NC STATE UNIVERSITY

Extracting Architectures from an Implementation (v1.1)

A.G.Dean

ECE 460/560

Embedded System Architectures

Embedded System Architectures © 2023 A.G. Dean

NC STATE UNIVERSITY

Overview

- Why?
  - Build road-maps of system to help debugging
    - Agans' Rule #1: **Understand the System**
    - Agans' Rule #4: **Divide and Conquer**

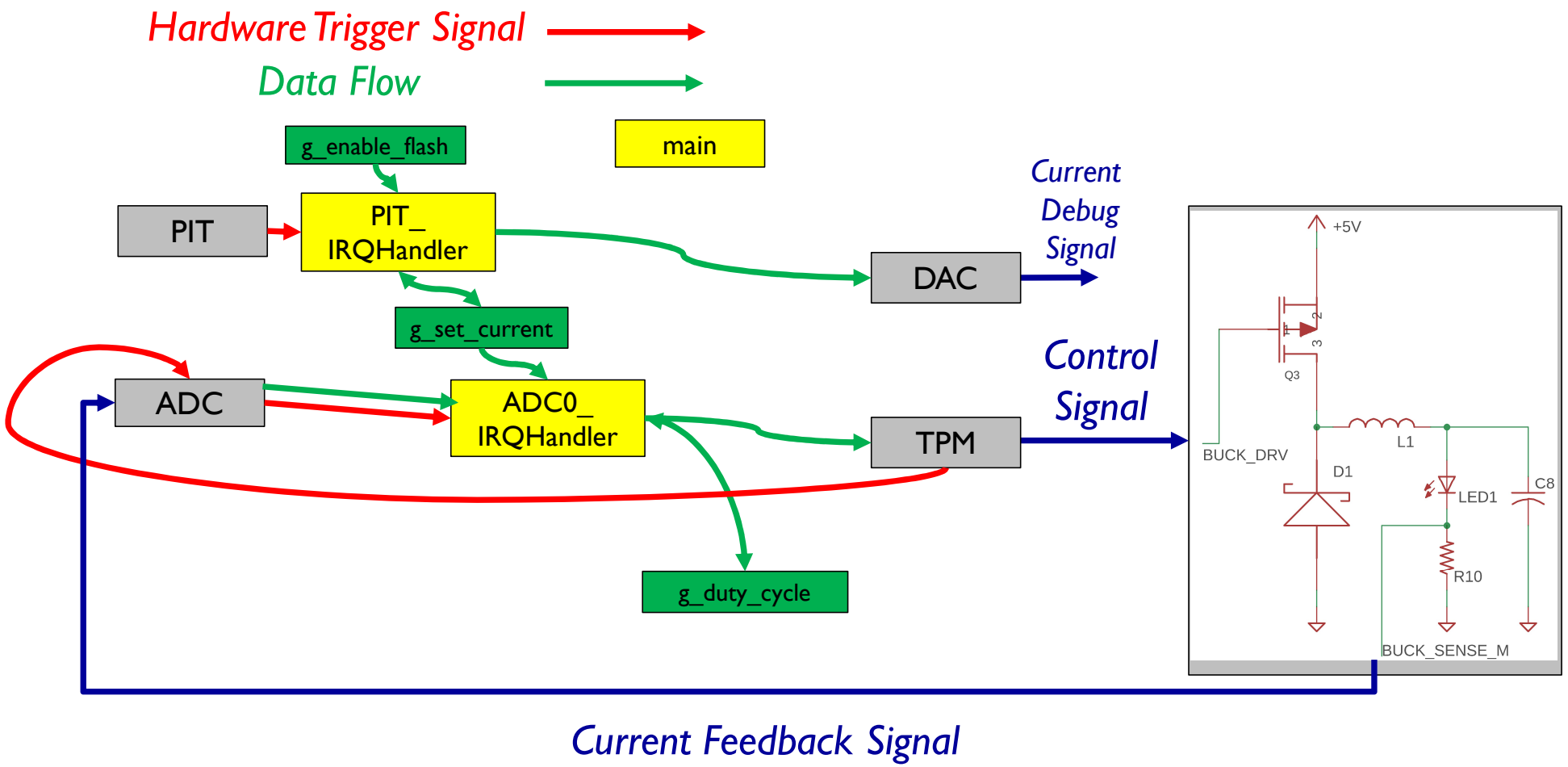
- How?
  - Extract …
    - Hardware aspects
    - Software aspects
  - Put them together

Extracting Architecture from an Implementation

- Hardware
  - Identify hardware input and output signals
    - Type: analog, digital, PWM, etc.
  - Identify which peripherals are used and how (use MCU manual)
    - Which input or output signals are used
    - How is peripheral used?
    - How is peripheral triggered? (HW signal or SW write?)
    - Does the peripheral generate interrupts or control signals for other peripherals?
- Software
  - Is there an explicit kernel/scheduler?
  - Identify threads and handlers (vectors in startup.s)
    - How are they triggered to execute?
    - What data is transferred one-way? (writer doesn't read the data). Where and how?
    - What data is shared two-way? (e.g. read/modify/write) Where and how?
  - Code structures for key threads and functions
    - Determine each thread's/handler's call graph
    - Determine each function's control flow graph (flow chart)

Embedded System Architectures © 2020 A.G. Dean

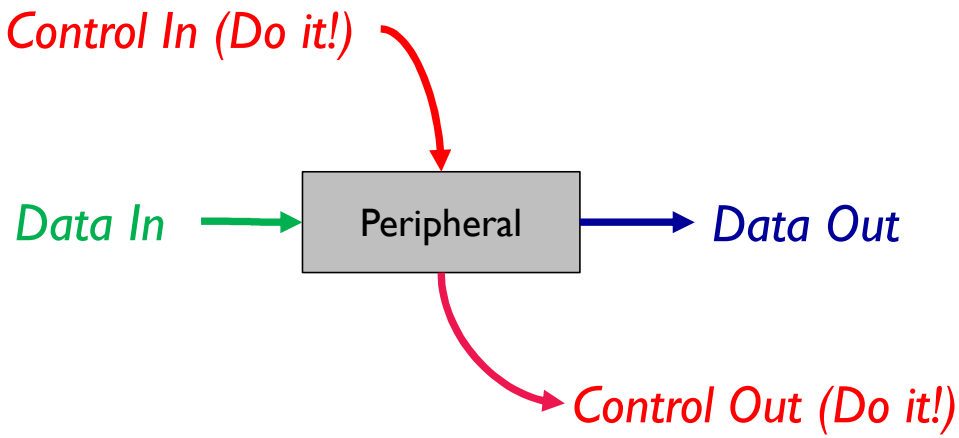
Reminder: Flashing Constant Current LED Driver



Embedded System Architectures © 2020 A.G. Dean

Hardware Aspects

- Identify system hardware input, outputs
  - Signal type (analog, digital, PWM, etc.)
  - Peripherals connected to those signals
- Examine peripheral use (cf. MCU manual)
  - What does peripheral do?
- Examine inputs and outputs
  - Data
  - Control
    - In: How is peripheral triggered?
      - HW signal, SW write or free-running (asynchronous)?
    - Out: What does the peripheral trigger?
      - Does the peripheral generate interrupts or trigger signals for other peripherals?

