respectively. If the parameters are *int* values, they are passed using register pairs (R7, R6), (R5, R4) and (R3, R2). If the number of arguments is greater than three, the first three arguments are passed through registers and rest is passed through fixed memory locations. Refer to C51 documentation for more details. Return values are usually passed through general purpose registers. R7 is used for returning *char* value and register pair (R7, R6) is used for returning *int* value. The 'C' subroutine can be invoked from the assembly program using the subroutine call Assembly instruction (Again cross compiler dependent).

```
E.g.  LCALL     _Cfunction
```

Where *Cfunction* is a function written in 'C'. The prefix _ informs the cross compiler that the parameters to the function are passed through registers. If the function is invoked without the _ prefix, it is understood that the parameters are passed through fixed memory locations.

9.2.3.3 Inline Assembly Inline assembly is another technique for inserting target processor/controller specific Assembly instructions at any location of a source code written in high level language 'C'. This avoids the delay in calling an assembly routine from a 'C' code (If the Assembly instructions to be inserted are put in a subroutine as mentioned in the section mixing assembly with 'C'). Special keywords are used to indicate that the start and end of Assembly instructions. The keywords are cross-compiler specific. C51 uses the keywords *#pragma asm* and *#pragma endasm* to indicate a block of code written in assembly.

```
E.g.    #pragma asm
        MOV A, #13H
        #pragma endasm
```


Important Note:

*The examples used for illustration throughout the section **Mixing Assembly & High Level Language** is Keil C51 cross compiler specific. The operation is cross compiler dependent and it varies from cross compiler to cross compiler. The intention of the author is just to give an overall idea about the mixing of Assembly code and High level language 'C' in writing embedded programs. Readers are advised to go through the documentation of the cross compiler they are using for understanding the procedure adopted for the cross compiler in use.*

9.3 PROGRAMMING IN EMBEDDED C

Whenever the conventional 'C' Language and its extensions are used for programming embedded systems, it is referred as '***Embedded C***' programming. Programming in 'Embedded C' is quite different from conventional Desktop application development using 'C' language for a particular OS platform. Desktop computers contain working memory in the range of Megabytes (Nowadays Giga bytes) and storage memory in the range of Giga bytes. For a desktop application developer, the resources available are surplus in quantity and s/he can be very lavish in the usage of RAM and ROM and no restrictions are imposed at all. This is not the case for embedded application developers. Almost all embedded systems are limited in both storage and working memory resources. Embedded application developers should be aware of this fact and should develop applications in the best possible way which optimises the code memory and working memory usage as well as performance. In other words, the hands of an embedded application developer are always tied up in the memory usage context☺.

9.3.1 'C' v/s. 'Embedded C'

'C' is a well structured, well defined and standardised general purpose programming language with extensive bit manipulation support. 'C' offers a combination of the features of high level language and assembly and helps in hardware access programming (system level programming) as well as business package developments (Application developments like pay roll systems, banking applications, etc). The conventional 'C' language follows ANSI standard and it incorporates various library files for different operating systems. A platform (operating system) specific application, known as, compiler is used for the conversion of programs written in 'C' to the target processor (on which the OS is running) specific binary files. Hence it is a platform specific development.

Embedded 'C' can be considered as a subset of conventional 'C' language. Embedded 'C' supports all 'C' instructions and incorporates a few target processor specific functions/instructions. It should be noted that the standard ANSI 'C' library implementation is always tailored to the target processor/controller library files in Embedded 'C'. The implementation of target processor/controller specific functions/instructions depends upon the processor/controller as well as the supported cross-compiler for the particular Embedded 'C' language. A software program called 'Cross-compiler' is used for the conversion of programs written in Embedded 'C' to target processor/controller specific instructions (machine language).

9.3.2 Compiler vs. Cross-Compiler

Compiler is a software tool that converts a source code written in a high level language on top of a particular operating system running on a specific target processor architecture (e.g. Intel x86/Pentium). Here the operating system, the compiler program and the application making use of the source code run on the same target processor. The source code is converted to the target processor specific machine instructions. The development is platform specific (OS as well as target processor on which the OS is running). Compilers are generally termed as '*Native Compilers*'. A native compiler generates machine code for the same machine (processor) on which it is running.

Cross-compilers are the software tools used in cross-platform development applications. In cross-platform development, the compiler running on a particular target processor/OS converts the source code to machine code for a target processor whose architecture and instruction set is different from the processor on which the compiler is running or for an operating system which is different from the current development environment OS. Embedded system development is a typical example for cross-platform development where embedded firmware is developed on a machine with Intel/AMD or any other target processors and the same is converted into machine code for any other target processor architecture (e.g. 8051, PIC, ARM etc). Keil C51 is an example for cross-compiler. The term 'Compiler' is used interchangeably with 'Cross-compiler' in embedded firmware applications. Whenever you see the term 'Compiler' related to any embedded firmware application, please understand that it is referring the cross-compiler.

