C for Embedded Systems

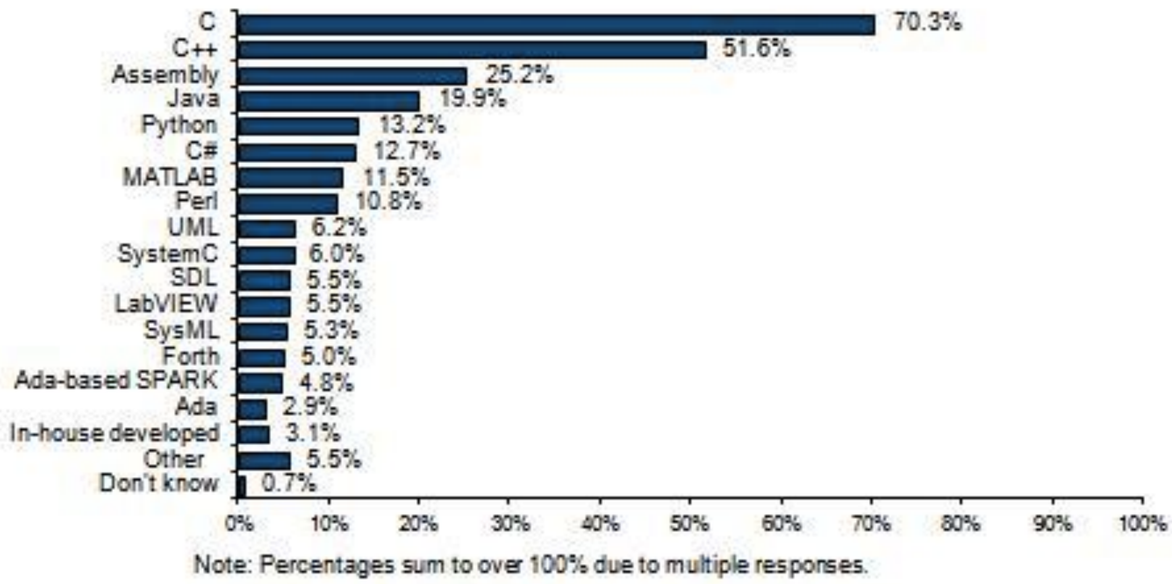
Baris Aksanli
Department of Electrical and Computer Engineering
San Diego State University

Outline

- C requirement in embedded environments
- C data types
- C variables, arrays
- Arithmetic/logic/shift operations
- C functions
- Direct memory access

Embedded Systems Programming

- Points of evaluation
  - Concurrency
  - Ability to specify thread execution times
  - Ability to control shared resources, queues etc.
  - Overhead
- Assembly
- C
- Ada
- Java



