

C programming for embedded microcontroller systems

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 "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
}

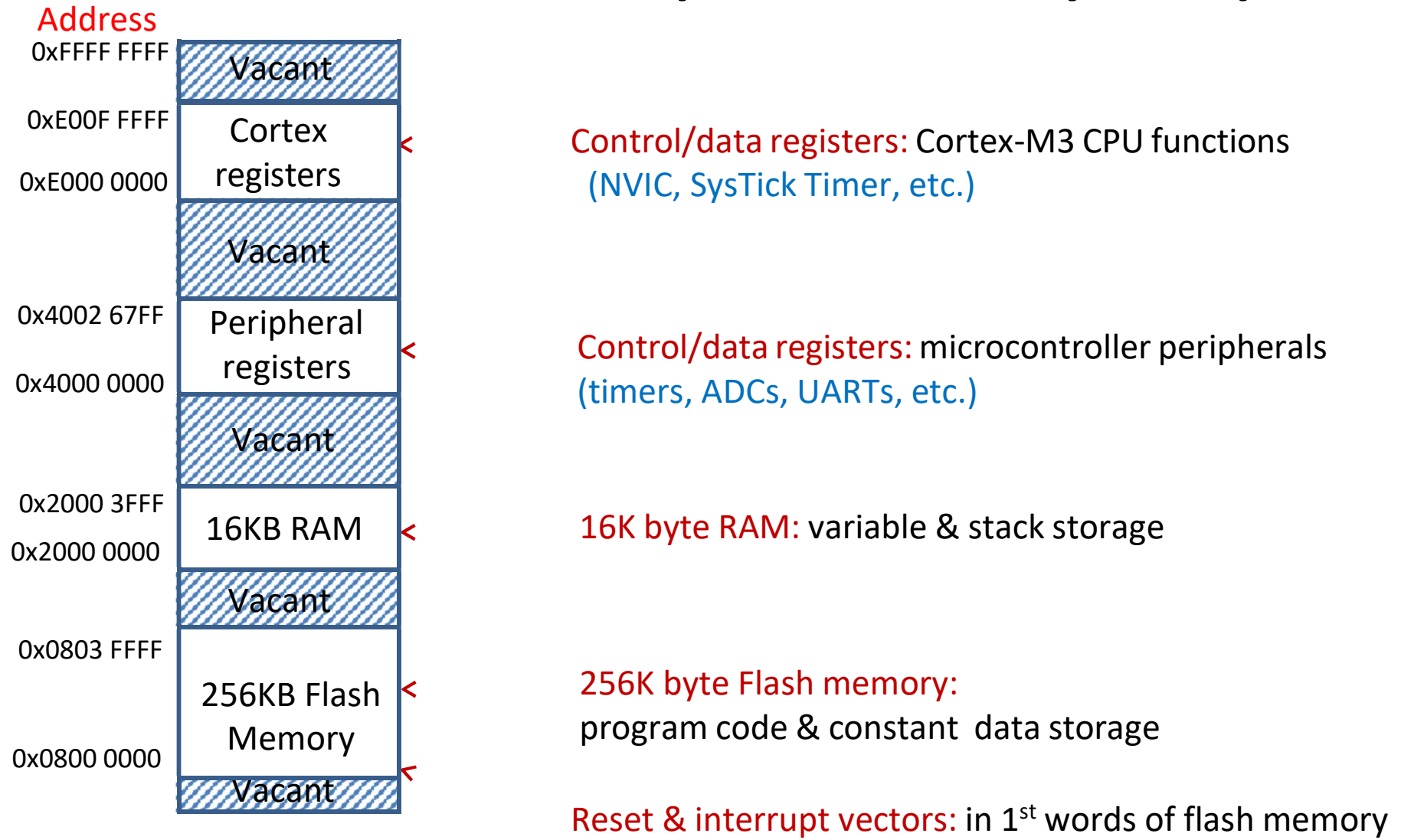
/* Main program */
void main(void) {
    unsigned char sw1; /*local (automatic) variable (stack or registers)
    int k; /*local (automatic) variable (stack or registers)
    /* Initialization section */
    -- instructions to initialize variables, I/O ports, devices, function registers
    /* Endless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

The diagram illustrates the structure of a C program with three main sections, each enclosed in a red dashed box:

- Global variables:** This section includes the header file `#include "STM32L1xx.h"` and global variables `int count, bob;`. A comment indicates that global (static) variables are placed in RAM.
- Function definitions:** This section contains a function definition `int function1(char x) { ... }`. Comments explain that the parameter `x` is passed to the function, which returns an integer value, and that local (automatic) variables are allocated to the stack or registers.
- Main program:** This section starts with `void main(void) {` and contains three sub-sections:
 - Declare local variables:** This part declares local (automatic) variables `unsigned char sw1;` and `int k;`, which are allocated to the stack or registers.
 - Initialize variables/devices:** This part contains the initialization section, where instructions to initialize variables, I/O ports, devices, and function registers are provided.
 - Body of the program:** This part contains the endless loop, which is a `while (1) { ... }` loop. Comments indicate that the loop can also be implemented using `for(;;) { ... }` and that the instructions inside the loop are to be repeated.

}

STM32L100RC μ C memory map



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 "STM32L1xx.h"           /* target uC information */

// GPIOA configuration/data register addresses are defined in STM32L1xx.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

C compiler data types

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

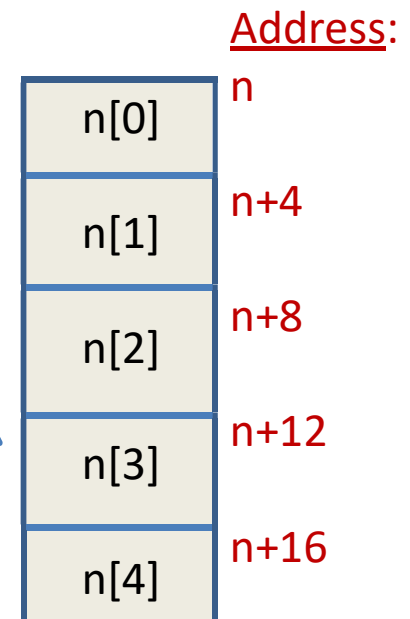
- A *variable* is an addressable storage location to information to be used by the program
 - Each variable must be *declared* to indicate size and type of information to be stored, plus name to be used to reference the information
 - int x,y,z; //declares 3 variables of type “int”*
 - char a,b; //declares 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.



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
    }
}
```

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) \wedge (XOR) \sim (Complement)

$C = A \& B;$
(AND)

A	0	1	1	0	0	1	1	0
B	1	0	1	1	0	0	1	1
C	0	0	1	0	0	0	1	0

$C = A | B;$
(OR)

A	0	1	1	0	0	1	0	0
B	0	0	0	1	0	0	0	0
C	0	1	1	1	0	1	0	0

$C = A \wedge B;$
(XOR)

A	0	1	1	0	0	1	0	0
B	1	0	1	1	0	0	1	1
C	1	1	0	1	0	1	1	1

$B = \sim A;$
(COMPLEMENT)

A	0	1	1	0	0	1	0	0
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
C	a	b	c	d	e	f	g	0

Clear selected bit of A

$C = A \ \& \ 0x01;$

A	a	b	c	d	e	f	g	h
0xFE	0	0	0	0	0	0	0	1
C	0	0	0	0	0	0	0	h

Clear all but the selected bit of A

$C = A \ | \ 0x01;$

A	a	b	c	d	e	f	g	h
0x01	0	0	0	0	0	0	0	1
C	a	b	c	d	e	f	g	1

Set selected bit of A

$C = A \ \wedge \ 0x01;$

A	a	b	c	d	e	f	g	h
0x01	0	0	0	0	0	0	0	1
C	a	b	c	d	e	f	g	h'

Complement selected bit of A

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

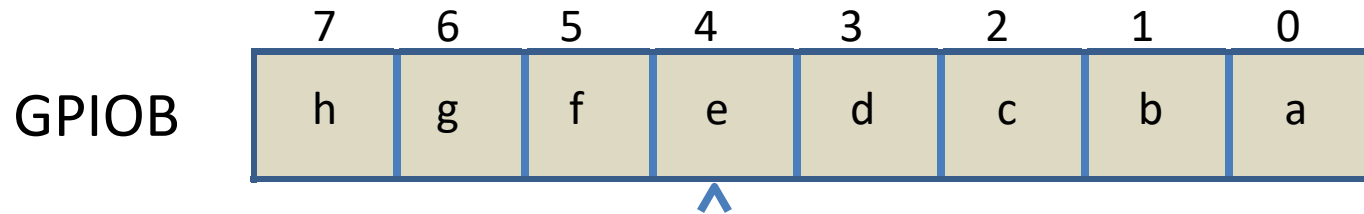
if ((PORTA & 0x01) == 0) // test bit 0 of PORTA

PORTA = c | 0x01; // write c to PORTA with bit 0 set to 1

Example of μ C register address definitions in *STM32Lxx.h* (*read this header file to view other peripheral functions*)