9.3.3 Using 'C' in 'Embedded C'

The author takes the privilege of assuming the readers are familiar with 'C' programming. Teaching 'C' is not in the scope of this book. If you are not familiar with 'C' language syntax and 'C' programming technique, please get a handle on the same before you proceed. Readers are advised to go through books by 'Brian W. Kernighan and Dennis M. Ritchie (K&R)' or 'E. Balagurusamy' on 'C' programming.

This section is intended only for giving readers a basic idea on how 'C' Language is used in embedded firmware development.

Let us brush up whatever we learned in conventional 'C' programming. Remember we will only go through the peripheral aspects and will not go in deep.

9.3.3.1 Keywords and Identifiers *Keywords* are the reserved names used by the 'C' language. All *keywords* have a fixed meaning in the 'C' language context and they are not allowed for programmers for naming their own variables or functions. ANSI 'C' supports 32 keywords and they are listed below. All 'C' supported keywords should be written in '*lowercase*' letters.

auto	Double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Identifiers are user defined names and labels. Identifiers can contain letters of English alphabet (both upper and lower case) and numbers. The starting character of an identifier should be a letter. The only special character allowed in identifier is underscore (_).

9.3.3.2 Data Types Data type represents the type of data held by a variable. The various data types supported, their storage space (bits) and storage capacity for 'C' language are tabulated below.

Data Type	Size (Bits)	Range	Comments
char	8	–128 to +127	Signed character
signed char	8	–128 to +127	Signed character
unsigned char	8	0 to +255	Unsigned character
short int	8	–128 to +127	Signed short integer
signed short int	8	–128 to +127	Signed short integer
unsigned short int	8	0 to +255	Unsigned short integer
int	16	–32,768 to +32,767	Signed integer
signed int	16	–32,768 to +32,767	Signed integer
unsigned int	16	0 to +65,535	Unsigned integer
long int	32	–2147,483,648 to +2,147,483,647	Signed long integer
signed long int	32	–2147,483,648 to +2,147,483,647	Signed long integer
unsigned long int	32	0 to +4,294,967,295	Unsigned long integer
float	32	3.4E-38 to 3.4E+38	Signed floating point
double	64	1.7E-308 to 1.7E+308	Signed floating point (Double precision)
long double	80	3.4E-4932 to 3.4E+4932	Signed floating point (Long Double precision)

The data type size and range given above is for an ordinary 'C' compiler for 32 bit platform. It should be noted that the storage size may vary for data type depending on the cross-compiler in use for embedded applications.

Since memory is a big constraint in embedded applications, select the optimum data type for a variable. For example if the variable is expected to be within the range 0 to 255, declare the same as an '*unsigned char*' or '*unsigned short int*' data type instead of declaring it as '*int*' or '*unsigned int*'. This will definitely save considerable amount of memory.

9.3.3.3 Storage Class Keywords related to storage class provide information on the scope (visibility or accessibility) and life time (existence) of a variable. 'C' supports four types of storage classes and they are listed below.

Storage class	Meaning	Comments
auto	Variables declared inside a function. Default storage class is auto	Scope and accessibility is restricted within the function where the variable is declared. No initialization. Contains random values at the time of creation
register	Variables stored in the CPU register of processor. Reduces access time of variable	Same as auto in scope and access. The decision on whether a variable needs to be kept in CPU register of the processor depends on the compiler
static	Local variable with life time same as that of the program	Retains the value throughout the program. By default initialises to zero on variable creation. Accessibility depends on where the variable is declared
extern	Variables accessible to all functions in a file and all files in a multiple file program	Can be modified by any function within a file or across multiple files (variable needs to be exported by one file and imported by other files using the same)

Apart from these four storage classes, 'C' literally supports storage class '*global*'. An '*auto or static*' variable declared in the public space of a file (declared before the implementation of all functions including main in a file) is accessible to all functions within that file. There is no explicit storage class for '*global*'. The way of declaration of a variable determines whether it is global or not.

9.3.3.4 Arithmetic Operations The list of arithmetic operators supported by 'C' are listed below

Operator	Operation	Comments
+	Addition	Adds variables or numbers
–	Subtraction	Subtracts variables or numbers
*	multiplication	multiplies variables or numbers
/	Division	Divides variables or numbers
%	Remainder	Finds the remainder of a division

9.3.3.5 Logical Operations Logical operations are usually performed for decision making and program control transfer. The list of logical operations supported by 'C' are listed below

Operator	Operation	Comments
&&	Logical AND	Performs logical AND operation. Output is true (logic 1) if both operands (left to and right to of && operator) are true
||	Logical OR	Performs logical OR operation. Output is true (logic 1) if either operand (operands to left or right of || operator) is true
!	Logical NOT	Performs logical Negation. Operand is complemented (logic 0 becomes 1 and vice versa)

9.3.3.6 Relational Operations Relational operations are normally performed for decision making and program control transfer on the basis of comparison. Relational operations supported by 'C' are listed below.

