Embedded C

Microcontroller Application and Development

Sorayut Glomglome

C programming for embedded microcontroller systems.

Assumes experience with assembly language programming.

V. P. Nelson

Outline

- Program organization and microcontroller memory
- Data types, constants, variables
- Microcontroller register/port addresses
- Operators: arithmetic, logical, shift
- Control structures: if, while, for
- Functions
- Interrupt routines

Basic C program structure

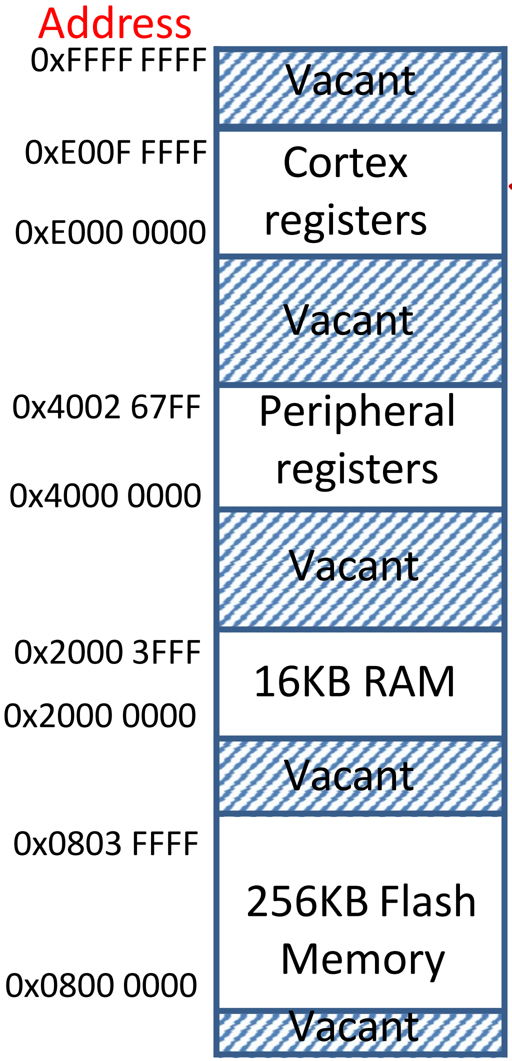
```
#include "stm32f767xx.h"      /* I/O port/register names/addresses for the STM32F767xx microcontrollers */
/* Global variables – accessible by all functions */
int count, bob;               //global (static) variables – placed in RAM
/* Function definitions*/
int function1(char x) {       //parameter 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
/* Infinite loop */
  while (1) {           //Can also use: for(;;) {
   -- instructions to be repeated
  } /* repeat forever */
}
```

Declare local variables

Initialize variables/devices

Body of the program

STM32L100RC µC memory map



Control/data registers: Cortex-M3 CPU functions
(NVIC, SysTick Timer, etc.)

Control/data registers: microcontroller peripherals
(timers, ADCs, UARTs, etc.)

16K byte RAM: variable & stack storage

256K byte Flash memory:
program code & constant data storage

Reset & interrupt vectors: in 1st words of flash memory

Microcontroller "header file"

- *Keil MDK-ARM* provides a *derivative-specific* "header file" for each microcontroller, which defines memory addresses and symbolic labels for CPU and peripheral function register addresses.

```
#include "stm32f767xx.h"        /* target uC information */

// GPIOA configuration/data register addresses are defined in stm32f767xx.h

void main(void) {
  uint16_t PAval;                              //16-bit unsigned variable

  GPIOA->MODER   &=  ~(0x00000003);            // Set GPIOA pin PA0 as input
  PAval = GPIOA->IDR;                          // Set PAval to 16-bits from GPIOA

  for(;;) {}          /* execute forever */
}
```

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 (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)