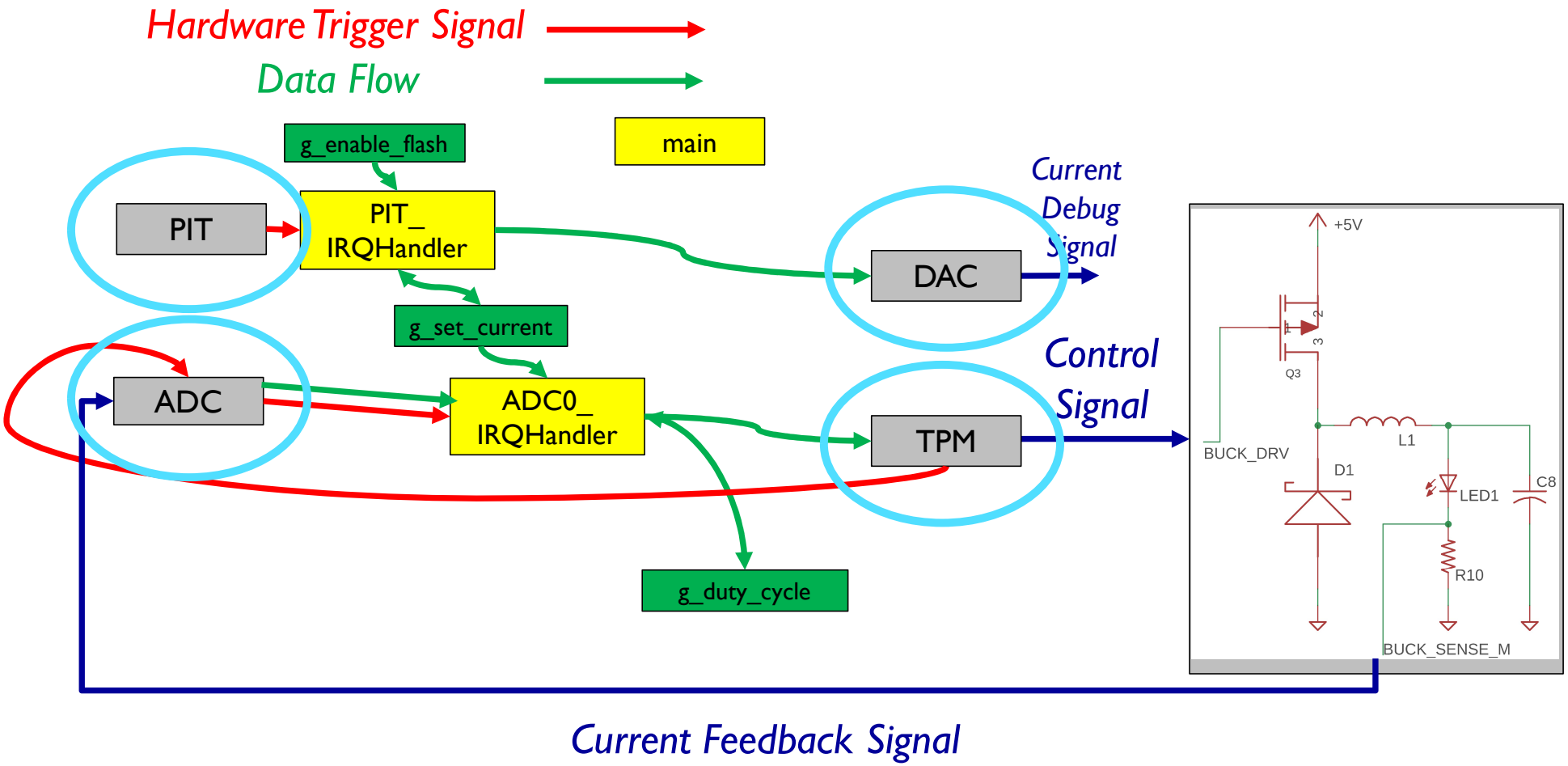


		Data	Control
Inputs	HW		
	SW		
Outputs	HW		
	SW		

Embedded System Architectures © 2020 A.G. Dean

NC STATE UNIVERSITY

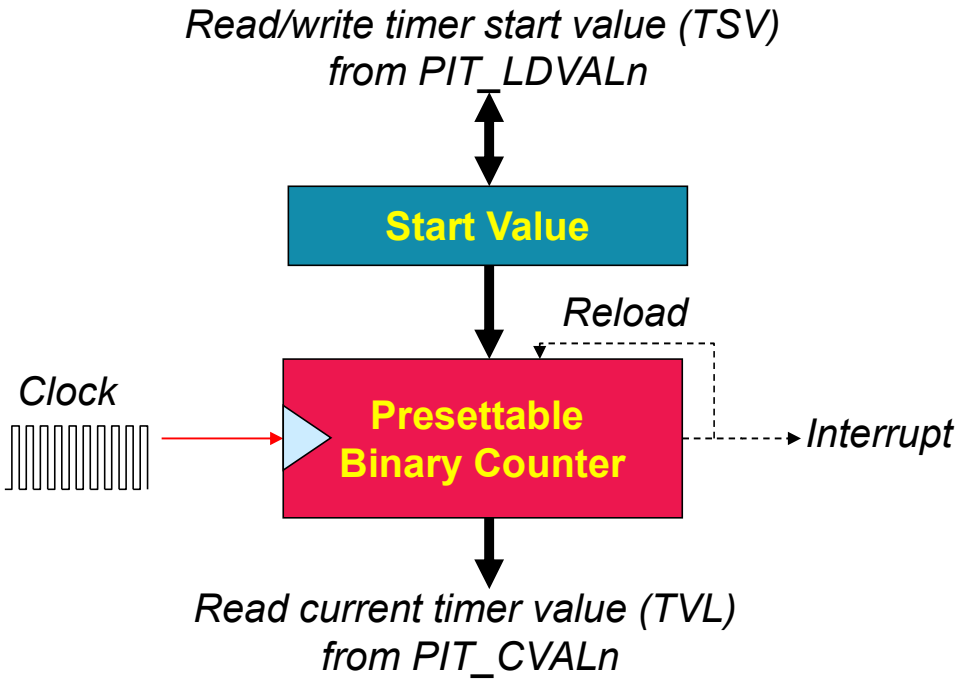
Hardware/Software Interactions



Embedded System Architectures © 2020 A.G. Dean

NC STATE UNIVERSITY

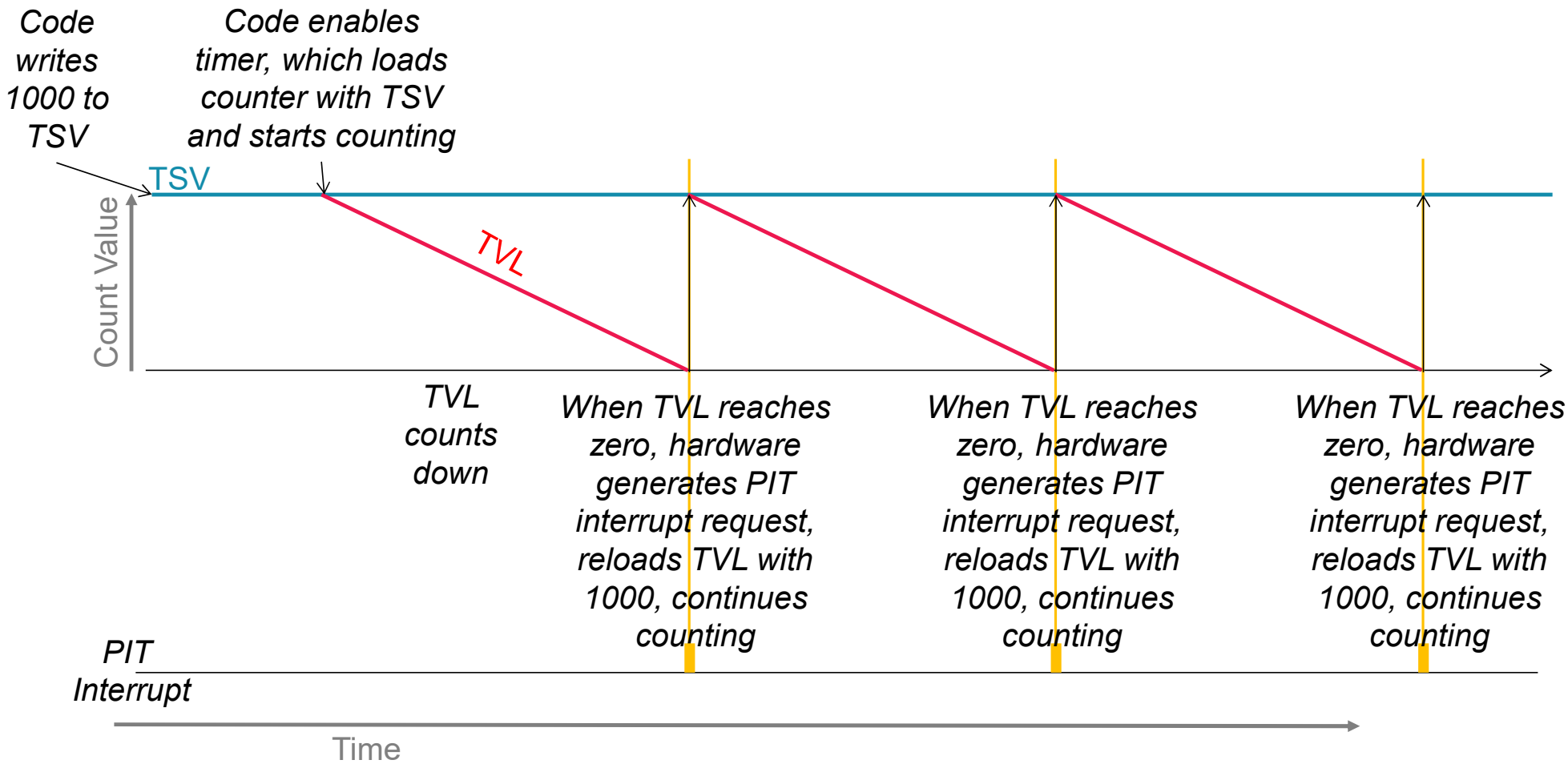
Periodic Interrupt Timer



- Generates periodic interrupts using a 32-bit counter
- Load start value (32-bit) from LDVAL
- Counter decrements with each clock pulse
  - Fixed clock source for PIT - Bus Clock from Multipurpose Clock Generator - e.g. 24 MHz
- When timer value (CVAL) reaches zero
  - Generates interrupt