Operator	Operation	Comments
<	less than	Checks whether the operand on the left side of '<' operator is less than the operand on the right side. If yes return logic one, else return logic zero
>	greater than	Checks whether the operand on the left side of '>' operator is greater than the operand on the right side. If yes return logic one, else return logic zero
<=	less than or equal to	Checks whether the operand on the left side of '<=' operator is less than or equal to the operand on the right side. If yes return logic one, else return logic zero
>=	greater than	Checks whether the operand on the left side of '>=' operator is greater than or equal to the operand on the right side. If yes return logic one, else return logic zero
==	Checks equality	Checks whether the operand on the left side of '==' operator is equal to the operand on the right side. If yes return logic one, else return logic zero
!=	Checks non-equality	Checks whether the operand on the left side of '!=' operator is not equal to the operand on the right side. If yes return logic one, else return logic zero

9.3.3.7 Branching Instructions Branching instructions change the program execution flow conditionally or unconditionally. Conditional branching depends on certain conditions and if the conditions are met, the program execution is diverted accordingly. Unconditional branching instructions divert program execution unconditionally.

Commonly used conditional branching instructions are listed below

Conditional branching instruction	Explanation

//if statement
 	
if (expression)
{
statement1;
statement2;
……;
}
statement 3;
……..;
	Evaluates the expression first and if it is true executes the statements given within the { } braces and continue execution of statements following the closing curly brace (}). Skips the execution of the statements within the curly brace { } if the expression is false and continue execution of the statements following the closing curly brace (}).
	
	One way branching

//if else statement
if (expression)
{
if_statement1;
if_statement2;
……..;
}
else
{
else_statement1;
	Evaluates the expression first and if it is true executes the statements given within the { } braces following if (expression) and continue execution of the statements following the closing curly brace (}) of else block. Executes the statements within the curly brace { } following the else, if the expression is false and continue execution of statements following the closing curly brace (}) of else.

else_statement2; ……; } statement 3;	Two way branching

//switch case statement
switch (expression)
{
case value1:
break;
case value2;
break;
default:
break;
}
	Tests the value of a given expression against a list of case values for a matching condition. The expression and case values should be integers. value1, value2, etc. are integers. If a match found, executes the statement following the case and breaks from the switch. If no match found, executes the default case.
	
	Used for multiple branching.

//Conditional operator
// ?exp1 : exp2
 	
(expression) ?exp1: exp2
 	
E.g.
if (x>y)
a=1;
else
a=0;
can be written using conditional operator as
a=(x>y)? 1:0
	Used for assigning a value depending on the (expression). (expression) is calculated first and if it is greater than 0, evaluates exp1 and returns it as a result of operation else evaluate exp2 and returns it as result. The return value is assigned to some variable.
	
	It is a combination of if else with assignment statement.
	
	Used for two way branching

//unconditional branching goto label	goto is used as unconditional branching instruction. goto transfers the program control indicated by a label following the goto statement. The label indicated by goto statement can be anywhere in the program either before or after the goto label instruction. goto is generally used to come out of deeply nested loops in abnormal conditions or errors.

9.3.3.8 Looping Instructions Looping instructions are used for executing a particular block of code repeatedly till a condition is met or wait till an event is fired. Embedded programming often uses the looping instructions for checking the status of certain I/o ports, registers, etc. and also for producing delays. Certain devices allow write/read operations to and from some registers of the device only when the device is ready and the device ready is normally indicated by a status register or by setting/clearing certain bits of status registers. Hence the program should keep on reading the status register till the device ready indication comes. The reading operation forms a loop. The looping instructions supported by 'C' are listed below.

Looping instruction	Explanation

//while statement
while (expression)
{
 	
body of while loop
 	
}
	Entry controlled loop statement.
	The expression is evaluated first and if it is true the body of the loop is entered and executed. Execution of 'body of while loop' is repeated till the expression becomes false.

// do while loop
 	
do
{
 	
body of do loop
 	
}
 	
while (expression);
	The 'body of the loop' is executed at least once. At the end of each execution of the 'body of the loop', the while condition (expression) is evaluated and if it is true the loop is repeated, else loop is terminated.

//for loop
 	
for (initialisation; test for condition; update variable)
{
body of for loop
}
	Entry controlled loop. Enters and executes the 'body of loop' only if the test for condition is true. *for* loop contains a loop control variable which may be initialised within the initialisation part of the loop. Multiple variables can be initialised with ',' operator.

//exiting from loop break; goto label	Loops can be exited in two ways. First one is normal exit where loop is exited when the expression/test for condition becomes false. Second one is forced exit. break and goto statements are used for forced exit. break exits from the innermost loop in a deeply nested loop, whereas goto transfers the program flow to a defined label.

//skipping portion of a loop
 	
while (expression)
{
……;
if (condition);
continue;
……;
}
	Certain situation demands the skipping of a portion of a loop for some conditions. The 'continue' statement used inside a loop will skip the rest of the portion following it and will transfer the program control to the beginning of the loop.
	//for loop with skipping
	
	for (initialisation; test for condition; update variable)
	{
	……;
	if (condition)
	continue;
	……;
	}

//do while with skipping
do
{
……;
if (condition)
continue;
……;
}
while (expression);	

Every 'for' loop can be replaced by a 'while' loop with a counter.

