The ARM Architecture

EEL 4745C: Microprocessor Applications II

Fall 2022

Md Jahidul Islam

July 10, 2022

The ARM Way: Background

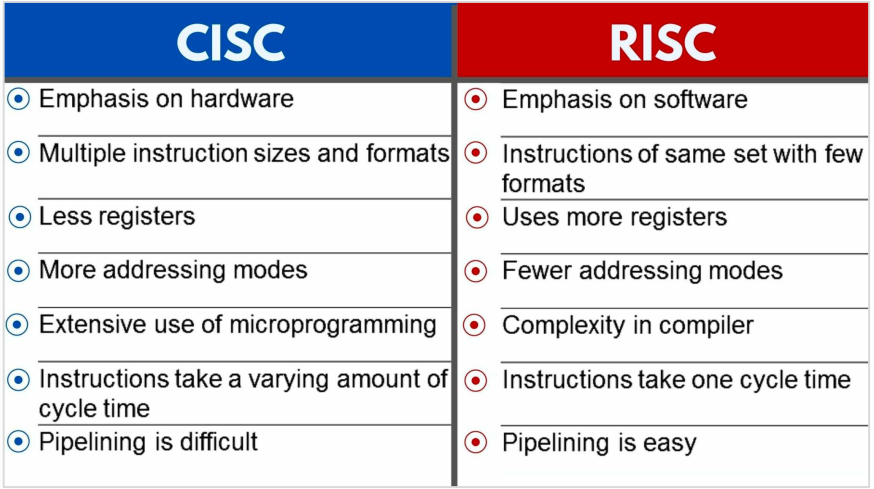
⇒ ARM the company vs ARM the processor

- A brief history of ARM: Part I, Part II
- Founded in 1990 as Advanced RISC Machines Ltd
  - Joint venture of Acorn Computers, Apple and VLSI Tech
  - TI and Nokia got onboard in early days
- Fabless company
  - Licenses design to other parties
  - AMD, Apple, NXP, Qualcomm, Samsung, TI
  - Loving the new M1 chips from Apple? See this video!
- See this talk from Dave Jaggar himself.

The ARM Way: RISC Features

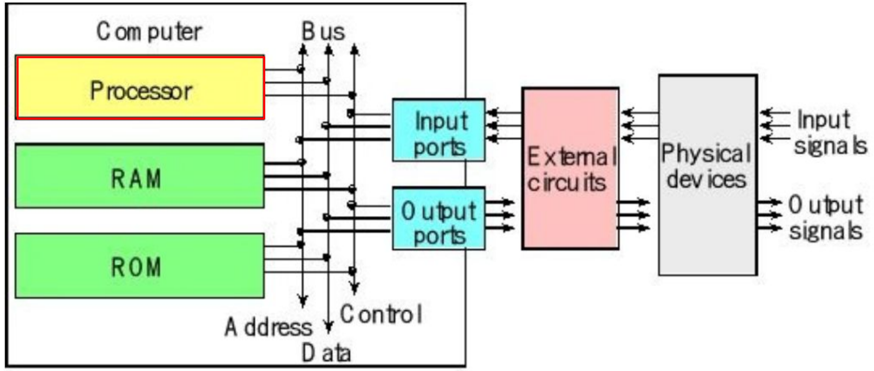
⇒ RISC (Reduced Instruction Set Computer) vs CISC:

- RISC emphasizes on software with
  - Small number of standardised instructions
  - Single clock-cycle instructions
  - Highly pipelined, makes heavy use of RAM
- Why the fuzz?
  - Low power, compact, and low cost
  - No fan, no heat sinks
- Perfect choice for custom RTOS (Real-time OS)
  - Mobile devices, IoT devices
  - Robotics!



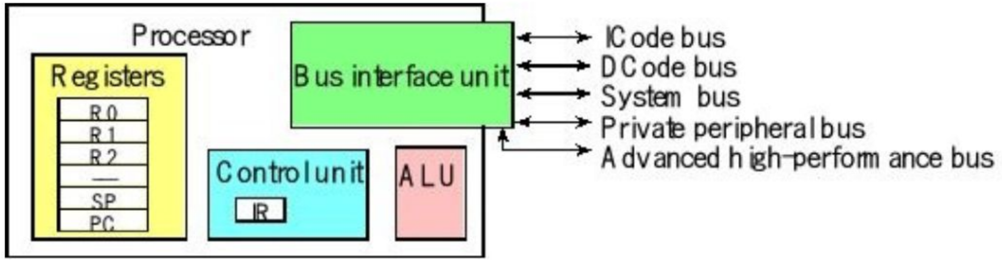
CISC	RISC
Emphasis on hardware	Emphasis on software
Multiple instruction sizes and formats	Instructions of same set with few formats
Less registers	Uses more registers
More addressing modes	Fewer addressing modes
Extensive use of microprogramming	Complexity in compiler
Instructions take a varying amount of cycle time	Instructions take one cycle time
Pipelining is difficult	Pipelining is easy

The Big Picture



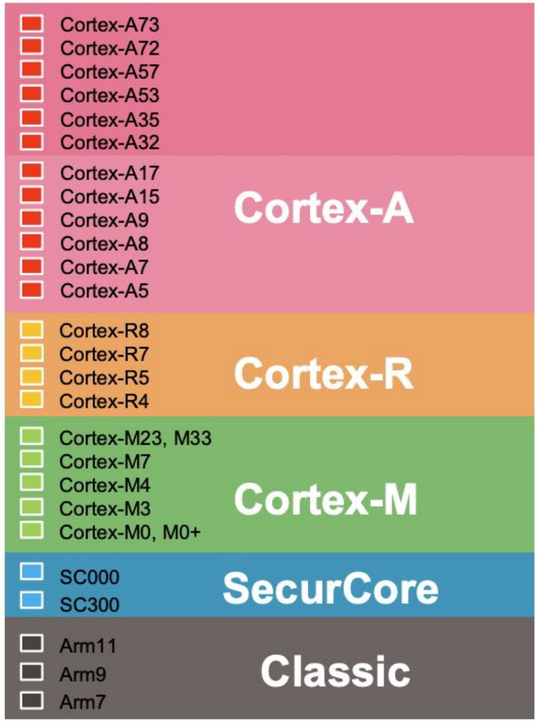
Regular OS	*Real-time OS*
Complex	Simple
Best effort	Guaranteed response
Fairness	Strict timing constraints
Average bandwidth	Minimum and maximum limits
Unknown components	Known components
Unpredictable behavior	Predictable behavior
Plug and play	Upgradable

OS is the software packages that manages all resources and hw-sw communication



Major components of a processor: CU, BIU, ALU, and Registers

Arm Family of Processors



Cortex
Low-Power Leadership from ARM

⇒ Cortex-A series (Application)

- High performance processors capable of full OS support
- Smartphones, digital TV, smart books

⇒ Cortex-R series (Real-time)

- High performance and reliability for real-time applications
- Automotive braking system, powertrains

⇒ Cortex-M series (Microcontroller)

- Cost-sensitive solutions for deterministic applications
- Microcontrollers, smart sensors

Arm Family of Processors

⇒ **ARM Architecture**

*** Describes the details of instruction set, memory map, and programming model*

*** See the Architecture Reference Manual*

⇒ **ARM Processor**

*** Developed using one of the architectures*

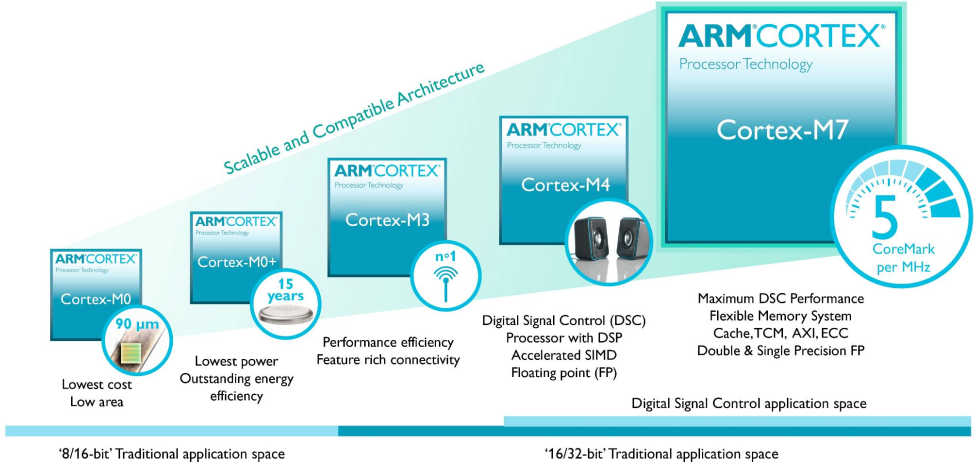
*** With platform-specific implementation*

*** See the Technical Reference Manual*

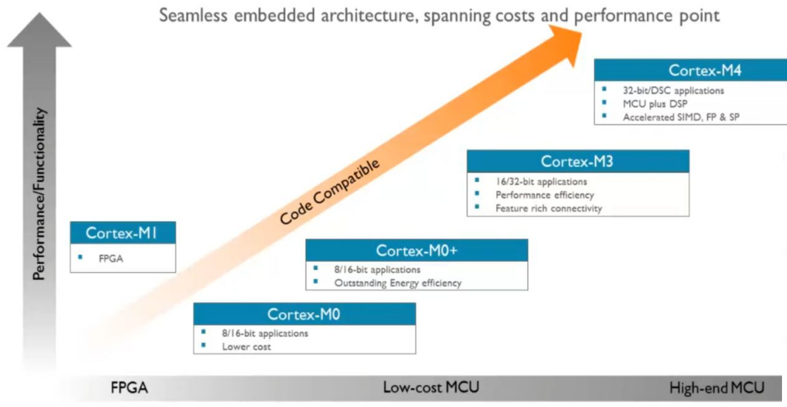
Arm Cortex-M Features

⇒ **Cortex-M series: M0, M3, M4, M7, M22, M23**

- Low cost, low power
- Fast interrupt response
- Inter-process communication
- **Energy-efficiency**
    - lower energy cost, longer battery life
- **Smaller code (Thumb mode instructions)**
    - Lower silicon costs
- **Ease of use**
    - Faster software development and reuse
    - Embedded and robotics applications


Scalable and Compatible Architecture
ARM CORTEX Processor Technology Cortex-M0
90 µm
Lowest cost
Low area
ARM CORTEX Processor Technology Cortex-M0+
15 years
Lowest power
Outstanding energy efficiency
ARM CORTEX Processor Technology Cortex-M3
n°1
Performance efficiency
Feature rich connectivity
ARM CORTEX Processor Technology Cortex-M4
Digital Signal Control (DSC)
Processor with DSP
Accelerated SIMD
Floating point (FP)
ARM CORTEX Processor Technology Cortex-M7
5 CoreMark per MHz
Maximum DSC Performance
Flexible Memory System
Cache, TCM, AXI, ECC
Double & Single Precision FP
Digital Signal Control application space
'8/16-bit' Traditional application space
'16/32-bit' Traditional application space

Arm Cortex-M Core Comparison

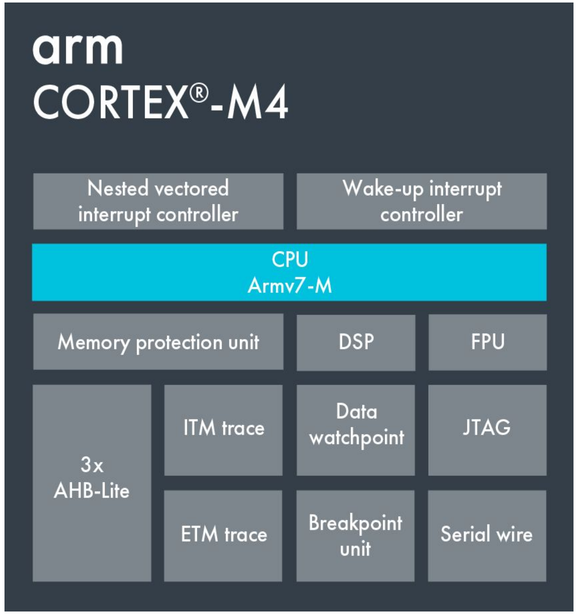


Cortex-M	Thumb	Thumb-2	HW MPY	HW DIV	Saturated math	DSP-extensions	FPU	ARM architecture
Cortex-M0	Most	Subset	1 or 32 cycle	No	No	No	No	ARMv6-M Von Neumann
Cortex-M0+	Most	Subset	1 or 32 cycle	No	No	No	No	ARMv6-M Von Neumann
Cortex-M1	Most	Subset	3 or 33 cycle	No	No	No	No	ARMv6-M Von Neumann
Cortex-M3	Entire	Entire	1 cycle	2-12 cycles	Yes	No	No	ARMv7-M Harvard
Cortex-M4	Entire	Entire	1 cycle	2-12 cycles	Yes	Yes	Optional **Yes for MSP432**	ARMv7E-M Harvard

