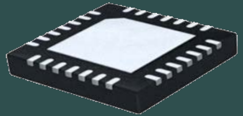
Basics of Embedded Programming

C

Course Flow

- Write a Program in Embedded C
- Bitwise operations in Embedded C
- Embedded C program Structure
- Control structures | Functions
- Variable | Constants | Number system | Data Types

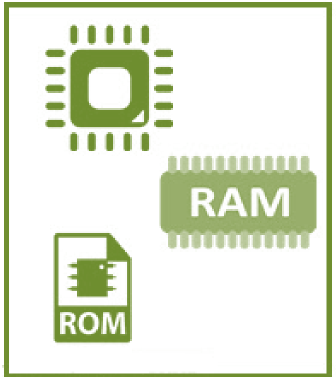
- What is an Embedded System
- Selection of programming language for embedded system
- Embedded C Compiler working



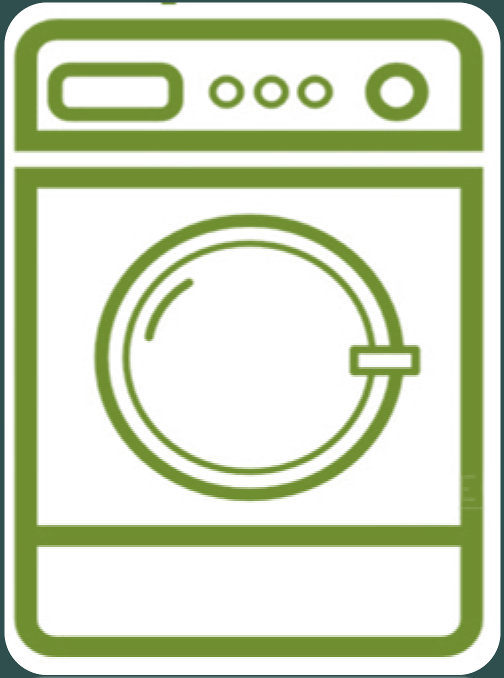
What is an Embedded System?

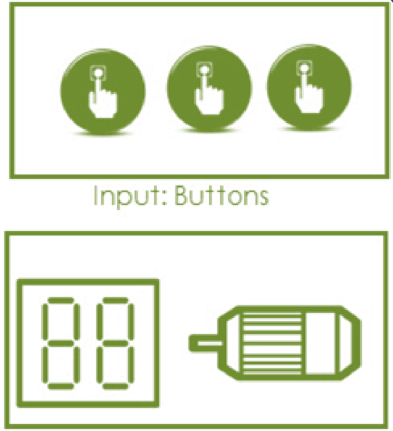
An Embedded System can be best described as a system which has both the hardware and software and is designed to do a specific task.