Program variables

- A *variable* is an addressable storage location to information to be used by the program
  - Each variable must be *declared* to indicate size and type of information to be stored, plus name to be used to reference the information

    int x,y,z; *//declares 3 variables of type "int"*

    char a,b; *//declares 2 variables of type "char"*

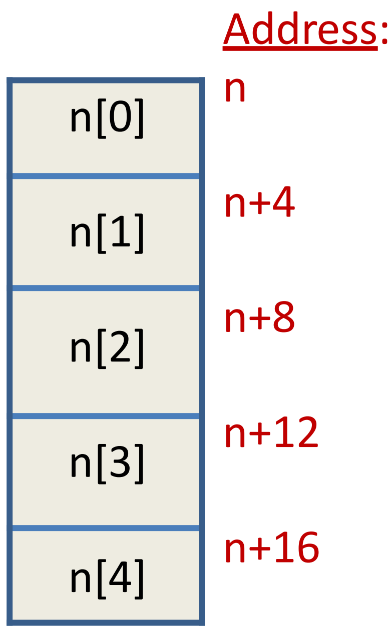
  - Space for variables may be allocated in registers, RAM, or ROM/Flash (for constants)
  - Variables can be *automatic* or *static*

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.

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, 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
  }
}
```

C statement types

- Simple variable assignments
  - Includes input/output data transfers
- Arithmetic operations
- Logical/shift operations
- Control structures
  - IF, WHEN, FOR, SELECT
- Function calls
  - User-defined and/or library functions

Arithmetic operations

- C examples – with standard arithmetic operators

```
int i, j, k;        // 32-bit signed integers
uint8_t m,n,p;      // 8-bit unsigned numbers
i = j + k;          // add 32-bit integers
m = n - 5;          // subtract 8-bit numbers
j = i * k;          // multiply 32-bit integers
m = n / p;          // quotient of 8-bit divide
m = n % p;          // remainder of 8-bit divide
i = (j + k) * (i - 2);  //arithmetic expression
```

*, /, % are higher in precedence than +, - (higher precedence applied 1st)

Example: j * k + m / n = (j * k) + (m / n)

Floating-point formats are not directly supported by Cortex-M3 CPUs.

Bit-parallel logical operators

Bit-parallel (bitwise) logical operators produce n-bit results of the corresponding logical operation:

& (AND) | (OR)   ^ (XOR)   ~ (Complement)

```
C = A & B;          A    0 1 1 0 0 1 1 0
(AND)               B    1 0 1 1 0 0 1 1
                    C    0 0 1 0 0 0 1 0

C = A | B;          A    0 1 1 0 0 1 0 0
(OR)                B    0 0 0 1 0 0 0 0
                    C    0 1 1 1 0 1 0 0

C = A ^ B;          A    0 1 1 0 0 1 0 0
(XOR)               B    1 0 1 1 0 0 1 1
                    C    1 1 0 1 0 1 1 1

B = ~A;             A    0 1 1 0 0 1 0 0
(COMPLEMENT)        B    1 0 0 1 1 0 1 1
```

Bit set/reset/complement/test

- Use a “mask” to select bit(s) to be altered

```
C = A & 0xFE;       A    a b c d e f g h
                 0xFE    1 1 1 1 1 1 1 0     Clear selected bit of A
                    C    a b c d e f g 0
```

```
C = A & 0x01;       A    a b c d e f g h
                 0xFE    0 0 0 0 0 0 0 1     Clear all but the selected bit of A
                    C    0 0 0 0 0 0 0 h
```

```
C = A | 0x01;       A    a b c d e f g h
                 0x01    0 0 0 0 0 0 0 1     Set selected bit of A
                    C    a b c d e f g 1
```

```
C = A ^ 0x01;       A    a b c d e f g h
                0x01     0 0 0 0 0 0 0 1     Complement selected bit of A
                    C    a b c d e f g h’
```

Bit examples for input/output

- Create a "pulse" on bit 0 of PORTA (assume bit is initially 0)

```
PORTA = PORTA | 0x01;   //Force bit 0 to 1
PORTA = PORTA & 0xFE;   //Force bit 0 to 0
```

- Examples:

```
if ( (PORTA & 0x80) != 0 ) //Or: ((PORTA & 0x80) == 0x80)
    bob();                  // call bob() if bit 7 of PORTA is 1
c = PORTB & 0x04;           // mask all but bit 2 of PORTB value
if ((PORTA & 0x01) == 0)    // test bit 0 of PORTA
    PORTA = c | 0x01;       // write c to PORTA with bit 0 set to 1
```

Bit examples for input/output

- Create a "pulse" on bit 0 of PORTA (assume bit is initially 0)

```
#define GPIO_PIN_0       ( (uint16_t) 0x0001U )
GPIOA -> ODR = GPIOA -> ODR | 0x01UL;  //Force bit 0 to 1
GPIOA -> ODR = GPIOA -> ODR & 0xFEUL;  //Force bit 0 to 0
GPIOA -> ODR = GPIOA -> ODR & ~(0x01UL);  //Force bit 0 to 0
```

- Examples:

```
if ( (GPIOA->IDR & 0x80) != 0UL ) //Or: ((GPIOA->IDR & 0x80UL) == 0x80UL)
     bob();                              // call bob() if bit 7 of PORTA is 1
c = GPIOB->IDR & 0x04UL;                 // mask all but bit 2 of PORTB value
if ((GPIOA->IDR & 0x01UL) == 0UL)        // test bit 0 of PORTA
     GPIOA->ODR  = c | 0x01;             // write c to PORTA with bit 0 set to 1
```

Example of µC register address definitions in *stm32f767xx.h*

(read this header file to view other peripheral functions)

```
#define PERIPH_BASE          ((uint32_t) 0x40000000UL)             //Peripheral base address in memory
#define AHB1PERIPH_BASE      (PERIPH_BASE + 0x00020000UL)          //AHB peripherals
/* Base addresses of blocks of GPIO control/data registers */
#define GPIOA_BASE       (AHB1PERIPH_BASE + 0x0000UL)              //Registers for GPIOA
#define GPIOB_BASE       (AHB1PERIPH_BASE + 0x0400UL)              //Registers for GPIOB
#define GPIOA            ((GPIO_TypeDef *) GPIOA_BASE)             //Pointer to GPIOA register block
#define GPIOB            ((GPIO_TypeDef *) GPIOB_BASE)             //Pointer to GPIOB register block
/* 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       */
  __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 uint32_t IDR;       /*!< GPIO port input data register,        Address offset: 0x10       */
  __IO uint32_t ODR;       /*!< GPIO port output data register,       Address offset: 0x14       */
  __IO uint32_t BSRR;      /*!< GPIO port bit set/reset register,     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;
```

Example: I/O port bits (using bottom half of GPIOB)

	7	6	5	4	3	2	1	0
GPIOB	h	g	f	e	d	c	b	a

Switch connected to bit 4 (PB4) of GPIOB

```
uint32_t sw;                    //32-bit unsigned type since GPIOB IDR and ODR = 32 bits
sw = GPIOB->IDR;                // sw = xxxxxxxxhgfedcba (upper 8 bits from PB15-PB8)
sw = GPIOB->IDR & 0x0010;       // sw = 000e0000 (mask all but bit 4)
                                // Result is sw = 00000000 or 00010000
if (sw == 0x01)                 // NEVER TRUE for above sw, which is 000e0000
if (sw == 0x10)                 // TRUE if e=1 (bit 4 in result of PORTB & 0x10)
if (sw == 0)                    // TRUE if e=0 in PORTB & 0x10 (sw=00000000)
if (sw != 0)                    // TRUE if e=1 in PORTB & 0x10 (sw=00010000)
GPIOB->ODR = 0x005a;            // Write to 16 bits of GPIOB; result is 01011010
GPIOB->ODR |= 0x10;             // Sets only bit e to 1 in GPIOB (GPIOB now hgf1dcba)
GPIOB->ODR &= ~0x10;            // Resets only bit e to 0 in GPIOB (GPIOB now hgf0dcba)
if ((GPIOB->IDR & 0x10) == 1)   // TRUE if e=1 (bit 4 of GPIOB)
```

Shift operators

Shift operators:

- x >> y (right shift operand x by y bit positions)
- x << y (left shift operand x by y bit positions)

Vacated bits are filled with 0's.

Shift right/left fast way to multiply/divide by power of 2