  - Reloads timer with start value

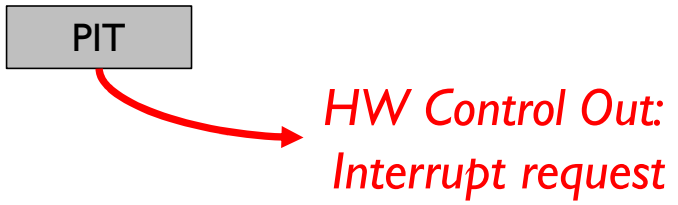
NC STATE UNIVERSITY

Periodic Interrupt Timer



Example: Periodic Interrupt Timer

- After initialization, timer's counter counts down
- When reaching zero (end of cycle),
  - Generates interrupt request
  - Reloads counter
- Resumes counting down



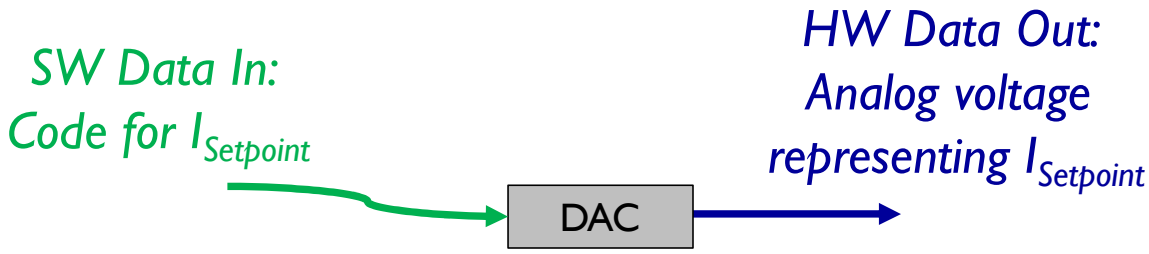
		Data	Control
Inputs	HW		
	SW		
Outputs	HW		Timer overflow signal (periodic)
	SW		

Embedded System Architectures © 2020 A.G. Dean

NC STATE UNIVERSITY

Example: Digital to Analog Converter

- Software writes code for DAC voltage to DAC data register
- DAC generates analog voltage based on data input