Example: Washing Machine



Control Unit:
Processor, RAM, ROM
with Software

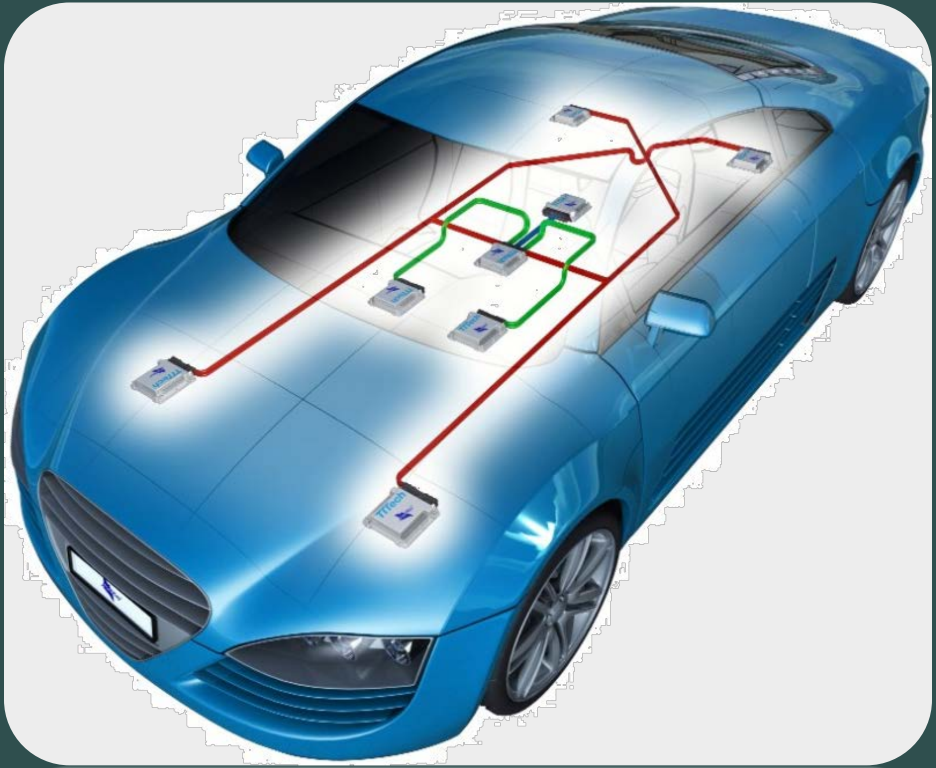




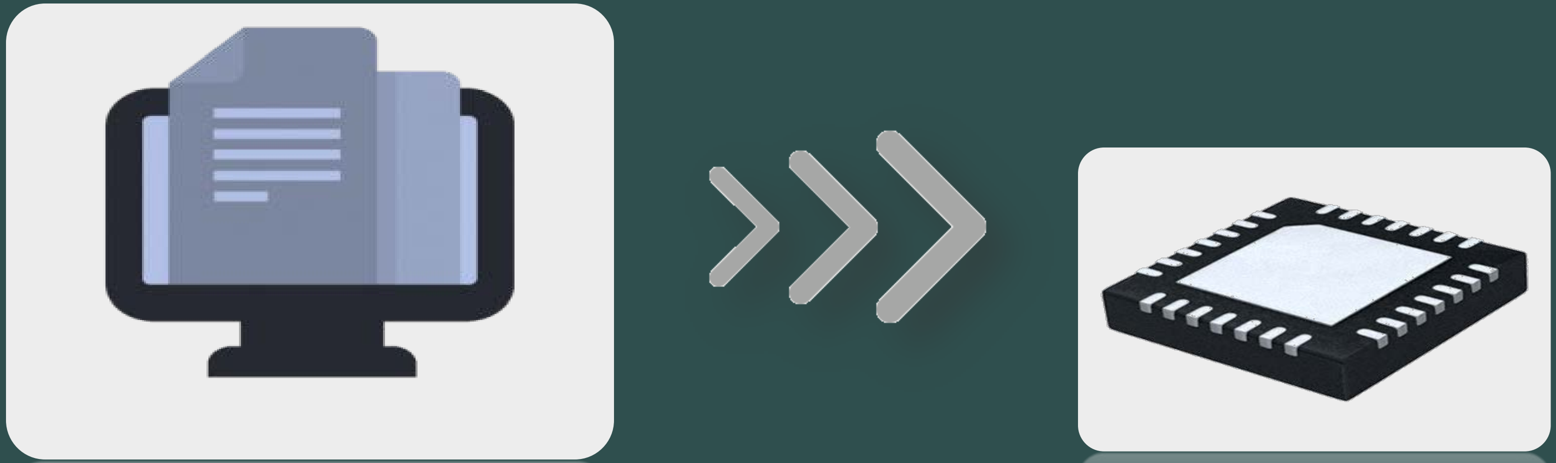
Input: Buttons

Output: Display, Motor

Example: Smart Car



Programming Embedded Systems

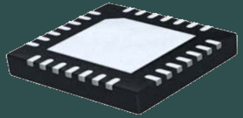


Embedded systems programming

- Embedded devices have resource constraints

- embedded systems typically uses smaller, less power consuming components.

- Embedded systems are more tied to the hardware.



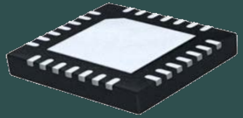
Factors for Selecting the Programming Language

Size: The memory that the program occupies is very important as Embedded Processors like Microcontrollers have a very limited amount of ROM.

Speed: The programs must be very fast i.e. they must run as fast as possible. The hardware should not be slowed down due to a slow running software.

Portability: The same program can be compiled for different processors.

Ease of Implementation
Ease of Maintenance
Readability



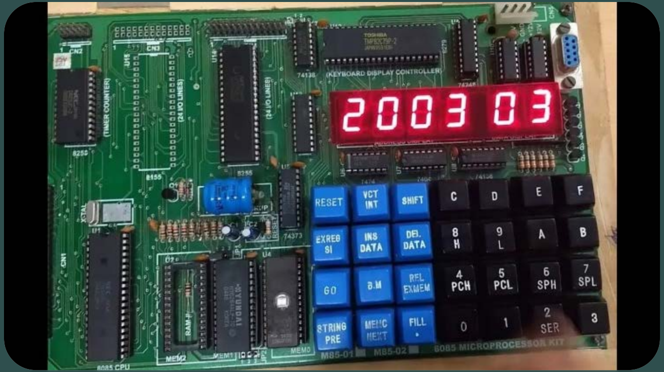
Language use for Embedded systems programming

- Machine Code
- Low level language, i.e., assembly
- High level language like C, C++, Java, Ada, etc.
- Application level language like Visual Basic, scripts, Access, etc.

Assembly Language Programming

```
HERE:   MOV   R0,#01H
        MOV   R1,#02H
        MOV   A,R0
        ADD   A,R1
        MOV   P0,A
        SJMP  HERE
```

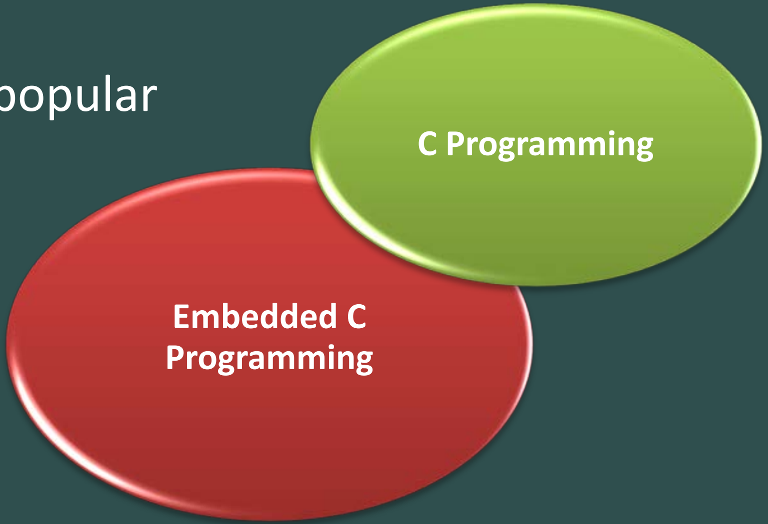
Address	Opcode	Operand
0000	78	01
0002	79	02
0004	E8	
0005	29	
0006	F5	80
0008	80	00

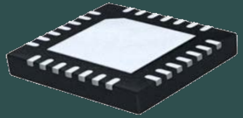


C Programming & Embedded C Programming

- The C Programming Language is the most popular and widely used programming language.

