To Do

- Currently this file is only a skeleton for how the expansion shield code uses the RTOS. Need to consider overlap with the **Architecture Design and Expansion Shield** slides before filling out.
- Add text notes per slide
- Convert hand drawings

Using the RTOS for the Expansion Shield Code

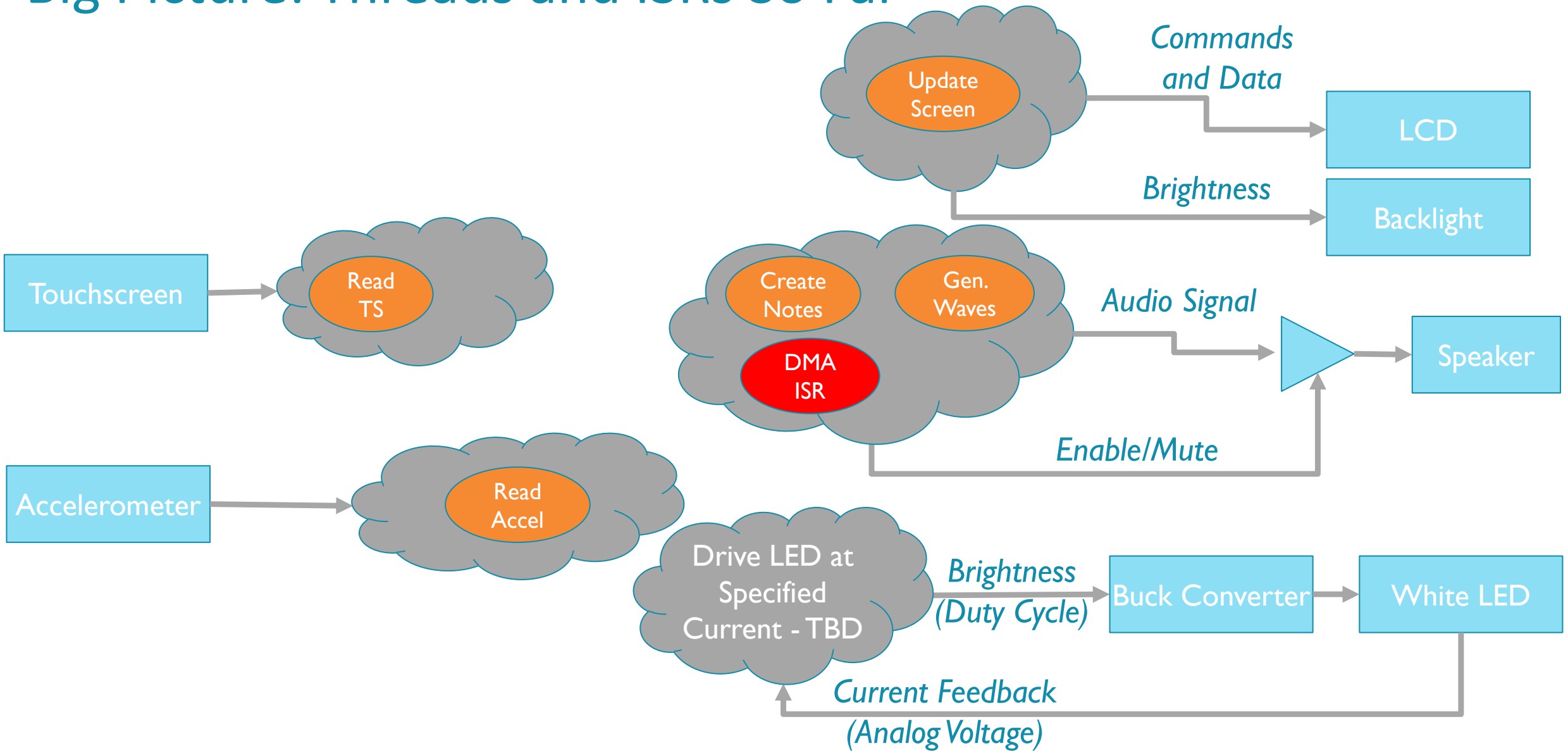
rev. 1/3/2024

Overview

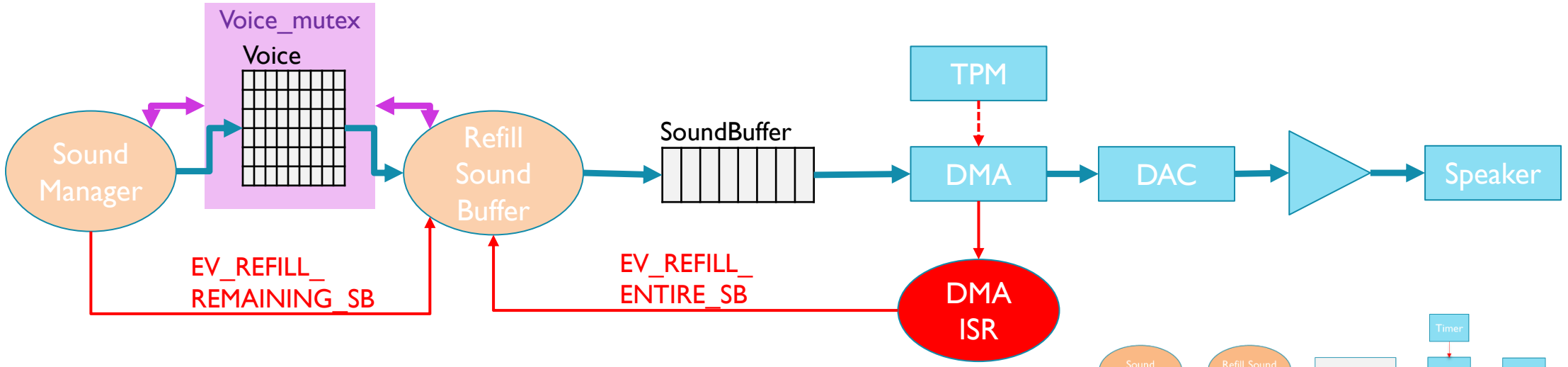
- How does RTOS help us implement the Expansion Shield *architecture*?
  - Threads and ISRs
  - Shared data and resources
  - Communication and synchronization
- Code on Github: Tools/TestCode/Shield_Base_v14
- Steps
  - Start up program and RTOS
  - Threads and periodic execution
  - Triggering threads with Event Flags
  - Triggering threads with Semaphores
  - Message Queues and Data Buffering
  - Restricting threads with Mutexes
  - ADC Server

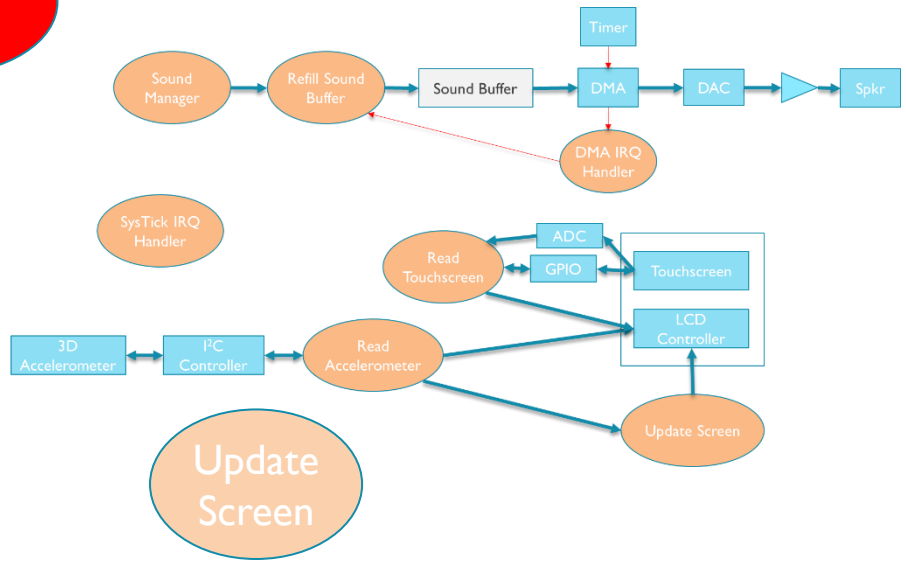
	ISRs	Delays	Event Flags	Semaphores	Message Queues	Mutexes
Sound Generation	x	x	x			x
Read Touchscreen		x			(x)	x
Read Accelerometer		x				
Update Screen		x				x
(CC LED Driver)	(x)				(x)	(x)

Big Picture: Threads and ISRs So Far



Shield Software and Hardware Architecture





Create initial arch diagram and then iteratively refine with more detail.