Let's consider a typical example for a looping instruction in embedded C application. I have a device which is memory mapped to the processor and I'm supposed to read the various registers of it (except status register) only after the contents of its status register, which is memory mapped at 0x3000 shows device is ready (say value 0x01 means device is ready). I can achieve this by different ways as given below.

```
//#############################################################
//using while loop
//#############################################################

char *status_reg = (char *) 0x3000; //Declares memory mapped register

while (*status_reg!=0x01);       //Wait till status_reg = 0x01

//#############################################################
//using do while loop
//#############################################################

char *status_reg = (char*) 0x3000;

do
{

} while (*status_reg!=0x01); Loop till status_reg = 0x01

//#############################################################
//using for loop
//#############################################################

char *status_reg = (char*) 0x3000;

for (;(*status_reg!=0x01););
```

The instruction *char *status_reg = (char*) 0x3000;* declares *status_reg* as a character pointer pointing to location 0x3000. The character pointer is used since the external device's register is only 8bit wide. We will discuss the pointer based memory mapping technique in a later section. In order to avoid compiler optimisation, the pointer should be declared as volatile pointer. We will discuss the same also in another section.

9.3.3.9 Arrays and Pointers Array is a collection of related elements (data types). Arrays are usually declared with data type of array, name of the array and the number of related elements to be placed in the array. For example the following array declaration

```
char arr [5];
```

declares a character array with name 'arr' and reserves space for 5 character elements in the memory as in Fig. 9.3†.

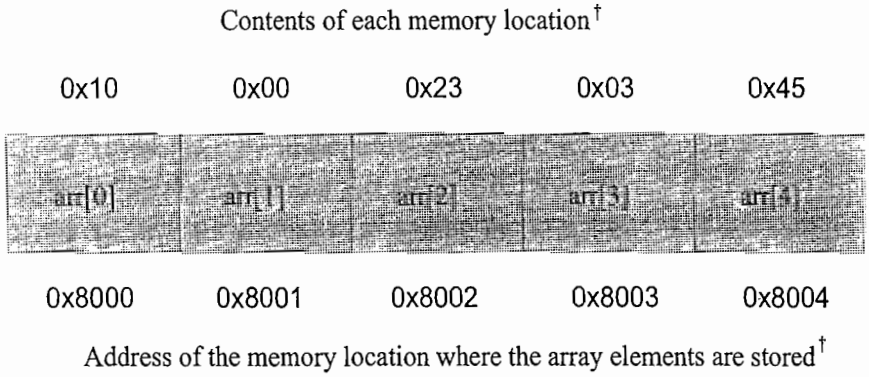

Fig. 9.3 **Array representation in memory**

The elements of an array are accessed by using the array index or subscript. The index of the first element is '0'. For the above example the first element is accessed by arr[0], second element by arr[1], and so on. In the above example, the array starts at memory location 0x8000 (arbitrary value taken for illustration) and the address of the first element is 0x8000. The '*address of*' operator (&) returns the address of the memory location where the variable is stored. Hence &arr[0] will return 0x8000 and &arr[1] will return 0x8001, etc. The name of the array itself with no index (subscript) always returns the address of the first element. If we examine the first element arr[0] of the above array, we can see that the variable arr[0] is allocated a memory location 0x8000 and the contents of that memory location holds the value for arr[0] (Fig. 9.4).

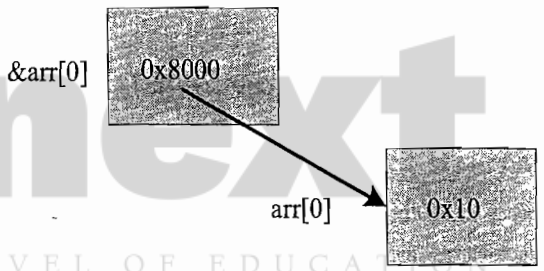

Fig. 9.4 **Array element address and content relationship**

Arrays can be initialised by two methods. The first method is initialising the entire array at the time of array declaration itself. Second method is selective initialisation where any member can be initialised or altered with a value.

```
//Initialization of array at the time of declaration

unsigned char arr[5] = {5, 10, 20, 3, 2};
unsigned char arr[ ] = {5, 10, 20, 3, 2};

//Selective initialization

unsigned char arr[5];

arr[0] = 5;
arr[1] = 10;
arr[2] = 20;
arr[3] = 3;
arr[4] = 2;
```

† Arbitrary value taken for illustration.

A few important points on Arrays

1. The '*sizeof()*' operator returns the size of an array as the number of bytes. E.g. '*sizeof (arr)*' in the above example returns 5. If arr[ ] is declared as an integer array and if the byte size for integer is 4, executing the '*sizeof (arr)*' instruction for the above example returns 20 (5 × 4).
2. The '*sizeof()*' operator when used for retrieving the size of an array which is passed as parameter to a function will only give the size of the data type of the array.

```
E.g.
    void test (char *p);              //Function declaration
    char arr [5] = {5, 10, 20, 3, 2};     //Array data type char

    void main ( )
    {
    test (arr);
    }

    void test (char *p)
    {
    printf ("%d", sizeof (*p));
    }
```

This code snippet will print '1' as the output, though the user expects 5 (size of arr) as output. The output is equivalent to sizeof (char), size of the data type of the array.