- Embedded C Programming Language is an extension of C Program Language

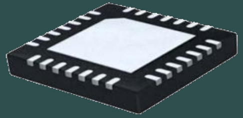




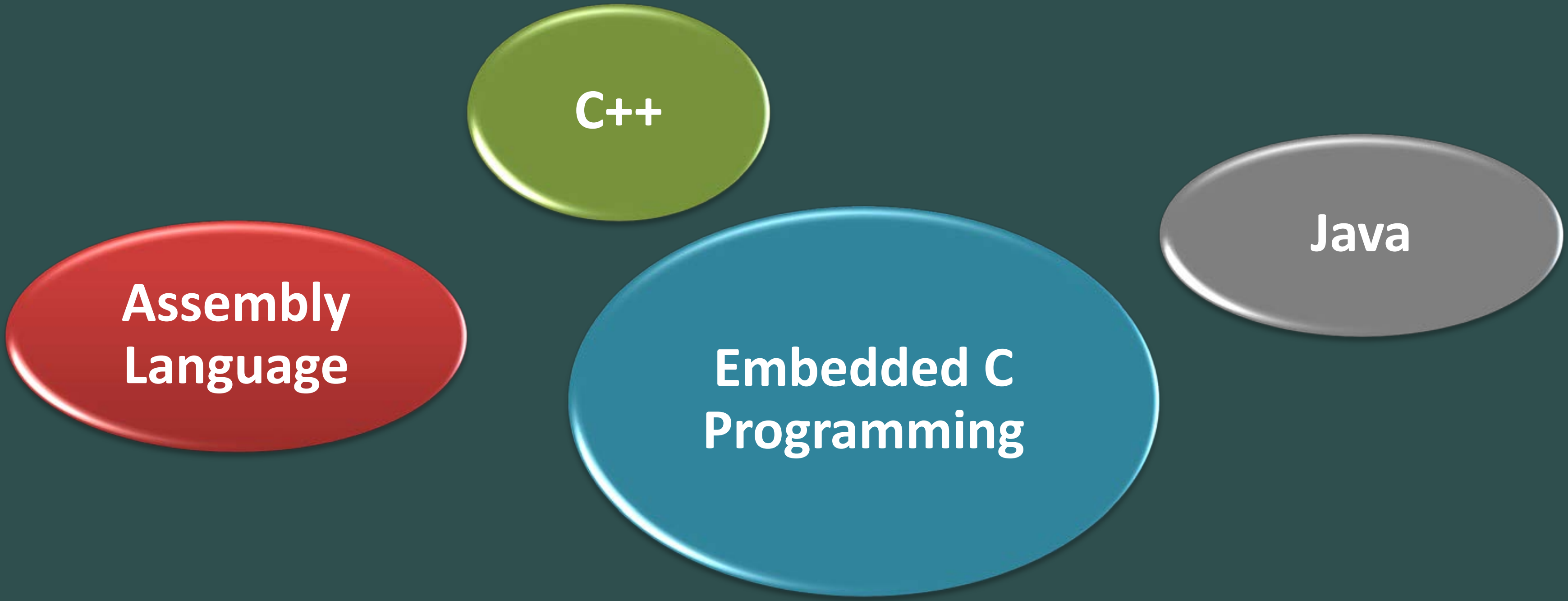
Use of C in embedded systems is driven by following advantages

- it is small and reasonably simpler to learn, understand, program and debug.
- C Compilers are available for almost all embedded devices
- C has advantage of processor-independence
- C combines functionality of assembly language and features of high level languages
- it is fairly efficient