Redraw diagram with less detail.

Decide on format/syntax of visual language.

Actors: Threads, ISRs, peripherals.

Interactions: Triggering, mutual exclusion, delays

Roadmap

- Start up program and RTOS
- Threads and periodic execution
- Triggering threads with Event Flags
- Triggering threads with Semaphores
- Message Queues and Data Buffering
- Restricting threads with Mutexes
- ADC Server

Start-Up

- main() function
  - Initializes system
  - Tests some peripherals
  - Initializes RTOS
  - Creates RTOS objects (threads, etc.)
  - Starts RTOS running (never returns)

Roadmap

- Start up program and RTOS
- Threads and periodic execution
- Triggering threads with Event Flags
- Triggering threads with Semaphores
- Message Queues and Data Buffering
- Restricting threads with Mutexes
- ADC Server

Threads: Telling RTOS about the Threads

- Must create at least one thread before starting RTOS
- System has multiple threads and other RTOS objects
  - Convenient to make a function to group together their creation
  - threads.c:Create_OS_Objects()
  - Tell OS about thread by calling osThreadNew with thread's root function and attributes
  - Should check return value for each OS call to detect, handle errors

Threads: Structure and Behavior

- Root functions
  - Structured as initialization + infinite loop
    - Wait, then work
- Debug signals
  - Indicate on logic analyzer when thread has started work but hasn't finished it
    - Waiting for work vs. "working"
- Periodic thread execution
  - Implemented with osDelay
  - Should really use osDelayUntil for better timing consistency – eliminates accumulation of most timing error

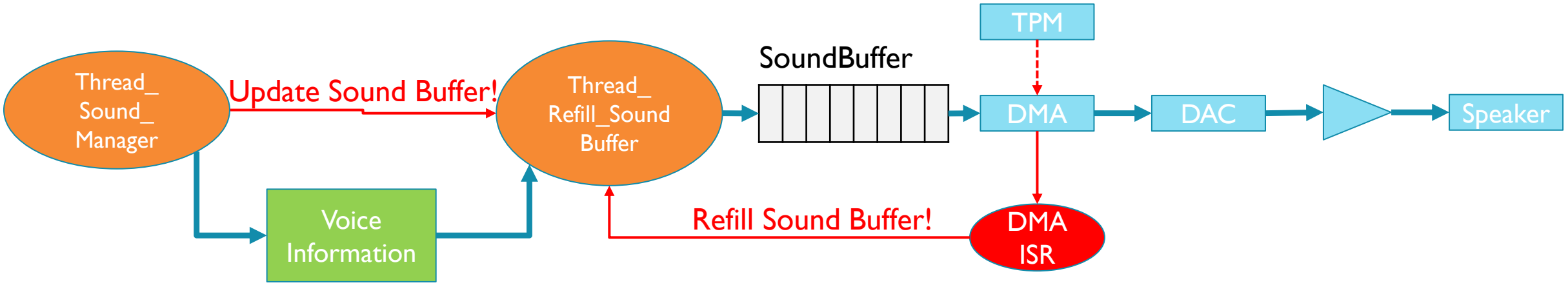
Threads: Idle Thread

- RTX_Config.c:osRtxIdleThread() runs when nothing else is ready
  - Infinite loop
  - Have added code to toggle a digital output bit to indicate on logic analyzer (LA) when idle thread is running
- Examine timing behavior with LA
  - Keep in mind that sampling effects may distort the displayed signal
    - Aliasing. Some signal transitions may not be shown.
    - Do not have infinite zoom. Non-vertical signal transition or gray rectangle indicate zoomed in too far. Retrigger with shorter time base.

Roadmap

- Start up program and RTOS
- Threads and periodic execution
- Triggering threads with Event Flags
- Triggering threads with Semaphores
- Message Queues and Data Buffering
- Restricting threads with Mutexes
- ADC Server

Triggering Threads with Event Flags



- Audio generation requires sound buffer to be filled with samples…
  - When DMA reaches end of buffer, fill **entire buffer**
  - When a new note is activated, fill **unplayed portion of buffer**
- What data to write?
  - When DMA reaches end of buffer, create new samples
  - When a new note is activated, update existing samples by adding waveform from new note
    - *Note: not implemented in code yet*

Event-Related Code

- Generate event – set event flag
  - DMA0_IRQHandler
  - Thread_Sound_Manager
- Await event, then do requested work
  - Thread_Refill_Soundbuffer

Confirm and Evaluate System Behavior and Timing

- Does it really work?
- How fast is it?
- Is there anything unexpected?
- Monitor the analog output too
  - Use mixed-signal display mode
  - Add spectrogram

Roadmap

- Start up program and RTOS
- Threads and periodic execution
- Triggering threads with Event Flags
- Triggering threads with Semaphores
- Message Queues and Data Buffering
- Restricting threads with Mutexes
- ADC Server

Roadmap

- Start up program and RTOS
- Threads and periodic execution
- Triggering threads with Event Flags
- Triggering threads with Semaphores
- Message Queues and Data Buffering
- Restricting threads with Mutexes
- ADC Server

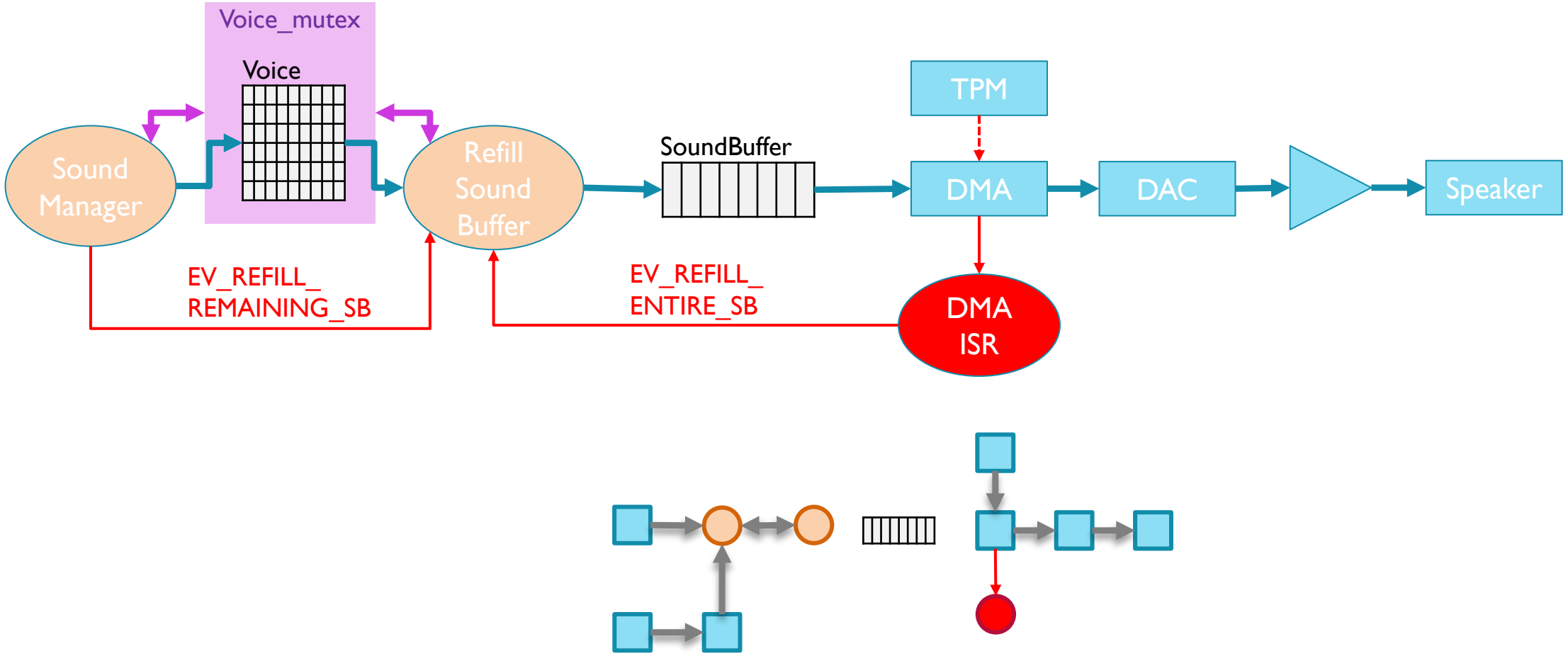
Roadmap

- Start up program and RTOS
- Threads and periodic execution
- Triggering threads with Event Flags
- Triggering threads with Semaphores
- Message Queues and Data Buffering
- Restricting threads with Mutexes
- ADC Server