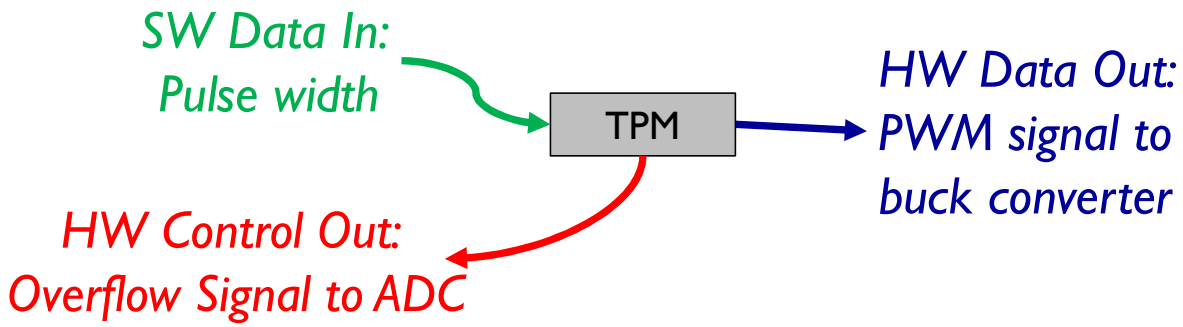


		Data	Control
Inputs	HW		
	SW	Code for $I_{Setpoint}$	
Outputs	HW	Analog voltage representing $I_{Setpoint}$	
	SW		

Embedded System Architectures © 2020 A.G. Dean

Example: Timer/PWM Module

- After initialization, timer runs independently of software
  - Generates PWM signal based on compare register and counter value
  - Generates HW control signal at end of cycle (e.g. overflow or underflow)
- To update desired pulse width, software writes value to compare register

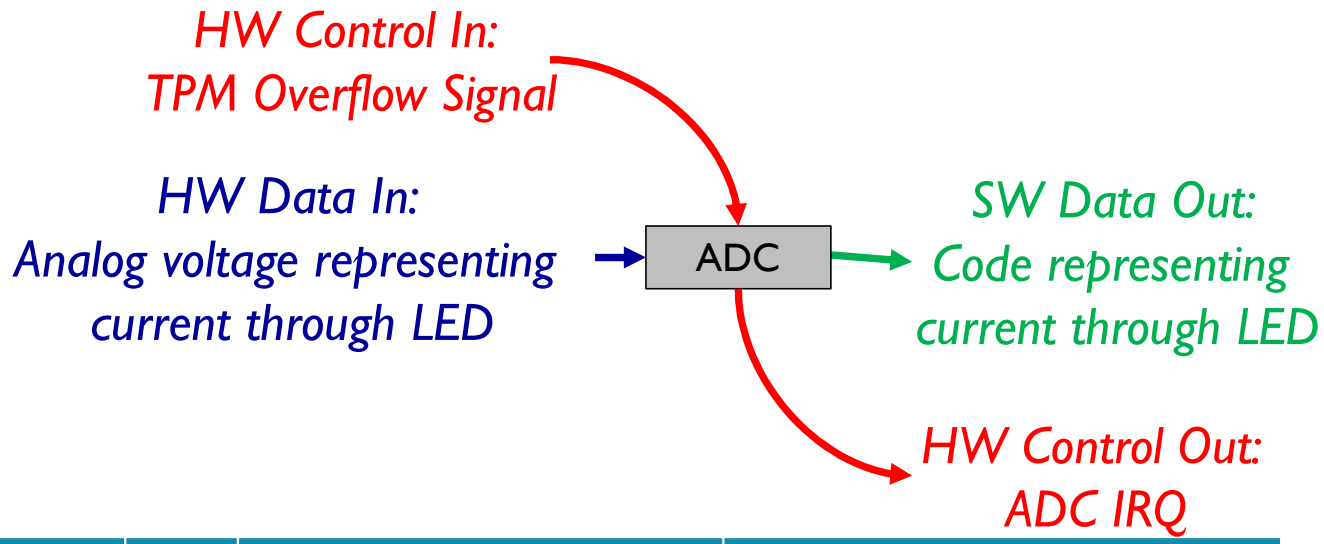


		Data	Control
Inputs	HW		
	SW	Pulse width value	
Outputs	HW	PWM signal	Timer overflow signal (periodic)
	SW		

Embedded System Architectures © 2020 A.G. Dean

Example: Analog to Digital Converter

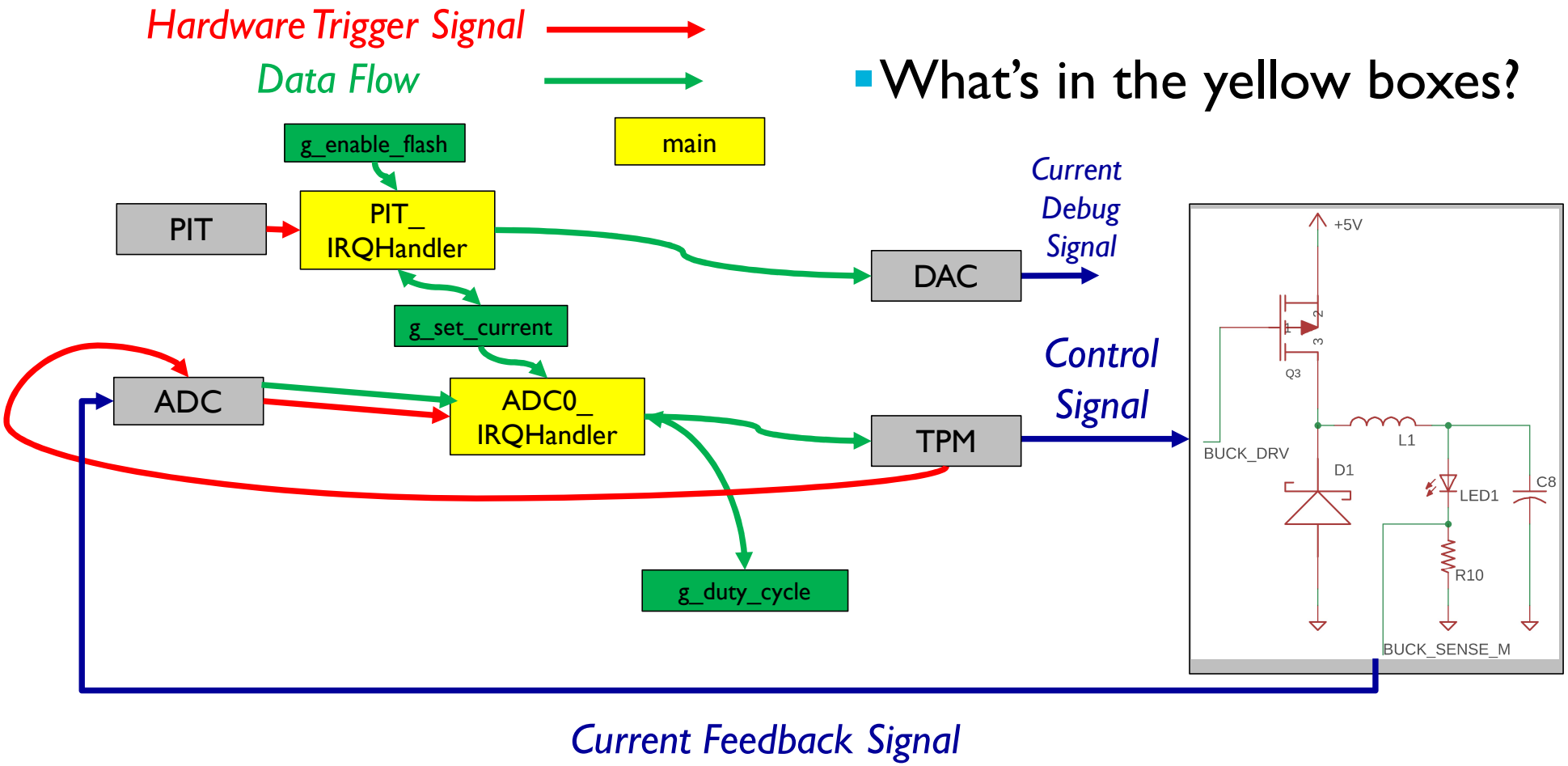
- Timer's HW control signal at end of cycle triggers ADC to start conversion
- ADC converts input (analog voltage representing LED current) to digital value
- ADC generates interrupt request when conversion is done
- ADC IRQ Handler (ISR) reads data from ADC



