RTOS2 (API)	🟩		2.1.3	CMSIS-RTOS API for Cortex-M, SC000, ai
Keil RTX5	✓	Source	5.5.4	CMSIS-RTOS2 RTX5 for Cortex-M, SC000
CMSIS Driver				Unified Device Drivers compliant to CM!
Compiler	🟩	ARM Compiler	1.7.2	Compiler Extensions for ARM Compiler
Event Recorder	✓	DAP	1.5.1	Event Recording and Component Viewe

The project pane should show the source files for the RTOS under CMSIS and those for the event recorder under Compiler.


Project
Project: HW3
Target 1
Application
LCD
Profiler
ThreadVis
Scripts
CMSIS
irq_armv6m.S (RTOS2:Keil RTX5)
rtx_delay.c (RTOS2:Keil RTX5)
rtx_evflags.c (RTOS2:Keil RTX5)
rtx_evr.c (RTOS2:Keil RTX5)
rtx_kernel.c (RTOS2:Keil RTX5)
rtx_lib.c (RTOS2:Keil RTX5)
rtx_memory.c (RTOS2:Keil RTX5)
rtx_mempool.c (RTOS2:Keil RTX5)
rtx_msgqueue.c (RTOS2:Keil RTX5)
rtx_mutex.c (RTOS2:Keil RTX5)
rtx_semaphore.c (RTOS2:Keil RTX5)
rtx_system.c (RTOS2:Keil RTX5)
rtx_thread.c (RTOS2:Keil RTX5)
rtx_timer.c (RTOS2:Keil RTX5)
os_systick.c (RTOS2:Keil RTX5)
RTX_Config.c (RTOS2:Keil RTX5)
RTX_Config.h (RTOS2:Keil RTX5)
Compiler
EventRecorder.c (Event Recorder)
EventRecorderConf.h (Event Recorder)
Device
Project
Books
Functions
Templates

METHODS FOR ANALYZING THREAD TIMING

We would like to understand the timing of the software. When does a thread run, and how long does it take? Is a thread preempted? When does an ISR run? How much time overhead does the RTOS take to switch between threads?



We can add *instrumentation* (measurement) code to the program to generate the debug signals which the logic analyzer and oscilloscope can then display. We will modify the code so that the MCU generates debug signals which indicate events such as when threads and handlers start or stop executing. Plugging the FRDM/Shield stack into the Analog Discovery tool connects these debug signals to the tool's DIO inputs, which are then read by the Waveforms program's logic analyzer and oscilloscope.

- ➢ If you are not familiar with the Analog Discovery device and the Waveforms software, please refer to this documentation on [Getting Started with Waveforms](), the [Oscilloscope]() ("scope") the [Logic Analyzer]() and [Using the Logic Analyzer]() (please ignore the Pattern Generator references in Sections 2.2 and 2.3).

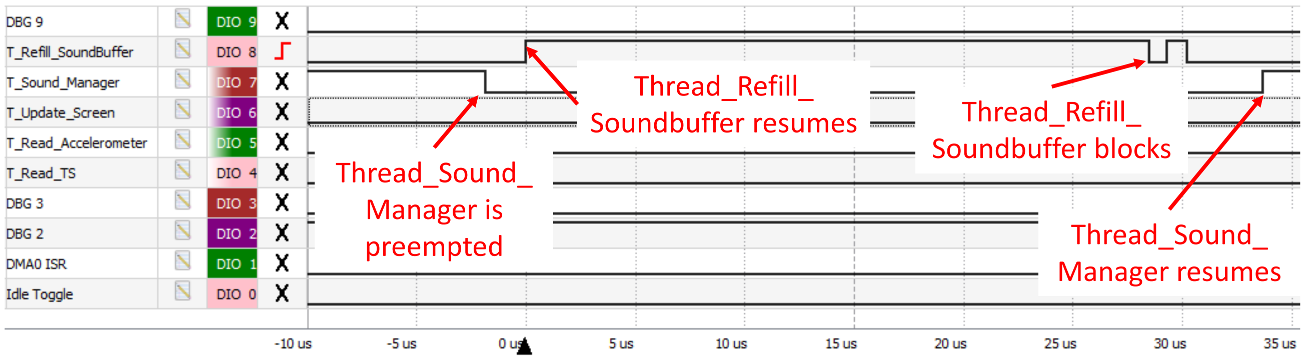


Figure 2. Example of thread visualizer signals. At -2 µs, Thread_Sound_Manager is preempted. At 0 µs, Thread_Refill_SoundBuffer resumes running (triggering the logic analyzer). At 28 µs, Thread_Refill_SoundBuffer blocks. At 34 µs, Thread_Sound_Manager resumes running.

The software in this exercise supports two types of debug signals: those automatically generated by the RTOS and those defined manually by the user.

- The RTOS can automatically generate thread visualization (TV) debug signals to indicate **thread events** such as **resuming running, being blocked,** or **being preempted**. These signals are **not** generated for interrupt and exception handlers. These RTOS-generated debug signals use code in ThreadVis\new_events.c and new_events.h, which work with debug.c and debug.h.
- You may have used user-defined debug signals before (e.g. ESA class). Defined in debug.c and debug.h., they must be manually added to the code. They do not directly show RTOS activity.

To summarize, the RTOS contains instrumentation code that will generate the debug signals for **thread events** in the system. We will need to add instrumentation code manually to monitor other processing (e.g. interrupt handlers). The table below shows the allocation of the debug signals and their use in this exercise; details are presented in the next two sub-sections (RTOS-Defined Debug Signals and User-Defined Debug Signals).

Num. Signal Name	**Descriptive Signal Name**	**osThreadNew Call Order**	**Use in this Code**	**AD2 DIO Signal**	**MCU Port Bit**
DBG_0	DBG_ISR_DMA	n/a, user defined	Student adds code to DMA0_IRQHandler to show when function starts and stops	DIO 0	D0
DBG_1	DBG_IDLE_LOOP	n/a, user defined	Student adds code to osRtxIdleThread to toggle each time through loop	DIO 1	D2
DBG_2	DBG_REFILL_ PENDING	n/a, user defined	Student adds code to show refill response time: ↑: DMA0_IRQHandler requests sound buffer refill ↓: Thread_Refill_Sound_Buffer has written first sample	DIO 2	D3
DBG_3	n/a	n/a, user defined	unused	DIO 3	D4
DBG_4	n/a	1st	TV: Thread_Read_TS (touchscreen)	DIO 4	B8
DBG_5	n/a	2nd	TV: Thread_Read_Accelerometer	DIO 5	B9
DBG_6	n/a	3rd	TV: Thread_Update_Screen	DIO 6	B10
DBG_7	n/a	4th	TV: Thread_Sound_Manager	DIO 7	B11
DBG_8	n/a	5th	TV: Thread_Refill_SoundBuffer	DIO 8	E2
DBG_9	n/a	6th	TV: osRtxIdleThread	DIO 9	E3
DBG_10	n/a	7th	TV: osRtxTimerThread	DIO 10	E1
DBG_11	n/a	8th	unused	DIO 11	E4

RTOS-DEFINED DEBUG SIGNALS (THREAD VISUALIZER)

As mentioned before, enabling the event recorder causes the RTOS to execute *hook* functions in new_events.c containing instrumentation code when certain events occur: thread created, blocked, switched in (resumed), or preempted.

Each thread's activity is represented by a debug output signal. As configured for this project, the debug code supports up to twelve debug signals. The kernel uses instrumentation code in new_events.c/h to automatically:

- Allocate a debug signal to a thread when it is created (via a call to osThreadNew). Signals are allocated to threads in the order that osThreadNew is called. The default signal allocation is shown in the table. Note that after all user threads have been created, the kernel calls osThreadNew twice to create the idle thread and the timer thread, which may result in up to two more TV signals being allocated.

- Set the bit to one when the thread **starts** or **resumes** executing.
- Clear the bit to zero when the thread is **preempted** and stops running.
- When the thread blocks and stops running, the bit behavior depends on the value of the symbol TV_UNIQUE_BLOCKED_SIGNAL, defined in new_events.h:
    - TV_UNIQUE_BLOCKED_SIGNAL != 0: Toggle the bit twice and then clear it to zero. This is the default behavior. Based on the sampling rate and display time scale of your logic analyzer, you many need to zoom in to distinguish between blocking and preemption. Note that this toggling takes about 1.7 µs, delaying other activities slightly.
    - TV_UNIQUE_BLOCKED_SIGNAL == 0: Clear the bit to zero. This signal is identical to the signal generated for preemption. It is used to measure execution times when there is no thread-to-thread preemption.

USER-DEFINED DEBUG SIGNALS

The fundamental macros and code for the debug signals are in debug.c and debug.h. To monitor something other than the thread events described above, follow these two steps.

First, select an available debug signal and give it a better name:

- Select an available user debug signal for the routine. Review debug.h to see DBG_0, DBG_1, etc. Select the first free signal (e.g. DBG_0). Note that when using the new_events.c/h, the RTOS skips over the first TV_NUM_USER_DBG_SIGNALS signals (defined in new_events.h, currently 4).
- To make the source code less confusing, it is good practice to define a meaningful name for the debug signal. In debug.h go to the section after // Define meaningful names…. and add #define DBG_RED_ALERT_ISR (DBG_0).