ARM Cortex-M4 Core

⇒ Implements ARMv7-M Architecture

- 32 bit RISC CPU with 16 integer registers
- Two instruction sets
  - ARM (Aarch32): 32 bit instructions, full access to register file
  - Thumb-2 (T32): 16 bit instructions, 12-cycle interrupt latency
- Optional Floating Point Unit (FPU)
- Two operating states
  - Privileged state: default state of CPU at reset
  - Non-privileged state: some instructions/memory N/A
- Two operating modes
  - Thread mode: standard mode (process stack / main stack)
  - Handler mode: interrupt (privileged mode, main stack)



Arm Cortex-M4 Instruction Architecture

Cortex-M4

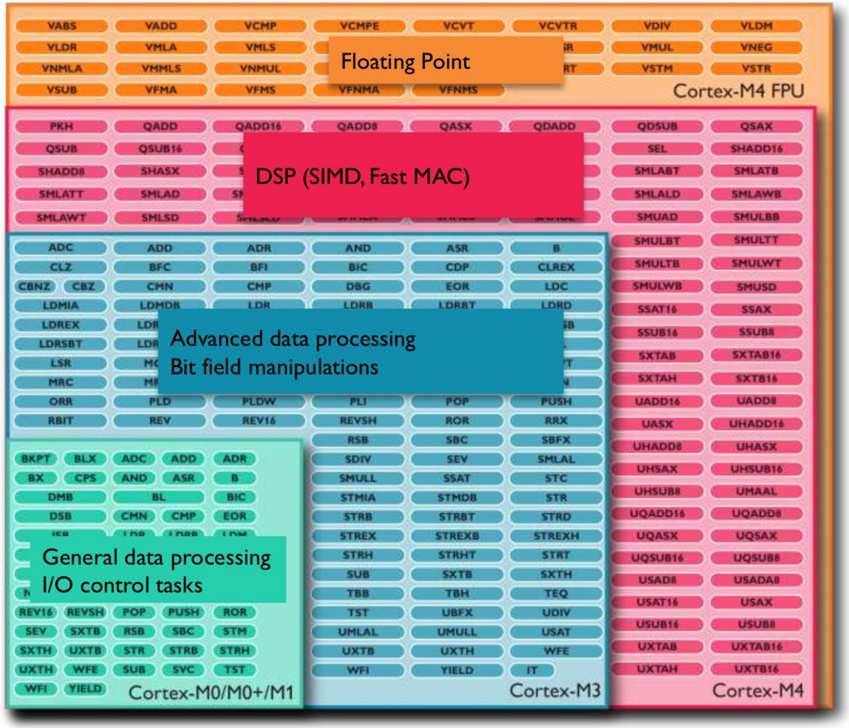
M4: *Adds DSP instructions, optional floating point unit (32.82 μW/MHz)*

Cortex-M3

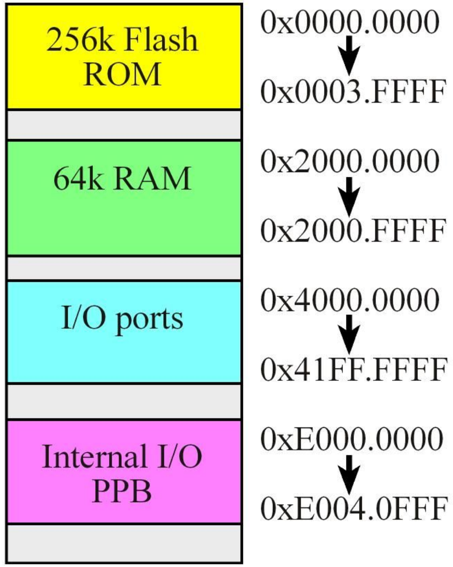
M3: *Thumb and Thumb-2 instruction single-cycle multiply (32 μW/MHz)*

Cortex-M0/M0+

M0: *Optimized for size and power (13 μW/MHz dynamic power)*
M0+: *Lower power + shorter pipeline (11 μW/MHz dynamic power)*

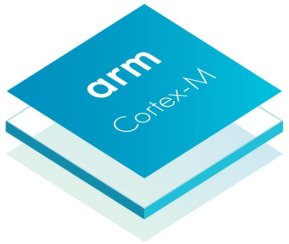


Memory Map: Tiva (TM4C123)



⇒ Memory and I/O

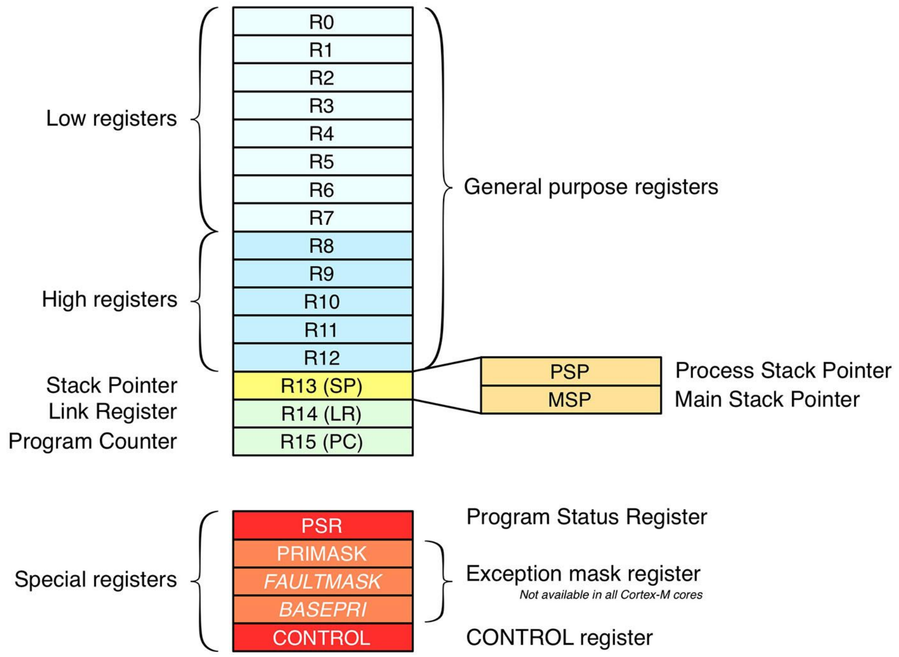
- 256K flash memory, 64K RAM
- 43 I/O, 64 pins



⇒ Memory-mapped architecture

- Single, flat address space
- Instruction fetches are half-word aligned (16 bit)
- All addresses are physical addresses
- Unaligned accesses may trigger a fault (no MMU)
- The ARM Architecture is bi-endian
- Cortex-M4 is configured as little-endian (endianness)

Registers and EABI



⇒ EABI: Embedded Application Binary Interface

- Set of rules for register usage
- Important when combining C & Assembly code

⇒ Rules!

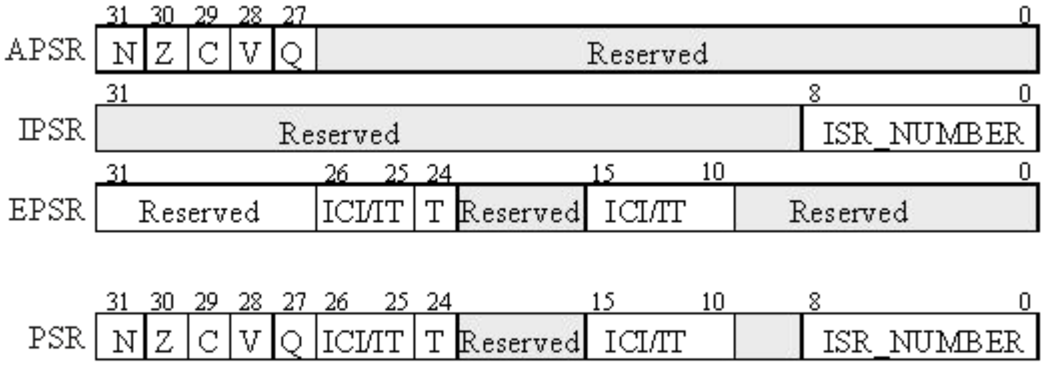
- Variable registers (r4-r11) are *callee* saved
- Scratch registers (r0-r3) are *caller* saved
  - Function arguments / returns
- Banked registers (msp/psp) must be accessed by special instructions
- Most Thumb2 instructions can only operate on lower half of register file (r0-r7)

Special Registers

Register	Name	Usage
IP (r12)	Intra-procedure call scratch register	Usage depends on tools and environment.
SP (r13)	Stack pointer	Points to bottom of stack ● MSP: used when CPU is in privileged state ● PSP: used when CPU is in non-privileged state
LR (r14)	Link Register	Points to subroutine return address
PC (r15)	Program Counter	Points to next instruction

⇒ PSR: Program Status Register

- APSR: Application PSR
- IPSR: Interrupt PSR
- EPSR: Execution PSR



Program Status Register (PSR)

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
apsr	N	Z	C	V	Q								GE																			
ipsr	Reserved																							Vector number								
epsr	Reserved					ICI/IT		T	Reserved								ICT/IT						Reserved									

Bit	Description
N	Negative Flag
Z	Zero Flag
C	Carry Flag
V	Overflow Flag
Q	DSP overflow and Saturation Flag
GE	Greater than or Equals Flag

Program Status Register (PSR)

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
apsr	N	Z	C	V	Q								GE																			
ipsr	Reserved																							Vector number								
epsr	Reserved					ICI/IT		T	Reserved								ICT/IT						Reserved									

Bit	Description
ICI	Interruptible-continuable instruction
IT	Execution state bits of the it instruction
T	Thumb state bit (must be preserved as 1)

Thumb-2: Address Mode

Address Mode	Mnemonic	Instruction	Equivalence
Immediate	MOV R0, #1	Data within the instruction	R0 = 1
Indexed	LDR R0, [R1]	Data pointed by register	R0 = R1
Indexed with offset	LDR R0, [R1, #4]	Data indexed by register + offset	R0 = *(R1+4)
PC-relative	BL Incr	Location is offset relative to PC	Jump to label
Register-list	PUSH {R1, LR, R4-R7}	List of registers	Stack operation

Thumb-2: Load/Store/Move

Mnemonic	Instruction	Equivalence
LDR Rd, [Rn]	Load 32-bit memory at [Rn] to Rd	Rd = Rn
STR Rt, [Rn]	Store Rt to 32-bit memory at [Rn]	*Rn = Rt
LDR Rd, [R1, #n]	Load 32-bit memory at [Rn+n] to Rd	Rd = *(Rn+n)
STR Rt, [Rn, #n]	Store Rt to 32-bit memory at [Rn+n]	*(Rn+n) = Rt
MOV Rd, Rn	Move the value of Rn to Rd register	Rd = Rn
MOV Rd, #imm12	#imm12 (12-bit constant) holds the value M	Rd = M (32-bit)

Thumb-2: Basic Operations

Mnemonic	Instruction	Equivalence
ADD Rd, Rn, Rm	Standard addition operation	Rd = Rn+Rm
ADD Rd, Rn, #imm1	#imm12 (12-bit constant) holds the value M	Rd = Rn+M
SUB Rd, Rn, Rm	Standard subtraction operation	Rd = Rn-Rm
SUBS Rd, Rn, Rm	Subtraction operation + update status register	Rd = Rn-Rm
EOR Rd, Rn, Rm	Standard XOR (Exclusive OR) operation	Rd = Rn ^ Rm