3. Use the syntax '*extern array type array name [ ]*' to access an array which is declared outside the current file. For example '*extern char arr[ ]*' for accessing the array '*arr[ ]*' declared in another file
4. Arrays are not equivalent to pointers. But the expression '*array name*' is equivalent to a pointer, of type specified by the array, to the first element of an array, e.g. '*arr*' illustrated for sizeof () operator is equivalent to a character pointer pointing to the first element of array '*arr*'.
5. Array subscripting is commutative in 'C' language and '*arr[k]*' is same as '**((arr)+(k))*' where '*arr[k]*' is the content of k^{th} element of array '*arr*' and '*(arr)*' is the starting address of the array arr and *k* is the index of the array or offset address for the 'k^{th}' element from the base address of array. '**((arr) + (k))*' is the content of array for the index *k*.

Pointers *Pointer* is a flexible at the same time most dangerous feature, capable of creating potential damages leading to firmware crash, if not used properly. Pointer is a memory pointing based technique for variable access and modification. Pointers are very helpful in

1. Accessing and modifying variables
2. Increasing speed of execution
3. Accessing contents within a block of memory
4. Passing variables to functions by eliminating the use of a local copy of variables
5. Dynamic memory allocation (Will be discussed later)

To understand the pointer concept, let us have a look at the data memory organisation of a processor. For a processor/controller with 128 bytes user programmable internal RAM (e.g. AT89C51), the memory is organised as

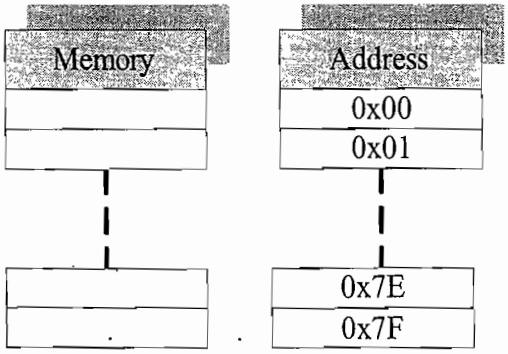

Fig. 9.5 **Data memory organisation for *8051***

If we declare a character variable, say *char input*, the compiler assigns a memory location to the variable anywhere within the internal memory 0x00 to 0x7F. The allocation is left to compiler's choice unless specified explicitly (Let it be 0x45 for our example). If we assign a value (say 10) to the variable ***input*** (input=10), the memory cell representing the variable input is loaded with 10.

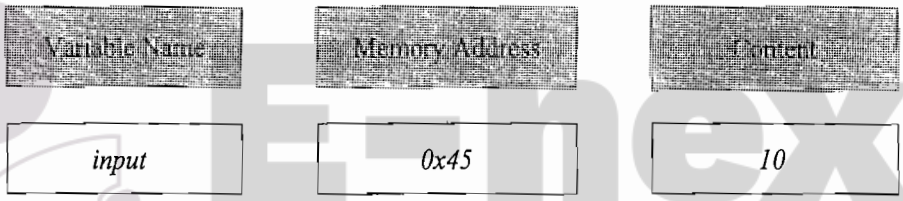

Fig. 9.6 **Relationship between variable name, address and data held by variable**

The contents of memory location 0x45 (representing the variable *input*) can be accessed and modified by using a pointer of type same as the variable (*char* for the variable *input* in the example). It is accomplished by the following method.

```
char input;     //Declaring input as character variable
char *p; //Declaring a character pointer p (* denotes p is a pointer)
p = &input      //Assigns the address of input as content to p
```

The same is diagrammatically represented as

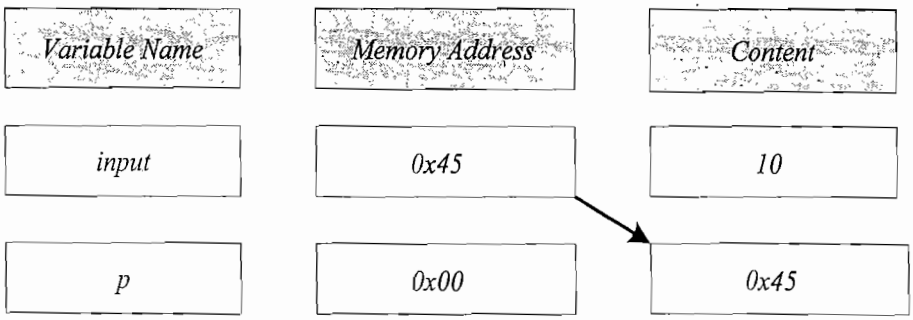

Fig. 9.7 **Pointer based memory access technique**

The compiler assigns a memory to the character pointer variable '*p*'. Let it be 0x00 (Arbitrary value chosen for illustration) and the memory location holds the memory address of variable *input* (0x45)

as content. In 'C' the address assignment to pointer is done using the address of operator '&' or it can be done using explicitly by giving a specific address. The pointer feature in 'C' is same as the indirect addressing technique used in *8051* Assembly instructions. The code snippet

```
MOV R0, #45H
MOV A,@R0 ; R0 & R1 are the indirect addressing registers.
```

is an example for 8bit memory pointer usage in *8051* Assembly and the code snippet

```
MOV DPTR, #0045H
MOV A,@DPTR ; DPTR is the 16bit indirect addressing register
```

is an example for 16bit memory pointer operation. The general form of declaring a pointer in 'C' is

```
data type *pointer; //'data type' is the standard data type like
                    //int, char, float etc.. supported by 'C' language.
```