Roadmap

- Start up program and RTOS
- Threads and periodic execution
- Triggering threads with Event Flags
- Triggering threads with Semaphores
- Message Queues and Data Buffering
- Restricting threads with Mutexes
- ADC Server

OLD SLIDES


Voice_mutex
Voice
Sound Manager
Refill Sound Buffer
SoundBuffer
TPM
DMA
DAC
Speaker
EV_REFILL_ REMAINING_SB
EV_REFILL_ ENTIRE_SB
DMA ISR

Sequencing Interactions

When two actors interact, is sequencing needed?

Interactions (2)

- asdf

How will parts interact to synchronize resource use and activity and sharing data?

	to Software	to Hardware
From Software	• *Shared variables, mutexes, event flags, semaphores, message queues, etc.*	• Software must write to control register fields
From Hardware	• Software reads status register fields (polling) • Request interrupt service (event-driven)	• Control signals between peripherals. Inputs: start A/D conversion trigger, count input, start DMA trigger, etc. Outputs: counter overflow, comparator output, timer match signal, A/D conversion complete, DMA transfer complete, etc.

Hardware/Software Interactions

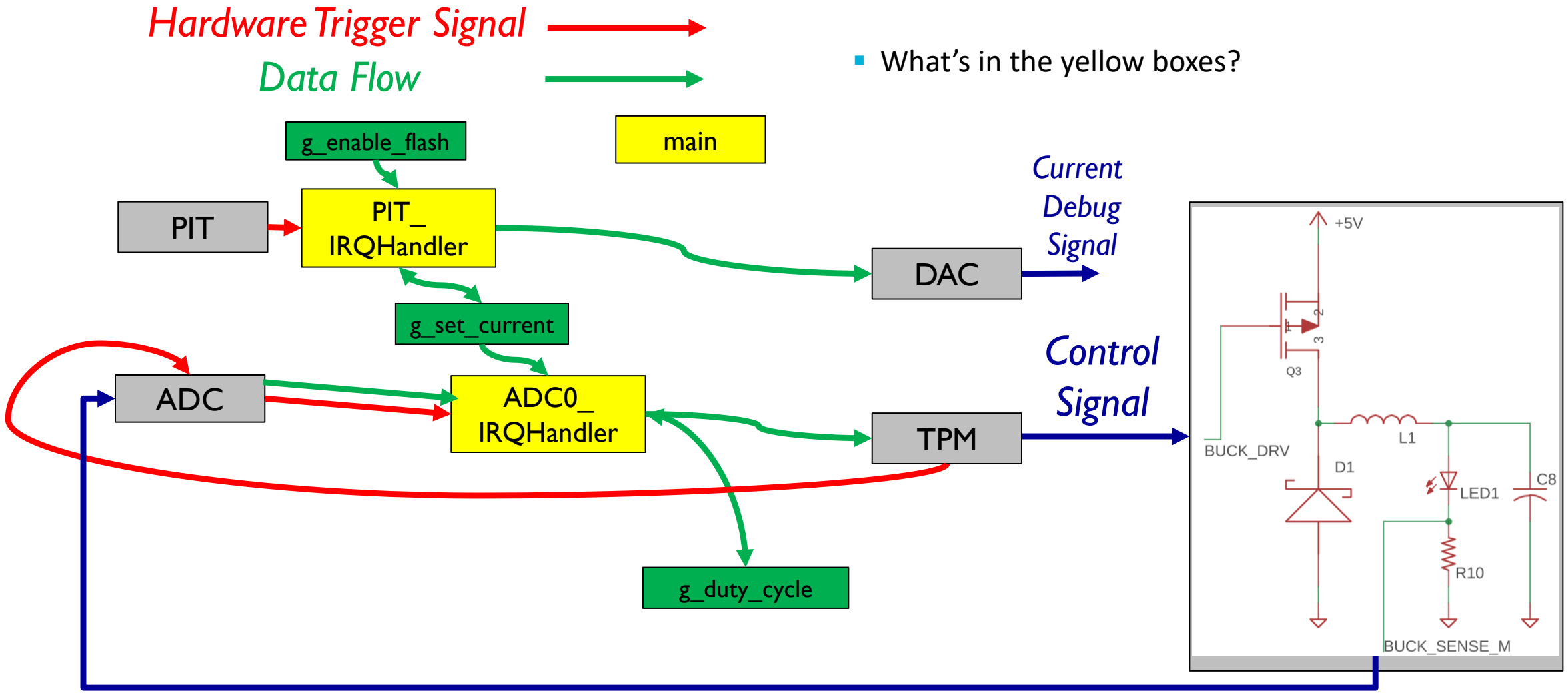
```
                PTD->PCOR = MASK(BLUE_LED_POS);
        }       else {
                PTD->PSOR = MASK(BLUE_LED_POS);
        }
}

#ifndef HBLED_H
#define HBLED_H

// Switching parameters
#define PWM_HBLED_CHANNEL (4)
#define PWM_PERIOD (400)
/* 48 MHz input clock.
       PWM frequency = 48 MHz/(PWM_PERIOD*2)
       Timer is in count-up/down mode. */

#define LIM_DUTY_CYCLE (PWM_PERIOD)

// Control approach configuration
#define USE_ASYNC_SAMPLING      0
#define USE_SYNC_NO_FREQ_DIV            1
#define USE_SYNC_SW_CTL_FREQ_DIV 0
#define USE_SYNC_HW_CTL_FREQ_DIV 0

#define SW_CTL_FREQ_DIV_FACTOR (1) // Software division in ISR
#define HW_CTL_FREQ_DIV_CODE (0) // Not used

#if USE_ASYNC_SAMPLING
#define       USE_TPM0_INTERRUPT 0
#define       USE_ADC_HW_TRIGGER 0
#define       USE_ADC_INTERRUPT 1
#endif

#if USE_SYNC_NO_FREQ_DIV
#define       USE_TPM0_INTERRUPT 0
#define       USE_ADC_HW_TRIGGER 1
#define       USE_ADC_INTERRUPT 1
#endif

#if USE_SYNC_SW_CTL_FREQ_DIV
#define       USE_TPM0_INTERRUPT 1
#define       USE_ADC_HW_TRIGGER 0
#define       USE_ADC_INTERRUPT 1
#endif

#if USE_SYNC_HW_CTL_FREQ_DIV
#define       USE_TPM0_INTERRUPT 0
#define       USE_ADC_HW_TRIGGER 1
#define       USE_ADC_INTERRUPT 1
#endif

// Control Parameters
// default control mode: OpenLoop, BangBang, Incremental, PID, PID_FX
#define DEF_CONTROL_MODE (Incremental)

// Incremental controller: change amount
#define INC_STEP (PWM_PERIOD/40)

// Proportional Gain, scaled by 2^8
#define PGAIN_8 (0x0028)

// PID (floating-point) gains. Guaranteed to be non-optimal.
#define I_GAIN_FL (0.000f)
#define P_GAIN_FL (0.600f)
#define D_GAIN_FL (0.000f)

// PID_FX (fixed-point) gains. Guaranteed to be non-optimal.
#define I_GAIN_FX (FL_TO_FX(0.0065f))
#define P_GAIN_FX (FL_TO_FX(0.0626f))
#define D_GAIN_FX (FL_TO_FX(0.1831f))

#define FLASH_PERIOD (20)
#define FLASH_CURRENT_MA (180)

// Hardware configuration
#define ADC_SENSE_CHANNEL (8)

#define R_SENSE (2.2f)
#define R_SENSE_MO ((int) (R_SENSE*1000))

#define V_REF (3.3f)
#define V_REF_MV ((int) (V_REF*1000))

#define ADC_FULL_SCALE (0x10000)
#define MA_SCALING_FACTOR (1000)

#define DAC_POS 30
#define DAC_RESOLUTION 4096

// #define MA_TO_DAC_CODE(i) (i*2.2*DAC_RESOLUTION/V_REF_MV) // Introduces timing delay and interesting bug!
#define MA_TO_DAC_CODE(i) ((i)*(2.2f*DAC_RESOLUTION/V_REF_MV))

#define MIN(a,b) ((a<b)?a:b)
#define MAX(a,b) ((a>b)?a:b)

#endif // HBLED_H

#ifndef DELAY_H
#define DELAY_H
#include <stdint.h>

extern void Delay(uint32_t dlyTicks);
extern void ShortDelay(uint32_t dlyTicks);
#endif
// ***********************************ARM University Program Copyright © ARM Ltd 2013*************************************
```

```
#include <MKL25Z4.H>

void Delay (uint32_t dly) {
  volatile uint32_t t;

        for (t=dly*10000; t>0; t--)
                ;
}

void ShortDelay (uint32_t dly) {
  volatile uint32_t t;

        for (t=dly; t>0; t--)
                ;
}

// ***********************************ARM University Program Copyright © ARM Ltd 2013*************************************

/*----------------------------------------------------------------------------
 *----------------------------------------------------------------------------*/
#include <MKL25Z4.H>
#include <stdio.h>
#include <stdint.h>

#include "gpio_defs.h"
#include "debug.h"

#include "timers.h"
#include "delay.h"
#include "LED5.h"

#include "HBLED.h"
#include "FX.h"

volatile int g_enable_control=1;
volatile int g_set_current=0;

volatile int measured_current;
volatile int16_t g_duty_cycle=0;  // global to give debugger access
volatile int error;
enum {OpenLoop, BangBang, Incremental, Proportional, PID, PID_FX}
        control_mode=DEF_CONTROL_MODE;

int32_t pGain_8 = PGAIN_8; // proportional gain numerator scaled by 2^8
volatile int g_enable_flash=1;

typedef struct {
        float dState; // Last position input
        float iState; // Integrator state
        float iMax, iMin; // Maximum and minimum allowable integrator state
        float iGain, // integral gain
                        pGain, // proportional gain
                        dGain; // derivative gain
} SPid;

typedef struct {
        FX16_16 dState; // Last position input
        FX16_16 iState; // Integrator state