What does meant by this instruction?
EORS *R1, R1, R1*

Explore these instructions: ***ADC***, ***SBC***, ***AND***, ***OR***, ***LSL***, ***LSR***, ***MUL***, ***CMP***, ***BIC***

Thumb-2: Branching

Mnemonic	Instruction	Explanation
B label	Branch	Branch to label
BX Rm	Branch and exchange	Branch indirect to location specified by Rm
BL label	Branch and link	Branch to subroutine at label
CMP Rt, Rd BNE label	Compare Rt and Rd Branch if Not Equal	Branch if Rt==Rd

Explore these instructions: ***BE, BZ, BNZ***

Thumb-2: Conditioning

Mnemonic	Instruction	Status of Flags
EQ	=	Z==1
NE	≠	Z==0
GE	≥	(Z==0) && (N==V)
GT	>	N==V
LE	≤	(Z==1) || (N!=V)
LT	<	N!=V

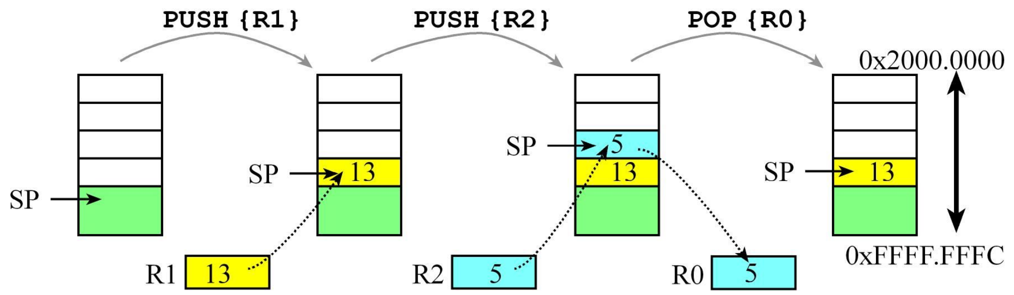
What is the final value of ***R0****?*

```
        MOV  R0, #2
        CMP  R0, #3
        ADDLT  R0, R0, #1
```

What is the final value of ***R2****?*

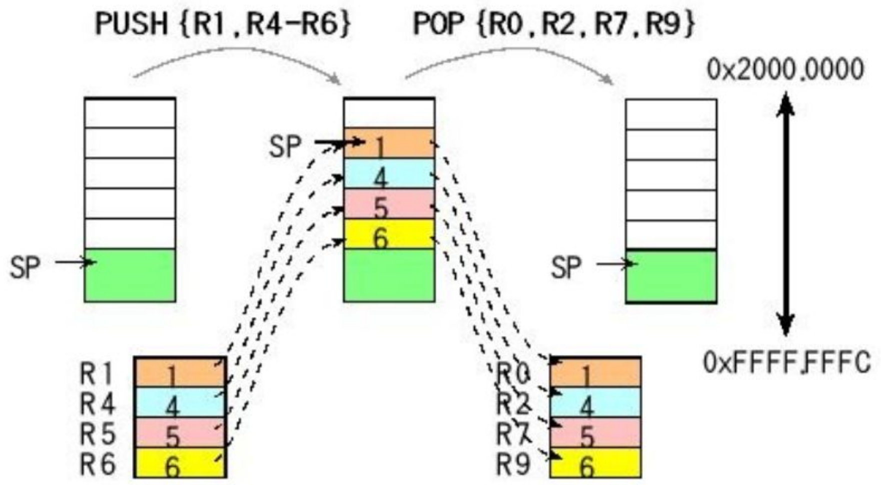
```
        MOVS  R0,  #1
        MOVS  R1,  #1
        MOVS  R2,  #1
        SUB   R1,  R2
        BNE   Label
        AND  R2, R2, #0
    Label:
        B  Label
```

Thumb-2: Stack Operations



⇒ FIFO: First In First Out

- **PUSH** 32-bit data: *SP* = *SP - 4* then store
- **POP** 32-bit data: load then *SP* = *SP* + *4*



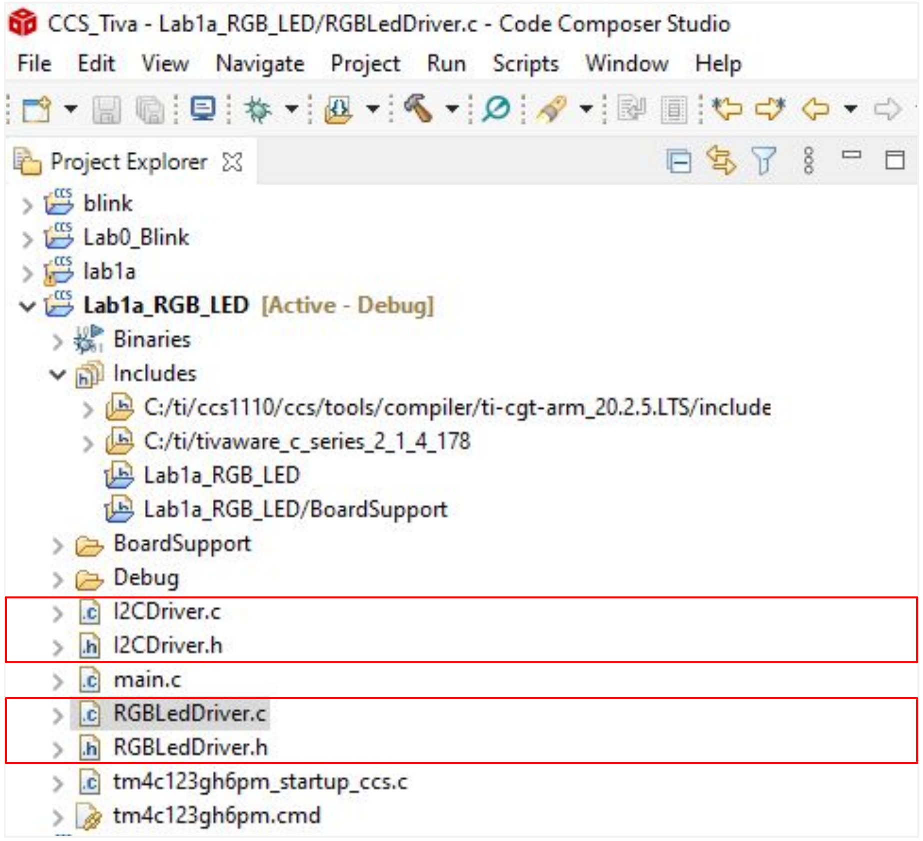
⇒ PUSH/POP Multiple Registers

- ***PUSH**: Registers are positioned in sorted order*
- ***POP**: values are loaded in sorted order*

Lab1a: LED Interfacing & I2C

TI **LP3943** LED Driver

Lab 1a: Interfacing LED Driver

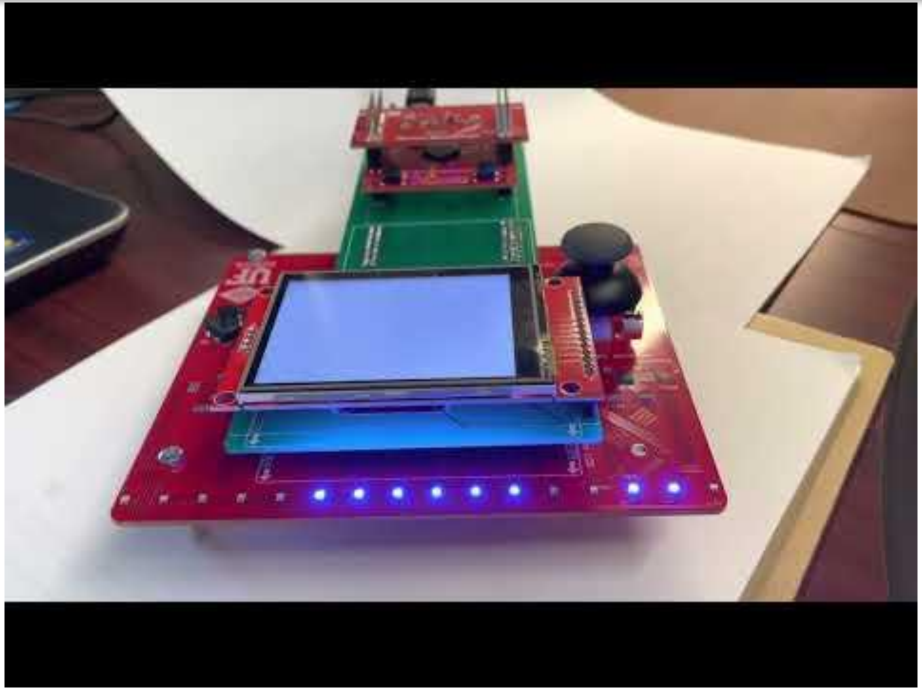


CCS_Tiva - Lab1a_RGB_LED/RGBLedDriver.c - Code Composer Studio

File Edit View Navigate Project Run Scripts Window Help

Project Explorer

- blink
- Lab0_Blink
- lab1a
- Lab1a_RGB_LED [Active - Debug]
  - Binaries
  - Includes
    - C:/ti/ccs1110/ccs/tools/compiler/ti-cgt-arm_20.2.5.LTS/include
    - C:/ti/tivaware_c_series_2_1_4_178
    - Lab1a_RGB_LED
    - Lab1a_RGB_LED/BoardSupport
  - BoardSupport
  - Debug
  - I2CDriver.c
  - I2CDriver.h
  - main.c
  - RGBLedDriver.c
  - RGBLedDriver.h
  - tm4c123gh6pm_startup_ccs.c
  - tm4c123gh6pm.cmd



⇒ *Bonus Point!* **+1**

- *If you can dim the LEDs in any pattern!*

TI LP3943 LED Driver

⇒ Independently control 16 LEDs

⇒ Two Internal PWM controls

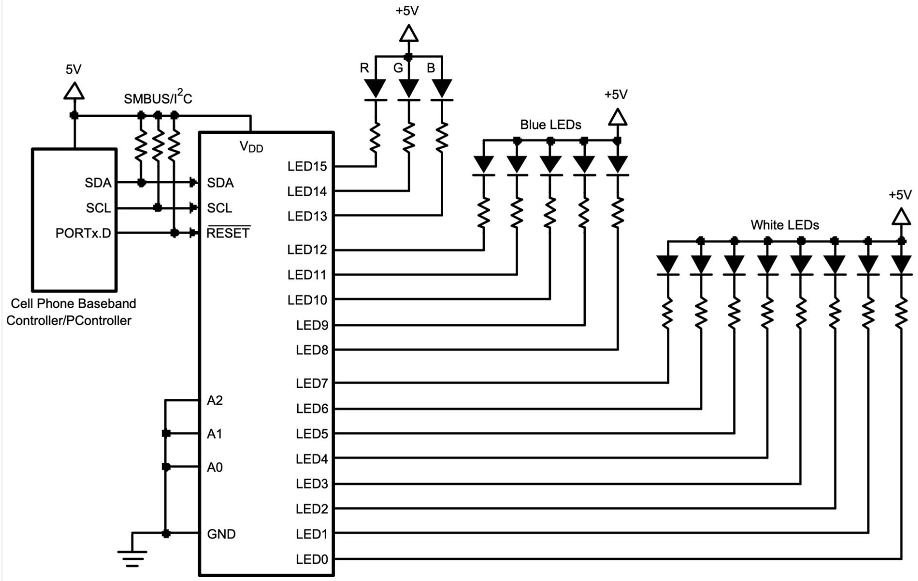
- Four led select registers (LS*n*)
- Two prescaler registers (PSC*n*)
- Two PWM registers (PWM*n*)
- Versatile duty-cycle control

⇒ I2C interface to MCU

⇒ **Applications**

- Digital cameras, indicator lamps
- GPIO expander in toys

⇒ More at http://www.ti.com/lit/ds/symlink/lp3943.pdf



LP3943 Configuration

- LED control on LS*n* registers

7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0
B1	B0	B1	B0	B1	B0	B1	B0
LED3		LED2		LED1		LED0	

The LS0 Register (address 0x06)

B1	B0	Configuration
0	0	Output is high impedance (LED is off)
0	1	Output set to ON state
1	0	Use PWM0/PSC0 for waveform
1	1	Use PWM1/PSC1 for waveform

