Developing and Debugging C Programs in MDK-ARM for the STM32L100RC Microcontroller

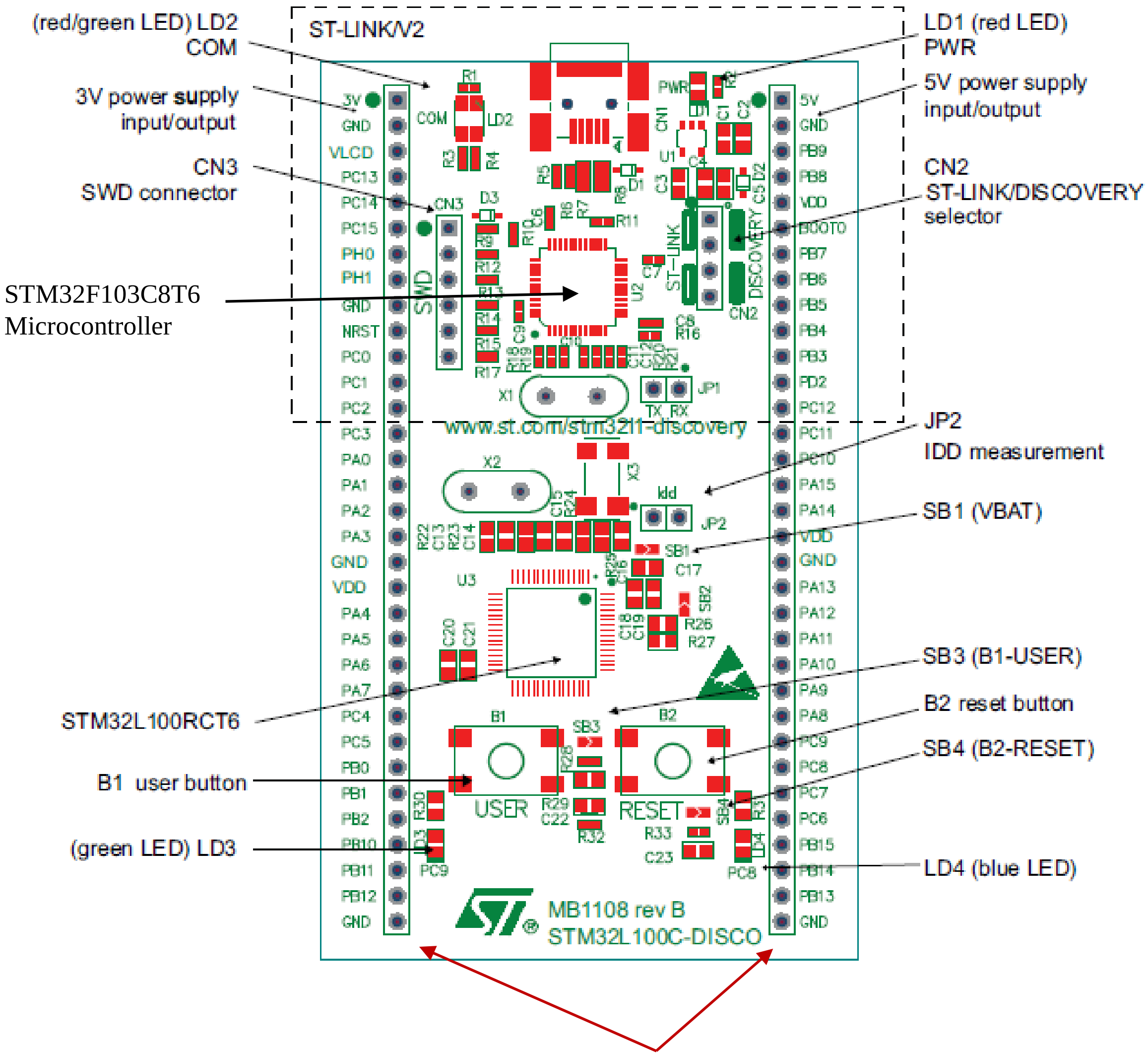
ELCE 3040/3050 – Lab Session 2 (write-up on course web page)

Important References (on course web page):

- *Tutorial: C programming for embedded microcontroller systems*
- *STM32L100C-Discovery User Manual (for the Discovery board)*
- *STM32L100 Microcontroller Data Sheet (for this specific microcontroller)*
- *STM32L100 Microcontroller Reference Manual (for all STM32Lxx microcontrollers)*