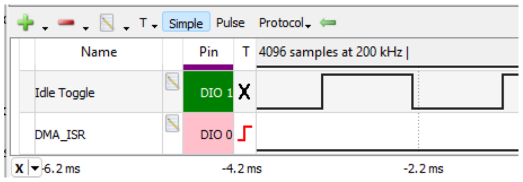
Second, add code to the event to monitor. There are several possible approaches, but here are two:

- **Show the time between a start event and an end event**. For example, when something is requested and then later serviced, or a block of code begins and then later completes (though it may be preempted in between):
    - Add code at the beginning of the processing/handler/ISR or at the request to set it to 1 using the macro DEBUG_START(DBG_RED_ALERT_ISR).
    - Add code at the end of the processing/handler/ISR or after the request is serviced to clear it to 0 using the macro DEBUG_STOP(DBG_RED_ALERT_ISR).
    - Make sure that each source file you modify this way #includes “debug.h”
- **Provide multiple progress or “heartbeat” notification events.** These events shows that certain code is really executing. For example, there is some delay between the RTOS announcing it will resume a thread, and the thread actually resuming. The RTOS events monitored above (thread resumed, blocked, preempted) don’t show the time overhead taken by the RTOS, so they don’t tell the whole story of when code actually starts or stops running.
    - Add code in the loop body to toggle (invert) output signal using the macro DEBUG_TOGGLE(DBG_LOOP37_ACTIVE). Each signal change shows when the loop ran.
    - Make sure that each source file you modify this way #includes “debug.h”

Now you will use these two approaches to further instrument the program.

➢ Add instrumentation code as described above to indicate DMA0_IRQHandler execution with a new signal named **DBG_ISR_DMA** (define it to use DBG_0).

- Add instrumentation code as described above to see when the RTOS idle thread is actually running. That thread (RTX_Config.c: osRtxIdleThread()) has an infinite loop which runs if there are no other threads ready to run. Add code to the loop osRtxIdleThread() to toggle (invert) output signal **DBG_IDLE_LOOP** (define it to use DBG_1) each time the loop executes.
    - Note that this signal will toggle at a high frequency (e.g. f_{signal}). If the scope/logic analyzer samples at less than $2*f_{signal}$ the signal will be aliased, corrupting frequency information. The signal will appear to be changing at a much lower frequency than reality, or even not changing (i.e. 0 Hz). Be sure to check the sampling rate which is shown in the upper left corner of the scope or analyzer window. In this figure, the sampling rate is shown to be 200 kHz (one sample every 5 us). The Idle Toggle trace seems to show a signal which is toggling every 1 ms. This is an example of aliasing; the actual frequency of that signal is much higher. Adjust the analyzer's time base to increase the sampling frequency enough to eliminate aliasing.



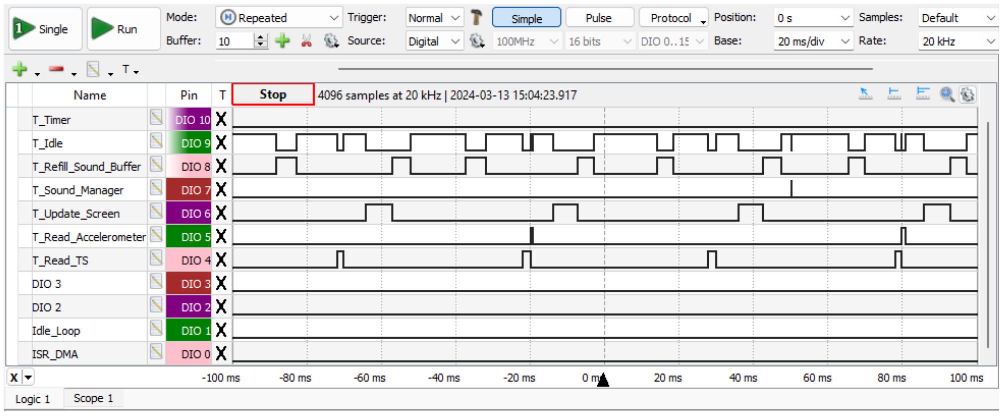
4096 samples at 200 kHz |

Name	Pin
Idle Toggle	DIO 1
DMA_ISR	DIO 0

BASIC EXPERIMENTAL TIMING ANALYSIS

VIEWING DEBUG SIGNALS WITH LOGIC ANALYZER

- Connect the Analog Discovery device to the FRDM/shield stack. Start the Waveforms program and select the logic analyzer program.
- Build, download and run the program. Run the logic analyzer without setting up any triggering. Adjust the sampling time base ("base") to about 20 ms/division. You will see something similar to this, but with many differences.



Mode: Repeated Trigger: Normal Position: 0 s Samples: Default
Buffer: 10 Source: Digital 100MHz 16 bits DIO 0..15 Base: 20 ms/div Rate: 20 kHz
4096 samples at 20 kHz | 2024-03-13 15:04:23.917

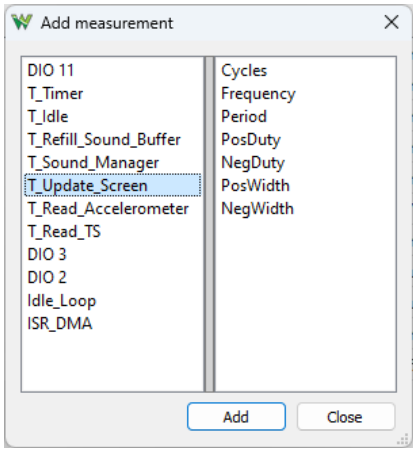
Name	Pin
T_Timer	DIO 10
T_Idle	DIO 9
T_Refill_Sound_Buffer	DIO 8
T_Sound_Manager	DIO 7
T_Update_Screen	DIO 6
T_Read_Accelerometer	DIO 5
T_Read_TS	DIO 4
DIO 3	DIO 3
DIO 2	DIO 2
Idle_Loop	DIO 1
ISR_DMA	DIO 0

- In order to simplify data analysis (and grading), rename the used DIO signals to something useful by double-clicking on the signal in the Name column and changing the name. The figure above shows an example.

AUTOMATIC SIGNAL STATISTICS FROM LOGIC ANALYZER

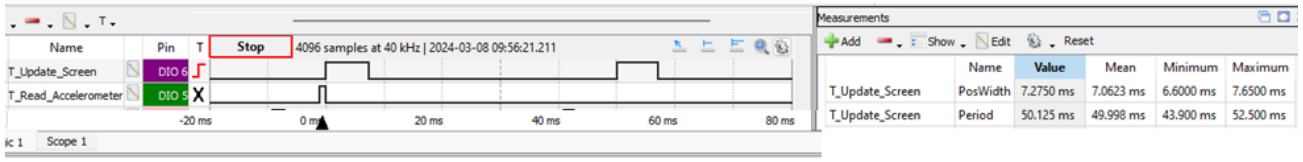
The Waveforms scope tool can measure characteristics of input signals, such as the positive width of a DIO signal. It can also compute that characteristic's statistics (e.g minimum, maximum, average) over one or multiple data acquisitions. We will use this to gather task execution time statistics. For example, let's see how Thread_Update_Screen behaves.

- In the Logic 1 View menu items, select Digital Measurements. This will enable (or disable) a Measurements window.
- Click the + Add button to get the Add measurement dialog box.



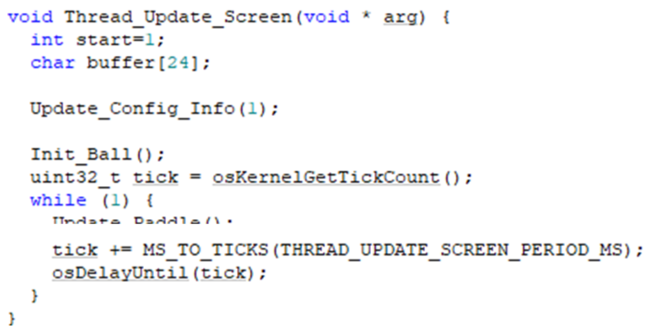
DIO 11	Cycles
T_Timer	Frequency
T_Idle	Period
T_Refill_Sound_Buffer	PosDuty
T_Sound_Manager	NegDuty
T_Update_Screen	PosWidth
T_Read_Accelerometer	NegWidth
T_Read_TS	
DIO 3	
DIO 2	
Idle_Loop	
ISR_DMA	

- In the left pane, select the digital signal to measure (e.g. T_Update_Screen, which we just renamed from DIO 6).
- In the right pane, select PosWidth (width of positive pulse) and click Add.
- In the right pane, select Period and click Add.
- Click Close.
- Click the Show button and check the Mean, Minimum and Maximum entries.
- Click the gear icon and check the Multiple Acquisitions box. This will accumulate the statistics over multiple acquisistions. If you need to reset the statistics, you'll press the Reset button.
- Make sure the HW3 program is running.
- Set the time per division to 10 ms for this example. Start the logic analyzer (press Run). You should see something similar to this, but waveforms and times will vary:



	Name	Value	Mean	Minimum	Maximum
T_Update_Screen	PosWidth	7.2750 ms	7.0623 ms	6.6000 ms	7.6500 ms
T_Update_Screen	Period	50.125 ms	49.998 ms	43.900 ms	52.500 ms

- As more data is captured, some of the statistics (e.g. Minimum and Maximum) should stabilize. Others may still change slowly (e.g. Mean).
- Check to see if the plots and measurements/statistics are reasonable and don't contradict each other. In some cases, you can also cross-check with the code.
  - **Computation time**: The signal plot shows T_Update_Screen takes somewhere between 5 and 10 ms each of the two times it ran. Note that the measurements and statistics shown support this, although the plot has much less timing resolution.
  - **Execution Period**: The signal plot also shows T_Update_Screen ran at about 0 ms and 50 ms, so the period should be 50 ms. The measurements support this. They also show some extreme cases not captured in this plot. For example, the minimum period of 43.9 ms would definitely be visible on this plot, but the current plot shows a period of about 50 ms. Looking at the source code for the thread, we see the main loop calls osDelayUntil() in order to try to run every MS_TO_TICKS(THREAD_UPDATE_ SCREEN_PERIOD_MS). Evaluating these #defines should give us a period of 50 ms.



```
void Thread_Update_Screen(void * arg) {
  int start=1;
  char buffer[24];

  Update_Config_Info(1);

  Init_Ball();
  uint32_t tick = osKernelGetTickCount();
  while (1) {
    Update_Paddle();

    tick += MS_TO_TICKS(THREAD_UPDATE_SCREEN_PERIOD_MS);
    osDelayUntil(tick);
  }
}
```

  - Note that the measurements are made based on each burst of acquired data. If we zoom in to 2 ms/division, the data burst will be so short that it won't contain a full signal period. The period can't be measured so the Period Value cell will be blanked out. If we zoom in even more to 200 us/division, the burst won't contain a full pulse, so the PosWidth will be blanked out. In these cases, the statistics are not being updated, so do not trust them. Zoom out, and reset the statistics as needed.