LED output modes

- Duty Cycle control in PWM*n*

7	6	5	4	3	2	1	0
1	0	0	0	0	0	0	0

The PWM0 Register (address 0x03)

- Frequency control in PSC*n*

7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0

The PSC0 Register (address 0x02)

⇒ PWMn and PSCn are integer registers: 8 bit PWM / 256 steps

⇒ Waveform period can range from 0.625 ms to 1.6 s; formula: $T = (PSCn + 1)/160$

LP3943 Configuration ++

- LEDs 0 to 7,1 Hz, 25%
- LEDs 8 to 12, 5 Hz, 25%
- LEDs 13 to 15 off

PSC0 (0x02)

7	6	5	4	3	2	1	0
1	0	0	1	1	1	1	1

PWM0 (0x03)

7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0

PSC1 (0x04)

7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	1

PWM1 (0x05)

7	6	5	4	3	2	1	0
1	0	0	0	0	0	0	0

LS0 (0x06)

7	6	5	4	3	2	1	0
1	0	1	0	1	0	1	0
LED3		LED2		LED1		LED0	

LS1 (0x07)

7	6	5	4	3	2	1	0
1	0	1	0	1	0	1	0
LED7		LED6		LED5		LED4	

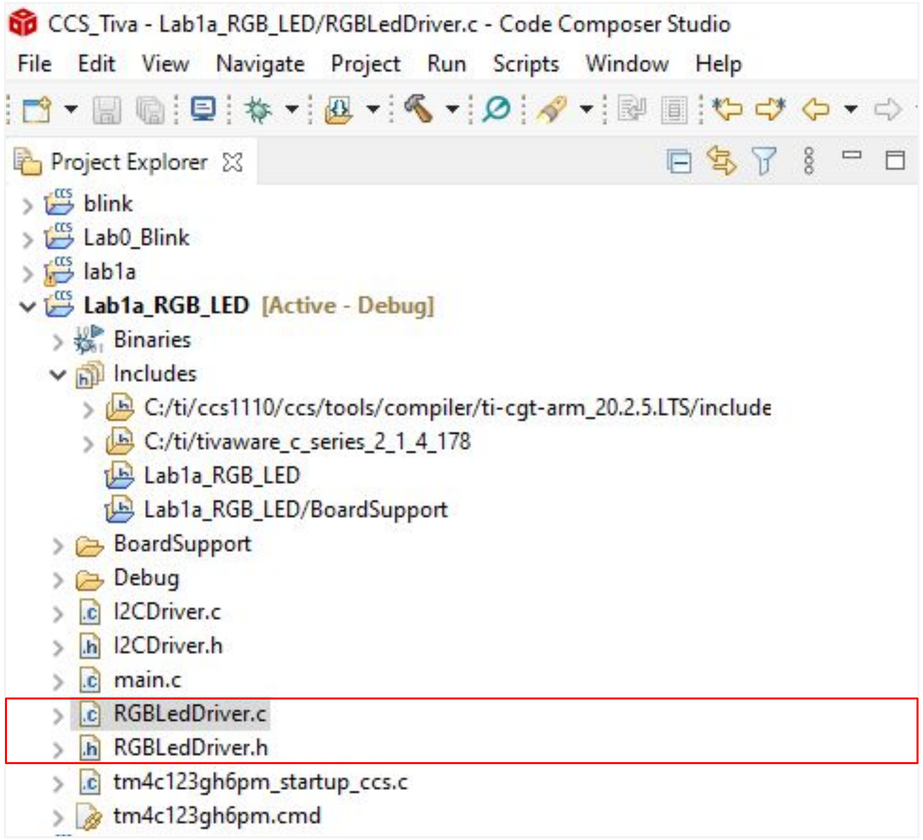
LS2 (0x08)

7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1
LED11		LED10		LED9		LED8	

LS3 (0x09)

7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	1
LED15		LED14		LED13		LED12	

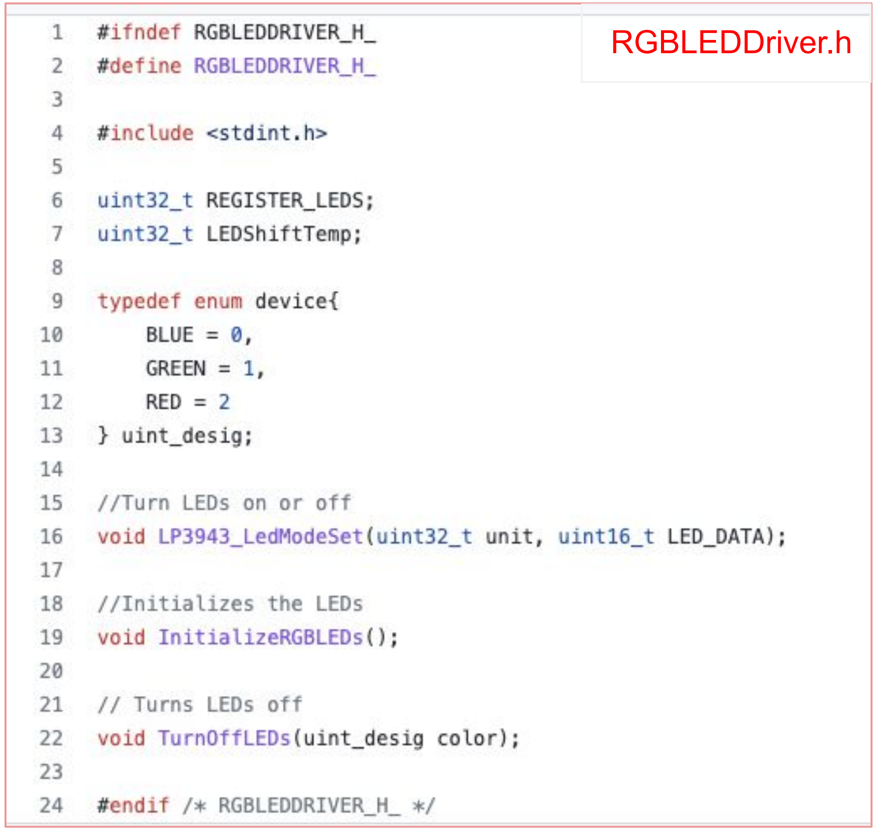
LED Driver Header


CCS_Tiva - Lab1a_RGB_LED/RGBLedDriver.c - Code Composer Studio

File Edit View Navigate Project Run Scripts Window Help

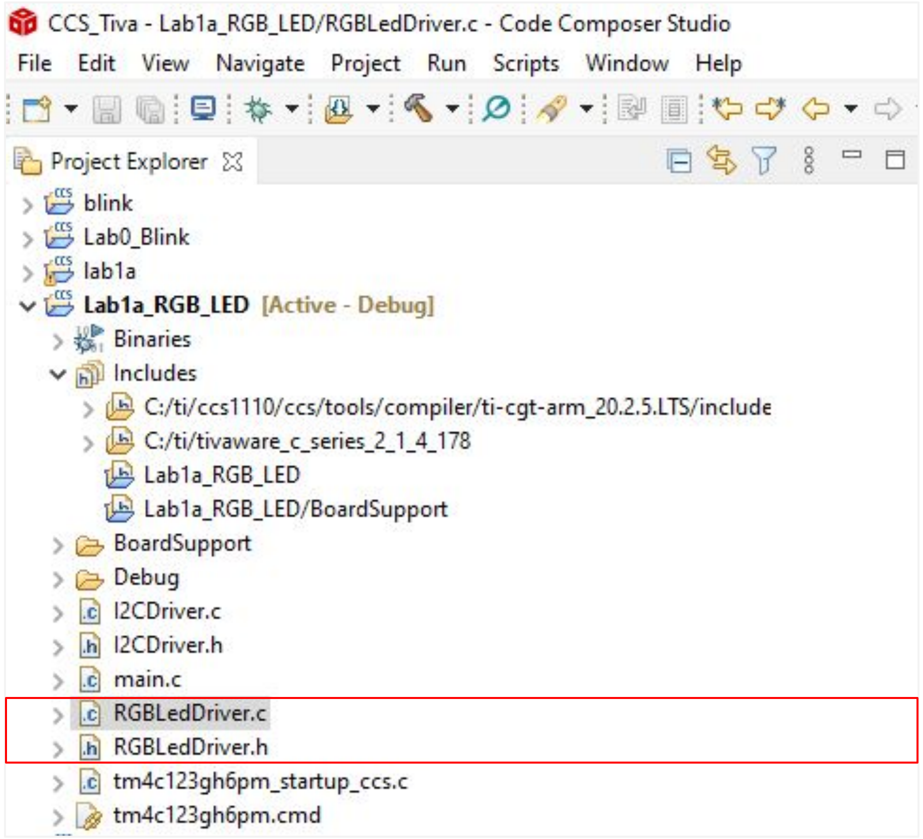
Project Explorer

- blink
- Lab0_Blink
- lab1a
- Lab1a_RGB_LED [Active - Debug]
  - Binaries
  - Includes
    - C:/ti/ccs1110/ccs/tools/compiler/ti-cgt-arm_20.2.5.LTS/include
    - C:/ti/tivaware_c_series_2_1_4_178
    - Lab1a_RGB_LED
    - Lab1a_RGB_LED/BoardSupport
  - BoardSupport
  - Debug
  - I2CDriver.c
  - I2CDriver.h
  - main.c
  - RGBLedDriver.c
  - RGBLedDriver.h
  - tm4c123gh6pm_startup_ccs.c
  - tm4c123gh6pm.cmd


RGBLEDDriver.h

```c
#ifndef RGBLEDDRIVER_H_
#define RGBLEDDRIVER_H_

#include <stdint.h>

uint32_t REGISTER_LEDS;
uint32_t LEDShiftTemp;

typedef enum device{
    BLUE = 0,
    GREEN = 1,
    RED = 2
} uint_desig;

//Turn LEDs on or off
void LP3943_LedModeSet(uint32_t unit, uint16_t LED_DATA);

//Initializes the LEDs
void InitializeRGBLEDs();

// Turns LEDs off
void TurnOffLEDs(uint_desig color);

#endif /* RGBLEDDRIVER_H_ */
```

LED Driver Functions


CCS_Tiva - Lab1a_RGB_LED/RGBLedDriver.c - Code Composer Studio
File Edit View Navigate Project Run Scripts Window Help
Project Explorer
blink
Lab0_Blink
lab1a
Lab1a_RGB_LED [Active - Debug]
Binaries
Includes
C:/ti/ccs1110/ccs/tools/compiler/ti-cgt-arm_20.2.5.LTS/include
C:/ti/tivaware_c_series_2_1_4_178
Lab1a_RGB_LED
Lab1a_RGB_LED/BoardSupport
BoardSupport
Debug
I2CDriver.c
I2CDriver.h
main.c
RGBLedDriver.c
RGBLedDriver.h
tm4c123gh6pm_startup_ccs.c
tm4c123gh6pm.cmd

RGBLEDDriver.c

```c
void LP3943_LedModeSet(uint32_t color, uint16_t LED_DATA)
{
    // Step 1: Set "SlaveAddress" for all colors
    // Hint: use the SetSlaveAddress function

    // Step 2: Convert LED_DATA

    // Send out LS register address
    StartTransmission(0x16);

    // Step 3: Send out LED Data
    // Hint: use the ContinueTransmission function

    // End communication
    EndTransmission();
}

void TurnOffLEDs(uint_desig color)
{
    LP3943_LedModeSet(color, 0x0000);
}

void InitializeRGBLEDs()
{
    // Step 1: initialize I2C

    // Step 2: turn off LEDs (Red, Green, and Blue)
}
```

I2C (Inter-Integrated Circuit) Basics

⇒ Designed for low-cost, medium data rate applications.