        FX16_16 iMax, iMin; // Maximum and minimum allowable integrator state
        FX16_16 iGain, // integral gain
                                pGain, // proportional gain
                                dGain; // derivative gain
} SPidFX;


SPid plantPID = {0, // dState
        0, // iState
        LIM_DUTY_CYCLE, // iMax
        -LIM_DUTY_CYCLE, // iMin
        I_GAIN_FL, // iGain
        P_GAIN_FL, // pGain
        D_GAIN_FL  // dGain
};

SPidFX plantPID_FX = {FL_TO_FX(0), // dState
        FL_TO_FX(0), // iState
        FL_TO_FX(LIM_DUTY_CYCLE), // iMax
        FL_TO_FX(-LIM_DUTY_CYCLE), // iMin
        I_GAIN_FX, // iGain
        P_GAIN_FX, // pGain
        D_GAIN_FX  // dGain
};


float UpdatePID(SPid * pid, float error, float position){
        float pTerm, dTerm, iTerm;

        // calculate the proportional term
        pTerm = pid->pGain * error;
        // calculate the integral state with appropriate limiting
        pid->iState += error;
        if (pid->iState > pid->iMax)
                pid->iState = pid->iMax;
        else if (pid->iState < pid->iMin)
                pid->iState = pid->iMin;
        iTerm = pid->iGain * pid->iState; // calculate the integral term
        dTerm = pid->dGain * (position - pid->dState);
        pid->dState = position;

        return pTerm + iTerm - dTerm;
}

FX16_16 UpdatePID_FX(SPidFX * pid, FX16_16 error_FX, FX16_16 position_FX){
        FX16_16 pTerm, dTerm, iTerm, diff, ret_val;

        // calculate the proportional term
        pTerm = Multiply_FX(pid->pGain, error_FX);
```

```
        // calculate the integral state with appropriate limiting
        pid->iState = Add_FX(pid->iState, error_FX);
        if (pid->iState > pid->iMax)
                pid->iState = pid->iMax;
        else if (pid->iState < pid->iMin)
                pid->iState = pid->iMin;

        iTerm = Multiply_FX(pid->iGain, pid->iState); // calculate the integral term
        diff = Subtract_FX(position_FX, pid->dState);
        dTerm = Multiply_FX(pid->dGain, diff);
        pid->dState = position_FX;

        ret_val = Add_FX(pTerm, iTerm);
        ret_val = Subtract_FX(ret_val, dTerm);
        return ret_val;
}

void Control_HBLED(void) {
        uint16_t res;
        FX16_16 change_FX, error_FX;

        FPTB->PSOR = MASK(DBG_CONTROLLER);

#if USE_ADC_INTERRUPT
        // already completed conversion, so don't wait
#else
        while (!(ADC0->SC1[0] & ADC_SC1_COCO_MASK))
                ; // wait until end of conversion
#endif
        res = ADC0->R[0];

        measured_current = (res*1500)>>16; // Extra Credit: Make this code work: V_REF_MV*MA_SCALING_FACTOR)/(ADC_FULL_SCALE*R_SENSE)

        switch (control_mode) {
                case OpenLoop:
                                // don't do anything!
                        break;
                case BangBang:
                        if (measured_current < g_set_current)
                                g_duty_cycle = LIM_DUTY_CYCLE;
                        else
                                g_duty_cycle = 0;
                        break;
                case Incremental:
                        if (measured_current < g_set_current)
                                g_duty_cycle += INC_STEP;
                        else
                                g_duty_cycle -= INC_STEP;
                        break;
                case Proportional:
                        g_duty_cycle += (pGain_8*(g_set_current - measured_current))/256; //  - 1;
                break;
                case PID:
                        g_duty_cycle += UpdatePID(&plantPID, g_set_current - measured_current, measured_current);
                        break;
                case PID_FX:
                        error_FX = INT_TO_FX(g_set_current - measured_current);
                        change_FX = UpdatePID_FX(&plantPID_FX, error_FX, INT_TO_FX(measured_current));
                        g_duty_cycle += FX_TO_INT(change_FX);
                break;
                default:
                        break;
        }

        // Update PWM controller with duty cycle
        if (g_duty_cycle < 0)
                g_duty_cycle = 0;
        else if (g_duty_cycle > LIM_DUTY_CYCLE)
                g_duty_cycle = LIM_DUTY_CYCLE;
        PWM_Set_Value(TPM0, PWM_HBLED_CHANNEL, g_duty_cycle);
        FPTB->PCOR = MASK(DBG_CONTROLLER);
}

#if USE_ADC_INTERRUPT
void ADC0_IRQHandler() {
        FPTB->PSOR = MASK(DBG_IRQ_ADC);
        Control_HBLED();
        FPTB->PCOR = MASK(DBG_IRQ_ADC);
}
#endif

void Set_DAC(unsigned int code) {
        // Force 16-bit write to DAC
        uint16_t * dac0dat = (uint16_t *)&(DAC0->DAT[0].DATL);
        *dac0dat = (uint16_t) code;
}

void Set_DAC_mA(unsigned int current) {
        unsigned int code = MA_TO_DAC_CODE(current);
        // Force 16-bit write to DAC
        uint16_t * dac0dat = (uint16_t *)&(DAC0->DAT[0].DATL);
        *dac0dat = (uint16_t) code;
}

void Init_DAC(void) {
  // Enable clock to DAC and Port E
        SIM->SCGC6 |= SIM_SCGC6_DAC0_MASK;
        SIM->SCGC5 |= SIM_SCGC5_PORTE_MASK;

        // Select analog for pin
        PORTE->PCR[DAC_POS] &= ~PORT_PCR_MUX_MASK;
        PORTE->PCR[DAC_POS] |= PORT_PCR_MUX(0);

        // Disable buffer mode
        DAC0->C1 = 0;
        DAC0->C2 = 0;

        // Enable DAC, select VDDA as reference voltage
        DAC0->C0 = DAC_C0_DACEN_MASK | DAC_C0_DACRFS_MASK;
        Set_DAC(0);
}

void Init_ADC(void) {
        // Configure ADC to read Ch 8 (FPTB 0)