Top view of the STM32L100C-Discovery Board

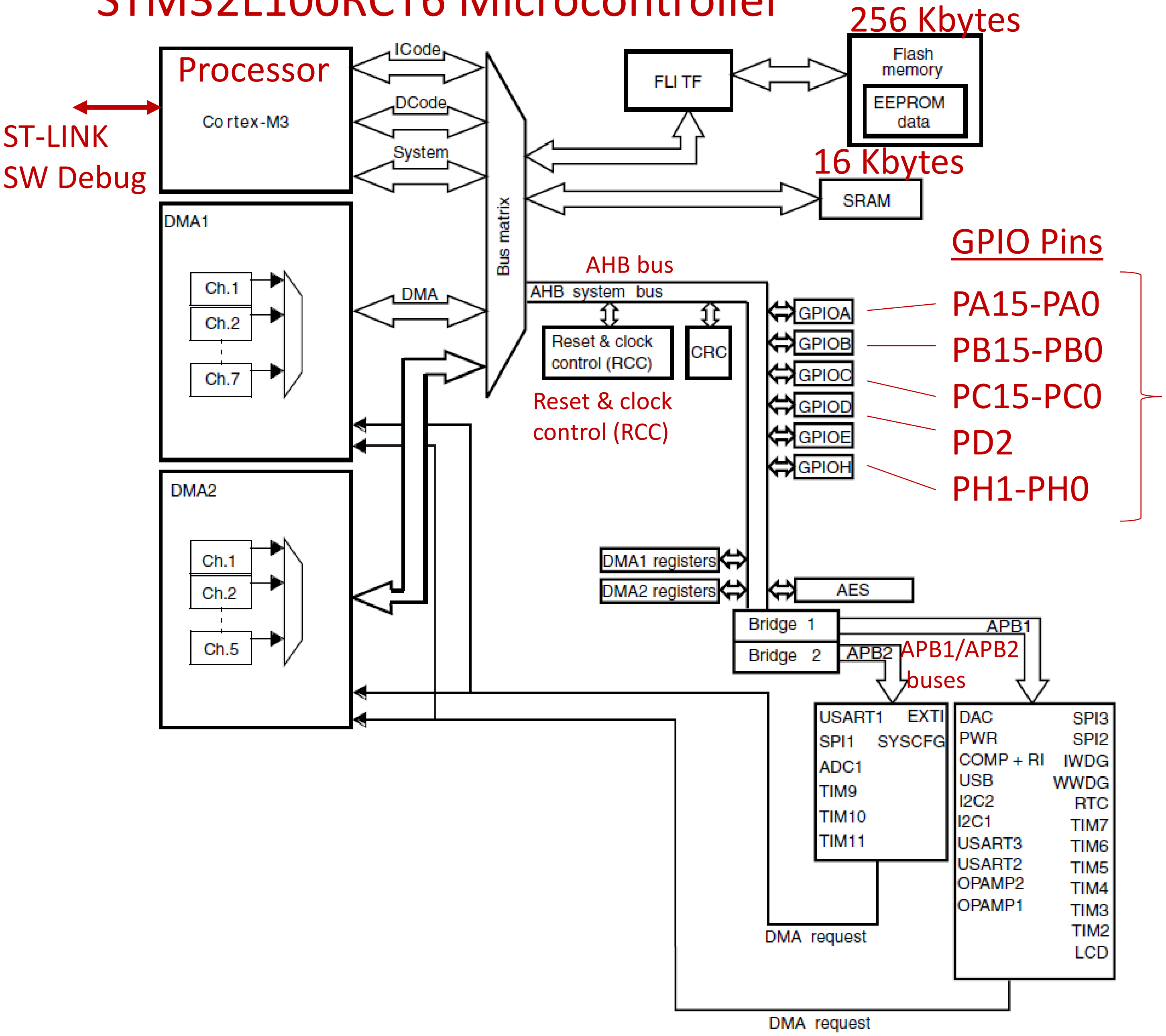
ST-LINK Microcontroller



Application Microcontroller

µC pins connect to pins on headers P1/P2

STM32L100RCT6 Microcontroller



Pins shared between GPIO and peripherals

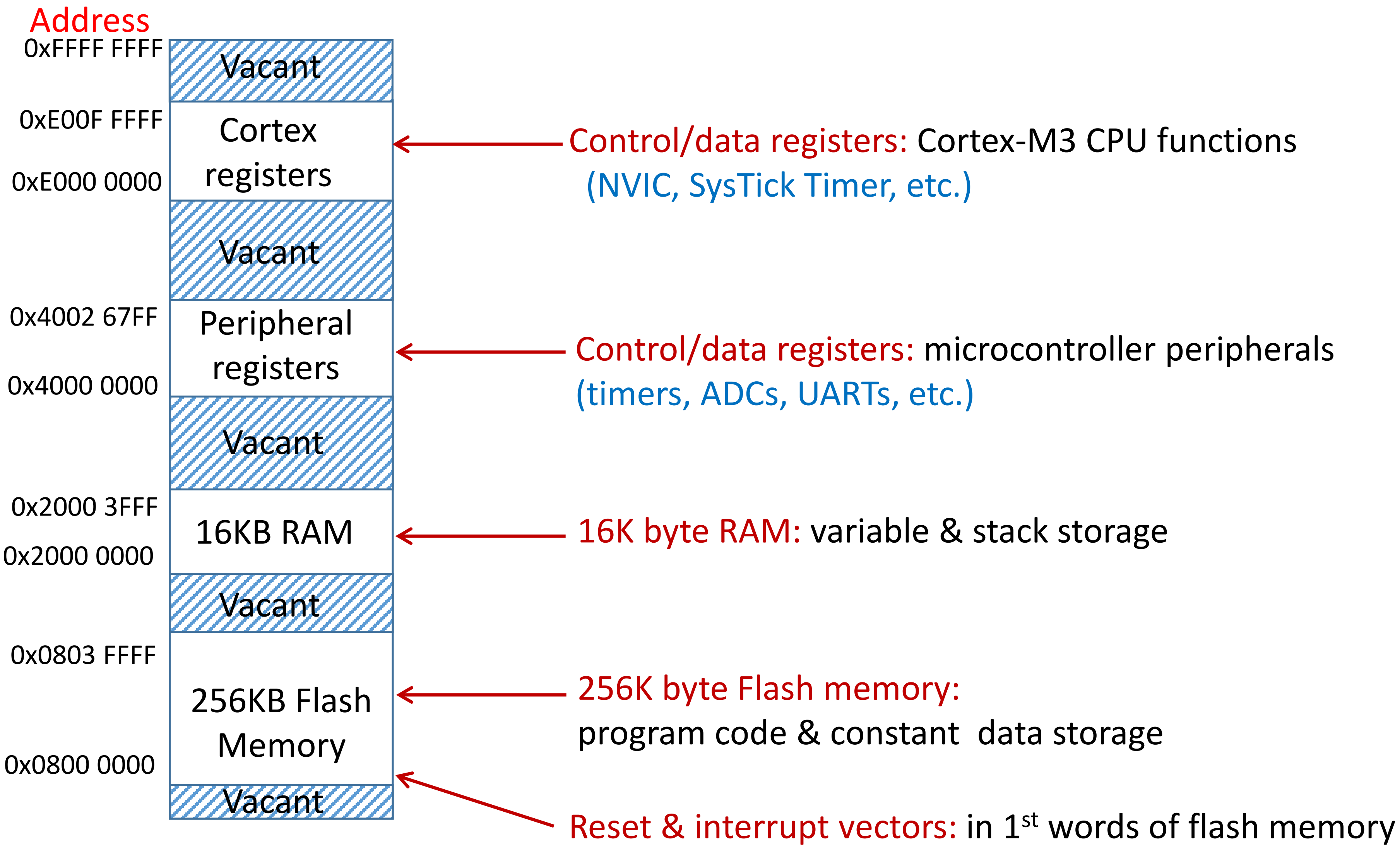
STM32L100C-Discovery Board

- PA0 ->User Button
- PC8 -> Blue LED
- PC9 -> Green LED
- PA13*/PA14* –> Serial-Wire (SW) Debug
- PH1*/PH0* –> 8MHz Clock input
- *** DO NOT CHANGE PIN MODES!**