```
B = A << 3;                A   1 0 1 0 1 1 0 1
(Left shift 3 bits)        B   0 1 1 0 1 0 0 0

B = A >> 2;                A   1 0 1 1 0 1 0 1
(Right shift 2 bits)       B   0 0 1 0 1 1 0 1

B = '1';                   B = 0 0 1 1 0 0 0 1 (ASCII 0x31)
C = '5';                   C = 0 0 1 1 0 1 0 1 (ASCII 0x35)
D = (B << 4) | (C & 0x0F);
        (B << 4)         = 0 0 0 1 0 0 0 0
       (C & 0x0F)        = 0 0 0 0 0 1 0 1
               D         = 0 0 0 1 0 1 0 1 (Packed BCD 0x15)
```

C control structures

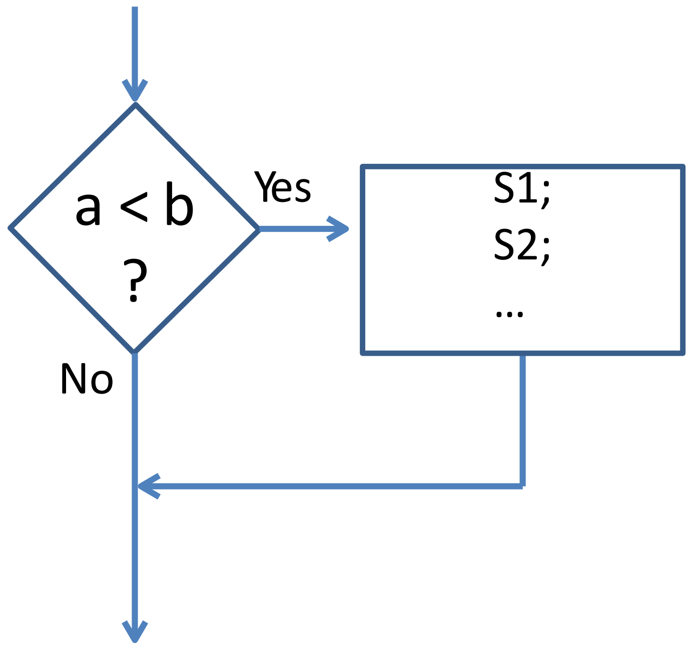
- Control order in which instructions are executed (program flow)
- Conditional execution
  - Execute a set of statements if some condition is met
  - Select one set of statements to be executed from several options, depending on one or more conditions
- Iterative execution
  - Repeated execution of a set of statements
    - A specified number of times, or
    - Until some condition is met, or
    - While some condition is true

IF-THEN structure

- Execute a set of statements if and only if some condition is met

TRUE/FALSE condition

```
if (a < b)
   {
     statement s1;
     statement s2;
     ....
   }
```



Relational Operators

- Test relationship between two variables/expressions

Test	TRUE condition	Notes
(m == b)	m equal to b	Double =
(m != b)	m not equal to b	
(m < b)	m less than b	1
(m <= b)	m less than or equal to b	1
(m > b)	m greater than b	1
(m >= b)	m greater than or equal to b	1
(m)	m non-zero	
(1)	always TRUE	
(0)	always FALSE	

1. Compiler uses signed or unsigned comparison, in accordance with data types

Example:

unsigned char a,b;
int j,k;
if (a < b) – unsigned
if (j > k) - signed

Boolean operators

- Boolean operators && (AND) and || (OR) produce TRUE/FALSE results when testing multiple TRUE/FALSE conditions

```
if ((n > 1) && (n < 5))    //test for n between 1 and 5
if ((c == 'q') || (c == 'Q')) //test c = lower or upper case Q
```

- Note the difference between Boolean operators &&, || and bitwise logical operators &, |