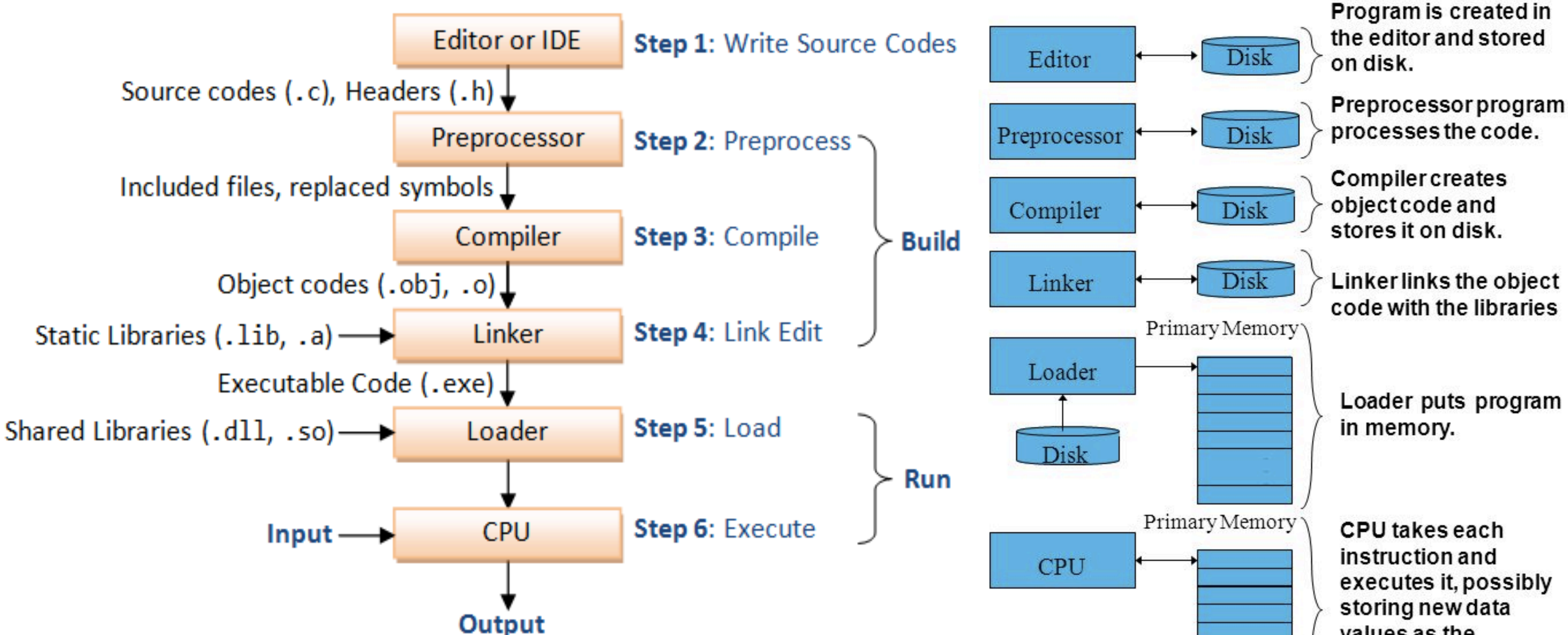
C vs. Assembler in Embedded Systems

- Convention: There already exists code written in C
- Assembly is hard to read and maintain
  - C programs can be clearer, easier to read
  - C has standardized syntax
- High-level languages (like C, compared to assembly) are more cost-effective
- C is more portable than assembly
- C allows both high-level and low-level programming