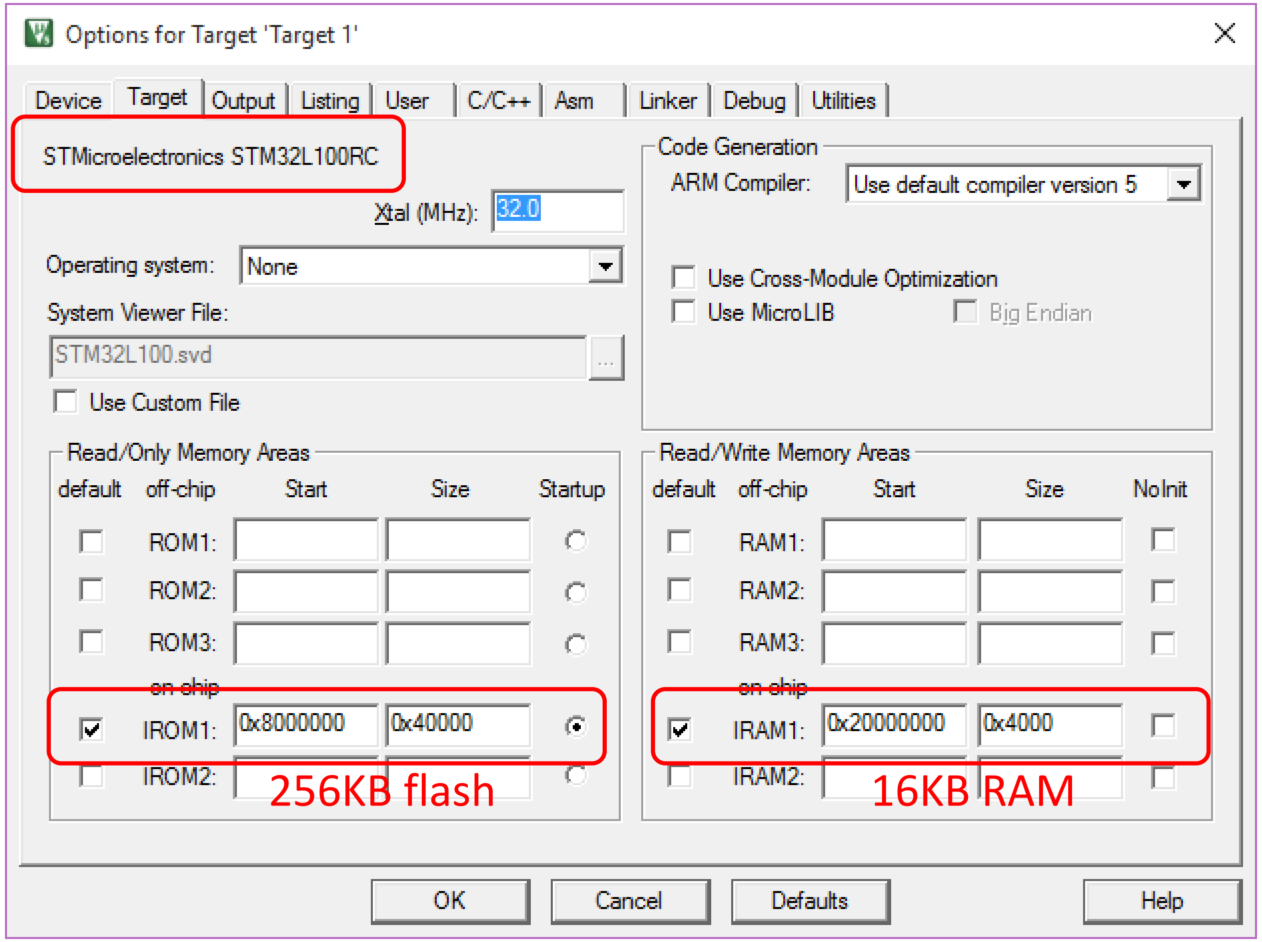
Peripheral Functions

- TIMx – timers
- ADCx – A/D converters
- DAC - D/A converter
- EXTI - external interrupts
- SYSCFG – system configuration
- SPIx - Serial Peripheral Interface
- I2Cx – Inter-Integrated Circuit bus
- USB – Universal Serial Bus
- USARTx – Univ. Sync/Async Receiver/Xmitter