The * (asterisk) symbol informs the compiler that the variable *pointer* is a pointer variable. Like any other variables, pointers can also be initialised in its declaration itself.

```
E.g.    char x, y;
        char *ptr = &x;
```

The contents pointed by a pointer is modified/retrieved by using * as prefix to the pointer.

```
E.g.    char x=5, y=56;
        char *ptr=&x;          //ptr holds address of x
        *ptr = y;              //x = y
```

Pointer Arithmetic and Relational Operations 'C' language supports the following Arithmetic and relational operations on pointers.

1. Addition of integer with pointer. e.g. ptr+2 (It should be noted that the pointer is advanced forward by the storage length supported by the compiler for the data type of the pointer multiplied by the integer. For example for integer pointer where storage size of int = 4, the above addition advances the pointer by $4 \times 2 = 8$)
2. Subtraction of integer from pointer, e.g. ptr–2 (Above rule is applicable)
3. Incremental operation of pointer, e.g. ++ptr and ptr++ (Depending on the type of pointer, the ++ increment context varies). For a character pointer ++ operator increments the pointer by 1 and for an integer pointer the pointer is incremented by the storage size of the integer supported by the compiler (e.g. pointer ++ results in pointer + 4 if the size for integer supported by compiler is 4)
4. Decrement operation of pointer, e.g. – –ptr and ptr– – (Context rule for decrement operation is same as that of incremental operation)
5. Subtraction of pointers, e.g. ptr1– ptr2
6. Comparison of two pointers using relational operators, e.g. ptr1 > ptr2, ptr1 < ptr2, ptr1 = = ptr2, ptr1! = ptr2 etc (Comparison of pointers of same type only will give meaningful results)

Note:
1. Addition of two pointers, say *ptr1* + *ptr2* is illegal
2. Multiplication and division operations involving pointer as numerator or denominator are also not allowed

A few important points on Pointers

1. The instruction *ptr++ increments only the pointer not the content pointed by the pointer '***ptr***' and *ptr– – decrements the pointer not the contents pointed by the pointer '*ptr*'
2. The instruction (*ptr) ++ increments the content pointed by the pointer '*ptr*' and not the pointer '*ptr*'. (*ptr)-- decrements the content pointed by the pointer '*ptr*' and not the pointer '*ptr*'
3. A type-casted pointer cannot be used in an assignment expression and cannot be incremented or decremented, e.g. ((int *ptr))++; will not work in the expected way
4. '***Null Pointer***' is a pointer holding a special value called '*NULL*' which is not the address of any variable or function or the start address of the allocated memory block in dynamic memory allocations
5. Pointers of each type can have a related null pointer viz. there can be character type null pointer, integer type null pointer, etc.
6. '*NULL*' is a preprocessor macro which is literally defined as zero or ((void *) 0). #define NULL 0 or #define NULL ((void *) 0)
7. '*NULL*' is a constant zero and both can be used interchangeably as Null pointer constants
8. A '*NULL*' pointer can be checked by the operator if ( ). See the following example

```
if (ptr)  //ptr is a pointer declared
printf ("ptr is not a NULL pointer");
else
printf ("ptr is a NULL pointer");
```

   The statement if (ptr) is converted to if (ptr! =0) by the (cross) compiler. Alternatively you can directly use the statement if (ptr! =0) in your program to check the 'NULL' pointer.
9. Null pointer is a 'C' language concept and whose internal value does not matter to us. '*NULL*' always guarantee a '0' to you but Null pointer need not be.

Pointers and Arrays—Are they related? Arrays are not equivalent to pointers and vice versa. But the expression array 'name [ ]' is equivalent to a pointer, of type specified by the array, to the first element of an array, e.g. for the character array '*char arr[5]*', '*arr[ ]*' is equivalent to a character pointer pointing to the first element of array '*arr*' (This feature is referred to as '*equivalence of pointers and* ***arrays***'). You can achieve the array features like accessing and modifying members of an array using a pointer and pointer increment/decrement operators. Arrays and pointer declarations are interchangeable when they are used as parameters to functions. Point 2 discussed under the section '**A few important points on Arrays**' for *sizeof()* operator usage explains this feature also.

9.3.3.10 Characters and Strings Character is a one byte data type and it can hold values ranging from 0 to 255 (unsigned character) or -128 to +127 (signed character). The term character literally refers to the alpha numeric characters (English alphabet A to Z (both small letters and capital letters) and number representation from '0' to '9') and special characters like *, ?, !, etc. An integer value ranging from 0 to 255 stored in a memory location can be viewed in different ways. For example, the hexadecimal number 0x30 when represented as a character will give the character '0' and if it is viewed as a decimal number it will give 48. String is an array of characters. A group of characters defined within a double quote represents a constant string.