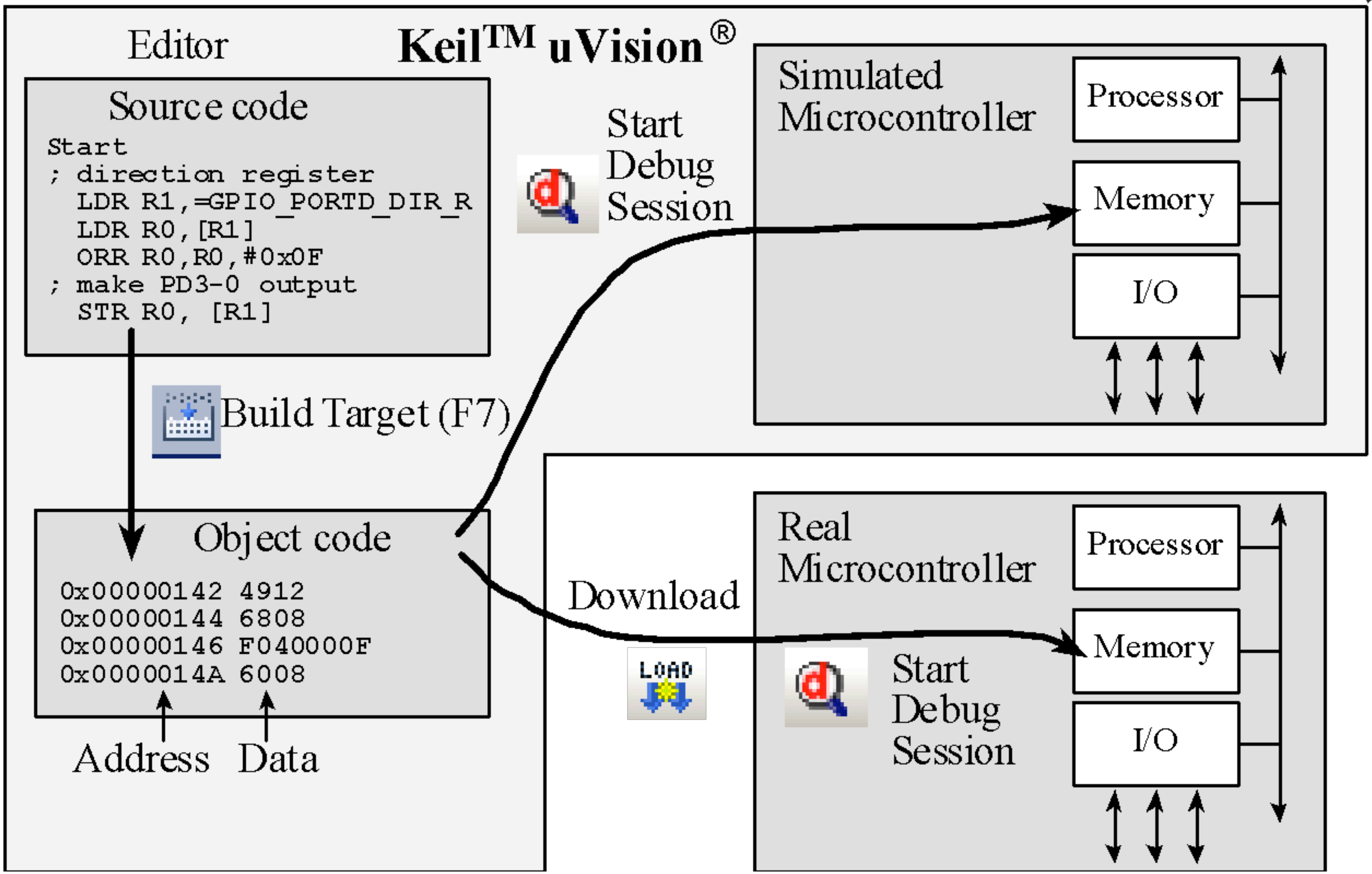
C Background

- First appeared in 1970s – to write an OS (Unix)
- Quickly became language of choice for pro "systems" programmers
- December 1989 ANSI standard formally defined
- To improve efficiency, ANSI extensions added
- Improved versions later: C++, C#, etc.
  - But we will not cover these in this class

C Development Process



C Development on Microcontrollers



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 Data Types

- Important: Always match data type and data characteristics!!
- Variable type determines how data is represented
  - #bits: range of numeric values
  - signed/unsigned: which arithmetic/relational operators are to be used by the compiler

C 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 Notes

- Range, resolution, accuracy
  - Many operations in embedded applications have very specific limits to the range and resolution
  - Use only the range and precision you really need
- Speed, code size
  - For 8 bit processors like AVR, use 8 bit variables
  - For 32 bit processors like ARM, use 32 bit variables
- For example, AVR is an 8-bit processor
  - To maximize speed, use 8 bit variables
  - Integers are the fastest
  - Floating point is slow, has a huge library
    - Some compilers don't support floating point
  - Fixed point arithmetic is much faster than floating point, uses integers. Requires more effort on the part of the programmer.

Data Type Notes

- Truncation
  - Be careful with variable truncation
  - Example: Add two 8 bit numbers & assign to a 16 bit value
  - This may result is truncation after addition then typecasting
  - Cast variables before addition
- Casting
  - Cast variables if there is any question about type of result
  - Example: z = (unsigned int)a+b; // if a, b are chars
- Avoid using floats and doubles when possible
  - They work slowly and take up a lot of space if implemented in software

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"

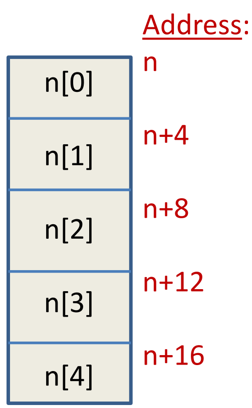
  char a,b; //declares 2 variables of type "char"
- Space for variables may be allocated in registers, RAM, or ROM/Flash (for constants)
- Variables can be *automatic* or *static*

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.

```
char   string1[20];
char   string2[] = "Enter a key when ready:";
int    result[10][10];
char   lookup_Table[] = {31, 35, 38, 0x20};
unsigned char day[][]= {"Mon", "Tue", "Wed"};
```

Automatic Variables

- Declared 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

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.