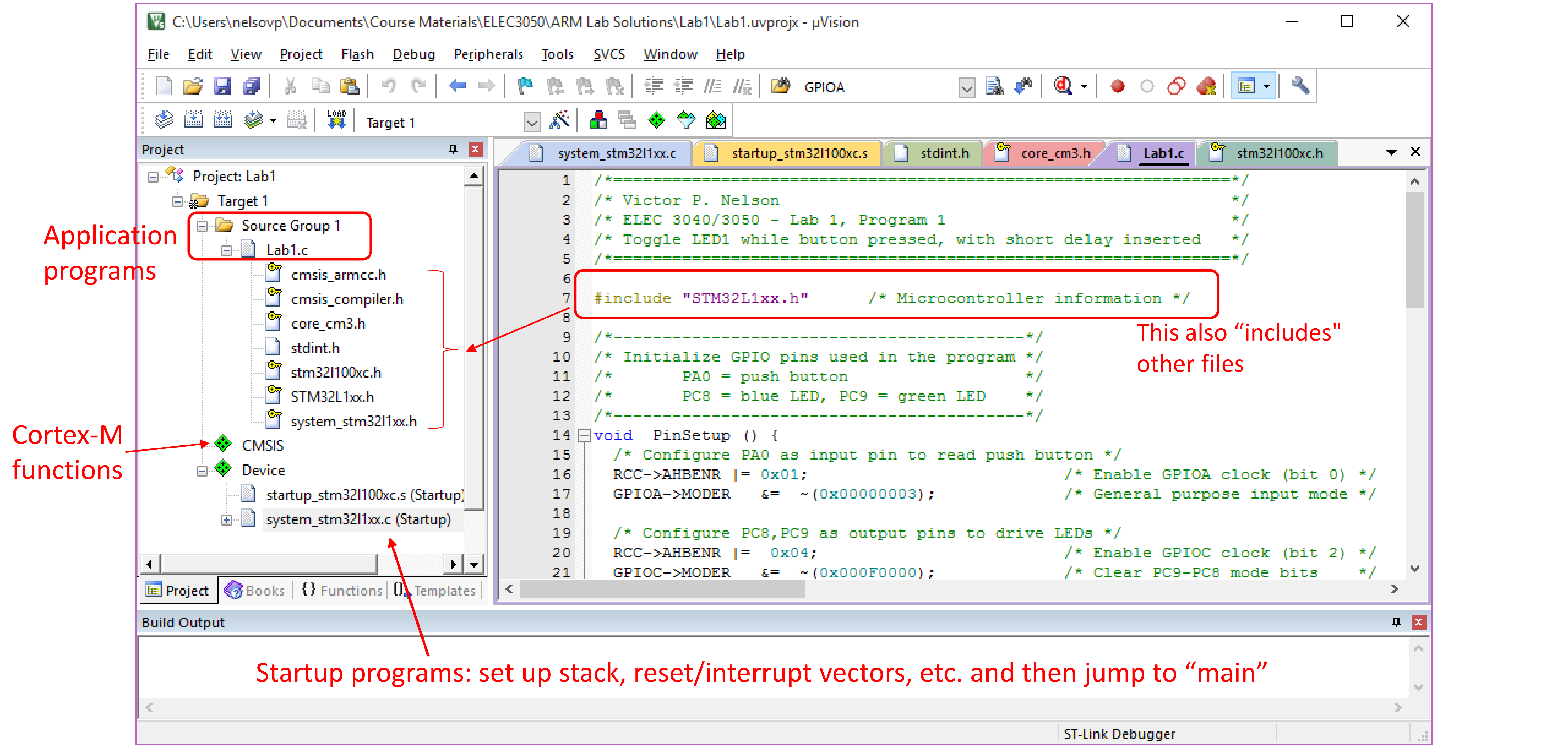
STM32L100RC µC memory map



Memory map automatically defined for the target microcontroller



C compiler automatically places program code in flash memory (IROM1) and data in RAM (IRAM1)


C:\Users\nelsovp\Documents\Course Materials\ELEC3050\ARM Lab Solutions\Lab1\Lab1.uvprojx - µVision
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
GPIOA
Target 1
Project
Project: Lab1
Target 1
Source Group 1
Lab1.c
cmsis_armcc.h
cmsis_compiler.h
core_cm3.h
stdint.h
stm32l100xc.h
STM32L1xx.h
system_stm32l1xx.h
CMSIS
Device
startup_stm32l100xc.s (Startup)
system_stm32l1xx.c (Startup)
Project Books Functions Templates
system_stm32l1xx.c startup_stm32l100xc.s stdint.h core_cm3.h Lab1.c stm32l100xc.h
/*==*/
/* Victor P. Nelson */
/* ELEC 3040/3050 - Lab 1, Program 1 */
/* Toggle LED1 while button pressed, with short delay inserted */
/*==*/
#include "STM32L1xx.h" /* Microcontroller information */
/*---*/
/* Initialize GPIO pins used in the program */
/* PA0 = push button */
/* PC8 = blue LED, PC9 = green LED */
/*---*/
void PinSetup () {
/* Configure PA0 as input pin to read push button */
RCC->AHBENR |= 0x01; /* Enable GPIOA clock (bit 0) */
GPIOA->MODER &= ~(0x00000003); /* General purpose input mode */
/* Configure PC8,PC9 as output pins to drive LEDs */
RCC->AHBENR |= 0x04; /* Enable GPIOC clock (bit 2) */
GPIOC->MODER &= ~(0x000F0000); /* Clear PC9-PC8 mode bits */
Build Output
ST-Link Debugger
Application programs
Cortex-M functions
This also "includes" other files
Startup programs: set up stack, reset/interrupt vectors, etc. and then jump to "main"

Basic C program structure

```
#include "STM32L1xx.h"      /* I/O port/register names/addresses for the STM32L1xx microcontrollers */

/* Global variables – accessible by all functions */
int count, bob;             //global (static) variables – placed in RAM

/* Function definitions*/
int function1(char x) {     //parameter x passed to the function, function returns an integer value
  int i,j;                  //local (automatic) variables – allocated to stack or registers
  -- instructions to implement the function
}

/* Main program */
void main(void) {
  unsigned char sw1;        //local (automatic) variable (stack or registers)
  int k;                    //local (automatic) variable (stack or registers)
/* Initialization section */
  -- instructions to initialize  variables, I/O ports, devices, function registers
/* Endless loop */
  while (1) {           //Can also use:  for(;;) {
   -- instructions to be repeated
  } /* repeat forever */
}
```

Declare local variables

Initialize variables/devices

Body of the program

C compiler data types

- Always match data type to data characteristics!