MEASURING TO CREATE THE PERIODIC TASK MODEL

We can now gather the task and handler computation times to build a periodic task model of the system's software timing.

- To simplify the timing analysis, set all threads to have the same priority. This way, no threads can interfere with our timing measurements. Make sure that each **THREAD_..._PRIO** declaration in **threads.h** and **sound.h** is **osPriorityNormal**. However, interrupt and exception handlers can still interfere.

- In new_events.h, ensure that TV_UNIQUE_BLOCKED_SIGNAL is defined as **0**. A thread blocking will be indicated with a single falling edge on its debug signal, simplifying timing analysis.
- Rebuild and download the program.

MEASURING INSTRUMENTED THREADS AND HANDLERS

1. Use the logic analyzer to complete the following table with the range of observed timing information for each thread or handler, except the Idle thread. Trigger the analyzer on the rising edge of the given thread/handler's debug signal.
    - You may need to adjust the sampling rate to accommodate short or long functions, and to see an entire period (two releases for the task).
    - Note that the period for **Thread_Sound_Manager** is reduced with each paddle hit or miss toward a minimum value. To determine that minimum value, check that function's source code (starting with **osDelayUntil()** and working backwards through the data). Enter that minimum value in the Expected column.

		Execution Duration C_i			**Execution Period T_i**			
		Measured				**Measured**		
Thread or Handler	**Symbols for thread/handler name abbreviation, computation time, execution period**	**Min.**	**Avg.**	**Max.**	**Expected**	**Min.**	**Avg.**	**Max.**
DMA0_IRQHandler	***I_DMA***, C_{I_DMA}, T_{I_DMA}							
Read Touchscreen	***T_RTS***, C_{T_RTS}, T_{T_RTS}							
Update Screen	***T_US***, C_{T_US}, T_{T_US}							
Refill SoundBuffer	***T_RSB***, C_{T_RSB}, T_{T_RSB}							
Sound Manager	***T_SM***, C_{T_SM}, T_{T_SM}							
Read Accelerometer	***T_RA***, C_{T_RA}, T_{T_RA}							

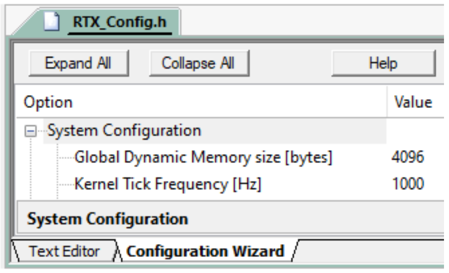
- In new_events.h, ensure that TV_UNIQUE_BLOCKED_SIGNAL is defined as **1**. A thread blocking will be indicated with a falling edge and a pulse, distinguishing it from preemption.
- Rebuild and download the program.

2. You should see only one thread being preempted. Which thread is preempted, and why is it preempted?
3. Which thread has the **largest absolute** execution time variation (Max C_i - Min C_i), and how large is that variation? What does the thread do differently between the minimum and maximum cases?

MEASURING WHAT IS NOT INSTRUMENTED

SYSTICK_IRQHANDLER

The RTOS uses the SysTick timer peripheral and its handler (SysTick_IRQHandler) to keep track of time. We expect the handler to run at a frequency set in RTX_Config.h (System Configuration). We need to add its timing information to our periodic task timing model to improve its accuracy.



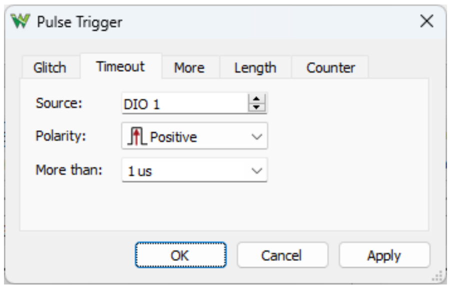
Option	Value
System Configuration	
Global Dynamic Memory size [bytes]	4096
Kernel Tick Frequency [Hz]	1000

System Configuration

The thread visualizer doesn't generate a debug signal for the SysTick interrupt handler. Because we are already recompiling the RTOS from source code, we could search to find the SysTick IRQ handler and manually instrument it. However, we'll use a different approach: looking at times when the idle thread should be toggling its user debug signal but is not.

We'll use the logic analyzer's **Pulse** (instead of Simple) trigger with a **Timeout** to trigger when the idle thread signal has a positive pulse longer than 1 µs.

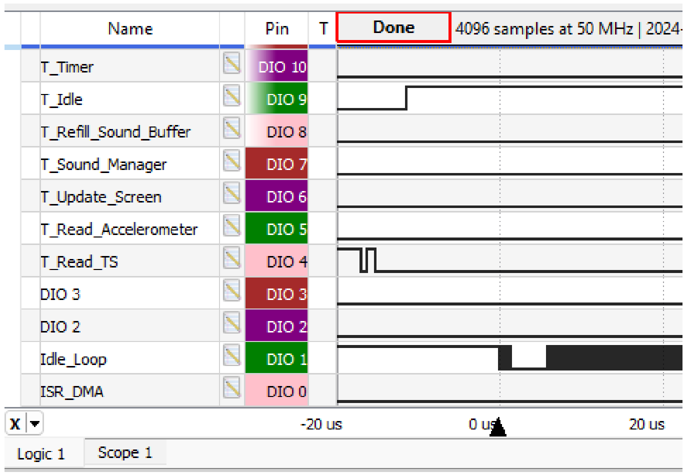
- ➢ Specify the signal source (the idle user debug signal), polarity (either positive or negative will work), and time More than 1 us. The analyzer will trigger when it sees a pulse of the given polarity from the source lasting long enough.
- ➢ Try using Single capture button Single (not Run) to get more time to examine the data and consider what is happening in the system.



Source: DIO 1
Polarity: Positive
More than: 1 us

This type of triggering may sometimes result in false positive cases; you will need to exclude them. In these cases, the CPU is executing another handler (e.g. the DMA ISR) or thread (anything except for the Idle thread), which usually shown by one of their debug signals being 1.

For example, these traces shows that the Read Touchscreen thread (DIO 4) was running until it blocked (a little after –20 us), the RTOS decided to resume the idle thread (DIO 9) at roughly –10 us, and DIO 1 shows that at 0 us the idle thread actually resumed running. This time is precise because that we are triggering on DIO 1. DIO 1 then shows that a little after 0 us the idle thread stopped running, but resumed before 10 us. What code was running in that last time gap?



Name	Pin
T_Timer	DIO 10
T_Idle	DIO 9
T_Refill_Sound_Buffer	DIO 8
T_Sound_Manager	DIO 7
T_Update_Screen	DIO 6
T_Read_Accelerometer	DIO 5
T_Read_TS	DIO 4
DIO 3	DIO 3
DIO 2	DIO 2
Idle_Loop	DIO 1
ISR_DMA	DIO 0

- It wasn't another thread, since the RTOS didn't decide to preempt the idle thread (DIO 9 is still 1).
- It wasn't the DMA ISR, because DIO 0 remains 0.
- It must have been preempted by an ISR/handler which is not instrumented with a debug signal. In our case it is the SysTick handler.

Trying to zoom out in time to see longer durations can lead to a sampling/aliasing problem. The AD2 and 3 can sample at f_{sample} = (100/n) MHz, with n being an integer. The logic analyzer reduces its sampling rate f_{sample} so that its buffer (4096 samples for AD2, 16384 samples for AD3) can hold all the samples for the acquisition. Capturing 1 ms of data limits the the sampling rate to at most 4096 samples/1 ms = 4.096 MHz for the AD2.

The idle toggle signal is an 8 MHz square wave (f_{toggle}). If the ratio f_{toggle} /f_{sample} is an integer, then aliasing will make the toggling signal look like a constant one or zero, with a frequency of 0 Hz. If the ratio is close to an integer, then aliasing will make the signal seem to change with the wrong frequency, interfering with your timing measurements.

You can eliminate this aliasing by lowering the toggle signal frequency f_{toggle} to meet both of these conditions:

- $f_{sample}/2 > f_{toggle}$, and
- f_{sample}/ f_{toggle} **is not** an integer and not near an integer.

- ➢ Change the duration of the idle thread's loop, which will change the toggle signal frequency. For example, adding this code to idle thread's loop should slow it down from 8 MHz to about 0.827 MHz.

```
        for (volatile int i=0; i<2; i++);
```

This change should clean up the idle toggle signal to make the execution period much more regular and easier to analyze, giving more consistent period measurements.