		Data	Control
Inputs	HW	Analog voltage representing current through LED	Timer overflow signal (periodic)
	SW		
Outputs	HW		ADC interrupt request
	SW	Digital code representing current through LED	

Embedded System Architectures © 2020 A.G. Dean

Hardware/Software Interactions



Embedded System Architectures © 2020 A.G. Dean

NC STATE UNIVERSITY

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 10
#define DAC_RESOLUTION 4096

// #define MA_TO_DAC_CODE(i) (i*2.2*DAC_RESOLUTION/V_REF_MV) // Introduces timing delay and in-
teresting 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 "LEDS.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_SCAL-
ING_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, meas-
ured_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 ac
        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] = A

#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
```

Software Aspects - Overview

- Threads and handlers
  - Which parts of the program can be running concurrently?
  - How are threads and handlers triggered?
  - How do threads and handlers interact with each other?
- Code structures within key threads and functions
  - Which functions can each thread call?
  - What is the possible flow of control within each function?

Embedded System Architectures © 2020 A.G. Dean

Thread Concepts

- Threads: Parts of the program which can be running concurrently
  - Execution sequence of thread A and thread B is independent
    - Can interleave execution sequence arbitrarily as long as A goes in order and B goes in order
    - Sometimes we will add constraints
  - Each thread has a "next instruction" to execute
    - Thread = "Execution context"
    - Each thread has its own program counter and function call stack so threads can proceed independently
- Threads vs. subroutine calls vs. ISRs
  - Subroutine calls (and software interrupts) are triggered by a specific instruction, are synchronous (synchronized) with program execution
  - Caller function always blocks (pauses) until callee or ISR finishes and relinquishes CPU control.
  - ISR is triggered by hardware event, is asynchronous (not synchronized) with program execution

Embedded System Architectures © 2020 A.G. Dean

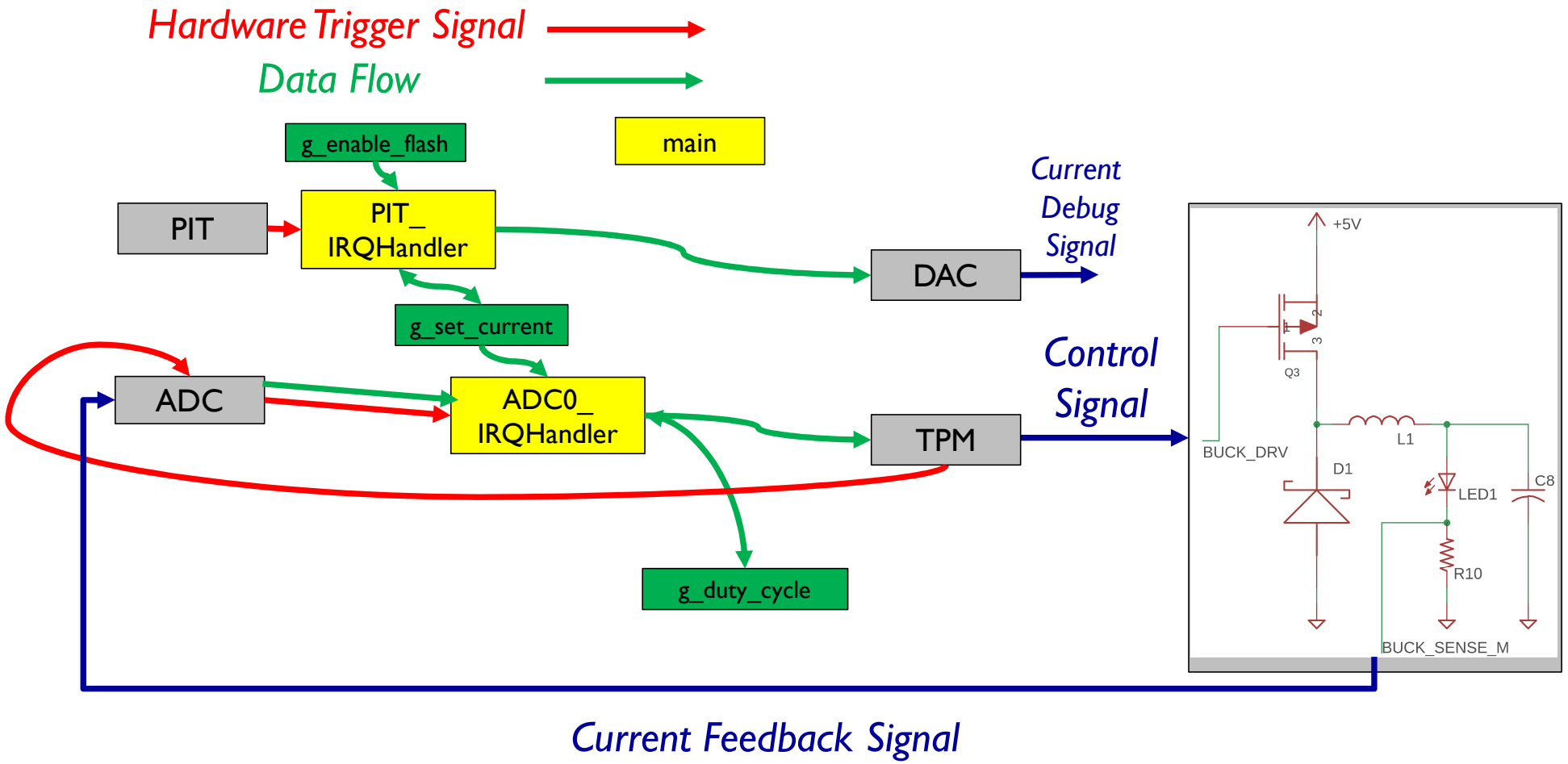
Software Aspects

- Threading
  - Is the system single-threaded (main + ISRs)?
  - Is the system multi-threaded? Need an explicit scheduler (in kernel)
    - Look in main for call for **osKernelStart**
- Identify threads and handlers
  - Threads created by calls to **osThreadNew**
  - Handler vectors identified in startup_MKL25Z4.s
- How are threads triggered to execute?
  - Hardware?
  - Other threads?
  - Kernel's scheduler (e.g. time-based)?
- What data is transferred…
  - One-way? Writer doesn't read the data, reader doesn't write the data.
  - Two-way? Thread reads data into register, modifies it, writes it back to memory.

Embedded System Architectures © 2020 A.G. Dean

NC STATE UNIVERSITY

Looking into the Software Boxes



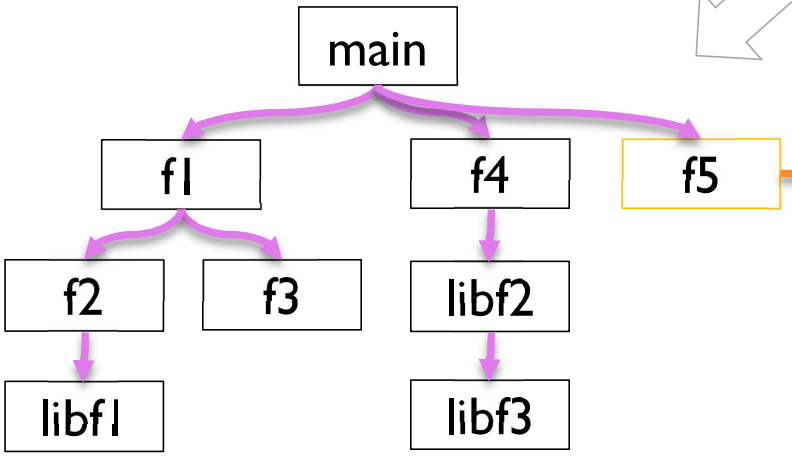
Embedded System Architectures © 2020 A.G. Dean

Structures Representing Code

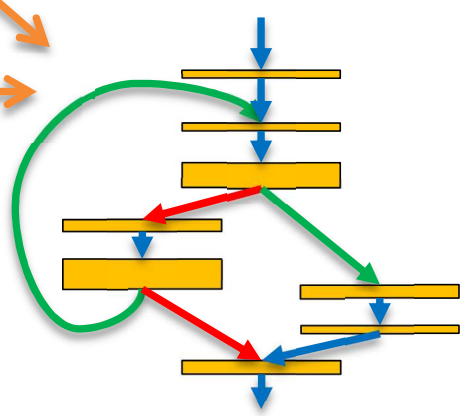
Source Code (.c)