Embedded C Programming



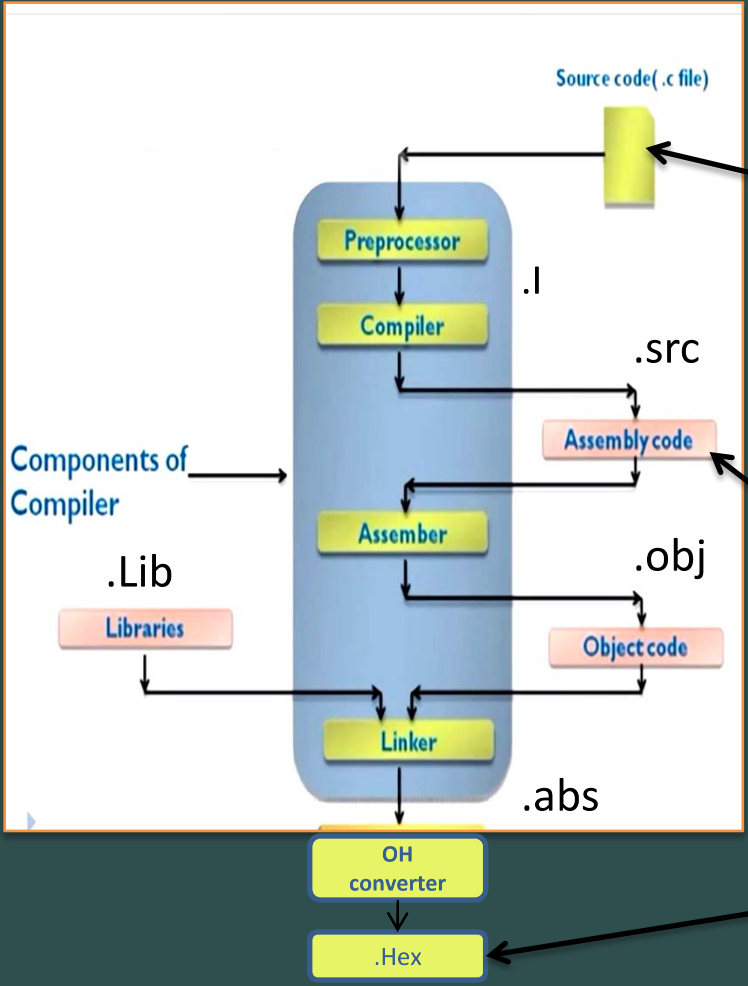
Basics of Embedded C Programming



Difference between C and Embedded C

Though **C and embedded C** appear different and are used in different contexts, they have more similarities than the differences. Most of the constructs are same; the difference lies in their applications.

Embedded C Compiler working




```
#include<reg51.h>
int main()
{
while(1)
{
char R0=1,R1=2,A;
P0=R0+R1;
}
return 0;
}
```

```
HERE:       MOV    R0,#01H
            MOV    R1,#02H
            MOV    A,R0
            ADD    A,R1
            MOV    P0,A
            SJMP   HERE
```

```
:03000000020800F3
:0C080000787FE4F6D8FD75810702000047
:0A00000078017902E829F58080F606
:00000001FF
```

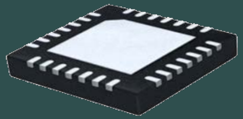
Embedded C	Assembly	Machine code

```c
#include<reg51.h>
int main()
{
while(1)
{
char R0=1,R1=2,A;
P0=R0+R1;
}
return 0;
}
```

```asm
HERE:   MOV    R0,#01H
        MOV    R1,#02H
        MOV    A,R0
        ADD    A,R1
        MOV    P0,A
        SJMP   HERE
```

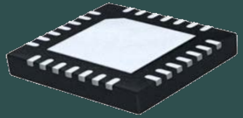
Address	Opcode	Operand
0000	78	01
0002	79	02
0004	E8	
0005	29	
0006	F5	80
0008	80	00

Note: Address of P0 = 80h



Basic Embedded C program structure

```
#include <reg51.h>  /* I/O port/register names/addresses
                       for the 8051xx microcontrollers  */

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

int fun_delay (int x)   /* Function definitions*/
                                    //parameter x passed to the function, function returns an integer value
{

int i;                              //local (automatic) variables – allocated to stack or registers

 for(i=0;i<=x;i++);                 // instructions to implement the function

 }
```

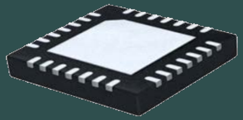


Basics of Embedded C Programming

```
void main(void)    /* Main program */
{

int k;                                  //local (automatic) variable (stack or registers)
P1=0x00;           /* Initialization section */ // instructions to initialize
k = 10;                                 //variables, I/O ports, devices, function registers

while (1)          /* Endless loop */
                                        //Can also use: for(;;)
{                  /* repeat forever */
P1=0x0FF;
Fun_delay( k );                         // function call
P1=0x00;
Fun_delay( k );                         // instructions to be repeated
}

}
```

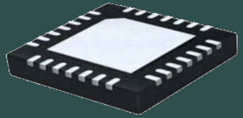
Basic data types in C51 compiler

Data Type	Bits	Bytes	Value Range
bit	1		0 to 1
signed char	8	1	-128 to +127
unsigned char	8	1	0 to 255
enum	16	2	-32768 to +32767
signed short	16	2	-32768 to +32767
unsigned short	16	2	0 to 65535
signed int	16	2	-32768 to +32767
unsigned int	16	2	0 to 65535
signed long	32	4	-2147483648 to 2147483647
unsigned long	32	4	0 to 4294967295
float	32	4	+/-1.175494E-38 to +/-3.402823E+38
sbit	1		0 to 1
sfr	8	1	0 to 255
sfr16	16	2	0 to 65535



Basic data types in ARM C compiler

Type	Size in bits	Natural alignment in bytes	Range of values
char	8	1 (byte-aligned)	0 to 255 (unsigned) by default. –128 to 127 (signed) when compiled with `--signed_chars`.
signed char	8	1 (byte-aligned)	–128 to 127
unsigned char	8	1 (byte-aligned)	0 to 255
(signed) short	16	2 (halfword-aligned)	–32,768 to 32,767
unsigned short	16	2 (halfword-aligned)	0 to 65,535
(signed) int	32	4 (word-aligned)	–2,147,483,648 to 2,147,483,647
unsigned int	32	4 (word-aligned)	0 to 4,294,967,295
(signed) long	32	4 (word-aligned)	–2,147,483,648 to 2,147,483,647
unsigned long	32	4 (word-aligned)	0 to 4,294,967,295
(signed) long long	64	8 (doubleword-aligned)	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
unsigned long long	64	8 (doubleword-aligned)	0 to 18,446,744,073,709,551,615



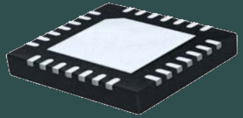
stdint.h header

uint8_t, int8_t: unsigned, signed 8-bit integer

uint16_t, int16_t: unsigned, signed 16-bit integer

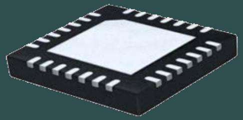
uint32_t, int32_t: unsigned, signed 32-bit integer

uint64_t, int64_t: unsigned, signed 64-bit integer



Bitwise Operators in C

Operator	Description
&	Binary AND Operator copies a bit to the result if it exists in both operands.
|	Binary OR Operator copies a bit if it exists in either operand.
^	Binary XOR Operator copies the bit if it is set in one operand but not both.
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.