- ➢ Use this approach to measure the timing characteristics of the SysTick handler for the next three questions.

4. Capture a logic analyzer screenshot and put it in your report.
5. What is the **shortest** time C_i for which the idle signal **stops** toggling?
6. Measure the parameters specified in the table below. The execution period T_i of the toggling gap should be close to the inverse of the SysTick frequency set in RTX_Config.h.

		Measured Execution Duration C_i			**Measured Execution Period T_i**		
Thread or Handler	**Symbols**	**Min.**	**Avg.**	**Max.**	**Min.**	**Avg.**	**Max.**
SysTick_IRQHandler	***I_ST***, C_{I_ST}, T_{I_ST}						

ECE 561 ONLY: RTOS AND DEBUG SIGNAL OVERHEAD

We can derive other useful information from this type of logic analyzer data:

- It takes a few microseconds to generate the signal (three transitions) showing that Thread_Read_TS blocked.
- It takes a few microseconds from the RTOS indicating that Thread_Read_TS blocked until it indicates that it will resume osRtxIdleThread (T_Idle).
- It takes over ten microseconds from when the RTOS indicates that it will resume osRtxIdleThread until the thread actually starts running (indicated by the toggling Idle Loop signal).

For each of the following five questions, run at least ten more tests to determine the minimum and maximum observed values. It may be helpful to change the trigger conditions as needed to get the data.

7. **Blocked Signal Duration**: What are the minimum and maximum durations you observed for a **blocked** signal transition (falling edge to falling edge)?
8. **Resume Delay:** What are the minimum and maximum times you observed between a thread **resume** transition for T_Idle and the Idle Loop signal **toggling** again? Hint: Trigger on T_Idle rising edge.
9. **Preemption Delay:** What are the minimum and maximum times you observed between the Idle Loop signal **stopping toggling** and a thread **preemption** transition for T_Idle? Omit any cases with ISR_DMA active between the end of toggling and the T_idle preemption signal.
10. **Preempted to Resume Delay:** What are the minimum and maximum times you observed between one thread's **preempted** transition and another thread's **resume** transition?
11. **Blocked to Resume Delay:** What are the minimum and maximum times you observed between the end of one thread's **blocked** transition and another thread's **resume** transition?

INTENTIONAL OMISSIONS

To simplify our analysis, our periodic task timing model ignores most of the RTOS, but we do consider SysTick_IRQHandler. This makes the model wrong, but it may still be useful. Here is some of what we leave out:

- RTOS and debug signal overhead times you just measured above! We leave them out of the model for simplicity, but now we know to distrust the model if these ignored times aren't tiny compared with the computation times or response times.
- The RTOS scheduler uses two additional types of exceptions and their handlers (SVC and PendSV) to do its work. We don't measure them directly here, but they do execute and take time.
- Other time taken by the operating system to do its work.
- Time taken to generate the debug signals.
- Time to execute osRtxTimerThread. This thread supports virtual timers by executing software-timer-driven callback functions managed with osTimerNew, osTimerStart, and osTimerDelete. These are not used in this application, so we can safely ignore this thread. (To make sure, you could load the program into the debugger, put a breakpoint in the event handling loop of osRtxTimerThread (in rtx_timer.c) and start it running).

EVALUATING THE WORKLOAD UTILIZATION

12. Enter the maximum expected CPU time utilization for each task (except the idle task) and handler as decimal value with three digits after the decimal point (e.g. ½ would be 0.500).

Thread or IRQ Handler	Condition	Maximum CPU Time Utilization (C_i/T_i)
SysTick_IRQHandler		
DMA0_IRQHandler		
Read Touchscreen	Not Touched	
Read Touchscreen	Touched	
Update Screen		
Refill SoundBuffer		
Sound Manager	Max. Period	
Sound Manager	Min. Period	
Read Accelerometer		

13. What is the system's expected maximum CPU time utilization if the **touchscreen is not pressed** and the **sound manager is running at the maximum period**?
14. What is the system's expected maximum CPU time utilization if the **touchscreen is pressed** and the **sound manager is running at the minimum period**?

EVALUATING THE SOUND BUFFER REFILL LATENCY

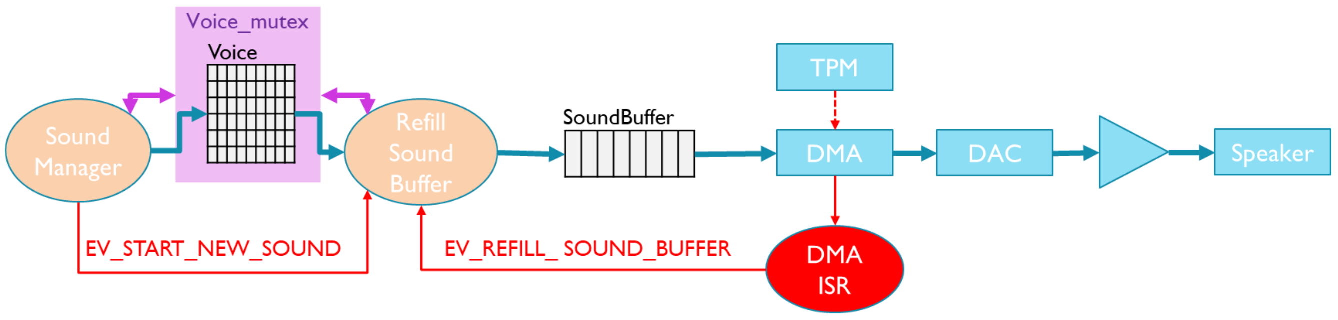
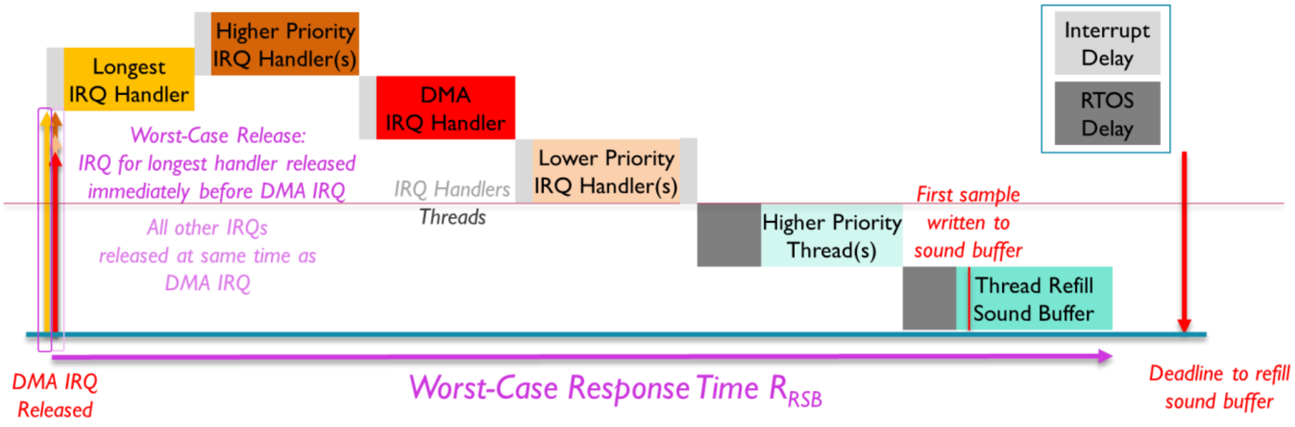


Figure 3. Sound generation architecture with default single buffer.

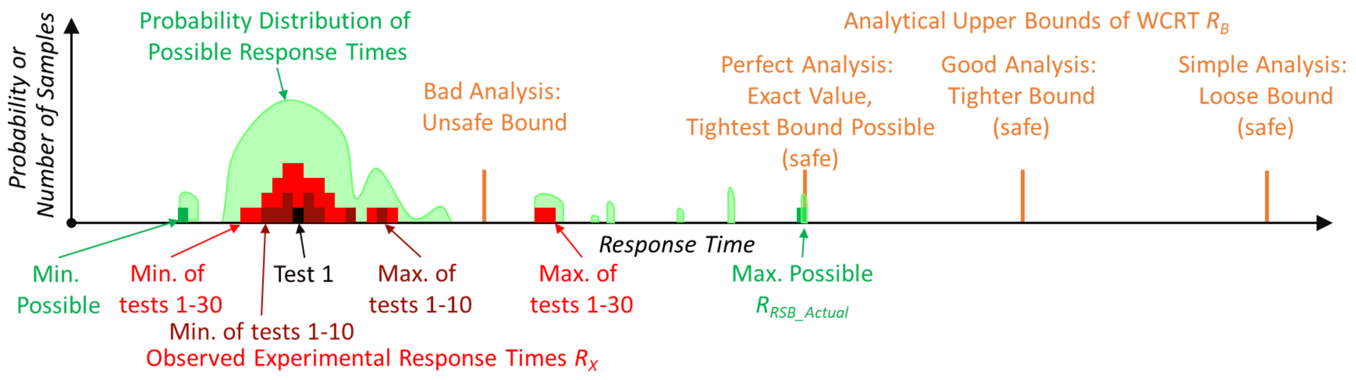
A time-critical operation in the shield code is refilling the sound buffer with audio samples before the DMA reads the next sample. By default, a single buffer is used. The sound buffer is refilled using this sequence of events:

- DMA peripheral transfers last sample from Sound Buffer, triggering IRQ.
- DMA IRQ handler runs, setting an event flag for Thread_Refill_Sound_Buffer (T_RSB).
- RTOS sees the set event flag and unblocks Thread_Refill_Sound_Buffer (T_RSB), allowing it to run eventually (once it is the highest priority ready thread).