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 sample_array[10];

Example

```c
#include <stdio.h>

void foo()
{
    int a = 10;
    static int sa = 10;

    a += 5;
    sa += 5;

    printf("a = %d, sa = %d\n", a, sa);
}


int main()
{
    int i;

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

OUTPUT:

```
a = 15, sa = 15
a = 15, sa = 20
a = 15, sa = 25
a = 15, sa = 30
a = 15, sa = 35
a = 15, sa = 40
a = 15, sa = 45
a = 15, sa = 50
a = 15, sa = 55
a = 15, sa = 60
```

Example 2

```
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
  }
}
```

What happens to the value of the variable j?

Other Variable Definitions

- **const** can be applied to the declaration of any variable to specify that its value will not be changed

- The **volatile** keyword can be used to state that a variable may be changed by hardware, the kernel, another thread etc.
  - For example, the volatile keyword may prevent unsafe compiler optimizations for memory-mapped input/output
    - Memory-mapped peripheral registers
    - Global variables modified by an interrupt service routine
    - Global variables accessed by multiple tasks within a multi-threaded application

Examples: const (with pointers)

2) Pointer to constant.

Pointer to constant can be declared in following two ways.

```
const in
```

or

```
int cons
```

We can ch
using point
or read wri

```
#include
int main
{
    int
    int
    cons

    prin
    *ptr

    ptr
    prin

    return 0;
}
```

3) Constant point

```
int *const ptr
```

Above declaration
but cannot change

```
#include <stdi
int main(void)
{
    int i = 10;
    int j = 20;
    int *const

    printf("ptr

    *ptr = 100;
    printf("ptr

    ptr = &j;
    return 0;
}
```

4) constant pointer to constant

```
const int *const ptr;
```

Run on IDE

Above declaration is constant pointer to constant variable which means we cannot change value pointed by pointer as well as we cannot point the pointer to other variable. Let us see with example.

```
#include <stdio.h>

int main(void)
{
    int i = 10;
    int j = 20;
    const int *const ptr = &i;        /* constant pointer to constant integer */

    printf("ptr: %d\n", *ptr);

    ptr = &j;           /* error */
    *ptr = 100;        /* error */

    return 0;
}
```

Examples: const (with pointers)

```c
int main(void) {
  int i = 42;
  int j = 28;

  const int *pc = &i;        //Also: "int const *pc"
  *pc = 41;                  //Wrong
  pc = &j;

  int *const cp = &i;
  *cp = 41;
  cp = &j;                   //Wrong

  const int *const cpc = &i;
  *cpc = 41;                 //Wrong
  cpc = &j;                  //Wrong
  return 0;
}
```

- `const int *p` is a pointer to a `const` int
- `int const *p` is also a pointer to a `const` int
- `int *const p` is a `const` pointer to an int
- `const int *const p` is a `const` pointer to a `const` int

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

Bit-parallel Logic Operations

- 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
```

Source: V.P. Nelson

Bit Set/Reset/Complement/Test

```
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
               0x01    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 ^ 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

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 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

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”

- Parameter passing
- **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

Pass-by-value Example

```
/* 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 (y) ;          //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
}
```

Pass-by-reference Example

```
/* 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.

Some Embedded Extensions

- Compiler specific extensions:
  `return_type func_name( parameters ) [{mem_model}]`
  `reentrant interrupt using…`

- Where:
  - return_type is the SINGLE value returned from the function
  - func_name is the name of the function
  - parameters are the arguments passed to the function
  - mem_model is small, compact, or large
  - reentrant indicates that function is recursive and reentrant
  - interrupt-n indicates the function is an ISR
  - using specifies the register bank used by the function arguments

Direct Register Access - 1

```
// GPIO Port A is located at 0x1000

uint32_t * Gpio_PortA = (uint32_t *) 0x1000U;

// Set the 0 bit high on PortA

*Gpio_PortA |= 0x01;           // Valid
Gpio_PortA++;                  // Invalid!

/* But this is not the safest way to do this
(for example the invalid line does not produce a
compile error) due to the fact that the contents
of this register might be changed by other
functions as well */

// See version 2 -> next slide
```

Direct Register Access - 2

```
uint32_t volatile * const Gpio_PortA = (uint32_t
*) 0x1000U;

// Set the 0 bit high on PortA

*Gpio_PortA |= 0x01;            // Valid
Gpio_PortA++;                   // Invalid!

/* This time the invalid line gives a compile
error. This is because the pointer (register
address) is defined constant, thus it cannot be
changed. */

/* Furthermore, the contents of this pointer is
defined as volatile, telling the compiler not to
assume anything about it. */
```

Some 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/
- http://www.geeksforgeeks.org/const-qualifier-in-c/