```
f1() {
  f2();
  f3();
}

f2() {
  …
}
f3() {
  …
}

f4() {
  …
}
f5() {
  …
}

main() {
  f1();
  f4();
  f5();
}
```

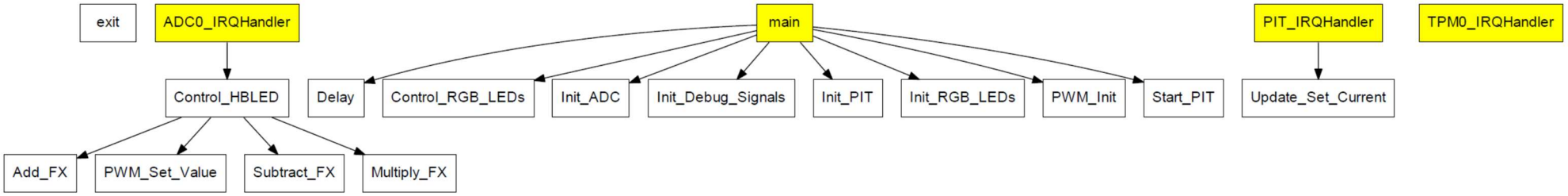
Function Call Graph of One Thread



Control Flow Graph of One Function's Code

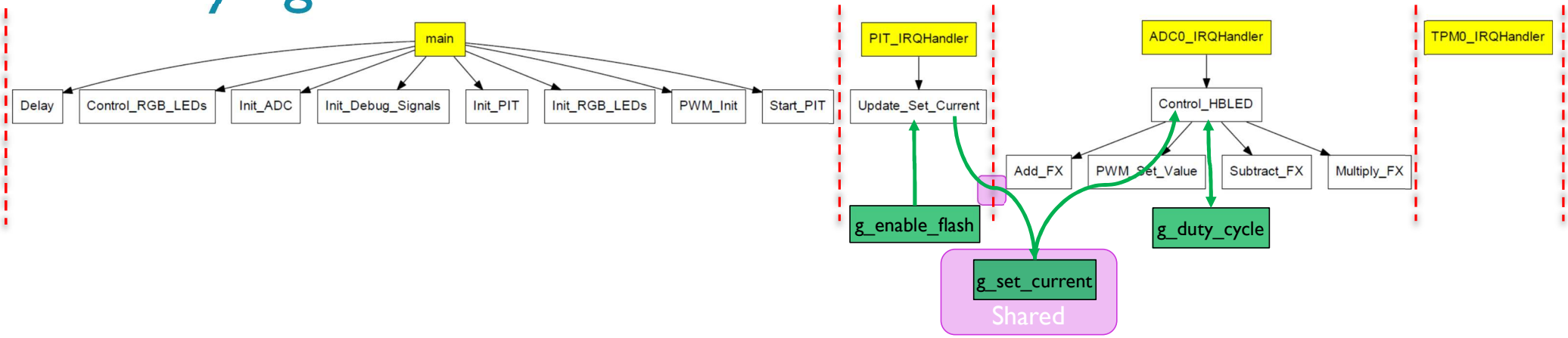


Function Call Graph



- Directed graph which shows function call relationships
  - Edges connect caller function (at tail of edge) to callee function (at head)
- Types
  - Static – show all possible function calls
  - Dynamic – shows which function calls actually occurred during a particular program run

Identifying Shared Data

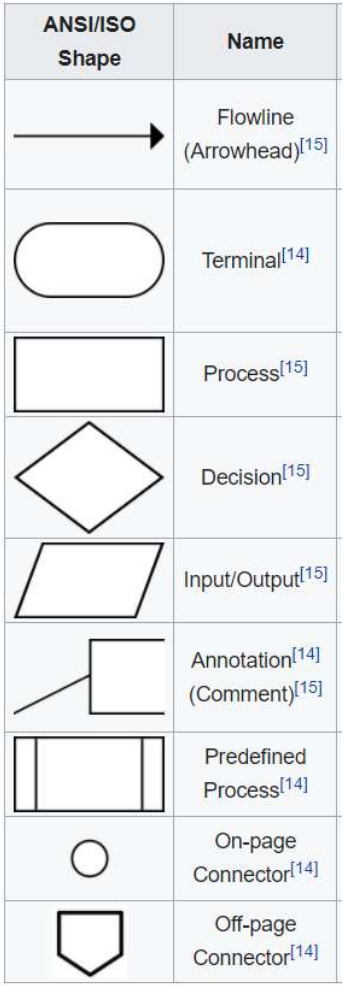


- Definitions
  - Resource: program variable, hardware peripheral, …
  - Used: read, written or both
  - Execution context: separate flow of program control with own program counter.
    - Main thread: main() function and every subroutine it calls (recursive: and every subroutine which *those* call)
    - Additional threads possible if scheduler supports them
    - Exception handlers, including ISRs
- A resource is shared if it **may be used by multiple execution contexts**
  - thread and thread
  - thread and handler
  - handler and handler
- "Slice and dice" the program
  - How deep is subroutine call nesting? Horizontal slices
  - Which execution context? Vertical slices
- ***Only variables potentially accessed by multiple execution context are shared***

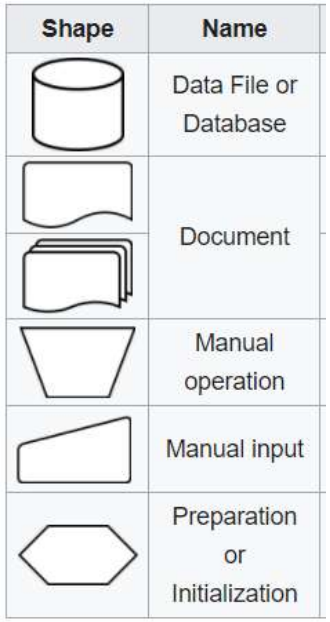
NC STATE UNIVERSITY

Flowcharts and Control Flow Graphs

- Both show possible paths of execution ("control flow") within a program or function
  - https://en.wikipedia.org/wiki/Flowchart
- Control flow graph is more detailed, has …
  - Basic blocks of machine instructions
  - Directed control flow edges
- We'll cover CFGs in ECE 461/561

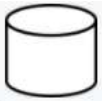
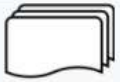
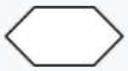


ANSI/ISO Shape	Name
	Flowline (Arrowhead)[15]
	Terminal[14]
	Process[15]
	Decision[15]
	Input/Output[15]
	Annotation[14] (Comment)[15]
	Predefined Process[14]
	On-page Connector[14]
	Off-page Connector[14]

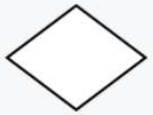
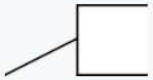


Shape	Name
	Data File or Database
	Document
	Manual operation
	Manual input
	Preparation or Initialization

ANSI/ISO Symbol Descriptions

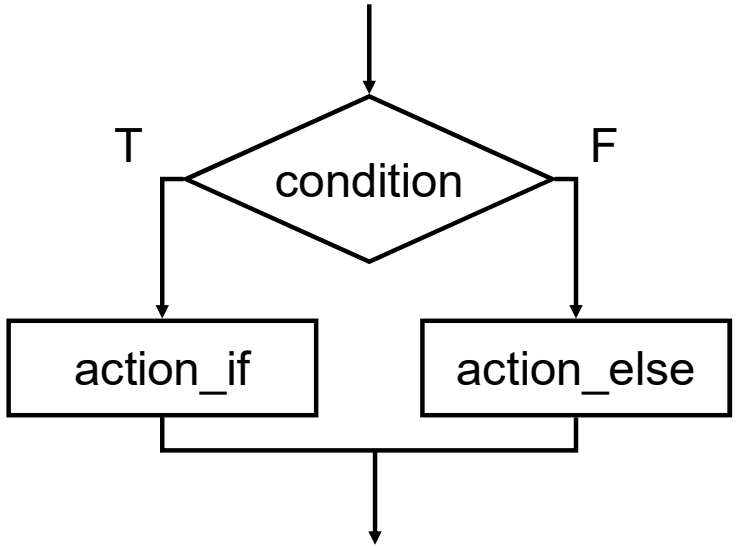
	Data File or Database	Data represented by a cylinder (disk drive).