```
if ( k && m)    //test if k and m both TRUE (non-zero values)
if ( k & m)     //compute bitwise AND between m and n,
                //then test whether the result is non-zero (TRUE)
```

Common error

- Note that == is a <u>relational</u> operator, whereas = is an <u>assignment</u> operator.

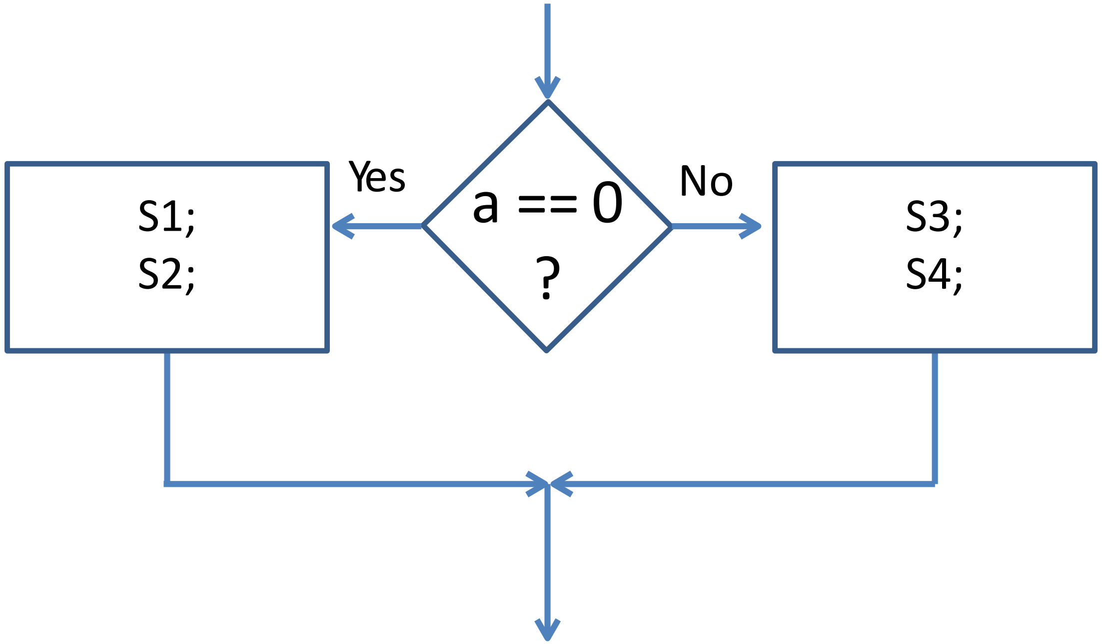
```
if ( m == n)  //tests equality of values of variables m and n
if (m = n)    //assigns value of n to variable m, and then
              //tests whether that value is TRUE (non-zero)
```

The second form is a common error (omitting the second equal sign), and usually produces unexpected results, namely a TRUE condition if n is 0 and FALSE if n is non-zero.

IF-THEN-ELSE structure

- Execute one set of statements if a condition is met and an alternate set if the condition is not met.

```
if (a == 0)
  {
    statement s1;
    statement s2;
  }
else
 {
   statement s3;
   statement s4:
 }
```



Multiple ELSE-IF structure

- Multi-way decision, with expressions evaluated in a specified order