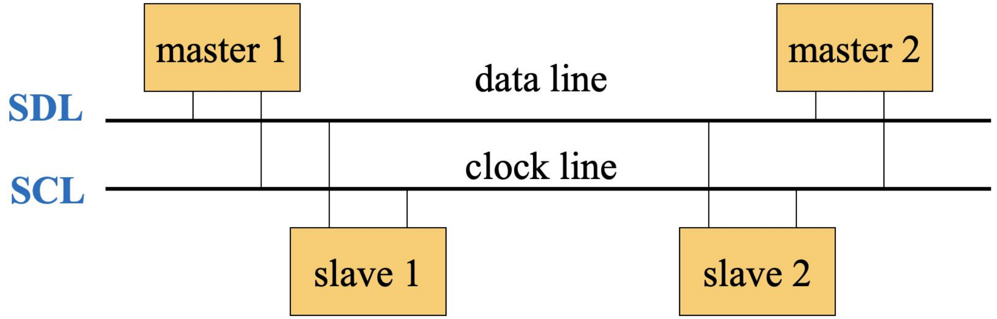
- In Phillips Semiconductor, 1980s
- Characteristics:
  - Serial, byte-oriented, bi-directional
  - Multiple slave/master communication

⇒ **I2C data link layer**

- *Master* transmit/receive & *Slave* transmi
- Every device has an unique address
  - 7 bits in address
  - 8th bit of address signals read (1) or write (0)

⇒ **I2C physical layer**

- Serial data line (SDA) and serial clock line (SCL)
- Open collector/drain drivers (default state high)
- No global master for clock



I2C Data Format

⇒ Clock Synchronization

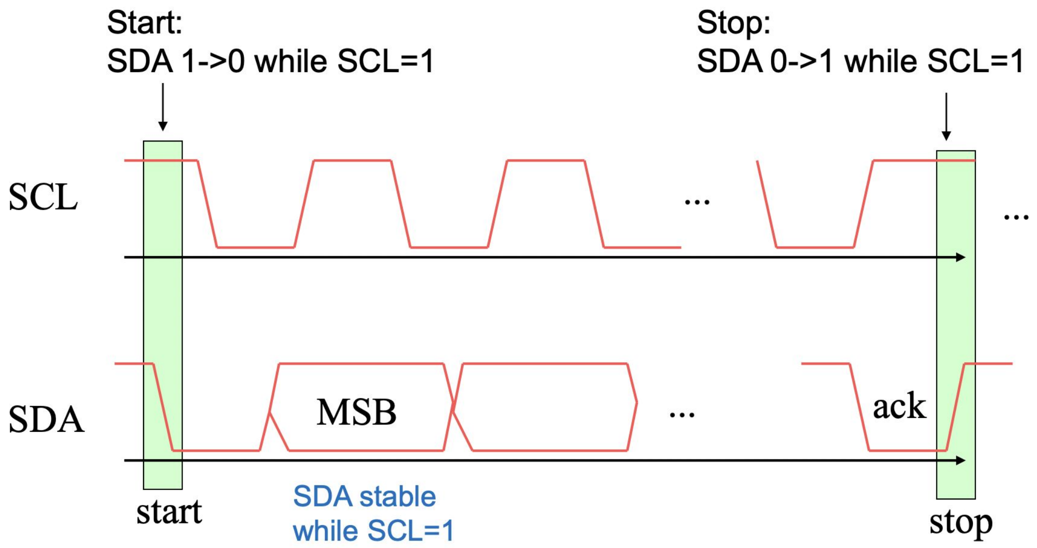
- Master generates its own clock on SCL
- Clock synchronization uses wired-AND

⇒ Bus Arbitration

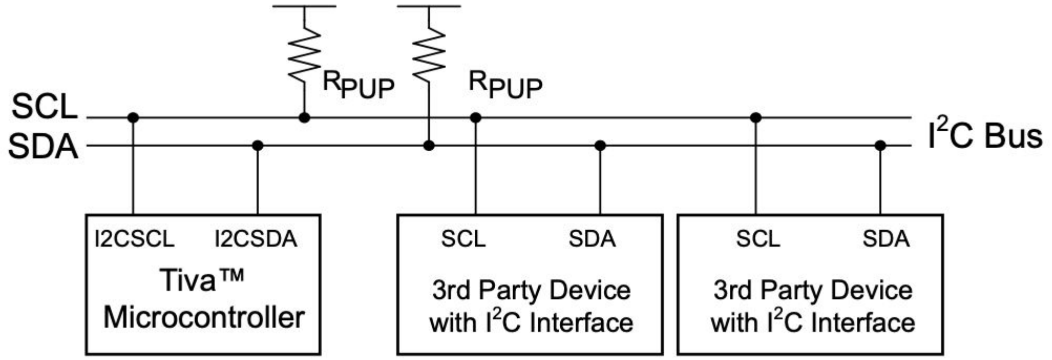
- Master may start sending if bus free
- Sender listens while sending
- Test SDA while SCL high
- Transmit 1 and hear 0 on SDA

⇒ Data Transfer

- Send 8-bit byte (MSB first)
- Each byte followed by acknowledge bit
- Master releases SDA(high) during ack
- Slave must pull SDA low during ack



I2C In Tiva C


SCL
SDA
RPUP
RPUP
I²C Bus
I2CSCL
I2CSDA
Tiva™ Microcontroller
SCL
SDA
3rd Party Device with I²C Interface
SCL
SDA
3rd Party Device with I²C Interface

⇒ Bi-directional data transfer through a two-wire design

- Serial data line (SDA) and serial clock line (SCL)

⇒ Four I2C modes

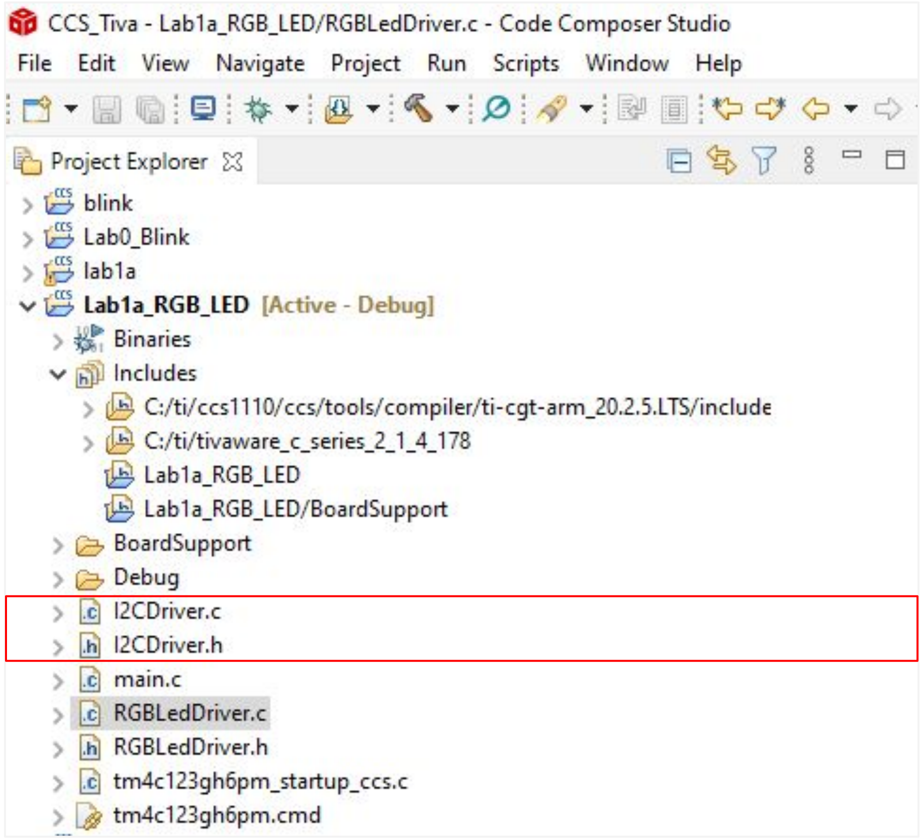
- Master transmit, Master receive
- Slave transmit, Slave receive

⇒ Four transmission speeds

⇒ **Related documentations**

- Tiva tm4c123gh6pm datasheet
  - Chapter 16
- These blogs
  - I2C basics
  - I2C in Tiva C series

Enabling I2C Communication



CCS_Tiva - Lab1a_RGB_LED/RGBLedDriver.c - Code Composer Studio

File Edit View Navigate Project Run Scripts Window Help

Project Explorer

- blink
- Lab0_Blink
- lab1a
- Lab1a_RGB_LED [Active - Debug]
  - Binaries
  - Includes
    - C:/ti/ccs1110/ccs/tools/compiler/ti-cgt-arm_20.2.5.LTS/include
    - C:/ti/tivaware_c_series_2_1_4_178
    - Lab1a_RGB_LED
    - Lab1a_RGB_LED/BoardSupport
  - BoardSupport
  - Debug
  - I2CDriver.c
  - I2CDriver.h
  - main.c
  - RGBLedDriver.c
  - RGBLedDriver.h
  - tm4c123gh6pm_startup_ccs.c
  - tm4c123gh6pm.cmd

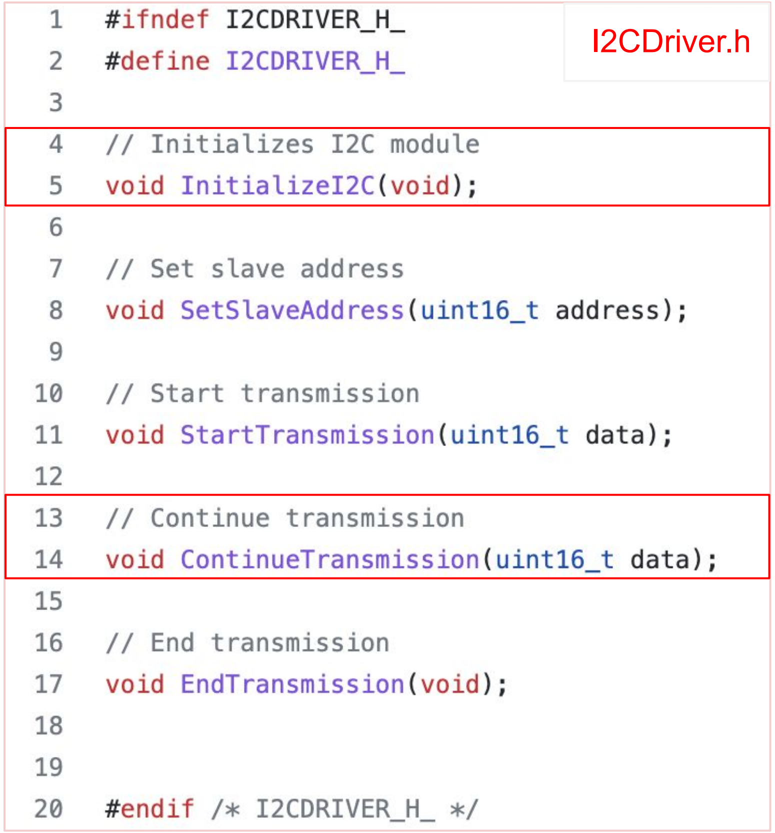
⇒ Most functionalities are already implemented

- See in the Board Support Package
- I2C.h and I2C.c

⇒ **Device-specific functions**

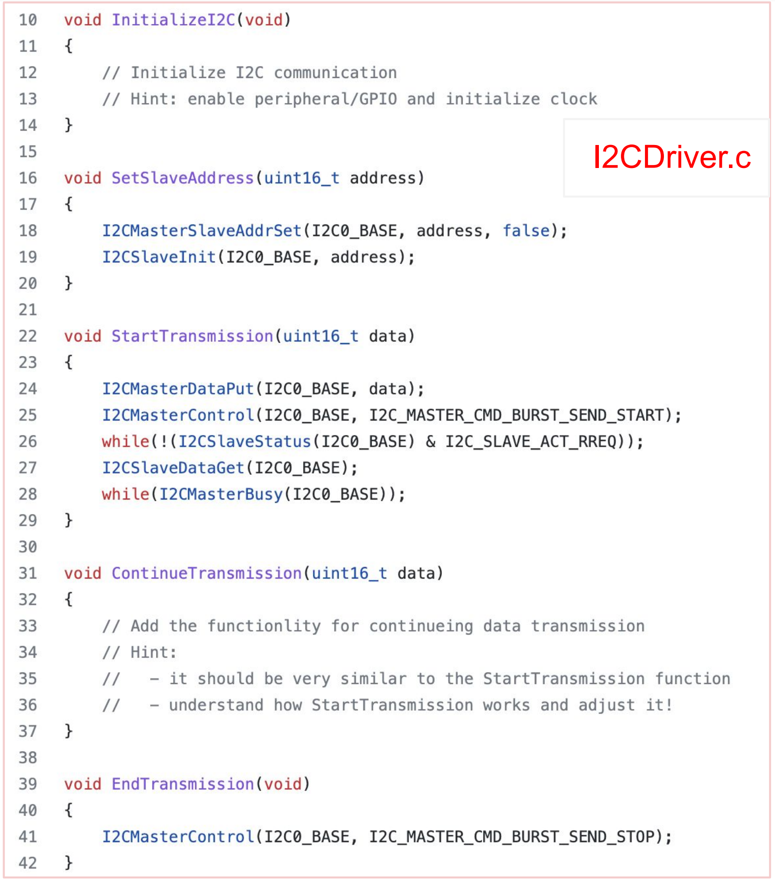
- I2CDriver.c and I2CDriver.h
- You need to know how to initialize and use I2C communication given the backbone!
  - *Initialize I2C*
  - *Start transmission given data*
  - *Continue transmission given data*
  - *End transmission*