	Document	Single documents represented a rectangle with a wavy base.
		Multiple documents represented stacked rectangle with a wavy base.
	Manual operation	Represented by a trapezoid with the longest parallel side at the top, to represent an operation or adjustment to process that can only be made manually.
	Manual input	Represented by quadrilateral, with the top irregularly sloping up from left to right, like the side view of a keyboard.
	Preparation or Initialization	Represented by an elongated hexagon, originally used for steps like setting a switch or initializing a routine.

- https://en.wikipedia.org/wiki/Flowchart

	Flowline (Arrowhead)[15]	Shows the process's order of operation. A line coming from one symbol and pointing at another.[14] Arrowheads are added if the flow is not the standard top-to-bottom, left-to right.[15]
	Terminal[14]	Indicates the beginning and ending of a program or sub-process. Represented as a stadium,[14] oval or rounded (fillet) rectangle. They usually contain the word "Start" or "End", or another phrase signaling the start or end of a process, such as "submit inquiry" or "receive product".
	Process[15]	Represents a set of operations that changes value, form, or location of data. Represented as a rectangle.[15]
	Decision[15]	Shows a conditional operation that determines which one of the two paths the program will take.[14] The operation is commonly a yes/no question or true/false test. Represented as a diamond (rhombus).[15]
	Input/Output[15]	Indicates the process of inputting and outputting data,[15] as in entering data or displaying results. Represented as a rhomboid.[14]
	Annotation[14] (Comment)[15]	Indicating additional information about a step in the program. Represented as an open rectangle with a dashed or solid line connecting it to the corresponding symbol in the flowchart.[15]
	Predefined Process[14]	Shows named process which is defined elsewhere. Represented as a rectangle with double-struck vertical edges.[14]
	On-page Connector[14]	Pairs of labeled connectors replace long or confusing lines on a flowchart page. Represented by a small circle with a letter inside.[14][18]
	Off-page Connector[14]	A labeled connector for use when the target is on another page. Represented as a home plate-shaped pentagon.[14][18]

Control Flow: If/Else

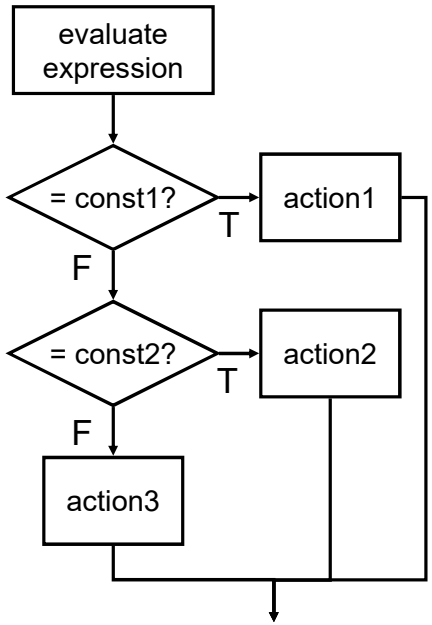
```
if (x){
    y++;
} else {
    y--;
}
```



NC STATE UNIVERSITY

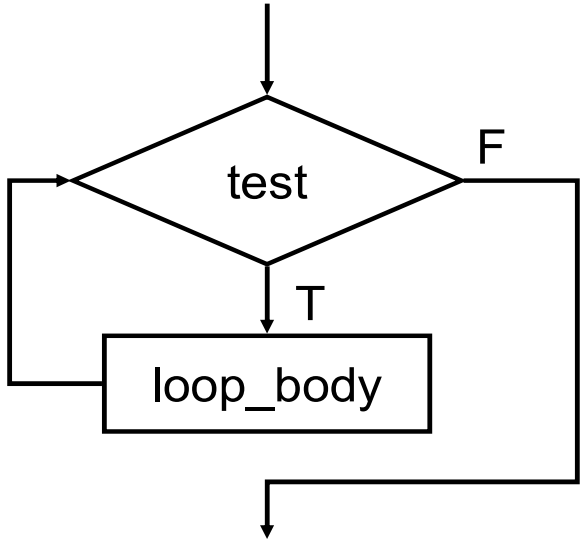
Control Flow: Switch