Longest IRQ Handler
Higher Priority IRQ Handler(s)
DMA IRQ Handler
Lower Priority IRQ Handler(s)
Interrupt Delay
RTOS Delay
Worst-Case Release: IRQ for longest handler released immediately before DMA IRQ
All other IRQs released at same time as DMA IRQ
IRQ Handlers
Threads
Higher Priority Thread(s)
First sample written to sound buffer
Thread Refill Sound Buffer
DMA IRQ Released
Worst-Case Response Time R_{RSB}
Deadline to refill sound buffer

Because the playback rate is one sample per 50 µs (assuming AUDIO_SAMPLE_FREQ is defined as 20000 in sound.h), the first sample needs to be in the buffer within 50 µs of the previous sample being used. This is our deadline.

ANALYTICALLY ESTIMATING UPPER BOUND OF REFILL LATENCY



Will the system always be able to refill the sound buffer before that deadline of 50 µs? This depends on the program, input data, sequencing of events, interrupts, etc. The actual maximum response time (call it $\boldsymbol{R_{RSB\_Actual}}$) is unknown.

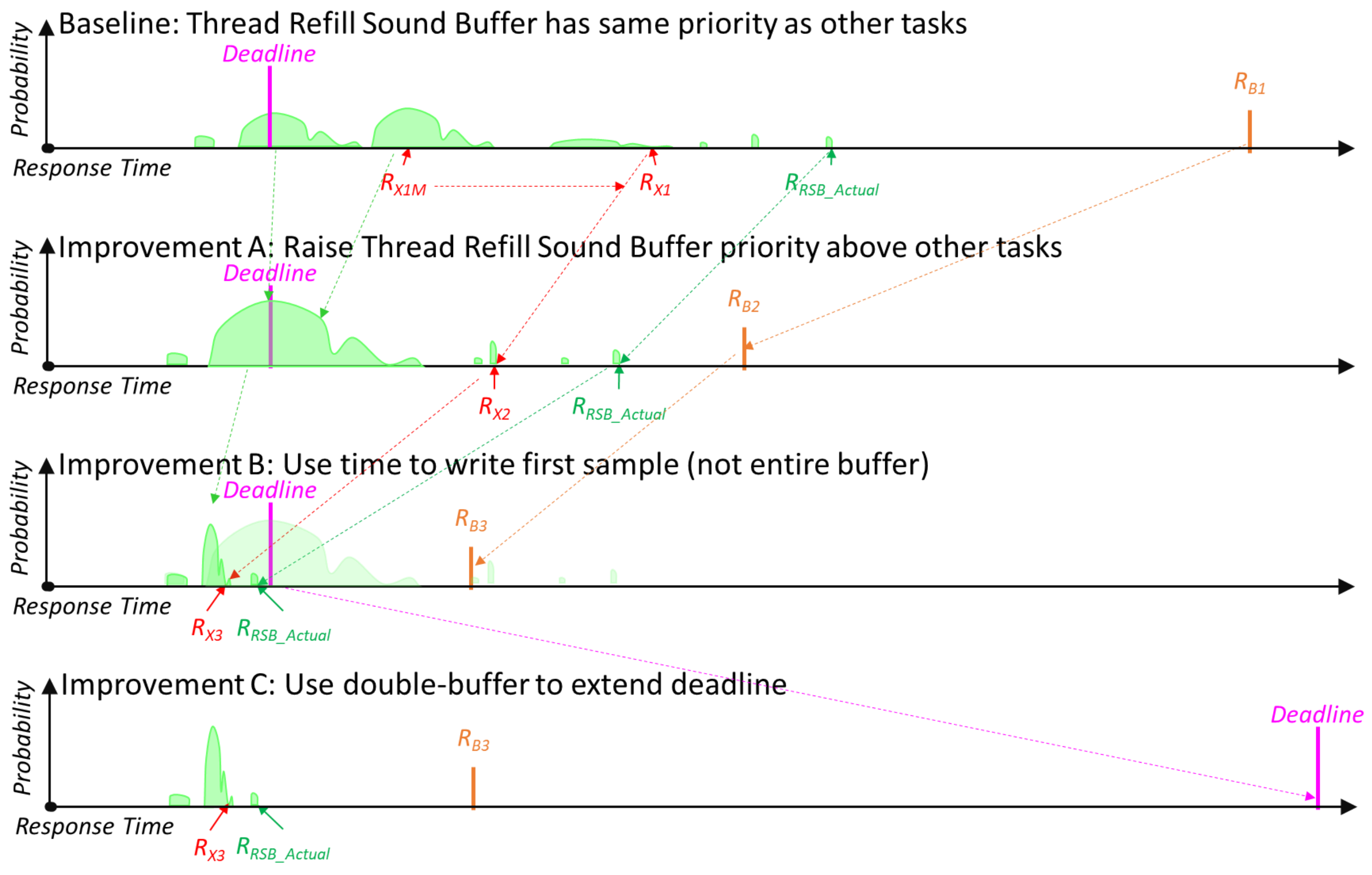
We can calculate an upper bound on the response time $\boldsymbol{R_{RSB\_UB}}$ for refilling the sound buffer. An upper bound will always be at least as large as the actual value: $\boldsymbol{R_{RSB\_UB}} >= \boldsymbol{R_{RSB\_Actual}}$. So if the upper bound is less than the deadline, then so is the actual value, and the deadline will always be met. For simplicity, we will omit RSB from response time subscripts, since all of our response time analysis here targets Thread_Refill_Sound_Buffer.

We will create the upper bound by applying the numerical response time analysis method which considers timing interference from other processing. Note that the interrupt/exception handlers all have higher priorities than the threads and can always preempt them.

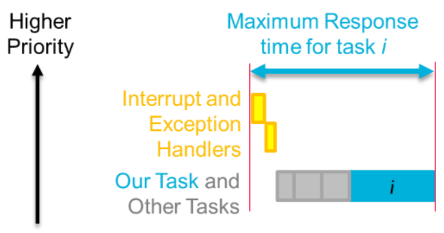
We start with a simple analysis to get R_{B1}, an easy but inaccurate (**loose**) bound of the WCRT of refilling the sound buffer. Then we will refine the analysis to move the bound closer to the actual WCRT (make it **tighter**). We raise the priority of T_RSB and calculate bound R_{B2}, which should be closer to the actual WCRT. Then we refine our analysis to get R_{B3} which considers that only the first buffer sample needs to be updated before the deadline, so we can ignore most of C_{T_RSB}.

		T_RSB Priority	**Who Can Preempt or Block T_RSB**	**Processing Required before Deadline**	**Estimated Max. Time for Processing**
Bound 1	R_{B1}	osPriorityNormal	All other application threads and handlers	Refill entire sound buffer	C_{T_RSB}
Bound 2	R_{B2}	osPriorityAboveNormal	All handlers	Refill entire sound buffer	C_{T_RSB}
Bound 3	R_{B3}	osPriorityAboveNormal	All handlers	Refill first sample of sound buffer	C_{T_RSB}/(Buffer size in Samples)

The available timing margin is the deadline minus the bound. A negative timing margin means that in the worst case the deadline will be missed (and maybe in other cases too).


Baseline: Thread Refill Sound Buffer has same priority as other tasks
Probability
Response Time
Deadline
R_{B1}
R_{X1M}
R_{X1}
R_{RSB_Actual}
Improvement A: Raise Thread Refill Sound Buffer priority above other tasks
Deadline
R_{B2}
R_{X2}
R_{RSB_Actual}
Improvement B: Use time to write first sample (not entire buffer)
Deadline
R_{B3}
R_{X3}
R_{RSB_Actual}
Improvement C: Use double-buffer to extend deadline
R_{B3}
Deadline
R_{X3}
R_{RSB_Actual}

BOUND 1: BASELINE CODE



Higher Priority
Maximum Response time for task *i*
Interrupt and Exception Handlers
Our Task and Other Tasks
i

$$R_i^0 = \boxed{C_i} + \boxed{\sum_{j \in handlers} C_j} + \boxed{\sum_{j \in tasks \cap j \neq i} C_j}$$

$$R_i^{n+1} = \boxed{C_i} + \boxed{\sum_{j \in handlers} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j} + \boxed{\sum_{j \in tasks \cap j \neq i} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j}$$

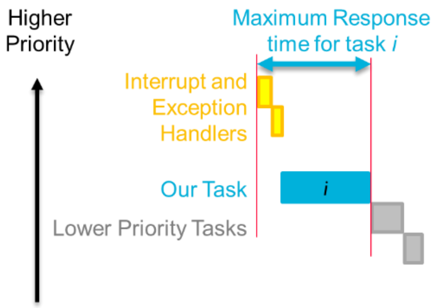
Initially all application threads have the same priority (osPriorityNormal). The OS thread osRtxIdle has a lower priority than the application threads and can always be preempted by an application thread. The numerical analysis must assume that T_RSB can be delayed by all threads at osPriorityNormal or higher.

- ➢ Use the equations above to calculate the worst-case response time for Thread_Refill_Sound_Buffer under these conditions. Calculate R_i^0, R_i^1, R_i^2, ... until the value stops changing. We will call the final value R_{B1}.