```
#define PERIPH_BASE      ((uint32_t)0x40000000)    //Peripheral base address in memory
#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
#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    */
    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 low registerBSRR, Address offset: 0x18    */
    __IO uint16_t BSRRH;    /*!< GPIO port bit set/reset high registerBSRR, Address offset: 0x1A    */
    __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)



Switch connected to bit 4 (PB4) of GPIOB

```
uint16_t sw; //16-bit unsigned type since GPIOB IDR and ODR = 16 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:

$x \gg y$ (right shift operand x by y bit positions)

$x \ll y$ (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

$B = A \ll 3;$
(Left shift 3 bits)

A	<u>1</u>	<u>0</u>	<u>1</u>	<u>0</u>	<u>1</u>	<u>1</u>	<u>0</u>	<u>1</u>
B	0	1	1	0	1	0	0	0

$B = A \gg 2;$
(Right shift 2 bits)

A	<u>1</u>	<u>0</u>	<u>1</u>	<u>1</u>	<u>0</u>	<u>1</u>	<u>0</u>	<u>1</u>
B	0	0	1	0	1	1	0	1

$B = '1';$

$C = '5';$

$D = (B \ll 4) \mid (C \ \& \ 0x0F);$
(B << 4)

$B = 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 0 \ 1$ (ASCII 0x31)

$C = 0 \ 0 \ 1 \ 1 \ 0 \ 1 \ 0 \ 1$ (ASCII 0x35)

$= 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0$

$$\begin{array}{lcl}
 (\text{C} \ \& \ 0\text{x}0\text{F}) & = & \underline{0 \ 0 \ 0 \ 0} \ \underline{0 \ 1 \ 0 \ 1} \\
 \text{D} & = & \underline{0 \ 0 \ 0 \ 1} \ \underline{0 \ 1 \ 0 \ 1} \ (\text{Packed BCD } 0\text{x}15)
 \end{array}$$

C control structures

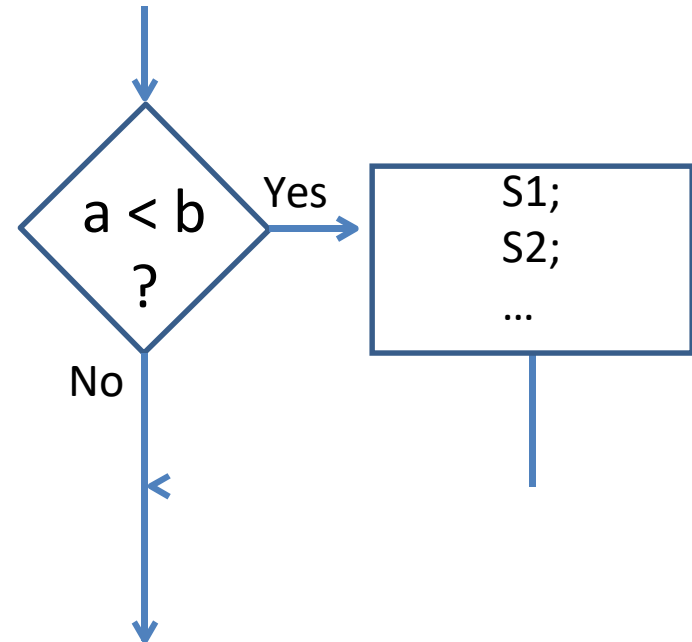
- Control order in which instructions are executed (program flow)
- Conditional execution
 - Execute a set of statements if some condition is met
 - Select one set of statements to be executed from several options, depending on one or more conditions
- Iterative execution
 - Repeated execution of a set 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 relational operator, whereas `=` is an assignment operator.

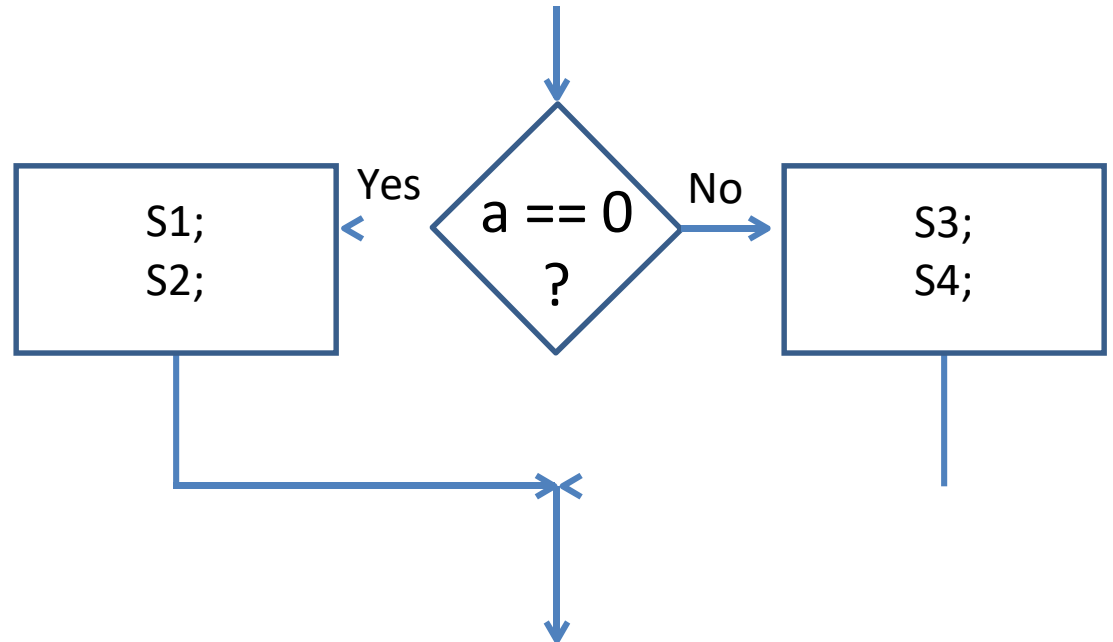
if (m == n) //tests equality of values of variables m and n
if (m = n) //assigns value of n to variable m, and then
//tests whether that value is TRUE (non-zero)

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;
}
```