```
switch (x) {
  case 1:
    y += 3;
    break;
  case 31:
    y -= 5;
    break;
  default:
    y--;
    break;
  }
```



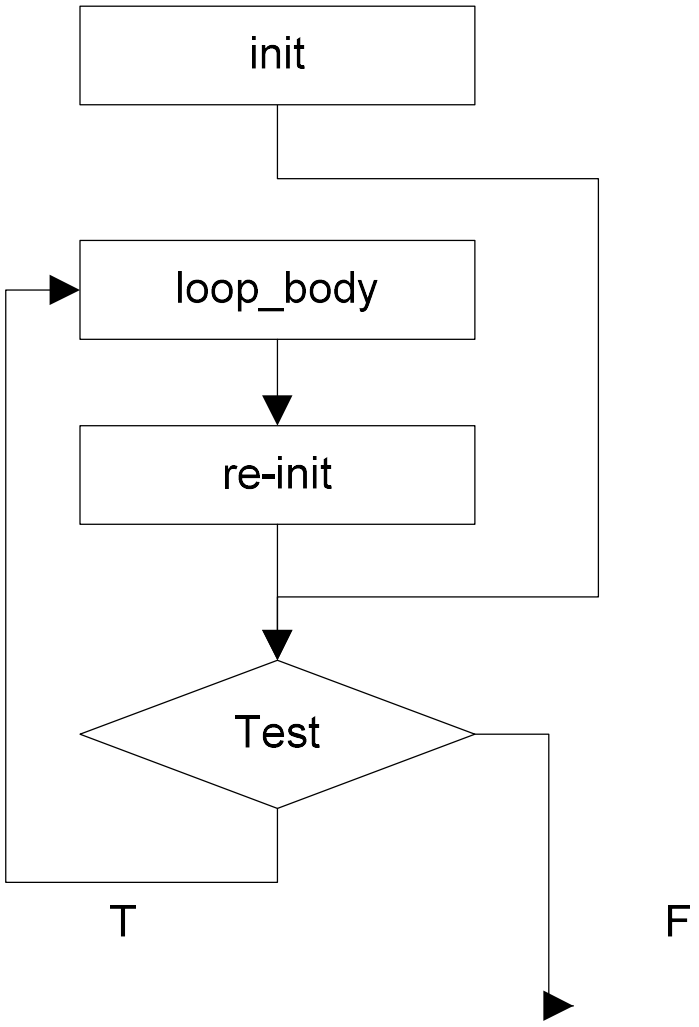
Iteration: While

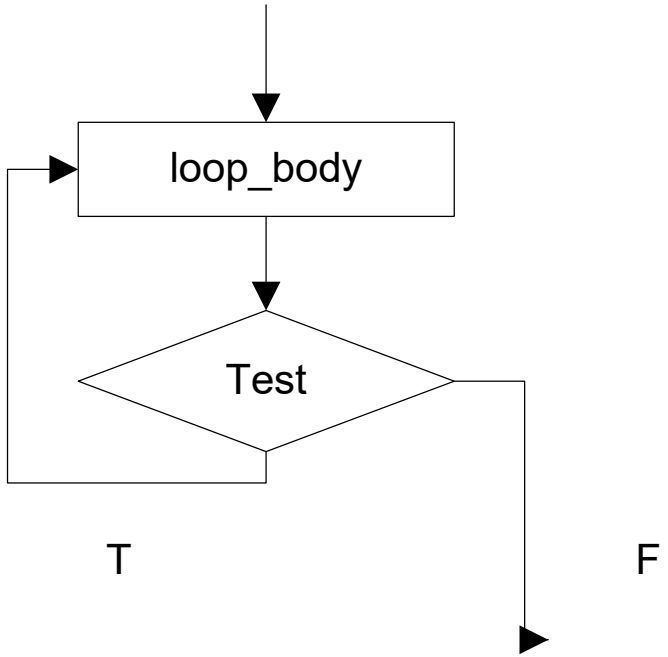
```
while (x<10) {
    x = x + 1;
}
```



Iteration: For

```
for (i = 0; i < 10; i++){
    x += i;
}
```



Iteration: Do/While

```
do {
    x += 2;
} while (x < 20);
```