        SIM->SCGC6 |= SIM_SCGC6_ADC0_MASK;
        ADC0->CFG1 = 0x0C; // 16 bit
        //     ADC0->CFG2 = ADC_CFG2_ADLSTS(3);
        ADC0->SC2 = ADC_SC2_REFSEL(0);

#if USE_ADC_HW_TRIGGER
        // Enable hardware triggering of ADC
        ADC0->SC2 |= ADC_SC2_ADTRG(1);
        // Select triggering by TPM0 Overflow
        SIM->SOPT7 = SIM_SOPT7_ADC0TRGSEL(8) | SIM_SOPT7_ADC0ALTTRGEN_MASK;
        // Select input channel
        ADC0->SC1[0] &= ~ADC_SC1_ADCH_MASK;
        ADC0->SC1[0] |= ADC_SC1_ADCH(ADC_SENSE_CHANNEL);
#endif

#if USE_ADC_INTERRUPT
        // enable ADC interrupt
        ADC0->SC1[0] |= ADC_SC1_AIEN(1);

        // Configure NVIC for ADC interrupt
        NVIC_SetPriority(ADC0_IRQn, 128); // 0, 64, 128 or 192
        NVIC_ClearPendingIRQ(ADC0_IRQn);
        NVIC_EnableIRQ(ADC0_IRQn);
#endif
}

void Update_Set_Current(void) {
        static int delay=FLASH_PERIOD;

        if (g_enable_flash){
                delay--;
                if (delay==1) {
                        FPTB->PSOR = MASK(DBG_LED_ON);
                        Set_DAC_mA(FLASH_CURRENT_MA);
                        g_set_current = FLASH_CURRENT_MA;
                } else if (delay==0) {
                        delay=FLASH_PERIOD;
                        FPTB->PSOR = MASK(DBG_LED_ON);
                        Set_DAC_mA(FLASH_CURRENT_MA/4);
                        g_set_current = FLASH_CURRENT_MA/4;
                } else {
                        FPTB->PCOR = MASK(DBG_LED_ON);
                        Set_DAC_mA(0);
                        g_set_current = 0;
                }
        }
}
/*----------------------------------------------------------------------------
  MAIN function
 *----------------------------------------------------------------------------*/
int main (void) {
        Init_Debug_Signals();
        Init_DAC();
        Init_ADC();
        Init_RGB_LEDs();

        // Configure driver for buck converter
        // Set up PTE31 to use for SMPS with TPM0 Ch 4
        SIM->SCGC5 |= SIM_SCGC5_PORTE_MASK;
        PORTE->PCR[31]  &= PORT_PCR_MUX(7);
        PORTE->PCR[31]  |= PORT_PCR_MUX(3);
        PWM_Init(TPM0, PWM_HBLED_CHANNEL, PWM_PERIOD, 40);

        Control_RGB_LEDs(1,1,0);
        Delay(100);
        Control_RGB_LEDs(0,0,1);

        // Configure flash timer
        Init_PIT(423456);
        Start_PIT();

        while (1) {
#if USE_ASYNC_SAMPLING
                // Start conversion
                ADC0->SC1[0] = ADC_SC1_AIEN(1) | ADC_SENSE_CHANNEL;
                Control_HBLED(); // Blocks until ADC done, then updates control
#endif
                // else do nothing but wait for interrupts
        }
}
#ifndef TIMERS_H
#define TIMERS_H
#include "MKL25Z4.h"

void PWM_Init(TPM_Type * TPM, uint8_t channel_num, uint16_t period, uint16_t duty);
void PWM_Set_Value(TPM_Type * TPM, uint8_t channel_num, uint16_t value);

void Init_PIT(unsigned period);
void Start_PIT(void);
void Stop_PIT(void);

#endif

#include "timers.h"
#include <MKL25Z4.h>
#include "HBLED.h"
#include "GPIO_defs.h"
#include "debug.h"
extern void Update_Set_Current(void);
```

```
void PWM_Init(TPM_Type * TPM,
{
        //turn on clock to TPM
        switch ((int) TPM) {
                case (int) TPM0:
                        SIM->SCGC6
                        break;
                case (int) TPM1:
                        SIM->SCGC6
                        break;
                case (int) TPM2:
                        SIM->SCGC6
                        break;
                default:
                        break;
        }
        //set clock source for
        SIM->SOPT2 |= (SIM_SOPT

        //load the counter and
        TPM->MOD = period;

        //set channel to center
        TPM->CONTROLS[channel_n

        //set TPM to up-down an
        TPM->SC = (TPM_SC_CPWMS

        //set trigger mode and
        TPM->CONF |= TPM_CONF_T

        // Set initial duty cyc
        TPM->CONTROLS[channel_n

#if USE_TPM0_INTERRUPT // if
        TPM0->SC |= TPM_SC_TOIE

        // Configure NVIC
        NVIC_SetPriority(TPM0_I
        NVIC_ClearPendingIRQ(TP
        NVIC_EnableIRQ(TPM0_IRQ
#endif

        // Start the timer coun
        TPM->SC |= TPM_SC_CMOD(
}

void PWM_Set_Value(TPM_Type *
        TPM->CONTROLS[channel_n
}
extern void Control_HBLED(voi

void TPM0_IRQHandler() {
        static uint32_t control

        FPTB->PSOR = MASK(DBG_I
        //clear pending IRQ fla
        TPM0->SC |= TPM_SC_TOF_

        control_divider--;
        if (control_divider ==
                control_divider
                // Start convers
                ADC0->SC1[0] = AI

        #if USE_ADC_INTERRUPT
                // can return im
        #else
                // Call control
                Control_HBLED();
        #endif
        }
        FPTB->PCOR = MASK(DBG_I
}


void Init_PIT(unsigned period
        // Enable clock to PIT
        SIM->SCGC6 |= SIM_SCGC6

        // Disable clocks for c
        PIT->MCR |= PIT_MCR_MDI

        // Initialize PITO to c
        PIT->CHANNEL[0].LDVAL =

        // No chaining
        PIT->CHANNEL[0].TCTRL &

        PIT->CHANNEL[0].TCTRL |

#if 1 // generate interrupts
        NVIC_SetPriority(PIT_IR
        NVIC_ClearPendingIRQ(PI
        NVIC_EnableIRQ(PIT_IRQn
#endif
}

void Start_PIT(void) {
// Enable counter
        PIT->CHANNEL[0].TCTRL |
        PIT->MCR &= ~PIT_MCR_MD

}

void Stop_PIT(void) {
// Disable counter
```