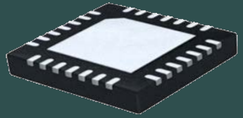
Constant/literal

Constants in C programming language, as the name suggests are the data that doesn't change. Constants are also known as literals.

Integer constants

```
123     /* decimal constant*/
0x9b    /* hexadecimal constant*/
O456    /* octal constant*/
```

For decimal literals :	no prefix is used.
Prefix used for hexadecimal:	0x / 0X
Prefix used for octal:	0



Character constants

Character constants hold a single character enclosed in single quotations marks

m = ‘a’;   //ASCII value 0x61
m= 0x01

String Constants/Literals

String constants consist of any number of consecutive characters in enclosed quotation marks (").

String(array) of characters:

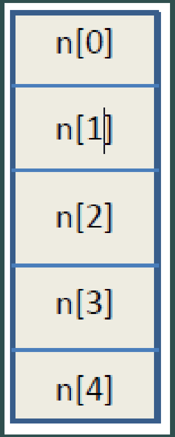
```
char my_string[] = "My String";

// Compiler will interpret the above statement as

char my_string[10] = {'M', 'y', ' ', 'S', 't', 'r', 'i', 'n', 'g', '\0'};
```

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 4^{th} array element



Program variables

```
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

Variables can be automatic or static

Automatic variables

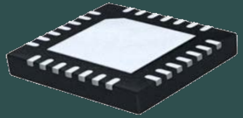
Declare within a function/procedure

Variable is visible (has scope) only within that function

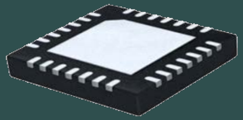
Space for the variable is allocated on the system stack when the procedure is entered
De-allocated, to be re-used, when the procedure is exited

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




```
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
   }
 }
```



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

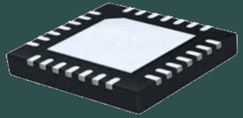
If declared outside a function, the variable is global in scope, i.e. known to all functions of the program

Use “normal” declarations.

Example: int count;

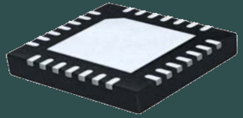
If declared within a function, insert key word static before the variable definition. The variable is local in scope, i.e. known only within this function.

Example: static int count;




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

```
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)
j = 0;                  //initialize static variable j first time math_op() entered
i= count;             //initialize automatic variable ieach time math_op() entered
j = j + i;              //change static variable j –value kept for next function call

}            //return & de-allocate space used by automatic variable i.
```



Arithmetic operations

```
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)

Bit-parallel logical operators

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

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

Number System

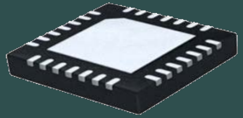
Decimal	Binary	Octal	Hexadecimal
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Numbering System		
System	Base	Digits
Binary	2	0, 1
Octal	8	0,1,2,3,4,5,6,7
Decimal	10	0,1,2,3,4,5,6,7,8,9
Hexadecimal	16	0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

Base	Prefix
Binary:	None
Decimal:	None
Hexadecimal:	**0x** or **0X**
Octal:	**0** (zero)

```
unsigned int n;
n = 0x64;                //HexaDecimal
n = 100;                 //Decimal
n = 0144                 //Octal
```

Decimal:	100	
Binary :	01100100	
Grouping:	001 100 100	0110 0100
	1 4 4	6 4
	Octal	**Hexadecimal**



Bit level Operations in C

1. Bitwise OR operator denoted by ‘**|**‘
2. Bitwise AND operator denoted by ‘**&**‘
3. Bitwise Complement or Negation Operator denoted by ‘**~**‘
4. Bitwise Right Shift & Left Shift denoted by ‘**>>**‘ and ‘**<<**‘ respectively
5. Bitwise XOR operator denoted by ‘**^**‘

AND Truth Table

Inputs		Output
A	B	Y = A.B
0	0	0
0	1	0
1	0	0
1	1	1

```
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
```

```
unsigned char A,B,C;    //we can declare an 8-bit number as a char
A = 0x66;               // binary A =  01100110;
B = 0xB3;               // binary  B =   10110011;
C = A & B;              // binary C =   00100010; i.e 0x22;
```

OR Truth Table

Inputs		Output
A	B	Y = A+B
0	0	0
0	1	1
1	0	1
1	1	1

```
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
```

```
unsigned int A,B,C;
A = 0x64;          //binary A = 01100100
B = 0x10;          //binary B = 00010000
C = A| B;          // C=0x74 which is binary 01110100
```

XOR Truth Table

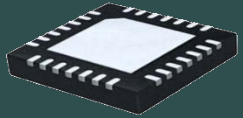
Inputs		Output
A	B	$Y = A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

```
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
```

```
unsigned int A,B,C;
A = 0x64;            //binary A = 01100100
B = 0xB3;            //binary B = 10110011
C = A^B;          // C = 0xD7 which is binary 11010111
```