15. What is the value of R_{B1}?
16. What is the timing margin (50 µs minus R_{B1})?

BOUND 2: IMPROVEMENT A – PRIORITIZE REFILL THREAD

Let's predict what happens if we raise the priority of T_RSB above all other threads. Now only the handlers should be able to delay the thread. Calculate R_{B2}



Higher Priority
Maximum Response time for task *i*
Interrupt and Exception Handlers
Our Task
Lower Priority Tasks
i

$$R_i^0 = \boxed{C_i} + \boxed{\sum_{j \in handlers} C_j}$$

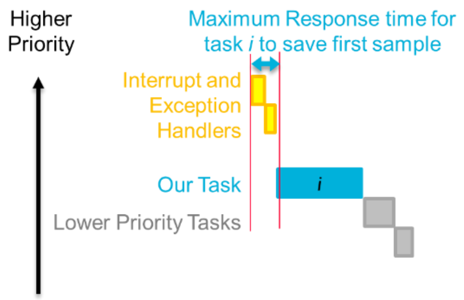
$$R_i^{n+1} = \boxed{C_i} + \boxed{\sum_{j \in handlers} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j}$$

- ➢ Use the equations above to calculate the worst-case response time for Thread_Refill_Sound_Buffer under these new conditions. Calculate R_i^0, R_i^1, R_i^2, ... until the value stops changing. Call the final value R_{B2}.

17. What is the value of R_{B2}?
18. What is the timing margin?

BOUND 3: IMPROVEMENT B – DEADLINE IS FIRST SAMPLE REFILL

Finally, let's estimate the worst-case response time for Thread_Refill_Sound_Buffer to update the **first sample** in the buffer, not all NUM_SAMPLES_PER_SOUND_BUFFER samples. Our standard numerical analysis bounds the time to update all of the samples



Higher Priority
Maximum Response time for task *i* to save first sample
Interrupt and Exception Handlers
Our Task
Lower Priority Tasks
i

$$R_i^0 = \boxed{\frac{C_i}{512}} + \boxed{\sum_{j \in handlers} C_j}$$

$$R_i^{n+1} = \boxed{\frac{C_i}{512}} + \boxed{\sum_{j \in handlers} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j}$$

(which is an over-estimate) because it uses the entire time $C_{Refill\_Sound\_Buffer}$ (but only part of that time is needed).

The equations above show an example with a sound buffer size of 512 samples, which may or may not match the value NUM_SAMPLES_PER_SOUND_BUFFER defined in your code.

Later in this exercise we will measure how long it takes for T_RSB to update the first sample. For now, let's just estimate. We assume that thread doesn't have any set-up time, and its $C_{Refill\_Sound\_Buffer}$, is spread evenly across the buffer's NUM_SAMPLES_PER_SOUND_BUFFER. Now we can estimate that the first sample will be written after the thread executes for a time $C_{Refill\_Sound\_Buffer}$/NUM_SAMPLES_PER_SOUND_BUFFER. We can update our response time analysis now, using $C_{Refill\_Sound\_Buffer}$/NUM_SAMPLES_PER_SOUND_BUFFER instead of $C_{Refill\_Sound\_Buffer}$ for C_i.

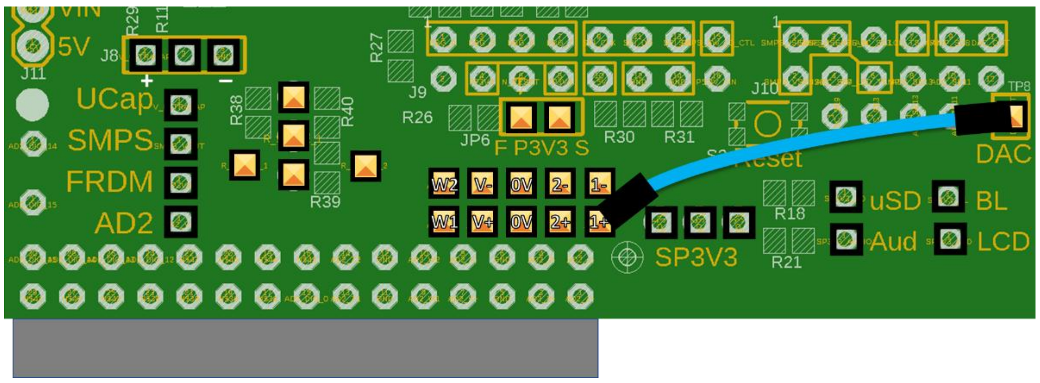
- Use the equations above to calculate the worst-case response time for Thread_Refill_Sound_Buffer under these new conditions. Calculate R_i^0, R_i^1, R_i^2, ... until the value stops changing. Call the final value R_{B3}.

19. What is the value of R_{B3}?
20. What is the timing margin?

EXPERIMENTALLY MEASURING REFILL LATENCY

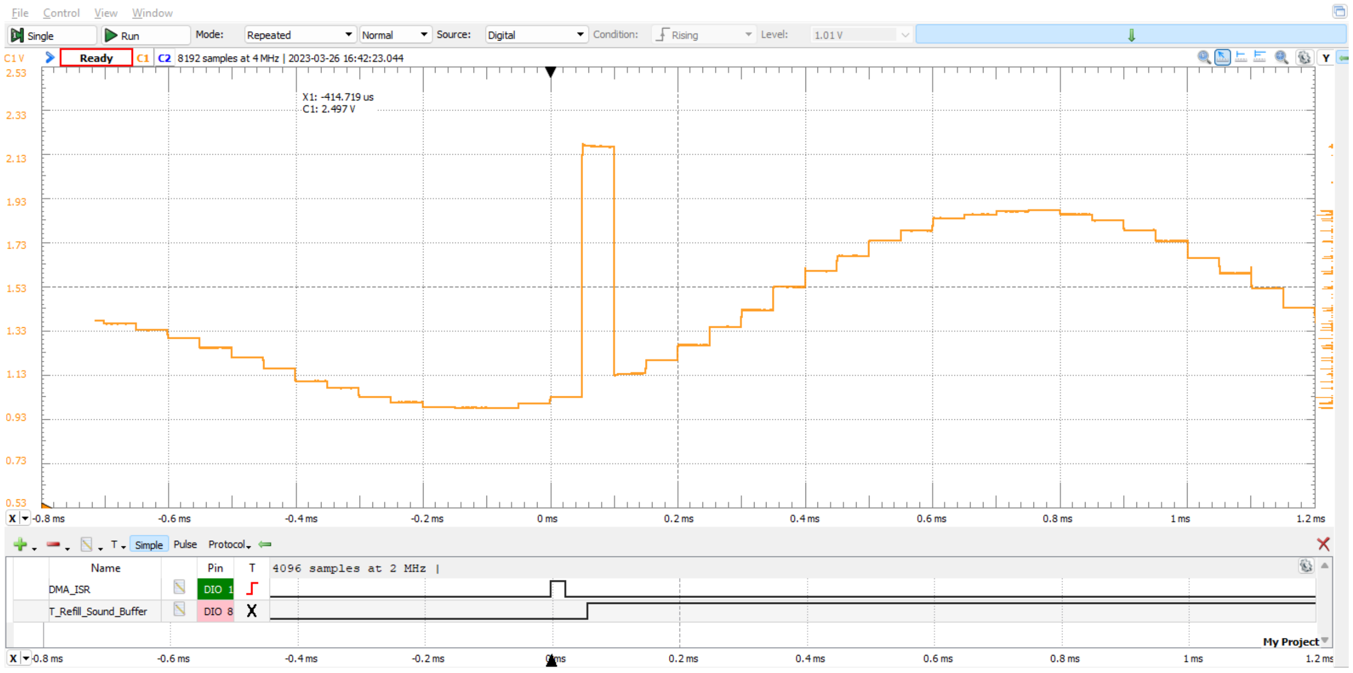
Let's see (and hear) how the audio generation components really behave. When the audio is not muted you'll hear the flawed output, with lots of buzzing and clicks which come from the sound buffer not being refilled in time. Let's look at the analog output of the DAC and directly see the impact of the delayed sound buffer refill. Use the Analog Discovery 2 to monitor the analog output (as well as the digital outputs).

- Connect Channel 1+ to the DAC output. Use a socket-socket jumper wire from as shown:



- Configure the Waveforms program to provide a mixed-signal display. On the Welcome tab, select **Scope**. In the View menu, select Digital (at the bottom of the list).
- In the newly-created logic analyzer window, press the green + to add the debug signals for the DMA IRQ Handler (DIO 0) and the TV signal for Thread_Refill_Sound_Buffer (DIO 8). You may also add the other debug signals as well.
- Trigger using the logic analyzer, on the rising edge of the DMA IRQ Handler. Start the program running on the MCU board and press Run in Waveforms.

You should see the C1 input glitch shortly after the DMA_ISR signal rises.



Notice the error in the oscilloscope trace – the bad data output sticks up like a sore thumb. The DAC output changes before Thread_Refill_Sound_Buffer has a chance to change the first sample in the buffer. The timebase is set to 200 us/division, so each division represents four output samples. This new DAC output is wrong - it is an old value which hasn't been updated yet. The DAC output should change again after 50 us to correct value, and subsequent values should also be correct.

➢ Listen to the audio (unmute if needed). There should be occasional clicking, indicating the bad data outputs.

BASELINE CODE

FIRST TRY

Let's measure the response time for refilling the sound buffer, which is approximately from the rising edge of DMA_ISR until the falling edge of T_RSB. With the scope running, zoom out until you can see the falling edge of T_RSB (e.g. 2 ms/division).