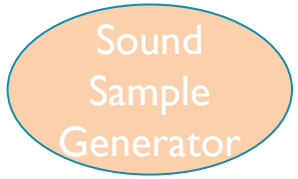
Hardware/Software Interactions

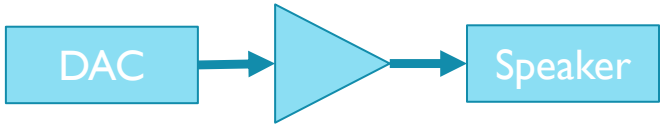


- What's in the yellow boxes?

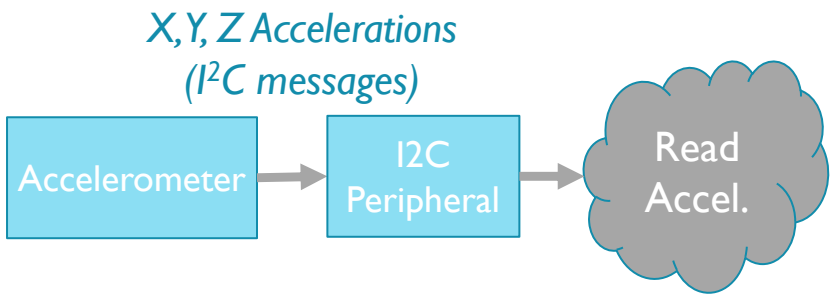
Embedded System Architectures © 2020 A.G. Dean

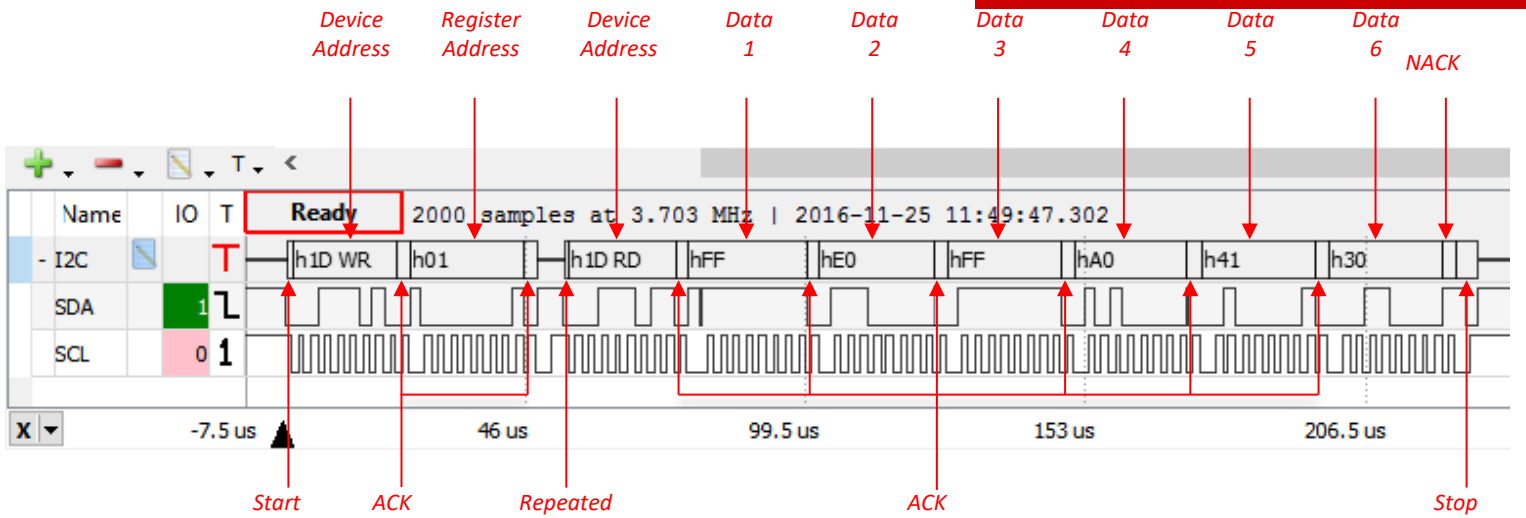
Evaluating Audio Software and Hardware Options





Refine: Accelerometer





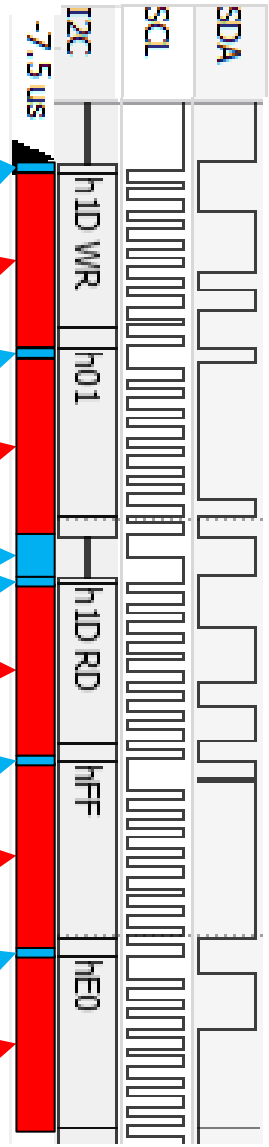
- How do we trigger the code to run?
  - Send message to read multiple acceleration bytes
- How do we talk to the I2C peripheral?
  - Examine documentation (KL25Z Ref. Man. Chapter 38)
  - Byte-oriented protocol. Software must run per byte.
  - Send start condition, send address byte, wait for ack, send data byte, wait for ack, etc. …. send stop condition
- Does I2C code have any internal delays?
  - Yes – limited by I^2C bus speed
  - How to implement these delays?
  - Try to reclaim that idle time? How much is there?

```
I2C_TRAN;                  /*set to transmit mode */
I2C_M_START;               /*send start    */
I2C0->D = dev;             /*send dev address        */
I2C_WAIT;                  /*wait for completion */
I2C0->D = address;         /*send read address */
I2C_WAIT;                  /*wait for completion */
I2C_M_RSTART;              /*repeated start */
I2C0->D = (dev|0x1);       /*send dev address (read) */
I2C_WAIT;                  /*wait for completion */
I2C_REC;                   /*set to recieve mode */
ACK;                       /*send ACK after read */
data = I2C0->D;            /*dummy read    */
I2C_WAIT;                  /*wait for completion */
data = I2C0->D;            /*read data     */
ACK;                       /*send ACK after read */
data = I2C0->D;            /*dummy read    */
I2C_WAIT;                  /*wait for completion */
data = I2C0->D;            /*read data     */
```

Refine: Accelerometer

```
void read_full_xyz()
{
  int i;
  uint8_t data[6];

  i2c_start();
  i2c_read_setup(MMA_ADDR , REG_XHI);

  for( i=0;i<5;i++)
    data[i] = i2c_repeated_read(0);
  data[i] = i2c_repeated_read(1);

  acc_X = (((int16_t) data[0])<<8) | data[1];
  acc_Y = (((int16_t) data[2])<<8) | data[3];
  acc_Z = (((int16_t) data[4])<<8) | data[5];
}
```

```
read_full_xyz())
    i2c_start()
        I2C_TRAN;              /*set to transmit mode */
        START;                 /*send start          */
    setup(dev, address)
    D = dev;                   /*send dev address (write)*/
    ait()
    ile ((I2C0->S&I2C_S_IICIF_MASK)==0);
    C0->S |= I2C_S_IICIC_MASK;
    D = address;               /*send read address */
    it();                      /*wait for completion */
    RSTART;                    /*repeated start */
    D = (dev|0x1);             /*send dev address (read)*/
    it();                      /*wait for completion */
    C;                         /*set to receive mode */
    ted_read(isLastRead)
        I2C0->D;               /*dummy read starts rx (if not
                                 already receiving) */
        I2C_WAIT;              /*wait for completion */
        data = I2C0->D;        /*read data, start next rx */
    i2c_repeated_read(isLastRead)
        data = I2C0->D;        /*dummy read starts rx (if not
                                 already receiving) */
        I2C_WAIT;              /*wait for completion */
        data = I2C0->D;        /*read data, start next rx */
```