IF-THEN-ELSE HCS12 assembly language vs C example

```
AD_PORT:      EQU  $91 ; A/D Data Port
MAX_TEMP:     EQU  128 ; Maximum temperature
VALVE_OFF:    EQU  0   ; Bits for valve off
VALVE_ON:     EQU  1   ; Bits for valve on
VALVE_PORT:   EQU  $258 ; Port P for the valve
```

...

; Get the temperature

```
ldaa AD_PORT
```

; IF Temperature > Allowed Maximum

```
cmpa #MAX_TEMP
```

```
bls ELSE_PART
```

; THEN Turn the water valve off

```
ldaa VALVE_OFF
```

```
staa VALVE_PORT
```

```
bra END_IF
```

; ELSE Turn the water valve on

```
ELSE_PART:
```

```
ldaa VALVE_ON
```

```
staa VALVE_PORT
```

```
END_IF:
```

; END IF temperature > Allowed Maximum

C version:

```
#define MAX_TEMP 128
```

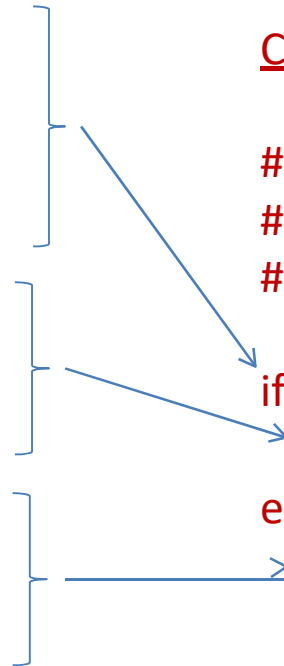
```
#define VALVE_OFF 0
```

```
#define VALVE_ON 1
```

```
if (AD_PORT <= MAX_TEMP)
    VALVE_PORT = VALVE_OFF;
```

```
else
```

```
    VALVE_PORT = VALVE_ON;
```