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

```
unsigned int A,B;
A= 0x64;                //binary A = 0b01100100
B = ~ A;               //  B= 0x9B which is binary 0b10011011
```



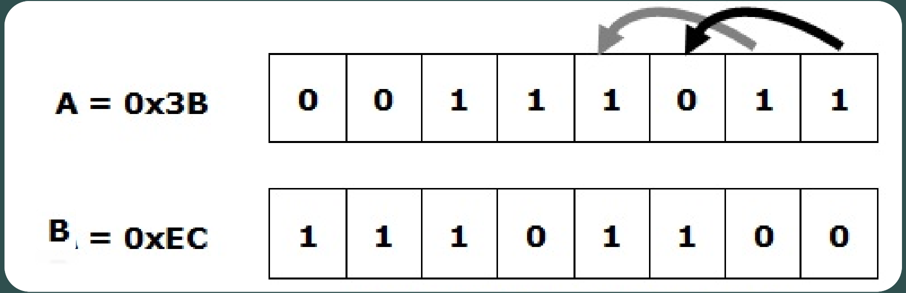
Shift operators

A >> B (right shift operand A by B bit positions)
A << B   (left shift operand A by B bit positions)

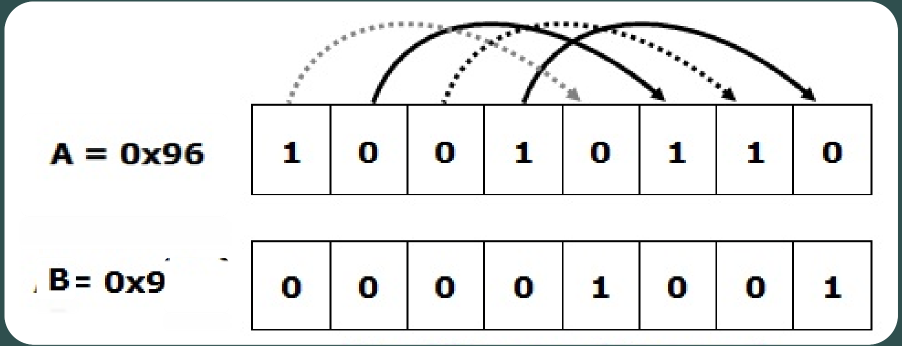
Vacated bits are filled with 0’s.

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

B = A << 2; // left shift A by 2

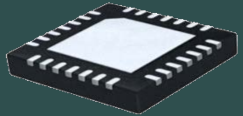


B = A >> 4; // right shift B by 4



Bit masking

When we assign value directly to any register. This may change the value of other bits which might be used to control other hardware feature. To avoid such scenario best practice is to use bit **masking**.



Basics of Embedded C Programming

REGT_8b

1 – Start
0 – Stop

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

REGT_8b Binary: **11100000** Hex: **0xE0**

REGT_8b = 0x04 // Binary: **00000100** LED 2 on

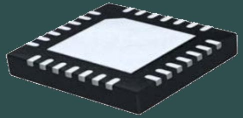
REGT_8b = REGT_8b | (1<<2); // REGT_8b |= (1<<2);

1=00000001

1<<2 = 00000100

REGT_8b | (1<<2) = 11100000 **|** 00000100 = 11100100

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



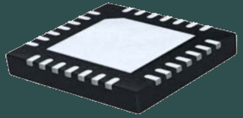
REGT_8b

1 – Start
0 – Stop

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

```
REGT_8b = REGT_8b & (~(1<<6));   //  REGT_8b &= ~(1<<6);
1=00000001
1<<6 = 0100 0000
~(1<<6) = 1011 1111
 REGT_8b & (~(1<<6)) = 11100100  &  10111111   = 10100000
```

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



Embedded C Programming

REGT_8b

1 – Start
0 – Stop

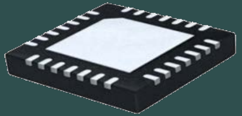
assume current value of REGT_8b as '11100011' which is 8 bit.

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

Stop LED 0 and LED 5 :

REGT_8b &= ~((1<<0) | (1<<5));
((1<<0) | (1<<5)) = ((00000001) | (00100000)) = 00100001
~(00100001) = 11011110
REGT_8b & 11011110 = (11100011) & (11011110) = 11000010

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



REGT_8b

1 – Start
0 – Stop

assume current value of REGT_8b as '11100011' which is 8 bit.

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

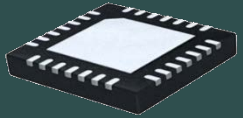
Start LED 3 and 4:

REGT_8b |= ((1<<3) | (1<<4));

((1<<3) | (1<<4)) = **((00001000) | (00010000)) = 00011000;**

REGT_8b | (00011000) is = **(11100011) | (00011000) = 11111011**

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



Embedded C Programming

REGT_8b

1 – Start
0 – Stop

assume current value of REGT_8b as '11100011' which is 8 bit.

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

Stop LED 7 and Start LED3:

REGT_8b = (REGT_8b | (1<<3)) & (~(1<<7));

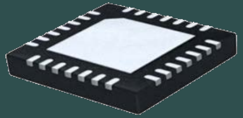
(1<<7) = **10000000** ~(1<<7) = 01111111

(1<<3) = 00001000

(REGT_8b | (1<<3)) = (11100011) | (00001000) = 11101011

(REGT_8b | (1<<3)) & (~(1<<7)) = (11101011) & (01111111) = 01101011

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



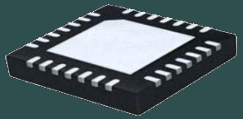
Monitoring Specific bit change in Registers

Many times we need to read certain Flags in a register that denotes change in Hardware state.