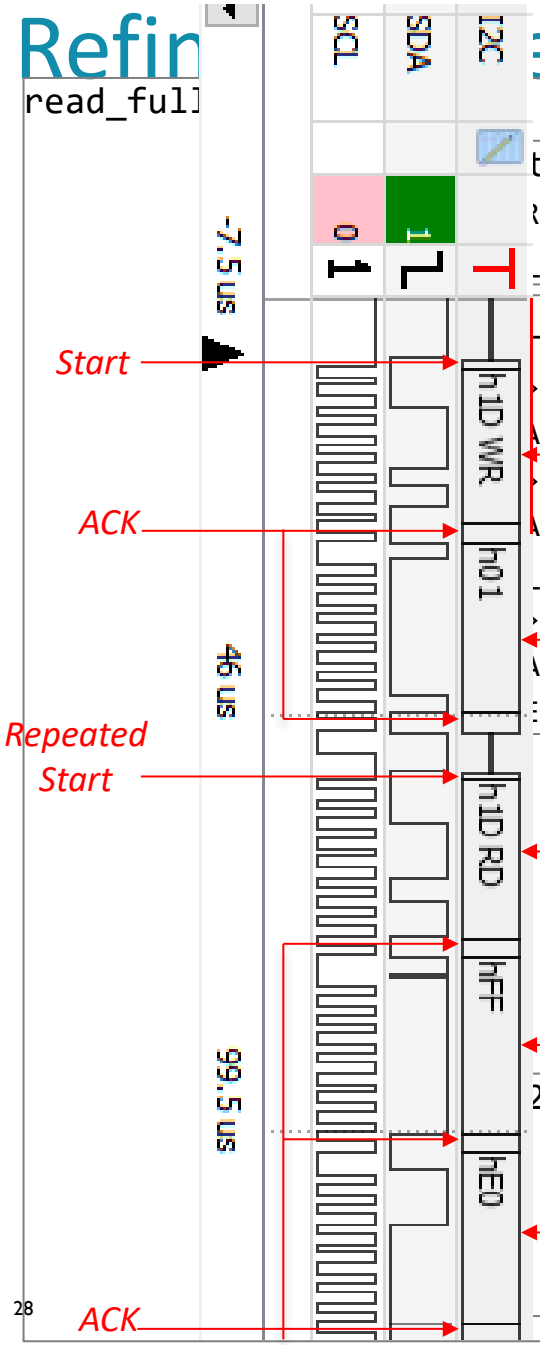
Refin erometer

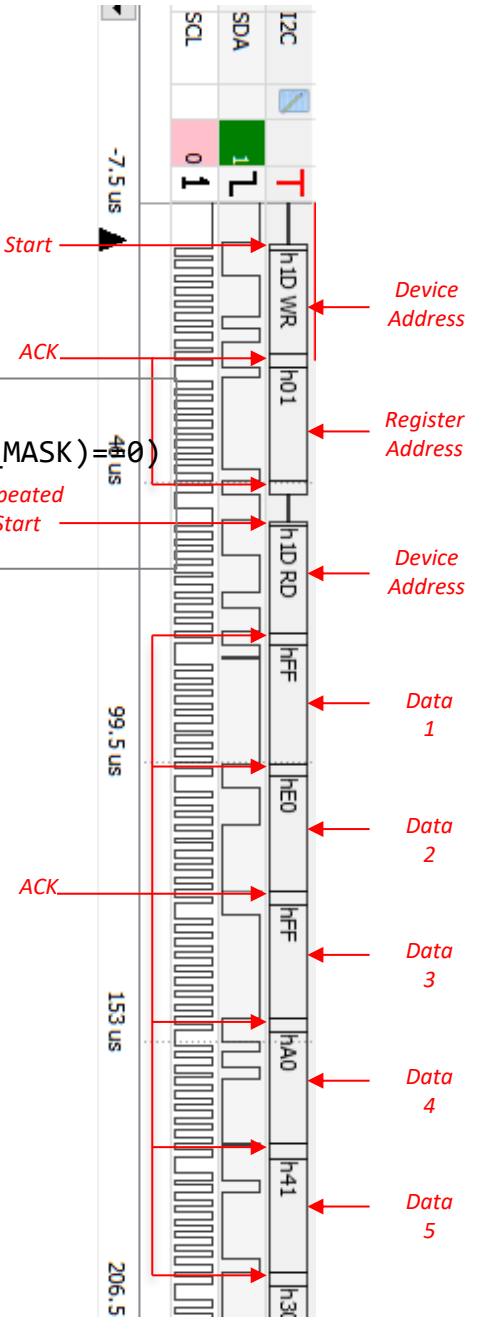
```
I2C_M_STOP;
/*send stop          */
data = I2C0->D;
/*read data          */
```

```
read_full
t()
RAN;              /*set to transmit mode */
_START;           /*send start          */
_setup(dev, address)
>D = dev;         /*send dev address (write)*/
AIT;              /*wait for completion */
>D = address;     /*send read address */
AIT;              /*wait for completion */
_RSTART;          /*repeated start */
>D = (dev|0x1);   /*send dev address (read)*/
AIT;              /*wait for completion */
:C;               /*set to receive mode */
```

```
i2c_wait()
    while ((I2C0->S & I2C_S_IICIF_MASK)==0)
       ;
    I2C0->S |= I2C_S_IICIC_MASK;
```

```
2c_repeated_read(isLastRead)
    ACK;                  /*send ACK after read */
    data = I2C0->D;       /*dummy read          */
    I2C_WAIT;             /*wait for completion */
    data = I2C0->D;       /*read data           */
```





I2C Code Close-Up

```
I2C_TRAN;          /*set to transmit mode */
I2C_M_START;       /*send start           */
I2C0->D = dev;     /*send dev address     */
I2C_WAIT;          /*wait for completion */
I2C0->D = address; /*send read address */
I2C_WAIT;          /*wait for completion */
I2C_M_RSTART;      /*repeated start */
I2C0->D = (dev|0x1); /*send dev address (read) */
I2C_WAIT;          /*wait for completion */
I2C_REC;           /*set to recieve mode */
NACK;              /*set NACK after read */
data = I2C0->D;    /*dummy read          */
I2C_WAIT;          /*wait for completion */
I2C_M_STOP;        /*send stop */
data = I2C0->D;    /*read data */
```

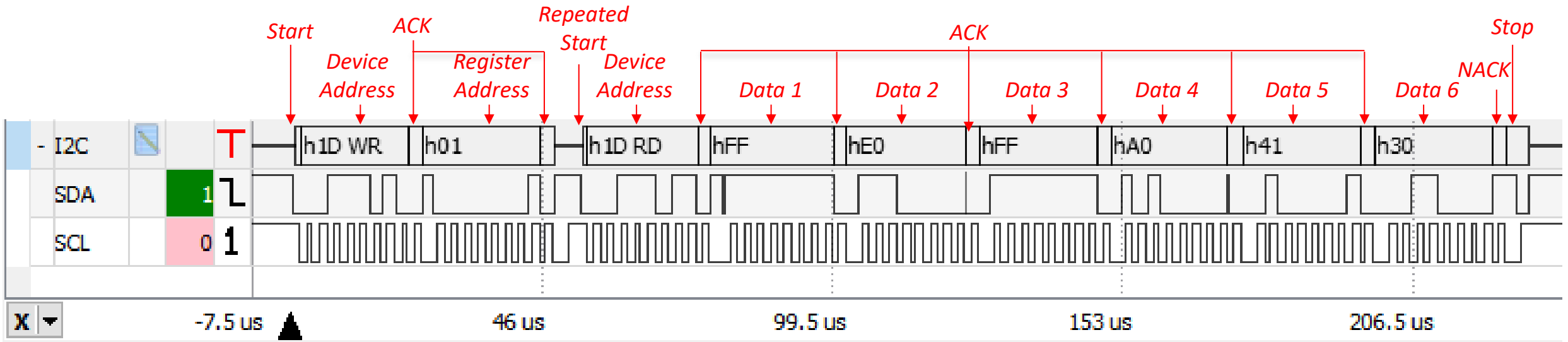
Refine: Accelerometer (with callgraph)

```
i2c_start()

    I2C_TRAN;                   */
    /*set to transmit mode */
    I2C_M_START;                */
    /*send start
```

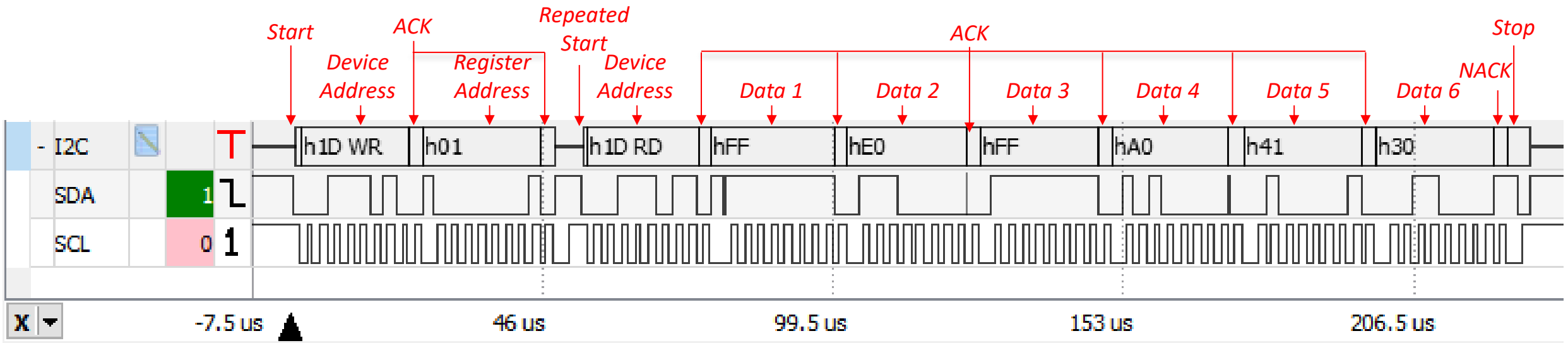
```
i2c_read_setup(dev, address)

I2C0->D = dev;                  */
/*send dev address
I2C_WAIT;
/*wait for completion */

    I2C0->D = address;
    /*send read address */
    I2C_WAIT;
    /*wait for completion */

I2C_M_RSTART;
/*repeated start */
I2C0->D = (dev|0x1);
/*send dev address (read)     */
I2C_WAIT;
/*wait for completion */
I2C_REC;
/*set to receive mode */
```

```
i2c_repeated_read(
    isLastRead)

ACK;
/*ACK after read */
data = I2C0->D;             */
/*dummy read
I2C_WAIT;
/*wait for completion */
I2C_M_STOP;                 */
/*send stop
data = I2C0->D;             */
/*read data
```