- Variable type indicates how data is represented
  - **#bits** determines range of numeric values
  - **signed/unsigned** determines which arithmetic/relational operators are to be used by the compiler
    - non-numeric data should be "unsigned"

- Header file "stdint.h" defines alternate type names for standard C data types
  - Eliminates ambiguity regarding #bits
  - Eliminates ambiguity regarding signed/unsigned

(Types defined on next page)

C compiler data types

Data type declaration *	Number of bits	Range of values
char k; unsigned char k; uint8_t k;	8	0..255
signed char k; int8_t k;	8	-128..+127
short k; signed short k; int16_t k;	16	-32768..+32767
unsigned short k; uint16_t k;	16	0..65535
int k; signed int k; int32_t k;	32	-2147483648.. +2147483647
unsigned int k; uint32_t k;	32	0..4294967295

* intx_t and uintx_t defined in *stdint.h*

Data type examples

- Read bits from GPIOA IDR (16 bits, non-numeric)
  - *uint16_t n; n = GPIOA->IDR; //or: unsigned short n;*
- Write TIM2 prescale value (16-bit unsigned)
  - *uint16_t t; TIM2->PSC = t; //or: unsigned short t;*
- Read 32-bit value from ADC (unsigned)
  - *uint32_t a; a = ADC; //or: unsigned int a;*
- System control value range [-1000…+1000]
  - *int32_t ctrl; ctrl = (x + y)*z; //or: int ctrl;*
- Loop counter for 100 program loops (unsigned)
  - *uint8_t cnt; //or: unsigned char cnt;*
  - *for (cnt = 0; cnt < 20; cnt++) {*

Constant/literal values

- Decimal is the default number format

```
int m,n;                //16-bit signed numbers
m = 453;   n = -25;
```

- Hexadecimal: preface value with 0x or 0X

```
m = 0xF312; n = -0x12E4;
```

- Octal: preface value with zero (0)

```
m = 0453; n = -023;
```

Don't use leading zeros on "decimal" values. They will be interpreted as octal.

- Character: character in single quotes, or ASCII value following "slash"

```
m = 'a';      //ASCII value 0x61
n = '\13';    //ASCII value 13 is the "return" character
```

- String (array) of characters:

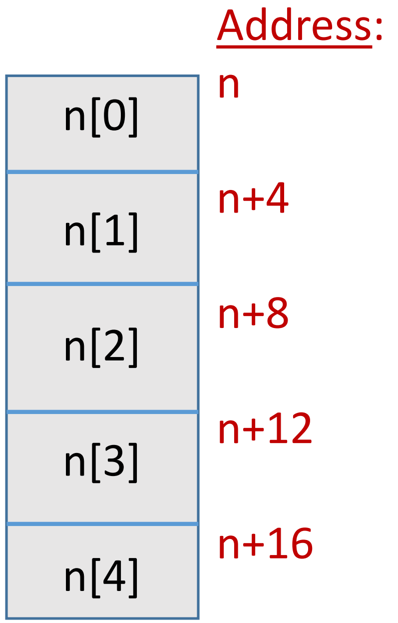
```
unsigned char k[7];
strcpy(m,"hello\n");  //k[0]='h', k[1]='e', k[2]='l', k[3]='l', k[4]='o',
                      //k[5]=13 or '\n' (ASCII new line character),
                      //k[6]=0 or '\0' (null character – end of string)
```

Variable arrays

- An *array* is a set of data, stored in consecutive memory locations, beginning at a named address
  - Declare array name and number of data elements, N
  - Elements are “indexed”, with indices [0 .. N-1]

int n[5]; //declare array of 5 “int” values

n[3] = 5; //set value of 4th array element



Note: Index of first element is always 0.

Automatic variables

- Declare within a function/procedure
- Variable is visible (has *scope*) only within that function
  - Space for the variable is allocated on the system *stack* when the procedure is entered
    - Deallocated, to be re-used, when the procedure is exited
  - If only 1 or 2 variables, the compiler may allocate them to registers within that procedure, instead of allocating memory.
  - Values are not retained between procedure calls

Automatic variable example

```
void delay () {
  int i,j;     //automatic variables – visible only within delay()
  for (i=0; i<100; i++) {          //outer loop
     for (j=0; j<20000; j++) {   //inner loop
     }                                          //do nothing
  }
}
```

Variables must be initialized each time the procedure is entered since values are not retained when the procedure is exited.

MDK-ARM (in my example): allocated registers r0,r2 for variables i,j

Static variables

- Retained for use throughout the program in RAM locations that are *not reallocated* during program execution.
- Declare either within or outside of a function
  - If declared outside a function, the variable is *global* in scope, i.e. known to all functions of the program
    - Use "normal" declarations. Example: *int count;*
  - If declared within a function, insert key word *static* before the variable definition. The variable is *local* in scope, i.e. known only within this function.

static unsigned char bob;

static int pressure[10];

Static variable example

```
unsigned char count;  //global variable is static – allocated a fixed RAM location
                      //count can be referenced by any function
void math_op () {
  int i;              //automatic variable – allocated space on stack when function entered
  static int j;       //static variable – allocated a fixed RAM location to maintain the value
  if (count == 0)     //test value of global variable count
     j = 0;           //initialize static variable j first time math_op() entered
  i = count;          //initialize automatic variable i each time math_op() entered
  j = j + i;          //change static variable j – value kept for next function call
}                     //return & deallocate space used by automatic variable i

void main(void) {
  count = 0;          //initialize global variable count
  while (1) {
    math_op();
    count++;          //increment global variable count
  }
}
```

Configuring GPIO pins (from Lab 1 program)

* Each peripheral module gets a separate clock signal from the RCC module.
(all clocks are initially disabled to save energy)