Enabling I2C Communication ++



I2CDriver.h

```c
#ifndef I2CDRIVER_H_
#define I2CDRIVER_H_

// Initializes I2C module
void InitializeI2C(void);

// Set slave address
void SetSlaveAddress(uint16_t address);

// Start transmission
void StartTransmission(uint16_t data);

// Continue transmission
void ContinueTransmission(uint16_t data);

// End transmission
void EndTransmission(void);


#endif /* I2CDRIVER_H_ */
```



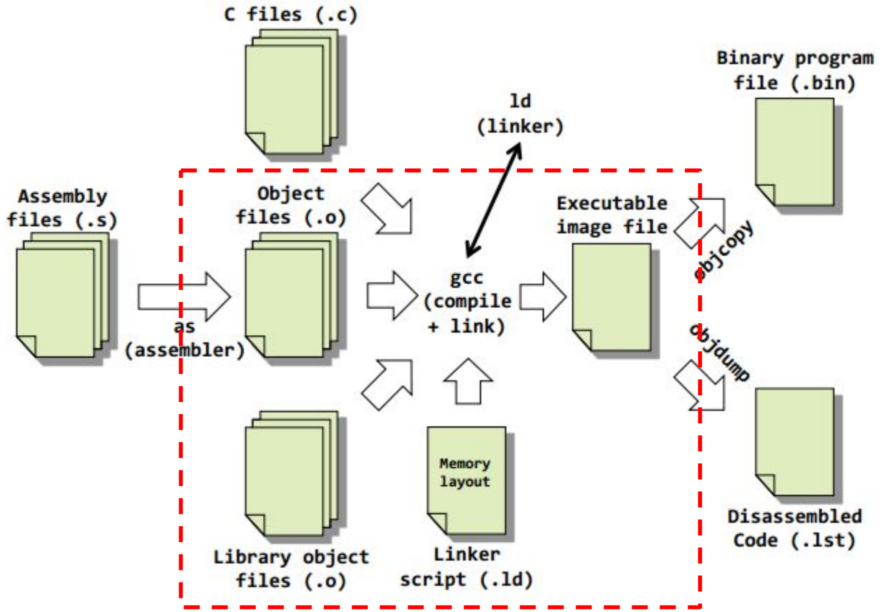
I2CDriver.c

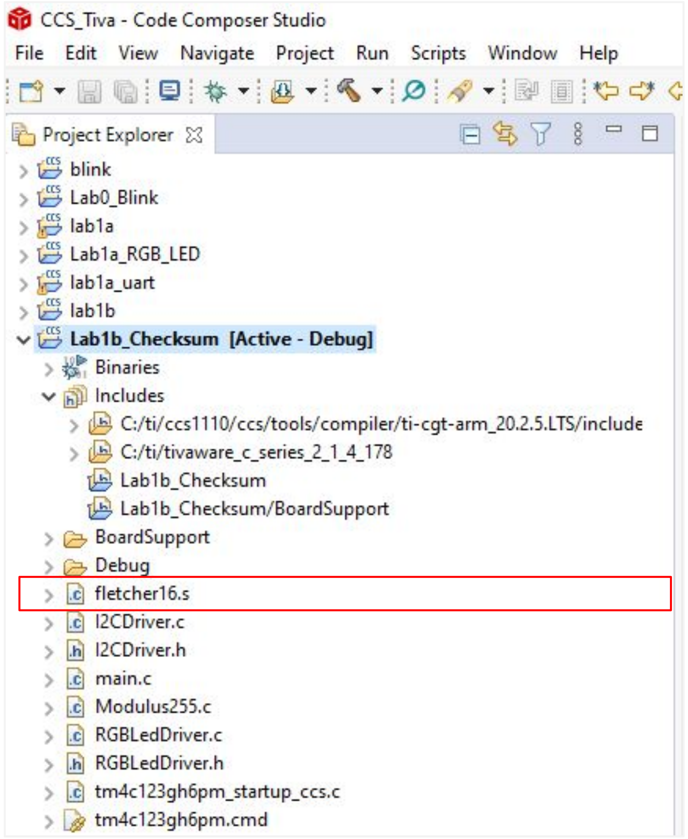
```c
void InitializeI2C(void)
{
    // Initialize I2C communication
    // Hint: enable peripheral/GPIO and initialize clock
}

void SetSlaveAddress(uint16_t address)
{
    I2CMasterSlaveAddrSet(I2C0_BASE, address, false);
    I2CSlaveInit(I2C0_BASE, address);
}

void StartTransmission(uint16_t data)
{
    I2CMasterDataPut(I2C0_BASE, data);
    I2CMasterControl(I2C0_BASE, I2C_MASTER_CMD_BURST_SEND_START);
    while(!(I2CSlaveStatus(I2C0_BASE) & I2C_SLAVE_ACT_RREQ));
    I2CSlaveDataGet(I2C0_BASE);
    while(I2CMasterBusy(I2C0_BASE));
}

void ContinueTransmission(uint16_t data)
{
    // Add the functionlity for continueing data transmission
    // Hint:
    //   - it should be very similar to the StartTransmission function
    //   - understand how StartTransmission works and adjust it!
}

void EndTransmission(void)
{
    I2CMasterControl(I2C0_BASE, I2C_MASTER_CMD_BURST_SEND_STOP);
}
```

Lab1b: Linking Assembly Functions

Fischer’s Checksum
In **C** and **Assembly**

Lab 1b: Linking Assembly Functions



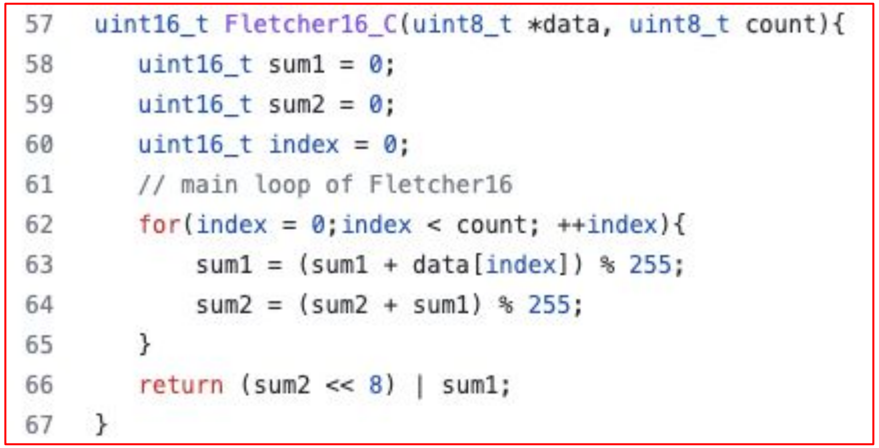

CCS_Tiva - Code Composer Studio

File Edit View Navigate Project Run Scripts Window Help

Project Explorer

- blink
- Lab0_Blink
- lab1a
- Lab1a_RGB_LED
- lab1a_uart
- lab1b
- Lab1b_Checksum [Active - Debug]
  - Binaries
  - Includes
    - C:/ti/ccs1110/ccs/tools/compiler/ti-cgt-arm_20.2.5.LTS/include
    - C:/ti/tivaware_c_series_2_1_4_178
    - Lab1b_Checksum
    - Lab1b_Checksum/BoardSupport
  - BoardSupport
  - Debug
  - fletcher16.s
  - I2CDriver.c
  - I2CDriver.h
  - main.c
  - Modulus255.c
  - RGBLedDriver.c
  - RGBLedDriver.h
  - tm4c123gh6pm_startup_ccs.c
  - tm4c123gh6pm.cmd

Fletcher16 Checksum in C

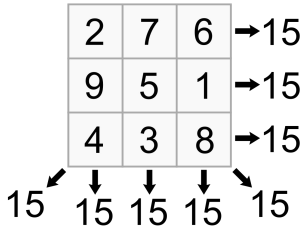


```c
uint16_t Fletcher16_C(uint8_t *data, uint8_t count){
    uint16_t sum1 = 0;
    uint16_t sum2 = 0;
    uint16_t index = 0;
    // main loop of Fletcher16
    for(index = 0;index < count; ++index){
        sum1 = (sum1 + data[index]) % 255;
        sum2 = (sum2 + sum1) % 255;
    }
    return (sum2 << 8) | sum1;
}
```

data[i]	**sum1**	**sum2**
2	2	2
7	9	11
6	15	26
9	24	50
5	29	79
1	30	109
4	34	143
3	37	180
8	45	225

⇒ MagicSq = {2, 7, 6, 9, 5, 1, 4, 3, 8}

⇒ Fletcher16_C (MagicSq, 9) = ?

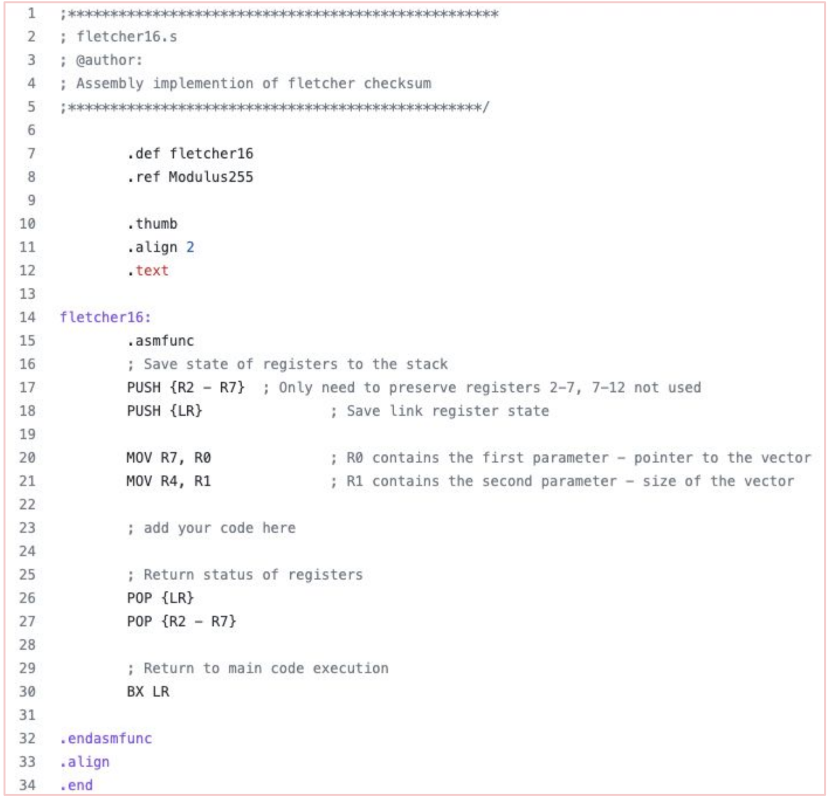


Return (225 << 8) || 45

= 57645

= $(1\ 1\ 1\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 1\ 0\ 1\ 1\ 0\ 1)_b$

Fletcher16 in Assembly



```
;*****************************************************
; fletcher16.s
; @author:
; Assembly implemention of fletcher checksum
;***************************************************/

        .def fletcher16
        .ref Modulus255

        .thumb
        .align 2
        .text

fletcher16:
        .asmfunc
        ; Save state of registers to the stack
        PUSH {R2 - R7}  ; Only need to preserve registers 2-7, 7-12 not used
        PUSH {LR}               ; Save link register state

        MOV R7, R0              ; R0 contains the first parameter - pointer to the vector
        MOV R4, R1              ; R1 contains the second parameter - size of the vector

        ; add your code here

        ; Return status of registers
        POP {LR}
        POP {R2 - R7}

        ; Return to main code execution
        BX LR

.endasmfunc
.align
.end
```