'*H*' is an example for a character, whereas "*Hello*" is an example for a string. String always terminates with a '\0' character. The '\0' character indicates the string termination. Whenever you declare a string using a character array, allocate space for the null terminator '\0' in the array length.

```
E.g.    char name [ ] = "SHIBU" ;
or      char name [6] = {'S', 'H', 'I', 'B', 'U', '\0'};
```

String operations are very important in embedded product applications. Many of the embedded products contain visual indicators in the form of alpha numeric displays and are used for displaying text. Though the conventional alpha numeric displays are giving way to graphic displays, they are deployed widely in low cost embedded products. The various operations performed on character strings are explained below.

Input & Output operations Conventional 'C' programs running on Desktop machines make use of the standard string inputting (*scanf()*) and string outputting (*printf()*) functions from the platform specific I/O library. Standard keyboard and monitor are used as the input and output media for desktop application. This is not the case for embedded systems. Embedded systems are compact and they need not contain a standard keyboard or monitor screen for I/O functions. Instead they incorporate application specific keyboard and display units (Alpha numeric/graphic) as user interfaces. The standard string input instruction supported by 'C' language is *scanf( )* and a code snippet illustrating its usage is given below.

```
char name [6];
scanf ("%s", name);
```

A standard ANSI C compiler converts this code according to the platform supported I/O library files and waits for inputting a string from the keyboard. The scanf function terminates when a white space (blank, tab, carriage return, etc) is encountered in the input string. Implementation of the *scanf()* function is (cross) compiler dependent. For example, for 8051 microcontroller all I/O operations are supposed to be executed by the serial interface and the C51 cross compiler implements the *scanf()* function in such a way to expect and receive a character/string (according to the scanf usage context (character if first parameter to *scanf()* is "%c" and string if first parameter is "%s")) from the serial port.

printf() is the standard string output instruction supported by 'C' language. A code snippet illustrating the usage of *printf( )* is given below.

```
char name [ ] = "SHIBU";
printf ("%s", name);
```

Similar to *scanf()* function, the standard ANSI C compiler converts this code according to the platform supported I/O library files and displays the string in a console window displayed on the monitor. Implementation of *printf()* function is also (cross) compiler specific. For 8051 microcontroller the C51 cross compiler implements the *printf()* function in such a way to send a character/string (according to the printf usage context (character if first parameter to *printf()* is "%c" and string if first parameter is "%s")) to the serial port.

String Concatenation Concatenation literally means '*joining together*'. String concatenation refers to the joining of two or more strings together to form a single string. 'C' supports built in functions for string operations. To make use of the built in string operation functions, include the header file '*string.h*' to your '*.c*' file. '*strcat()*' is the function used for concatenating two strings. The syntax of '*strcat()*' is illustrated below.

```
strcat (str1, str2); //'str1' and 'str2' are the strings to
                     //be concatenated.
```

On executing the above instruction the null character ('\0') from the end of '*str1*' is removed and '*str2*' is appended to '*str1*'. The string '*str2*' remains unchanged. As a precautionary measure, ensure that str1 (first parameter of '*strcat( )*' function) is declared with enough size to hold the concatenated string. '*strcat()*' can also be used for appending a constant string to a string variable.

```
E.g.   strcat(str1, "Hello!");
```

> ***Note:***
> 1. Addition of two strings, say *str1+str2* is not allowed
> 2. Addition of a string variable, say str1, with a constant string, say "*Hello*" is not allowed

String Comparison Comparison of strings can be performed by using the '*strcmp()*' function supported by the string operation library file. Syntax of *strcmp()*' is illustrated below.

```
strcmp (str1, str2); //str1 and str2 are two character strings
```

If the two strings which are passed as arguments to '*strcmp()*' function are equal, the return value of '*strcmp()*' will be zero. If the two strings are not equal and if the ASCII value of the first non-matching character in the string *str1* is greater than that of the corresponding character for the second string *str2*, the return value will be greater than zero. If the ASCII value of first non-matching character in the string *str1* is less than that of the corresponding character for the second string *str2*, the return value will be less than zero. '*strcmp()*' function is used for comparing two string variables, string variable and constant string or two constant strings.

```
E.g.   char str1 [ ] = "Hello world";
       char str2 [ ] = "Hello World!";
       int n;
       n= strcmp(str1, str2);
```

Executing this code snippet assigns a value which is greater than zero to *n*. (since ASCII value of '*w*' is greater than that of '*W*'). *n= strcmp(str2, str1);* will assign a value which is less than zero to *n*. (since ASCII value of '*W*' is less than that of '*w*').

The function *stricmp()* is another version of *strcmp()*. The difference between the two is, *stricmp()* is not case sensitive. There won't be any differentiation between upper and lowercase letters when *stricmp()* is used for comparison. If *stricmp()* is used for comparing the two strings str1 and str2 in the above example, the return value of the function *stricmp* (str1, str2) will be zero.