The falling edge of T_RSB should move around, showing a large amount of timing variability. Try pressing the **Single** button (or key **F4**) so the scope acquires one screen of data and then pauses, giving you time to evaluate the waveforms. Do this ten times and record the maximum response time.

21. What is the maximum response time you observed from these ten acquisitions? We will call this ***R_{X1M}*** (experimental value 1, manual measurement).

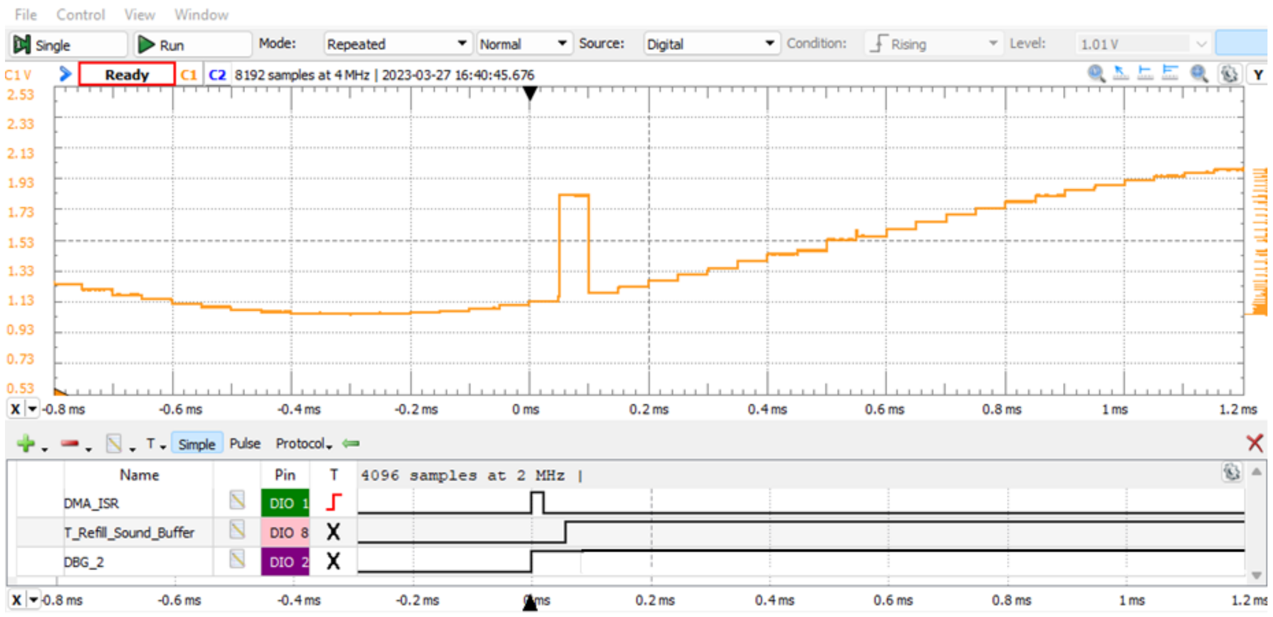
SIMPLIFY MEASUREMENTS WITH AUTOMATION

We will use features in Waveforms and our program to automate some time measurement and statistical analysis. These will let us run the code longer (executing more test cases), filter out irrelevant test cases and get a more accurate view of system behavior.

MAKING THE REFILL LATENCY VISIBLE

Let's simplify the timing delay analysis by making a debug signal explicitly show the time from the refill request to when the first sample is written.

- In debug.h, #define a new symbol DBG_REFILL_PENDING to be DBG_2 (which is connected DIO 2 signal of the AD2).
- Add code so the DMA IRQ Handler starts debug signal DBG_REFILL_PENDING.
- Modify Thread_Refill_Sound_Buffer to stop debug signal DBG_REFILL_PENDING immediately before the comment // Clear updated flags for all voices.
- In the logic analyzer window, press the green + to add DIO 2. Rename the signal from DIO 2 it to Refill Pending. Trigger the scope on its rising edge and start the scope running. You should see something similar to this, with the C1 (orange) and T_Refill_SoundBuffer and Refill Pending traces varying as the program runs:

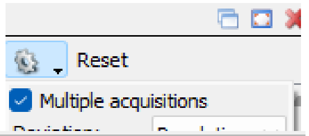


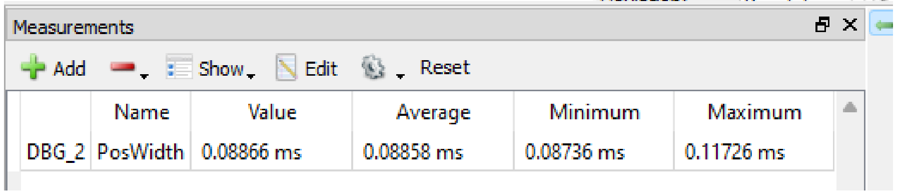
22. Let the scope run for a few seconds. What is the longest observed duration (call it $\boldsymbol{R_{X1}}$) of the Refill Pending pulse?

AUTOMATIC MEASUREMENT

The Waveforms scope tool can measure characteristics of input signals, such as positive pulse width.

- In the Scope 1 View menu items, select Digital Measurements. This will create a Measurements window.
- Click the + Add button to get the Add measurement dialog box. Select the digital signal to measure (DIO 2) and select PosWidth (width of positive pulse). Click Add and Close.
- Click the Show button and check the Average, Minimum and Maximum entries.
- Click the gear icon and check the Multiple Acquisitions box.
- Start the scope running and observe the values. Note that the Maximum value is limited to the maximum time range shown on the horizontal axis. Increase the time per division as the scope is running until the maximum stops increasing.





	Name	Value	Average	Minimum	Maximum
DBG_2	PosWidth	0.08866 ms	0.08858 ms	0.08736 ms	0.11726 ms

SECOND TRY

Let's use this new method to repeat the evaluation of sound buffer refill timing. How long do we need to measure the program? This depends on how long it takes for the maximum to stabilize, which depends on the program and input data. So let's do some analysis first.

- ➢ In Waveforms, Click Reset in the Measurements pane to reset the statistics.
- ➢ Press and hold the shield's touchscreen to increase the application program's processing load.
- ➢ In Waveforms again, press Run so the scope gathers data, and then press Stop after about **one second**.

23. What is the maximum value of PosWidth of Refill Pending for about **one second** of program execution?
24. Reset the measurements and measure the maximum value of PosWidth of Refill Pending for about **ten seconds** of program execution.
25. Reset the measurements and measure the maximum value of PosWidth of Refill Pending for about **30 seconds** of program execution.
26. Reset the measurements and measure the maximum value of PosWidth of Refill Pending for about **120 seconds** of program execution. We will call this $\boldsymbol{R_{X1}}$.

- ➢ Use this information to decide how long to gather data (1, 10, 30, 120 seconds?) in the upcoming measurement tests.

27. What is the time difference $\boldsymbol{R_{X1}}$ - $\boldsymbol{R_{X1M}}$ between the times measured in the manual experiment ($\boldsymbol{R_{X1M}}$) and the last automated baseline experiment ($\boldsymbol{R_{X1}}$)? A large difference shows you missed large pulse width values when sampling the data for $\boldsymbol{R_{X1M}}$.

COMPARING THE MODEL WITH REALITY

The periodic task model we created for the analytical bounding is wrong because we made many assumptions. However, it may still be useful, depending on how we use it. The calculated value $\boldsymbol{R_{B1}}$ is supposed to be an *upper* bound on the actual response time values, so the bound should be *at least as large* as all observed (measured) values, including the maximum observed value $\boldsymbol{R_{X1}}$. The ratio $\boldsymbol{R_{B1}/R_{X1}}$ lets us evaluate how close the bound is to the experimentally observed maximum:

- < 1: Bound is unsafe (smaller than actual value), as experiments prove it is wrong.
- ≥1: Bound seems to be safe (given the tests which were run). The larger the ratio, the looser the bound and the more it overestimates the maximum observed time $\boldsymbol{R_{X1}}$.

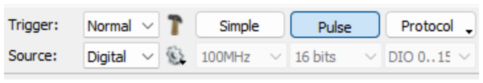
28. What is the ratio $\boldsymbol{R_{B1}/R_{X1}}$? (format: 12.34%)
29. Is the bound $\boldsymbol{R_{B1}}$ safe, given the observed maximum value $\boldsymbol{R_{X1}}$?
30. **561 Only**: Your data should show that $\boldsymbol{R_{X1}}$ is significantly smaller than $\boldsymbol{R_{B1}}$. The periodic timing model you used above to create $\boldsymbol{R_{B1}}$ relies on several simplifying assumptions (refer to the slides and lecture videos for details). Which of those assumptions is most likely to cause the large difference between $\boldsymbol{R_{X1}}$ and $\boldsymbol{R_{B1}}$?

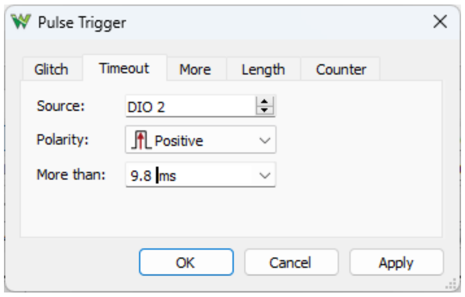
EXAMINING REFILL DELAYS WITH SCOPE

What sequence of events and processing pushes the response time up to $\boldsymbol{R_{X1}}$? Why doesn't it reach $\boldsymbol{R_{B1}}$? Let's get a screenshot of the processor activity so we can see what is happening and how long each part takes.