Consider a 8 bit Register **P0**

Switch = 1 button pressed
Switch = 0 button released

P0. 7	P0. 6	P0. 5	P0. 4	P0. 3	P0. 2	P0. 1	P0.0
0	0	Switch	0	0	0	0	0



Monitoring Specific bit change in Registers

To monitor for the change in 5th bit from 0 to 1

```
While ( P0  &  (1<<5) )        //wait indefinitely until 5th bit changes from 0 to 1
{                              //do something //exit loop
}
```

1= 00000001

1<<5 = 00100000

P0. 7	P0. 6	P0. 5	P0. 4	P0. 3	P0. 2	P0. 1	P0.0
0	0	Switch 0	0	0	0	0	0

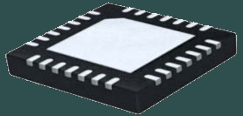
P0= 0x00 = 0000000

P0 & (1<<5) = 00000000 & 00100000 = 00000000

P0. 7	P0. 6	P0. 5	P0. 4	P0. 3	P0. 2	P0. 1	P0.0
0	0	Switch 1	0	0	0	0	0

P0= 0x20 = 00100000

P0 & (1<<5) = 00100000 & 00100000 = 00100000



Embedded C Programming

Monitoring Specific bit change in Registers

To monitor for the change in 5th bit from 1 to 0

```
while ( ! (P0 & (1<<5) ) )      //wait indefinitely until 5th bit changes from 0 to 1
{                               //do something //exit loop
}
```

1= 00000001

1<<5 = 00100000

P0. 7	P0. 6	P0. 5	P0. 4	P0. 3	P0. 2	P0. 1	P0.0
0	0	Switch 1	0	0	0	0	0

P0= 0x20 = 0010000

P0 & (1<<5) = 00100000 & 00100000 = 00100000

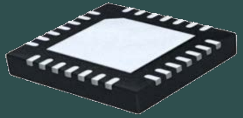
! (P0 & (1<<5)) = ! (00100000) = 00000000

P0. 7	P0. 6	P0. 5	P0. 4	P0. 3	P0. 2	P0. 1	P0.0
0	0	Switch 0	0	0	0	0	0

P0= 0x00 = 00000000

P0 & (1<<5) = 00000000 & 00100000 = 00000000

! (P0 & (1<<5)) = ! (00000000) = 00000001



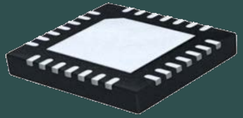
Monitoring Specific bit change in Registers

To monitor for the change in 5th bit from 0 to 1

```
While ( P0  &  (1<<5) )        //wait indefinitely until 5th bit changes from 0 to 1
{
                                  //do something //exit loop
}
```

To monitor for the change in 5th bit from 1 to 0 we just Negate the condition inside while loop .

```
while ( ! (P0 & (1<<5) ) )       //wait indefinitely until 12th bit changes from 1 to 0
 {                                //do something //exit loop

 }
```



Extracting Bits REGHL_16

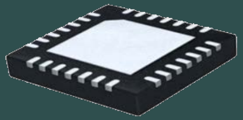
assume current value of REGHL_16 as ‘1000000110101011' which is 16 bit.

LED 15	bit 14	bit 13	bit 12	bit 11	bit 10	bit 9	bit 8	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
1	0	0	0	0	0	0	1	1	0	1	0	1	0	1	1

Now we have asked to extract lower 8-bits and upper 8 bit register into REGH_8 and REGL_8

REGH_8 = (REGHL_16 & 0XFF00) >>8; // binary 10000001
REGHL_16 & 0XFF00 = 10000001 00000000
(REGHL_16 & 0XFF00) >>8 = 00000000 10000001

REGL_8 = (REGHL_16 & 0X00FF) ; // binary 10101011



C control structures

Control order in which instructions are executed

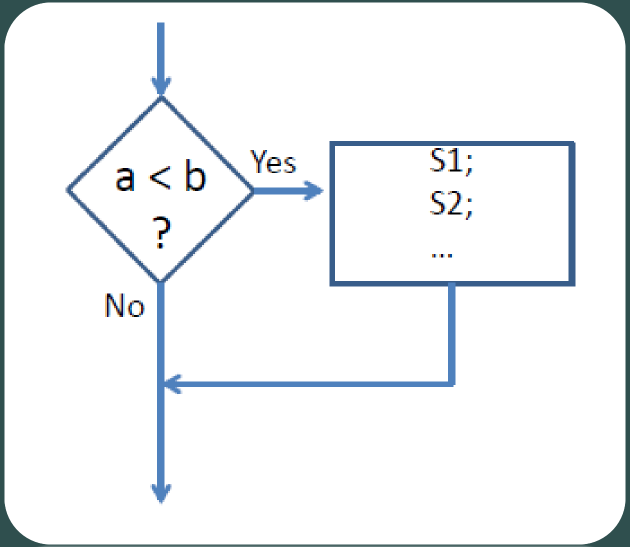
- Conditional execution
  - Execute a set of statements if some condition is met
  - Select one set of statements to be executed from several options, depending on one or more conditions