```
if (n == 1)
   statement1;     //do if n == 1
 else if (n == 2)
   statement2;   //do if n == 2
 else if (n == 3)
   statement3;  //do if n == 3
else
   statement4;  //do if any other value of n (none of the above)
```

Any “statement” above can be replaced with a set of statements: {s1; s2; s3; …}

SWITCH statement

- Compact alternative to ELSE-IF structure, for multi-way decision that tests one variable or expression for a number of constant values

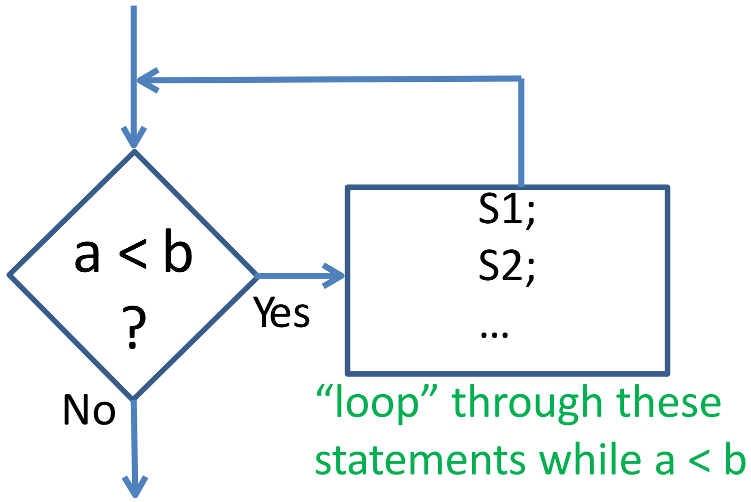
```
/* example equivalent to that on preceding slide */
 switch ( n) {          //n is the variable to be tested
     case 0: statement1; //do if n == 0
     case 1: statement2; // do if n == 1
     case 2: statement3; // do if n == 2
     default: statement4; //if for any other n value
}
```

Any “statement” above can be replaced with a set of statements: {s1; s2; s3; …}

WHILE loop structure

- Repeat a set of statements (a “loop”) as long as some condition is met

```
while (a < b)
  {
    statement s1;
    statement s2;
    ….
  }
```



Something must eventually cause a >= b, to exit the loop

WHILE examples

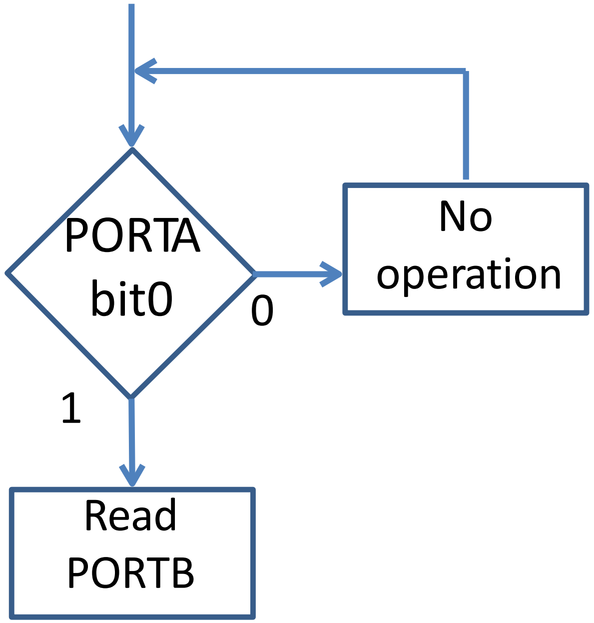
/* Add two 200-element arrays. */

```
int M[200],N[200],P[200];
int k;

/* Method 1 – using DO-WHILE */
k = 0;                          //initialize counter/index
do {
   M[k] = N[k] + P[k];          //add k-th array elements
   k = k + 1;                   //increment counter/index
} while (k < 200);              //repeat if k less than 200

/* Method 2 – using WHILE loop
k = 0;                          //initialize counter/index
while (k < 200} {               //execute the loop if k less than 200
   M[k] = N[k] + P[k];          //add k-th array elements
   k = k + 1;                   //increment counter/index
}
```

WHILE example



Wait for a 1 to be applied
to bit 0 of GPIOA
and then read GPIOB

```
while ( (GPIOA->IDR & 0x0001) == 0)   // test bit 0 of GPIOA
    {}                                 // do nothing & repeat if bit is 0
c = GPIOB->IDR;                        // read GPIOB after above bit = 1
```

FOR loop structure

- Repeat a set of statements (one "loop") while some condition is met
  - often a given # of iterations

Initialization(s)

Condition for execution

Operation(s) at end of each loop