.def : *function or variable created, accessed from other functions*

.ref : *external function/variable reference*

.thumb*: we are using thumb mode*

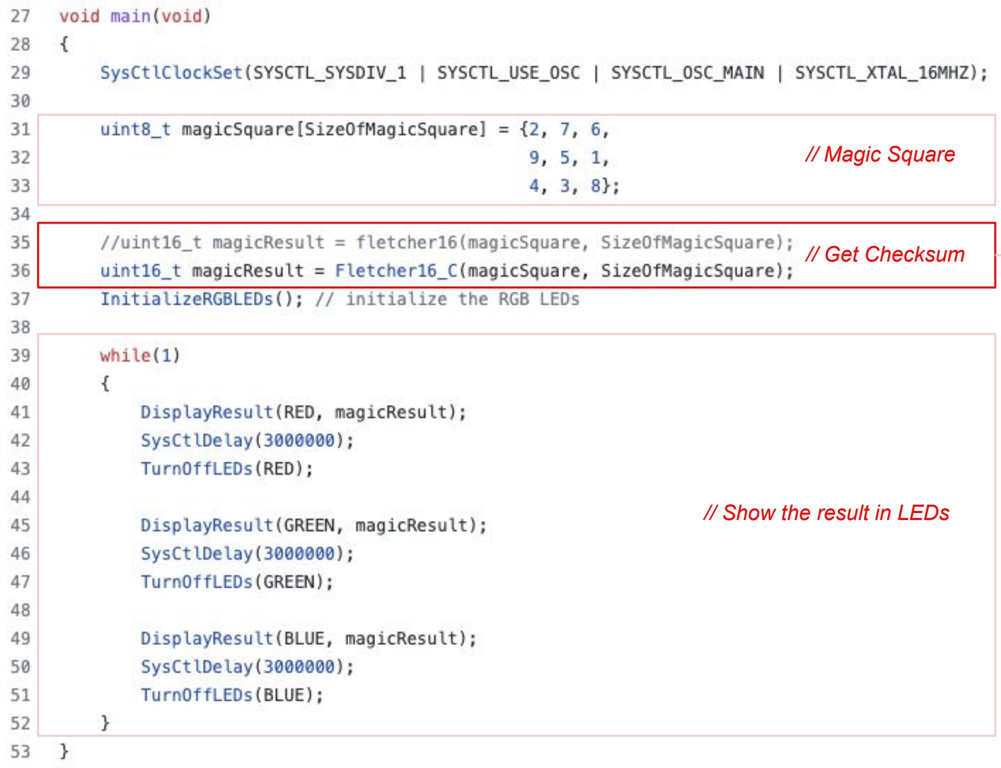
.align2*: in thumb mode, instructions are 16 bit rather than 32 bits*

.text: *start of code section*

fletcher16*: name of the function and works like a normal label*

.asmfunc*: starting a function rather than a label*

Fletcher16 in C and Assembly

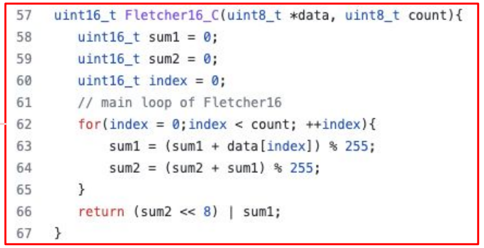


```c
void main(void)
{
    SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN | SYSCTL_XTAL_16MHZ);

    uint8_t magicSquare[SizeOfMagicSquare] = {2, 7, 6,
                                              9, 5, 1,
                                              4, 3, 8};

    //uint16_t magicResult = fletcher16(magicSquare, SizeOfMagicSquare);
    uint16_t magicResult = Fletcher16_C(magicSquare, SizeOfMagicSquare);
    InitializeRGBLEDs(); // initialize the RGB LEDs

    while(1)
    {
        DisplayResult(RED, magicResult);
        SysCtlDelay(3000000);
        TurnOffLEDs(RED);

        DisplayResult(GREEN, magicResult);
        SysCtlDelay(3000000);
        TurnOffLEDs(GREEN);

        DisplayResult(BLUE, magicResult);
        SysCtlDelay(3000000);
        TurnOffLEDs(BLUE);
    }
}
```

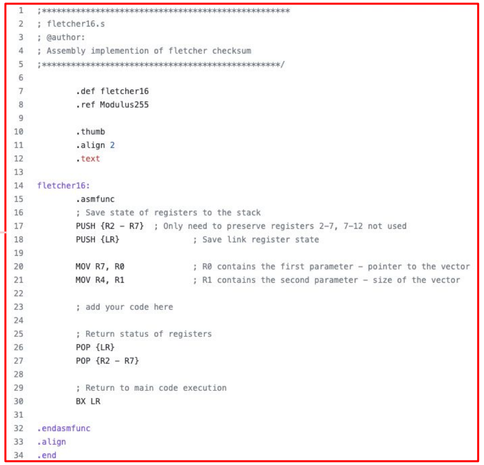
// Magic Square

// Get Checksum

// Show the result in LEDs

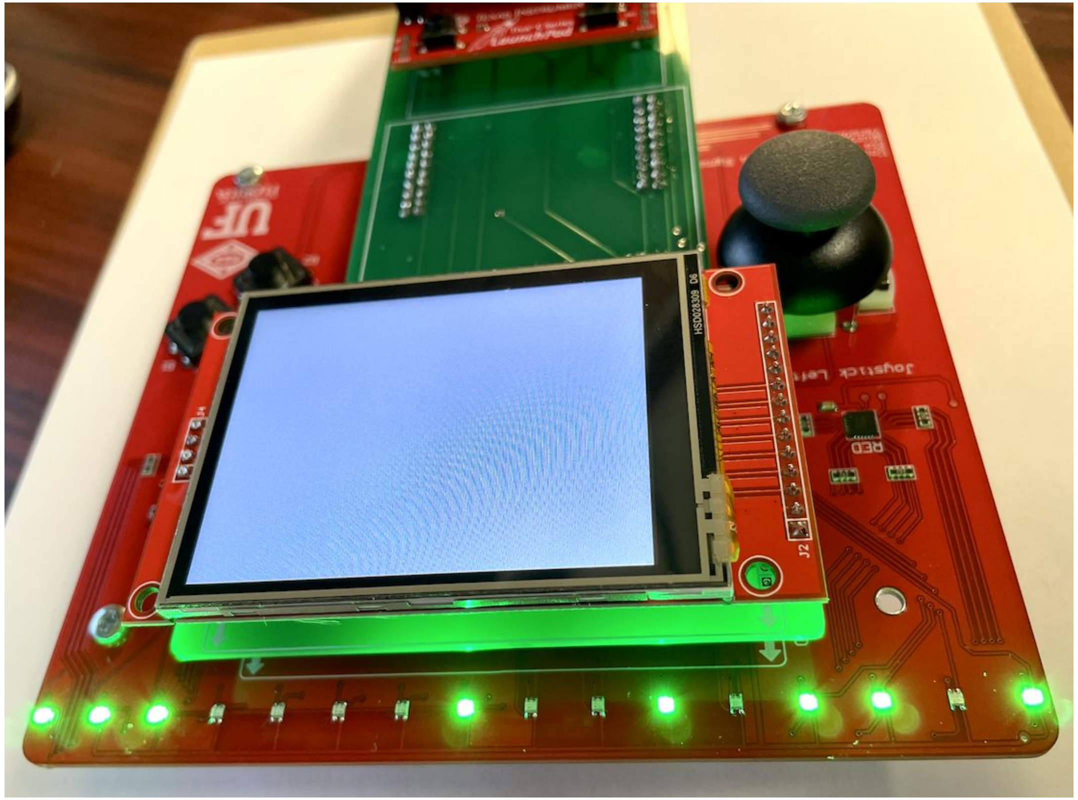


```c
uint16_t Fletcher16_C(uint8_t *data, uint8_t count){
    uint16_t sum1 = 0;
    uint16_t sum2 = 0;
    uint16_t index = 0;
    // main loop of Fletcher16
    for(index = 0;index < count; ++index){
        sum1 = (sum1 + data[index]) % 255;
        sum2 = (sum2 + sum1) % 255;
    }
    return (sum2 << 8) | sum1;
}
```



```
;**************************************************
; fletcher16.s
; @author:
; Assembly implemention of fletcher checksum
;*************************************************/

        .def fletcher16
        .ref Modulus255

        .thumb
        .align 2
        .text

fletcher16:
        .asmfunc
        ; Save state of registers to the stack
        PUSH {R2 - R7}  ; Only need to preserve registers 2-7, 7-12 not used
        PUSH {LR}               ; Save link register state

        MOV R7, R0              ; R0 contains the first parameter - pointer to the vector
        MOV R4, R1              ; R1 contains the second parameter - size of the vector

        ; add your code here

        ; Return status of registers
        POP {LR}
        POP {R2 - R7}

        ; Return to main code execution
        BX LR

.endasmfunc
.align
.end
```

Checksum Output In LEDs



data[i]	**sum1**	**sum2**
2	2	2
7	9	11
6	15	26
9	24	50
5	29	79
1	30	109
4	34	143
3	37	180
8	45	225