Ambiguous ELSE association

```
if (n > 0)
    if (a > b)
        z = a;
else
    z = b;
```

//else goes with nearest previous “if” (a > b)

```
if (n > 0) {
    if (a > b)
        z = a;
} else {
    z = b;
}
```

Braces force proper association

//else goes with first “if” (n > 0)

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

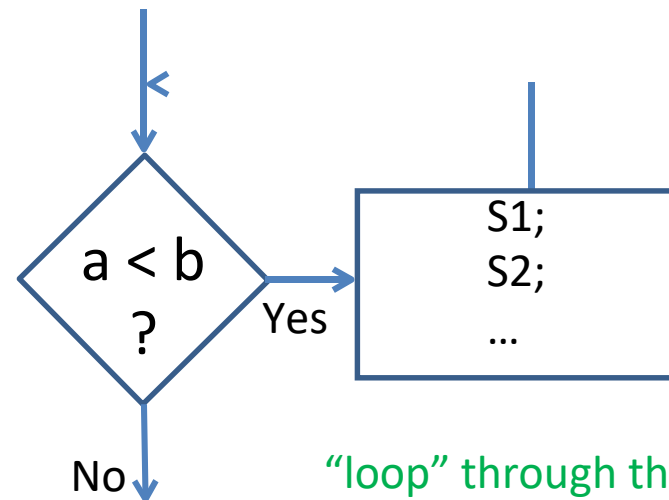
```
/* example equivalent to that on preceding slide */  
switch ( n ) { //n is the variable to be tested  
    case 0: statement1; //do if n == 0  
    case 1: statement2; // do if n == 1  
    case 2: statement3; // do if n == 2  
    default: statement4; //if for any other n value  
}
```

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;  
  
  ....  
}
```



“loop” through these
statements while a < b

Something must eventually cause $a \geq b$, to exit the loop

WHILE loop example: C vs. HCS12 Assembly Language

C version:

```
#define MAX_ALLOWED 128
#define LIGHT_ON 1
#define LIGHT_OFF 0

while (AD_PORT <= MAX_ALLOWED)
{
    LIGHT_PORT = LIGHT_ON;
    delay();
    LIGHT_PORT = LIGHT_OFF;
    delay();
}
```

```
AD_PORT: EQU $91 ; A/D Data port
MAX_ALLOWED: EQU 128 ; Maximum Temp
LIGHT_ON: EQU 1
LIGHT_OFF: EQU 0
LIGHT_PORT: EQU $258 ; Port P
; ---
; Get the temperature from the A/D
    ldaa AD_PORT
; WHILE the temperature > maximum allowed
WHILE_START:
    cmpa MAX_ALLOWED
    bls END_WHILE
; DO - Flash light 0.5 sec on, 0.5 sec off
    ldaa LIGHT_ON
    staa LIGHT_PORT ; Turn the light
    jsr delay ; 0.5 sec delay
    ldaa LIGHT_OFF
    staa LIGHT_PORT ; Turn the light off
    jsr delay
; End flashing the light, Get temperature from the A/D
    ldaa AD_PORT
; END_DO
    bra WHILE_START
END_WHILE:
```

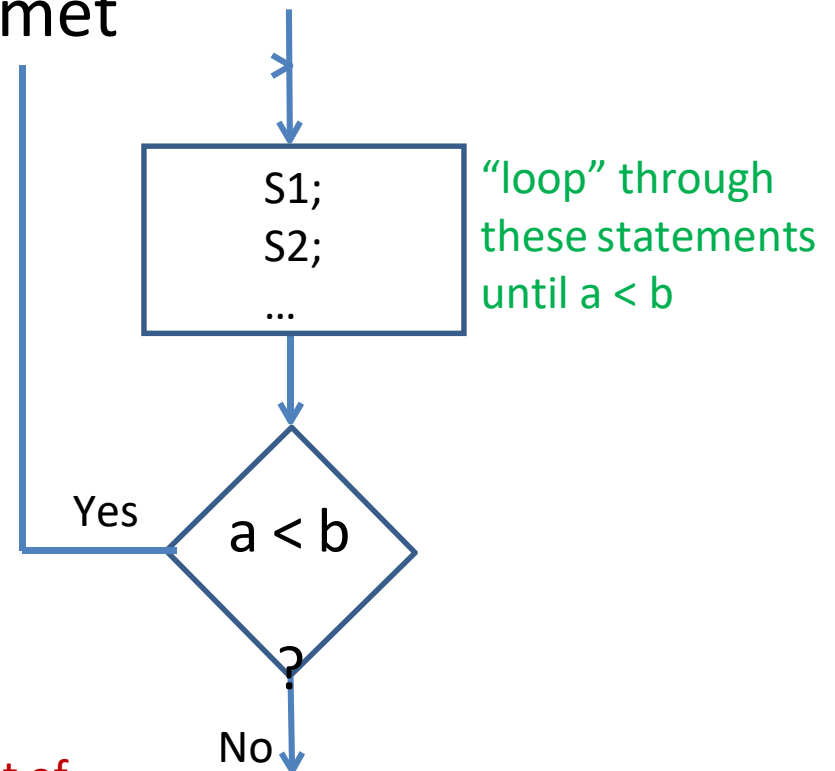
DO-WHILE loop structure

- Repeat a set of statements (one “loop”)
until some condition is met