```
for (m = 0; m < 200; m++)
  {
    statement s1;
    statement s2;
  }
```

FOR loop structure

- FOR loop is a more compact form of the WHILE loop structure

```
/* execute loop 200 times */
for (m = 0; m < 200; m++)
   {
     statement s1;
     statement s2;
   }
```

```
/* equivalent WHILE loop */
m = 0;   //initial action(s)
while (m < 200)  //condition test
   {
     statement s1;
     statement s2;
     m = m + 1;  //end of loop action
   }
```

FOR structure example

```
/* Read 100 16-bit values from GPIOB into array C */
/* Bit 0 of GPIOA (PA0) is 1 if data is ready, and 0 otherwise */
uint16_t  c[100];
uint16_t  k;

for (k = 0; k < 200; k++) {
      while ((GPIOA->IDR & 0x01) == 0)   //repeat until PA0 = 1
          {}                              //do nothing if PA0 = 0
      c[k] = GPIOB->IDR;                  //read data from PB[15:0]
}
```

FOR structure example

```
/* Nested FOR loops to create a time delay */

for (i = 0; i < 100; i++) {            //do outer loop 100 times
    for (j = 0; j < 1000; j++) {       //do inner loop 1000 times
    }                                  //do "nothing" in inner loop
}
```

C functions

- Functions partition large programs into a set of smaller tasks
  - Helps manage program complexity
  - Smaller tasks are easier to design and debug
  - Functions can often be reused instead of starting over
  - Can use of “libraries” of functions developed by 3rd parties, instead of designing your own

C functions

- A function is “called” by another program to perform a task
  - The *function* may return a result to the caller
  - One or more arguments may be passed to the function/procedure

Function definition

Type of value to be returned to the caller*

Parameters passed by the caller

```
int math_func (int k; int n)
{
  int j;                //local variable
  j = n + k - 5;        //function body
  return(j);            //return the result
}
```

* If no return value, specify “void”

Function arguments

- Calling program can pass information to a function in two ways
  - By value: pass a constant or a variable value
    - function can use, but not modify the value
  - By reference: pass the address of the variable
    - function can both read and update the variable
  - Values/addresses are typically passed to the function by pushing them onto the system stack
    - Function retrieves the information from the stack

Example – pass by value

```
/* Function to calculate x^2 */
int square ( int x ) {   //passed value is type int, return an int value
   int y;                //local variable – scope limited to square
   y = x * x;            //use the passed value
   return(x);            //return the result
}

void main {
  int k,n;               //local variables – scope limited to main
  n = 5;
  k = square(n);         //pass value of n, assign n-squared to k
  n = square(5);         // pass value 5, assign 5-squared to n
}
```

Example – pass by reference

```
/* Function to calculate x² */
void square ( int x, int *y ) {   //value of x, address of y
  *y = x * x;                     //write result to location whose address is y
}

void main {
  int k,n;            //local variables – scope limited to main
  n = 5;
  square(n, &k);      //calculate n-squared and put result in k
  square(5, &n);      // calculate 5-squared and put result in n
}
```

In the above, *main* tells *square* the location of its local variable, so that *square* can write the result to that variable.

Example – receive serial data bytes

```
/* Put string of received SCI bytes into an array */
Int rcv_data[10];            //global variable array for received data
Int rcv_count;               //global variable for #received bytes

void SCI_receive ( ) {
  while ( (SCISR1 & 0x20) == 0) {}   //wait for new data (RDRF = 1)
  rcv_data[rcv_count] = SCIDRL;      //byte to array from SCI data reg.
  rcv_count++;                       //update index for next byte
}
```

Other functions can access the received data from the global variable array rcv_data[].

Some on-line C tutorials

- http://www.cprogramming.com/tutorial/c-tutorial.html
- http://www.physics.drexel.edu/courses/Comp_Phys/General/C_basics/
- http://www.iu.hio.no/~mark/CTutorial/CTutorial.html
- http://www2.its.strath.ac.uk/courses/c/