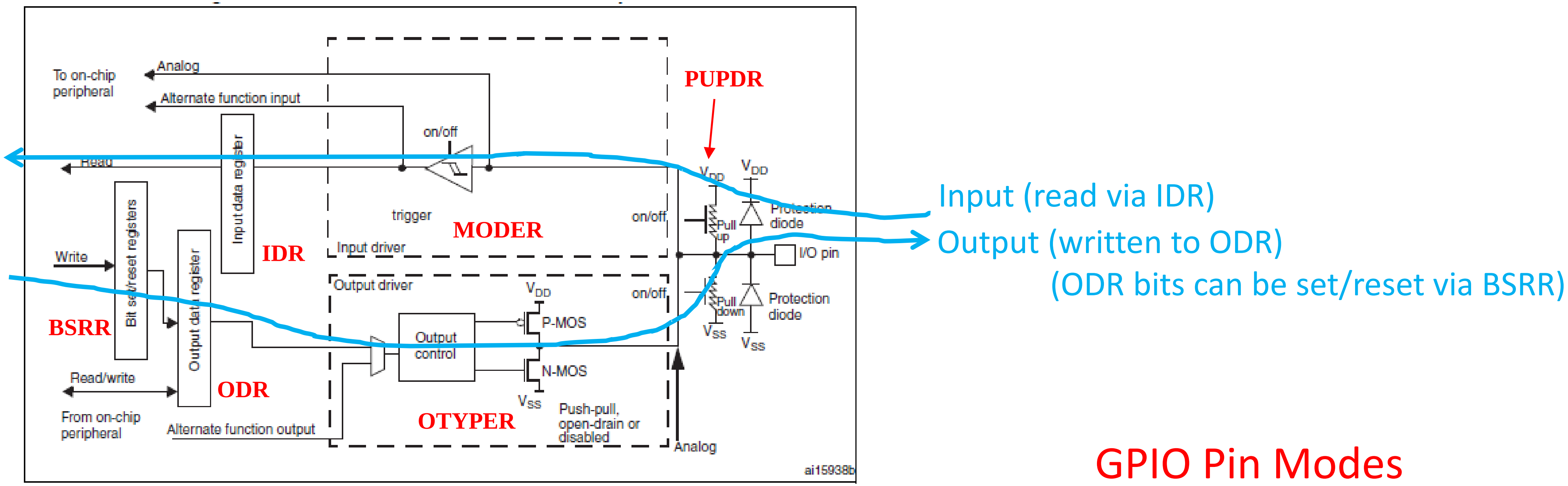
```
/* Initialize GPIO pins in program for Lab 1 */
void  PinSetup () {
    /* Configure PA0 as input pin to read push button */
(1) RCC->AHBENR |= 0x01;                        /* Enable GPIOA clock* (bit 0) */
(2) GPIOA->MODER    &=  ~(0x00000003);          /* General purpose input mode */

    /* Configure PC8,PC9 as output pins to drive LEDs */
(1) RCC->AHBENR |=  0x04;                       /* Enable GPIOC clock* (bit 2)  */
(2) GPIOC->MODER    &=  ~(0x000F0000);          /* Clear PC9-PC8 mode bits      */
    GPIOC->MODER     |=    (0x00050000);        /* General purpose output mode*/
(1) GPIOC->ODR        |=    (0x0300);           /* Initialize PC9-PC8 to 00       */
}
```

1. Turn on clock to GPIO module
2. Configure modes of GPIO pins
3. Initialize output values

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

GPIOC->MODER

GPIO Pin Electronics



GPIO Mode Register (MODER)

Two bits for each of 16 pins

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

GPIO Pin Modes

MODERn[1:0] = 00: Digital input mode **
(n = pin#) 01: General-purpose digital output mode
10: Alternate function mode
11: Analog mode

** Reset state – except PA[15:13], PB[4:3]

From header file ***stm32l100xc.h*** (automatically included in project)

Allows GPIO registers to be accessed as variables of a "record" structure in C

```
/* Base address of all STM microcontroller peripherals */
#define PERIPH_BASE ((uint32_t)0x40000000)          //Peripheral base address
#define AHBPERIPH_BASE (PERIPH_BASE + 0x20000)  //AHB peripherals

/* Base addresses of blocks of GPIO control/data registers */
#define GPIOA_BASE (AHBPERIPH_BASE + 0x0000)     //Registers for GPIOA
#define GPIOB_BASE (AHBPERIPH_BASE + 0x0400)     //Registers for GPIOB

/* Address offsets from GPIO base address –   block of registers defined as a "structure" */
typedef struct
{
  __IO uint32_t MODER;        // GPIO port mode register,                    Address offset: 0x00
  __IO uint16_t OTYPER;       // GPIO port output type register,             Address offset: 0x04
       uint16_t RESERVED0;    // Reserved,                                                   0x06
  __IO uint32_t OSPEEDR;      // GPIO port output speed register,            Address offset: 0x08
  __IO uint32_t PUPDR;        // GPIO port pull-up/pull-down register,       Address offset: 0x0C
  __IO uint16_t IDR;          // GPIO port input data register,              Address offset: 0x10
       uint16_t RESERVED1;    // Reserved,                                                   0x12
  __IO uint16_t ODR;          // GPIO port output data register,             Address offset: 0x14
       uint16_t RESERVED2;    // Reserved,                                                   0x16
  __IO uint16_t BSRR;         // GPIO port bit set/reset register BSRR,      Address offset: 0x18
  __IO uint32_t LCKR;         // GPIO port configuration lock register,      Address offset: 0x1C
  __IO uint32_t AFR[2];       // GPIO alternate function low register,  Address offset: 0x20-0x24
} GPIO_TypeDef;
```

```
GPIOA->ODR = 0x113A;
N = GPIOB->IDR;
GPIOA->MODER &= 0xFFFFFFFC;
```

Base -> Offset

Address

```
/* Declare the peripherals */
#define GPIOA          ((GPIO_TypeDef *) GPIOA_BASE)
#define GPIOB          ((GPIO_TypeDef *) GPIOB_BASE)
```

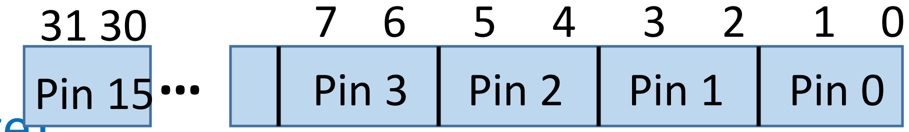
Create pointers to the GPIO module base address