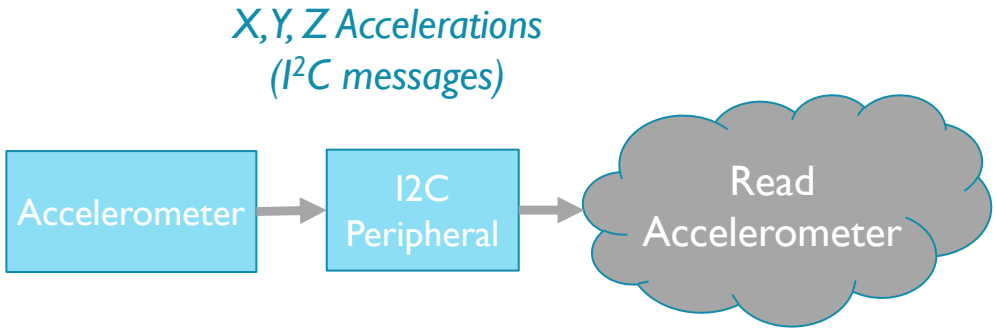
```
I2C_M_STOP;                 */
/*send stop
data = I2C0->D;             */
/*read data
```



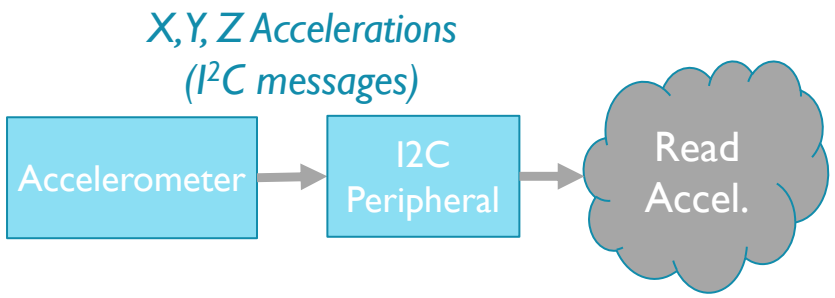
Refine: Accelerometer (with callgraph)

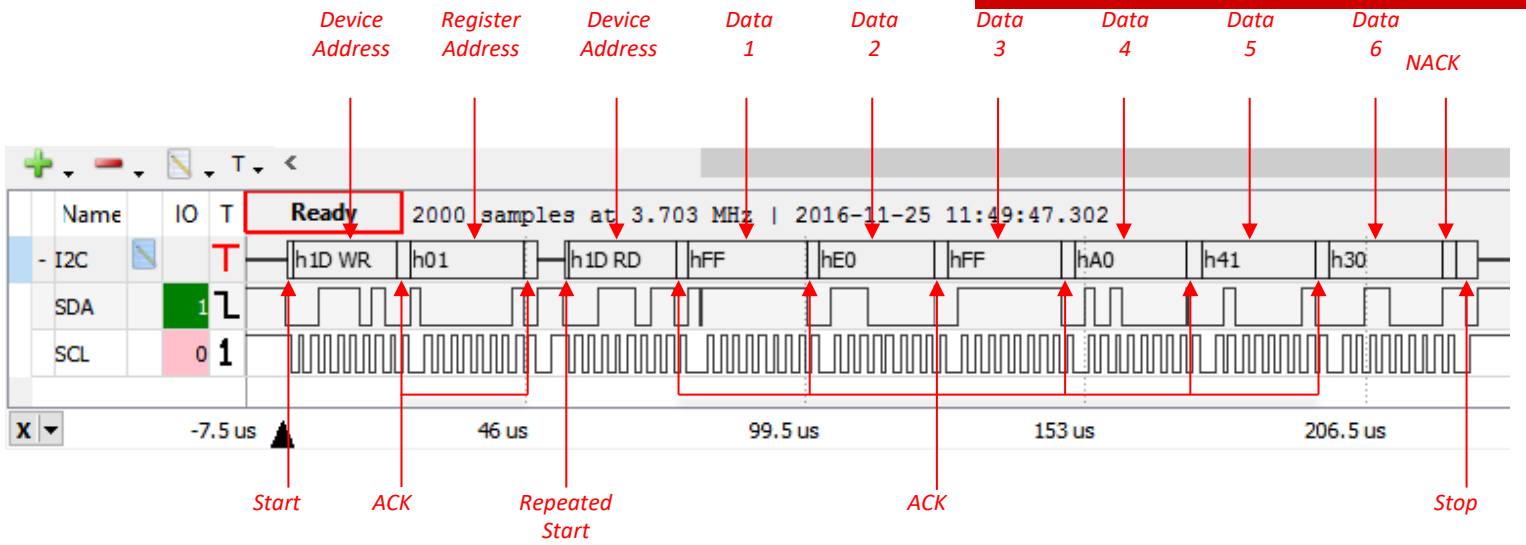


Refine: Accelerometer



Refine: Accelerometer





- How do we trigger the code to run?
  - Send message to read multiple acceleration bytes
- How do we talk to the I2C peripheral?
  - Examine documentation (KL25Z Ref. Man. Chapter 38)
  - Byte-oriented protocol. Software must run per byte.
  - Send start condition, send address byte, wait for ack, send data byte, wait for ack, etc. …. send stop condition
- Does I2C code have any internal delays?
  - Yes – limited by I^2C bus speed
  - How to implement these delays?
  - Try to reclaim that idle time? How much is there?

```
I2C_TRAN;                   /*set to transmit mode */
I2C_M_START;                /*send start    */
I2C0->D = dev;              /*send dev address        */
I2C_WAIT;                   /*wait for completion */
I2C0->D = address;          /*send read address */
I2C_WAIT;                   /*wait for completion */
I2C_M_RSTART;               /*repeated start */
I2C0->D = (dev|0x1);        /*send dev address (read) */
I2C_WAIT;                   /*wait for completion */
I2C_REC;                    /*set to recieve mode */
NACK;                       /*set NACK after read */
data = I2C0->D;             /*dummy read    */
I2C_WAIT;                   /*wait for completion */
I2C_M_STOP;                 /*send stop     */
data = I2C0->D;             /*read data     */
```

```
#define I2C_M_START
I2C0->C1 |= I2C_C1_MST_MASK
#define I2C_M_RSTART
I2C0->C1 |= I2C_C1_RSTA_MASK
#define I2C M STOP
I2C0->C1 &= ~I2C C1 MST MASK
#define I2C_TRAN
I2C0->C1 |= I2C_C1_TX_MASK
#define I2C_REC
I2C0->C1 &= ~I2C_C1_TX_MASK
#define ACK
I2C0->C1 &= ~I2C_C1_TXAK_MASK
#define NACK
I2C0->C1 |= I2C_C1_TXAK_MASK
```

```
#define ACK
I2C0->C1 &= ~I2C_C1_TXAK_MASK

#define I2C_M_RSTART
I2C0->C1 |= I2C_C1_RSTA_MASK

#define I2C M STOP
I2C0->C1 &= ~I2C C1 MST MASK

#define I2C_TRAN
I2C0->C1 |= I2C_C1_TX_MASK

#define I2C_REC
I2C0->C1 &= ~I2C_C1_TX_MASK

#define I2C_M_START
I2C0->C1 |= I2C_C1_MST_MASK

#define NACK
I2C0->C1 |= I2C_C1_TXAK_MASK
```