Return (225 << 8) || 45

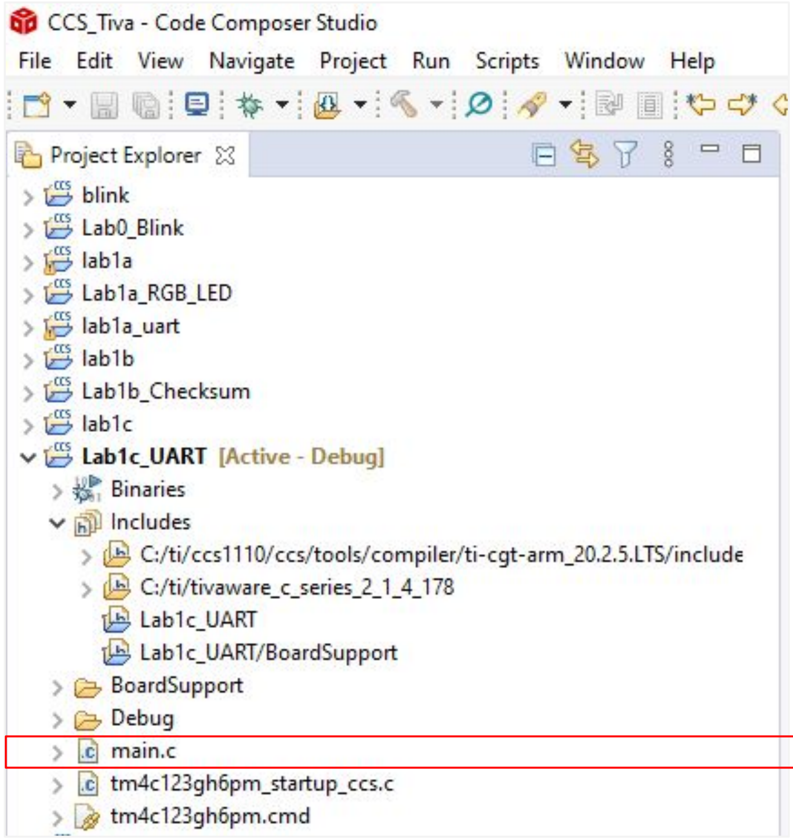
= 57645

*= (**1 1 1** 0 0 0 0 **1** 0 0 **1** 0 **1 1** 0 **1**)$_b$*

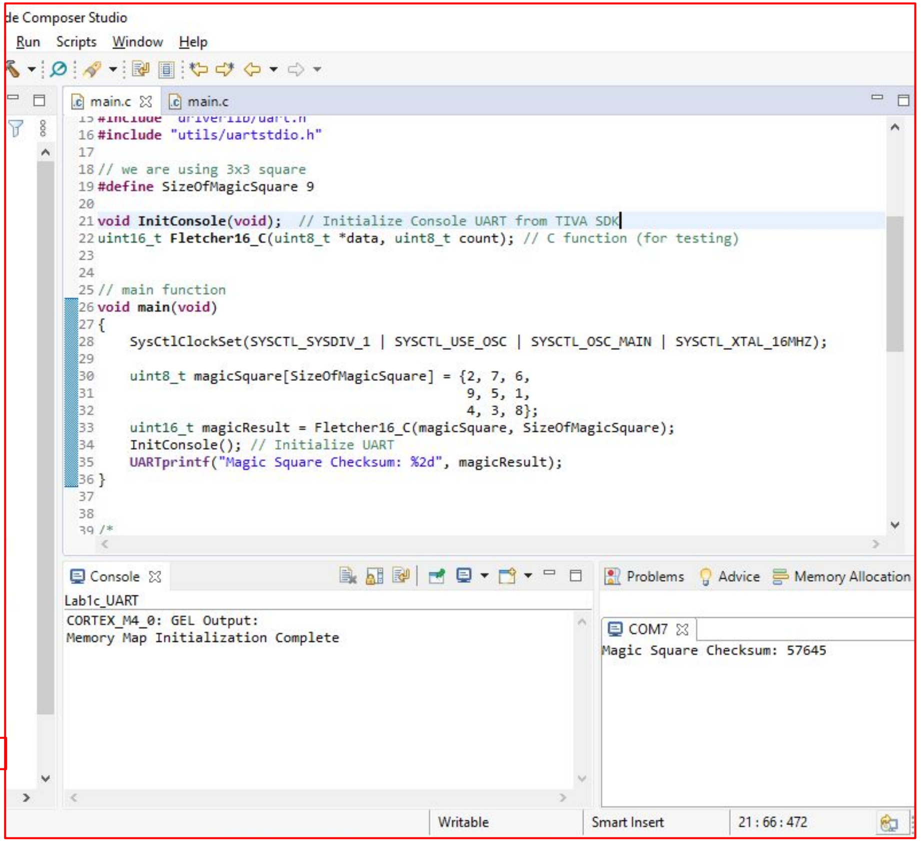
Lab1b: Basic UART Communication

Fischer's Checksum
In Console

UART: Console I/O


CCS_Tiva - Code Composer Studio
File Edit View Navigate Project Run Scripts Window Help
Project Explorer

```
> blink
> Lab0_Blink
> lab1a
> Lab1a_RGB_LED
> lab1a_uart
> lab1b
> Lab1b_Checksum
> lab1c
v Lab1c_UART [Active - Debug]
    > Binaries
    v Includes
        > C:/ti/ccs1110/ccs/tools/compiler/ti-cgt-arm_20.2.5.LTS/include
        > C:/ti/tivaware_c_series_2_1_4_178
          Lab1c_UART
          Lab1c_UART/BoardSupport
    > BoardSupport
    > Debug
    > main.c
    > tm4c123gh6pm_startup_ccs.c
    > tm4c123gh6pm.cmd
```

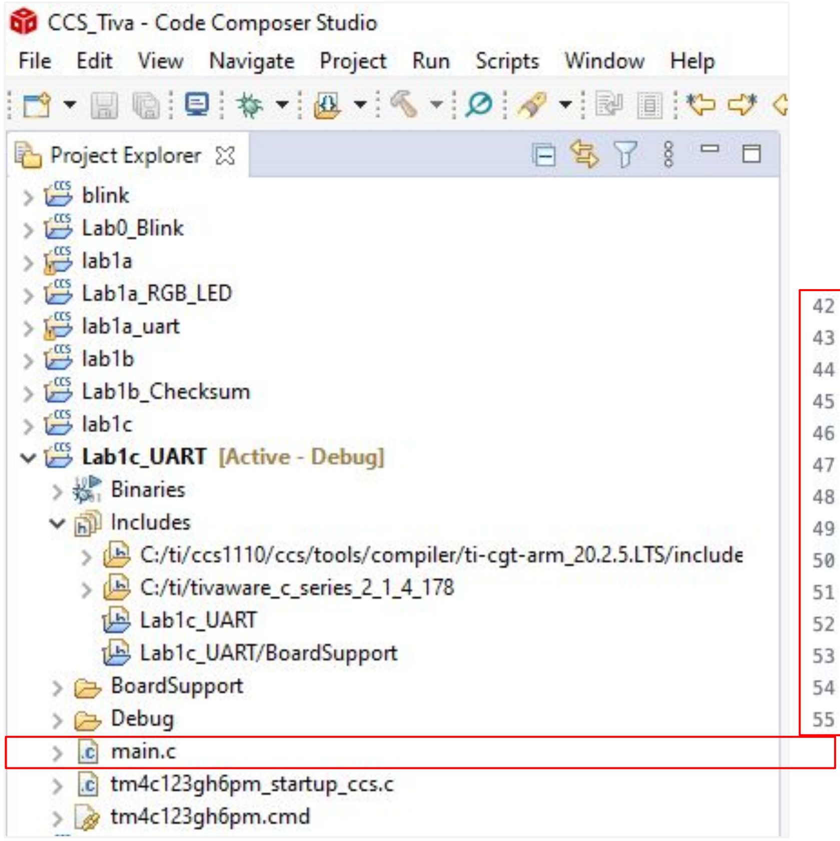


```c
#include "driverlib/uart.h"
#include "utils/uartstdio.h"

// we are using 3x3 square
#define SizeOfMagicSquare 9

void InitConsole(void);  // Initialize Console UART from TIVA SDK
uint16_t Fletcher16_C(uint8_t *data, uint8_t count); // C function (for testing)


// main function
void main(void)
{
    SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN | SYSCTL_XTAL_16MHZ);

    uint8_t magicSquare[SizeOfMagicSquare] = {2, 7, 6,
                                               9, 5, 1,
                                               4, 3, 8};
    uint16_t magicResult = Fletcher16_C(magicSquare, SizeOfMagicSquare);
    InitConsole(); // Initialize UART
    UARTprintf("Magic Square Checksum: %2d", magicResult);
}


/*
```

```
Lab1c_UART
CORTEX_M4_0: GEL Output:
Memory Map Initialization Complete
```

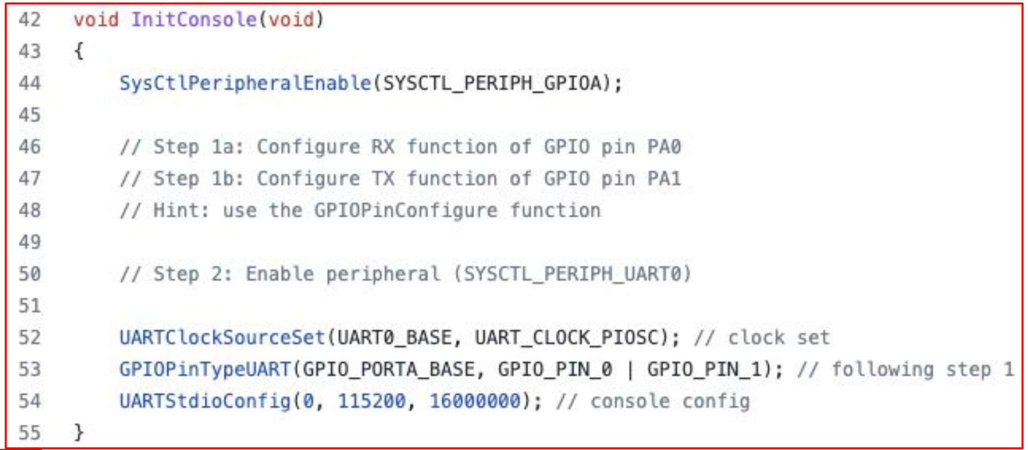
```
Magic Square Checksum: 57645
```

UART: Initializing Console



CCS_Tiva - Code Composer Studio

- blink
- Lab0_Blink
- lab1a
- Lab1a_RGB_LED
- lab1a_uart
- lab1b
- Lab1b_Checksum
- lab1c
- Lab1c_UART [Active - Debug]
  - Binaries
  - Includes
    - C:/ti/ccs1110/ccs/tools/compiler/ti-cgt-arm_20.2.5.LTS/include
    - C:/ti/tivaware_c_series_2_1_4_178
    - Lab1c_UART
    - Lab1c_UART/BoardSupport
  - BoardSupport
  - Debug
  - main.c
  - tm4c123gh6pm_startup_ccs.c
  - tm4c123gh6pm.cmd



```c
void InitConsole(void)
{
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);

    // Step 1a: Configure RX function of GPIO pin PA0
    // Step 1b: Configure TX function of GPIO pin PA1
    // Hint: use the GPIOPinConfigure function

    // Step 2: Enable peripheral (SYSCTL_PERIPH_UART0)

    UARTClockSourceSet(UART0_BASE, UART_CLOCK_PIOSC); // clock set
    GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1); // following step 1
    UARTStdioConfig(0, 115200, 16000000); // console config
}
```

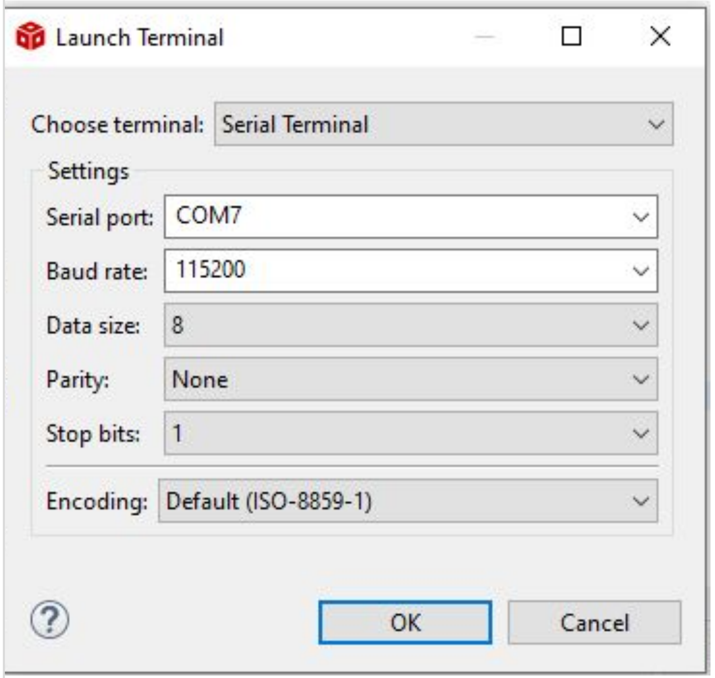
UART: Console I/O Configuration

⇒ All UART functionalities are implemented

- See in the Board Support Package
- uart.h and uart.c

⇒ **We will only use UART for console I/O**

- In CCS, open the terminal (serial port)
- Find the COM port to which Tiva is connected
- Configure that COM port
  - Baud-rate: 115200
  - Data size: 8
  - Parity: none

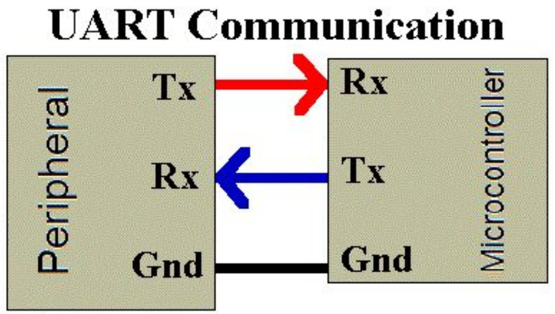


Choose terminal:	Serial Terminal
Settings	
Serial port:	COM7
Baud rate:	115200
Data size:	8
Parity:	None
Stop bits:	1
Encoding:	Default (ISO-8859-1)

UART (Universal Asynchronous Receiver Transmitter) Basics

⇒ UART communication

- Serial asynchronous communication
- Transmission of 8 bits serially
- Requires Tx-Rx agreement ahead-of-time
- Data rate < 115.2 Kbps



⇒ UART in Tiva

- 8 UART Tx/Rx pairs
- Programmable baud-rate generator
- Programmable FIFO trigger levels

⇒ Related documentations

- Tiva tm4c123gh6pm datasheet
  - Chapter 14
- These blogs
  - UART basics
  - UART in Tiva C series

Question / Comments / Suggestions?