GPIO "mode" register

- GPIOx->MODER selects operating mode for each pin

  x = A…I (GPIOA, GPIOB, …, GPIOI)

- 2 bits per pin:



00 – Input mode (reset state):

Pin value captured in IDR every bus clock (through Schmitt trigger)

01 – General purpose output mode:

- Write pin value to ODR
- Read IDR to determine pin state
- Read ODR for last written value

10 – Alternate function mode:

Select alternate function via AF mux/register (see later slide)

11 – Analog mode:

Disable output buffer, input Schmitt trigger, pull resistors

(so as not to alter the analog voltage on the pin)

GPIO data registers

- 16-bit data registers for each port GPIOx

  x = A…I (GPIOA, GPIOB, …, GPIOI)

- GPIOx->IDR
  - Data input through the 16 pins
  - Read-only
- GPIOx->ODR
  - Write data to be output to the 16 pins
  - Read last value written to ODR
  - Read/write (for read-modify-write operations)
- C examples:

```
GPIOA->ODR = 0x45;        //send data to output pins
N = GPIOA->IDR;  //copy data from in pins to N
```

GPIO port bit set/reset registers

- GPIO output bits can be <u>individually</u> set and cleared

  (*without affecting other bits in that port)*
- GPIOx_BSRR (Bit Set/Reset Register)
  - Bits [15..0] = Port x **set** bit y (y = 15..0) (BSRRL)
  - Bits [31..16] = Port x **reset** bit y (y = 15..0) (BSRRH)
  - Bits are *write-only*
    - 1 = Set/reset the corresponding GPIOx bit
    - 0 = No action on the corresponding GPIOx bit

  (“set” has precedence if bit=1 in both BSSRL and BSSRH)
- C examples:

```
GPIOA->BSRRL = (1 << 4);  //set bit 4 of GPIOA
GPIOA->BSRRH = (1 << 5);  //reset bit 5 of GPIOA
```

Transferring data to/from GPIO pins

Reading input pin states via the IDR (test state of pin PA0):

```
uint16_t  bob;              //16-bit variable matches IDR size
bob = GPIOA->IDR;           //read states of all 16 PA[15:0] pins
bob = bob & 0x0001;         //mask all but bit 0 to test PA0
if (bob == 0x0001)...       //do something if PA0=1
```

Alternatively:

```
if ((GPIOA->IDR & 0x0001) == 0x0001)...  //do something if PA0=1
```

Common error :

```
if (GPIOA->IDR == 0x0001)  //TRUE only if all 16 pin states match this pattern
```

Write to the 16 pins via ODR (only affects pins configured as outputs – other pin values ignored)

```
GPIOB->ODR = 0x1234;  //set PB[15:0] = 0001001000110100
```

Reading ODR returns last value written to it:

```
GPIOB->ODR &= 0xFFFE; //reset PB0=0 (without changing PB[15:1])
GPIOB->ODR |= 0x0001; //set PB0=1 (without changing PB[15:1])
GPIOB->ODR ^= 0x0001; //complement PB0 state
```

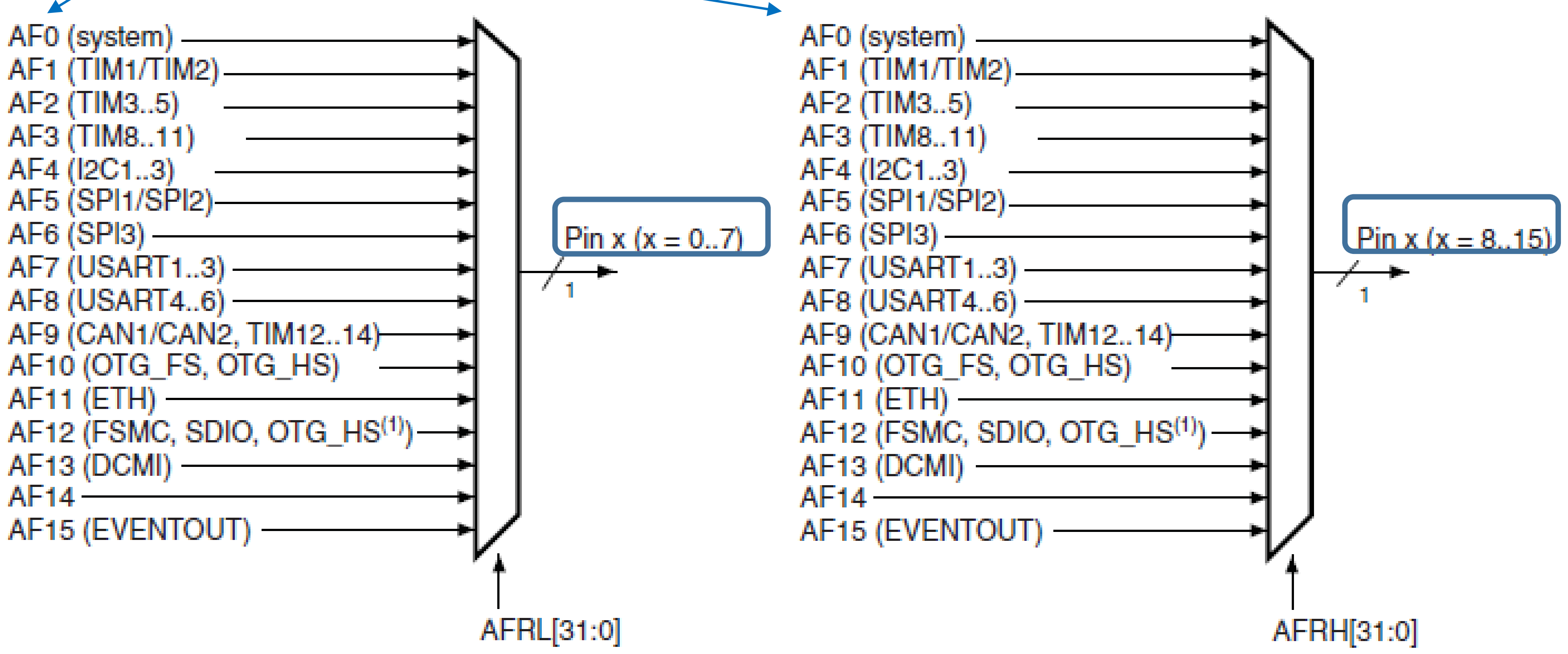
1 written to a bit of BSRR[15:0] sets corresponding GPIO bit Px15-Px0 (writing 0s has no effect)