- 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

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



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

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

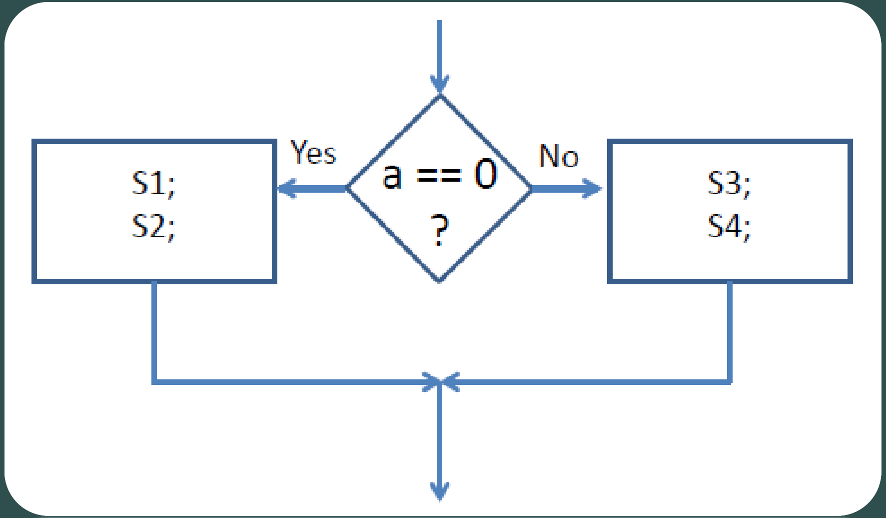
Note the difference between
Boolean operators &&, || and
bitwise logical operators &, |

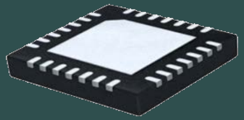
Note that ==is a relational operator,
whereas =is an assignment operator

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:
}
```

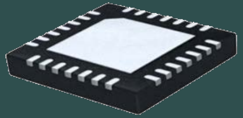




Multiple ELSE-IF structure

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
}
```



SWITCH statement

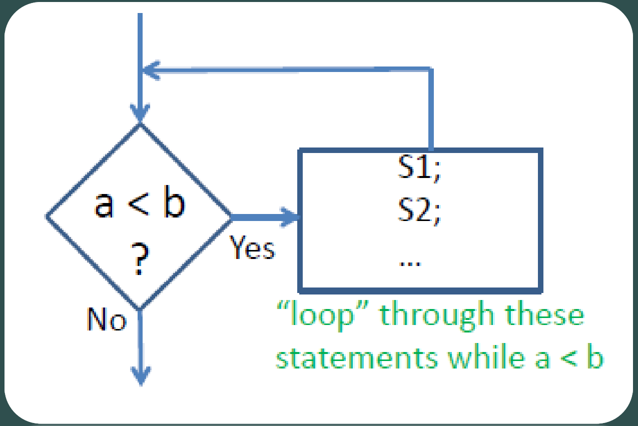
Compact alternative to ELSE-IF structure, for multiway decision that tests one variable or expression for a number of constant values.

```
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
 }
```

WHILE loop structure

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

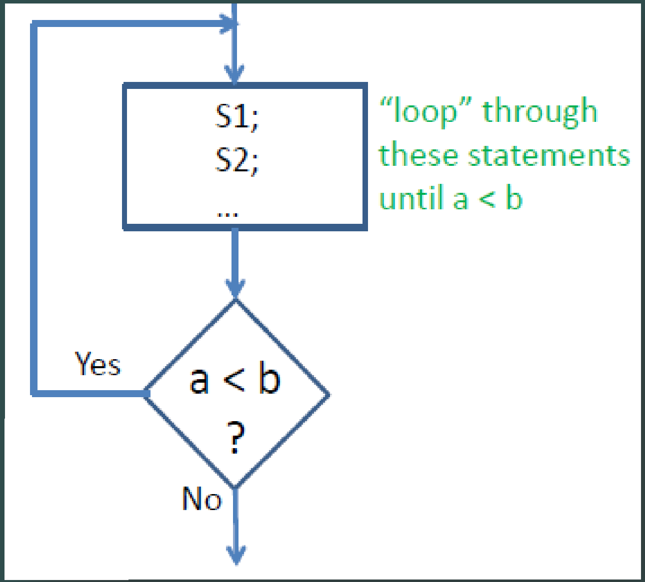
```
while (a < b)
{
statement s1;
statement s2;
….
}
```

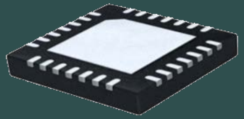


DO-WHILE loop structure

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

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





FOR loop structure

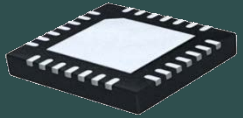
FOR loop is a more compact form of the WHILE loop structure

Initialization(s)

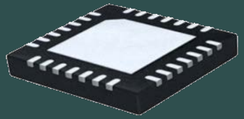
Condition for execution

Operation(s) at end of each loop

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



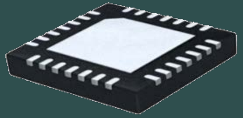

```
/* Nested FOR loops to create a time delay */

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 function

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 3rdparties, instead of designing your own
- The function may return a result to the caller
- One or more arguments may be passed to the function/procedure



Embedded C Programming

```
#include<reg51.h>

Int  math_func( int k; int  n)
```

Function Declaration

```
Void main()
{
Int a,b,c;
a = 10; b =20;
c=math_func (a,b);
}
```

Function call

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

Function definition

Bit-parallel logical operators

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

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