```
do  
{  
  statement s1;  
  statement s2;  
  ....  
}
```

```
while (a < b);
```

The condition is tested after executing the set of statements, so the statements are guaranteed to execute at least once.



DO-WHILE example

C version:

```
#define MAX_ALLOWED 128
#define LIGHT_ON 1
#define LIGHT_OFF 0

do {
    LIGHT_PORT = LIGHT_ON;
    delay();
    LIGHT_PORT = LIGHT_OFF;
    delay();
} while (AD_PORT <= MAX_ALLOWED);
```

```
;HCS12 Assembly Language Version
; DO
; Flash light 0.5 sec on, 0.5 sec off
    ldaa    LIGHT_ON
    staa    LIGHT_PORT    ; Turn light on
    jsr     delay         ; 0.5 sec delay
    ldaa    LIGHT_OFF
    staa    LIGHT_PORT    ; Turn light off
    jsr     delay
; End flashing the light
; Get the temperature from the A/D
    ldaa    AD_PORT
; END_DO
    bra     WHILE_START
; END_WHILE:
; END_WHILE temperature > maximum allowed
; Dummy subroutine
delay:    rts
```

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
```

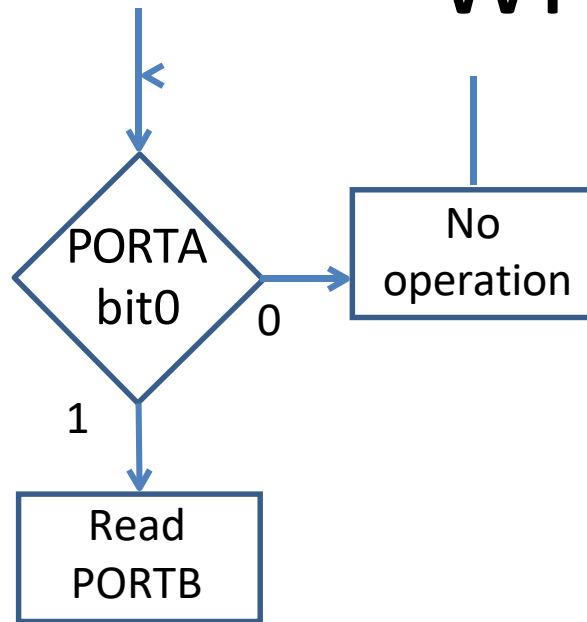
```
while (k < 200) {                       //execute the loop if k less than 200
```

```
    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

<pre><i>/* execute loop 200 times */</i> <i>for (m = 0; m < 200; m++)</i> { <i>statement s1;</i> <i>statement s2;</i> }</pre>	<pre><i>/* equivalent WHILE loop */</i> <i>m = 0; //initial action(s)</i> <i>while (m < 200) //condition test</i> { <i>statement s1;</i> <i>statement s2;</i> <i>m = m + 1; //end of loop action</i> }</pre>
--	---

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”

Parameters passed to

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 x2 */  
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/>