1 written to a bit of BSRR[31:16] resets/clears corresponding bits Px15-Px0 (writing 0s has no effect)

```
GPIOB->BSRR = 0x0021;       //set   PB5 and PB0 to 1
GPIOB->BSRR = 0x0021 << 16; //reset PB5 and PB0 to 0 (write to upper 16 bits of BSRR)
```

Alternate function selection

Each pin defaults to GPIO pin at reset (mux input 0)



GPIOx_AFRL
(low pins 0..7)

GPIOx_AFRH
(high pins 8..15)

Other GPIO pin options

Modify these registers for other than default configuration

- GPIOx_OTYPER – output type
  - 0 = push/pull (reset state)
  - 1 = open drain
- GPIOx_PUPDR – pull-up/down
  - 00 – no pull-up/pull-down (reset state)
  - 01 – pull-up
  - 10 – pull-down
- GPIOx_OSPEEDR – output speed
  - 00 – 2 MHz low speed (reset state)
  - 01 – 25 MHz medium speed
  - 10 – 50 MHz fast speed
  - 11 – 100 MHz high speed (on 30 pf)

Setting/Clearing Selected Register Bits

Example: Configure pin PC5 as an input pin and PC8 as an output pin,
without changing the operating modes of the other 14 GPIOC pins.

Clear corresponding MODER bits for each pin, using a logical AND operator with a mask to force these bits to 00.
Use logical OR operator with another mask to force selected MODER bits to 1 to produce the desired 2-bit values.
For example, to set bits 5:4 of a register to the value "mn":

	Bit#:	9876543210
Current register bits:		abcdefghij
AND with mask to clear bits 5-4:		1111001111
	Result:	abcd00ghij
OR with mask to set bits 5-4 to mn:		0000mn0000
	Result:	abcdmnghij

Mode Register (MODER)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Mode bits MODER5 (bits 11:10) and MODER8 (bits 17:16), for pins PC5 and PC8, are forced to 00 by reading the current MODER and applying a logical AND operator to clear those bits in one of the following ways:

```
GPIOC->MODER = GPIOC->MODER &  0xFFFCF3FF;  //MODER8=MODER5=00  (digital input mode)
GPIOC->MODER = GPIOC->MODER & ~0x00030C00;  //MODER8=MODER5=00
GPIOC->MODER &=  0xFFCFF3FF;                //MODER8=MODER5=00
GPIOC->MODER &= ~0x00300C00;                //MODER8=MODER5=00
```

0xFFFCF3FF (= ~0x00030C00) contains 0's in bits corresponding to pins PC8 and PC5.

To configure PC8 in output mode (MODER8 = 01), use the logical OR operator to set the low bit of MODER8 to 1:

```
GPIOC->MODER  = GPIOC->MODER | 0x00010000;  //MODER8=01
GPIOC->MODER |= 0x00010000;                 //MODER8=01
```

Although we could simply write a 32-bit pattern to MODER to configure all 16 pins in one step, it is good practice to change only those bits for the specific pins to be configured, using logical AND/OR operators, and thereby avoid inadvertently changing the previously-configured modes of other pins.

Lab 1: Main Program

```
int main(void) {
  unsigned char sw1;                    //state of SW1
  unsigned char led1;                   //state of LED1
  PinSetup();                           //Configure GPIO pins
  led1 = 0;                             //Initial LED state
  toggles = 0;                          //#times LED state changed
  /* Endless loop */
  while (1) {       //Can also use:  for(;;) {
        if (led1 == 0)                                  //LED off?
              GPIOC->BSRR = 0x0100 << 16;               //Reset PC8=0 to turn OFF blue LED (in BSRR upper half)
        else                                            //LED on
              GPIOC->BSRR = 0x0100;                     //Set PC8=1 to turn ON blue LED (in BSRR low half)
        sw1 = GPIOA->IDR & 0x01;                        //Read GPIOA and mask all but bit 0

        /* Wait in loop until SW1 pressed */
        while (sw1 == 0) {                              //Wait for SW1 = 1 on PE0
              sw1 = GPIOA->IDR & 0x01;                  //Read GPIOA and mask all but bit 0
        }
        delay();                                        //Time delay for button release
        led1 = ~led1;                                   //Complement LED1 state
        toggles++;                                      //Increment #times LED toggled
  } /* repeat forever */
}
```

Lab 1: Delay Function

```
/*-------------------------------------------------------*/
/* Delay function - do nothing for about 1 second */
/*-------------------------------------------------------*/
void delay () {
  int i,j,n;                          //local variables –always undefined when function entered
  for (i=0; i<20; i++) {              //outer loop
      for (j=0; j<20000; j++) {       //inner loop
          n = j;                      //dummy operation for single-step test
      }                               //do nothing
  }
}
```

Lab 2 Exercise
(See posted lab write-up)

- Decimal up/down counter (0-9) displayed on LEDs
- Switch S1: start (1) or stop (0) counting
- Switch S2: count down (0) or up (1)   (roll over between 0-9)
- S1/S2 = "virtual switches" in the *Waveforms* Static I/O Tool
- Show count on "virtual LEDs" in the *Waveforms* Static I/O Tool
- Main program + three "functions"
    - Main program does initialization and then enters an "endless loop"
        - Call delay function, check S2 to set direction, call count function if S1=1
    - Delay function: half-second time delay (do nothing for half a second)
    - Counting function: increment or decrement the count
    - GPIO initialization function
- **Exercise debug features**