It would be very helpful for the scope to trigger **only** when the Refill Pending signal is longer than a specified time (e.g. 99% of R_{X1}), filtering out all the other shorter instances of Refill Pending. The logic analyzer supports this kind of triggering on digital inputs, but the scope doesn't, so we will switch tools.

- Stop the scope and select the Logic 1 tab.
- Rename the DIO signals to match the names you entered on the scope's digital window.
- Click the **Pulse** trigger button, then select the **Timeout** tab.
- Set **Source** to DIO 2 (i.e. Refill Buffer), **Polarity** to Positive, and **More than** to just a little less than R_{X1} (you can type in a numerical value with units, e.g. 9.8 ms). Click **OK**.
- Start the the logic analyzer running. It will trigger (t =0 ms) at the **end** of each pulse on DIO 2 which is active (asserted) for more than the time you specified.


Trigger: Normal
Source: Digital 100MHz 16 bits DIO 0..15


Pulse Trigger
Source: DIO 2
Polarity: Positive
More than: 9.8 ms

Now you have an example of the events and processing that lead to a response time near R_{X1}. You should repeat this several times to see if there is a single sequence or multiple sequences (e.g. different threads preempting the refill).

31. Take a screenshot of the logic analyzer for the most common sequence and include it in your report. Be sure the screenshot shows the full positive pulse width of Refill Pending (DIO 2) and the signals DIO 0 through DIO 10 (including their names).
32. Examine your screenshot and list the name and duration of each thread/handler which executes while Refill Pending is active (1).
33. **ECE 561 Only:** Identify which periodic timing model assumption is ~~wrongest~~ most likely to cause the gap between R_{X1} and R_{B1}. and explain why your screenshot supports this.

IMPROVEMENT A: PRIORITIZE REFILL THREAD

One problem with this system is that Thread_Refill_Sound_Buffer is not given priority over other threads, making its response time longer than necessary. Let's raise the priority of Thread_Refill_Sound_Buffer above all other threads.

- Examine the structure Refill_Sound_Buffer_attr in sound.c to see that its .priority field is set to THREAD_RSB_PRIO. Change the definition of THREAD_RSB_PRIO in sound.h from osPriorityNormal to osPriorityAboveNormal. Valid priority levels are defined in cmsis_os2.h (search for osPriority) and online here.
- Save the files, rebuild the project, download and run it.

EXPERIMENTAL MEASUREMENT

- In the Waveforms scope, click Reset in the Measurements window to reset the statistics. We will call the maximum observed value of PosWidth of Refill Pending R_{X2}. Let the scope run until the displayed value for R_{X2} stabilizes.

34. How large is R_{X2}?

- Now use the logic analyzer to perform single triggers until the actual response time is the previously observed maximum R_{X2}.

35. Take a screenshot for your report showing processor activities over the entire response time (e.g. from t=0 to t=R_{X2}).
36. List in order the names of threads and/or handlers which are delaying the refill request in your specific screenshot case.

COMPARISON WITH BOUND 2

37. What is the ratio R_{B2}/R_{X2}? (format: 12.34%)
38. Is the bound R_{B2} safe, given the observed maximum value R_{X2}?
39. **561 Only**: Your data should show that R_{X2} is only slightly smaller than R_{B2}. Explain why this priority improvement tightened up the bound so much.

IMPROVEMENT B: DEADLINE IS FIRST SAMPLE REFILL

Let's simplify the experimental timing delay analysis by making a debug signal explicitly show the time from the refill request to when the **first** sample is written to the sound buffer (instead of after the **last** sample is written). This will let us evaluate the third bound (R_{B3}) which we calculated.

- Modify Thread_Refill_Sound_Buffer to stop debug signal DBG_REFILL_PENDING immediately after each sample is written: SoundBuffer[WriteSBNum][i] = sum; at the comment // Sample Written. Note that there are two instances of writing to SoundBuffer; add the signal after the first instance, not the instance in the #if USE_DOUBLE_BUFFER code structure.
- Build, download and run the program.

EXPERIMENTAL MEASUREMENT

- Measure the timing with Waveforms. Trigger on the rising edge of ISR_DMA. Let the scope run and gather data for at least 30 seconds, being sure to touch the screen to trigger different execution cases.

40. What is the longest observed duration (call it R_{X3}) of the Refill Pending pulse?
41. What is the time delay between T_Refill_Sound_Buffer rising and Refill Pending falling? This is the time for the thread get set up and update the first sample.
42. What is the time delay between Refill Pending falling and T_Refill_Sound_Buffer falling? This is the time to compute the remaining 511 samples in the sound buffer.

COMPARISON WITH BOUND 3

43. What is the ratio R_{B3}/R_{X3}? (format: 12.34%)
44. Is the bound R_{B3} safe, given the observed value R_{X3}?
45. **ECE 561 Only**: Your data should show that R_{X3} is significantly **larger** than R_{B3}. The periodic timing model you used above to create R_{B3} relies on several simplifying assumptions (refer to the slides and lecture videos for details). Which of those assumptions is most likely to cause the large difference between R_{X3} and R_{B3}?

TAKING A CLOSER LOOK

Let's use the logic analyzer to take a closer look and find the biggest mismatch between the model and reality. Configure the logic analyzer to trigger when DIO 2 (Refill Pending) is about as long as R_{X3}.

46. Take a screenshot for your report showing ISR_DMA (signal rise and fall), Refill Pending (rise and fall), and the start of T_Refill_Sound_Buffer (activities until Refill Pending falls).
47. Which thread (call it "X") was running when the DMA ISR started running (indicated by ISR_DMA rising)?

48. What is the time delay between ISR_DMA rising and falling?
49. What is the time delay between ISR_DMA falling and thread X being preempted?
50. What is the time delay between thread X being preempted and T_Refill_Soundbuffer resuming?
51. What is the time delay between T_Refill_Sound_Buffer rising and Refill Pending falling?

Consider these measurements and examine the source code for Thread_Refill_Sound_Buffer between osThreadFlagsWait() and SoundBuffer[WriteSBNum][i] = sum;.

52. Which simplifying assumption caused the **largest** difference between R_{X3} and R_{B3}? Explain the sequence of events in your screenshot, your measurements of their timing, and how these support your answer about the top simplifying assumption.
53. **ECE 561 Only**: Which assumption caused the **second-largest** difference between R_{X3} and R_{B3}? Again, explain how your evidence supports your answer.

IMPROVEMENT C: EASE THE DEADLINE WITH DOUBLE-BUFFERING

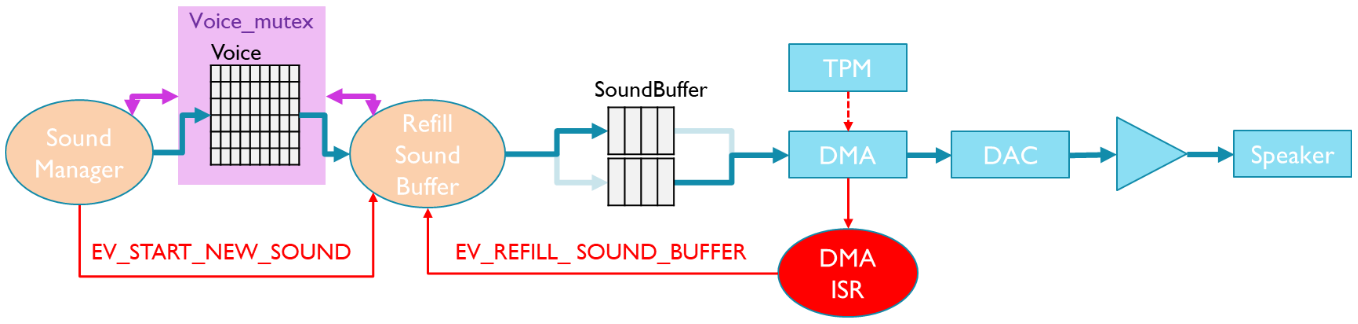


Figure 4. Sound generation architecture with double buffer.

The code has support for double-buffering the sound output. This raises the deadline for refilling the first SoundBuffer entry from 1*T_{Sample} = 50 µs to NUM_SAMPLES_PER_SOUND_BUFFER*T_{Sample}.

- To enable it, change the definition of USE_DOUBLE_BUFFER (in sound.h) from 0 to 1. To keep the total buffer memory size the same, each buffer is half the size of the original buffer. This reduction will double the release frequency of DMA controller interrupt, its handler, and Thread_Refill_Sound_Buffer.
- Save the files, rebuild the code and download it.

EXPERIMENTAL MEASUREMENT

- Switch back to the scope window. Click Reset in the Measurements window to reset the statistics. Start the scope running again.

54. What is the maximum positive pulse width of Refill Buffer (i.e. delay to update the first SoundBuffer sample)? It should be well under the new deadline.
55. Watch the DAC output on the scope for at least 20 seconds. Do you ever see a corrupted sample?
56. Press Unmute on the LCD and listen to the audio. Is the sound quality better? Most of the clicking should be gone, but there may be other bugs remaining in the system.

TAKING A CLOSER LOOK

- ➢ Switch to the logic analyzer and set the timeout to a little less than the maximum delay you just measured. Use the logic analyzer to capture the events which cause the maximum latency (i.e. Refill Buffer pulse width) which you saw above.

57. Take a screenshot of the logic analyzer and include it in your report.
58. Which threads and/or ISRs are delaying the refill request in your specific screenshot case? Describe the sequence of events.

REVISION HISTORY

V1.01

Added spreadsheet link, corrected report question numbers.

V1.02

Added instructions to change to simple TV blocked signal when measuring task durations, change back afterwards