> ***Note:***
> 1. Comparison of two string variables using '= =' operator is invalid, e.g. if (*str1 == str2*) is invalid
> 2. Comparison of a string variable and a string constant using '= =' operator is also invalid, e.g. if (*str1 == "Hello"*) is invalid

Finding String length String length refers to the number of characters except the null terminator character '\0' present in a string. String length can be obtained by using a counter combined with a search operation for the '\0' character. 'C' supplies a ready to use string function ***strlen()*** for determining the length of a string. Its syntax is explained below.

```
strlen (str1);          //where str1 is a character string
```

```
e.g.   char str1 [ ] = "Hello World!"  ;
       int n;
       n = strlen (str1);
```

Executing this code snippet assigns the numeric value 12 to integer variable '*n*'.

Copying Strings *strcpy()* function is the 'C' supported string operation function for copying a string to another string. Syntax of *strcpy( )* function is

```
strcpy (str1, str2);   //str1, str2 are character strings.
```

This function assigns the contents of *str2* to *str1*. The original content of str1 is overwritten by the contents of *str2*. *str2* remains unchanged. The size of the character array which is passed as the first argument the *strcpy( )* function should be large enough to hold the copied string. *strcpy()* function can also be used to assign constant string to string variables.

```
e.g.   strcpy(str1, "Hello!")
```

> ***Note:***
> A string variable cannot be copied to another string variable using the '=' operator e.g. *str1 = str2* is illegal

A few important points on Characters and Strings

1. The function *strcat()* is used for concatenating two string variables or string variable and string constants. Characters cannot be appended to a string variable using *strcat()* function. For example *strcat (str1, 'A')* may not give the expected result. The same can be achieved by *strcat (str1, "A")*
2. Strings are character arrays and they cannot be assigned directly to a character array (except the initialisation of arrays using string constants at the time of declaring character arrays)

```
//#############################################################
E.g.    unsigned char str1 [] = 'Hello"; // is valid;
        unsigned char str1 [6];
        str1= "Hello"; //is invalid.
```

3. Whenever a character array is declared to hold a string, allocate size for the null terminator character '\0' also in the character array

9.3.3.11 Functions Functions are the basic building blocks of modular programs. A function is a self-contained and re-usable code snippet intended to perform a particular task. 'Embedded *C*' supports two different types of functions namely, *library functions* and *user defined functions*.

Library functions are the built in functions which is either part of the standard 'Embedded C' library or user created library files. 'C' provides extensive built in library file support and the library files are categorised into various types like I/O library functions, string operation library functions, memory allocation library functions etc. *printf()*, *scanf ()*, etc. are examples of I/O library functions. *strcpy()*, *strcmp()*, etc. are examples for string operations library functions. *malloc()*, *calloc()* etc are examples of memory allocation library functions supported by 'C'. All library functions supported by a particular library is implemented and exported in the same. A corresponding header ('*.h*') file for the library file provides information about the various functions available to user in a library file. Users should include the header file corresponding to a particular library file for calling the functions from that library in the

'C' source file. For example, if a programmer wants to use the standard I/O library function *printf()* in the source file, the header file corresponding to the I/O library file ("*stdio.h*" meant for standard i/o) should be included in the 'c' source code using the #include preprocessor directive.

E.g.

```
#include <stdio.h>
void main (  )
{
    printf("Hello World");
}
```

"***string.h***" is the header file corresponding to the built in library functions for string operations and "***malloc.h***" is the header file for memory allocation library functions. Readers are requested to get info on header files for other required libraries from the standard ANSI library. As mentioned earlier, the standard 'C' library function implementation may be tailored by cross-compilers, keeping the function name and parameter list unchanged depending on Embedded 'C' application requirements. ***printf()*** function implemented by C51 cross compiler for *8051* microcontroller is a typical example. The library functions are standardised functions and they have a unique style of naming convention and arguments. Users should strictly follow it while using library functions.

User defined functions are programmer created functions for various reasons like modularity, easy understanding of code, code reusability, etc. The generic syntax for a function definition (implementation) is illustrated below.

```
Return type  function name (argument list)
{
    //Function body (Declarations & statements)
    //Return statement
}
```

Return type of a function tells—what is the data type of the value returning by the function on completion of its execution. The return type can be any of the data type supported by 'C' language, viz. *int*, *char*, *float*, *long*, etc. *void* is the return type for functions which do not return anything. Function name is the name by which a function is identified. For user defined functions users can give any name of their interest. For library functions the function name for doing certain operations is fixed and the user should use those standard function names. Parameters are the list of arguments to be passed to the function for processing. Parameters are the inputs to the functions. If a function accepts multiple parameters, each of them are separated using ',' in the argument list. Arguments to functions are passed by two means, namely, pass by value and pass by reference. Pass by value method passes the variables to the function using local copies whereas in pass by reference variables are passed to the function using pointers. In addition to the above-mentioned attributes, a function definition may optionally specify the function's linkage also. The linkage of the function can be either '*external*' or '*internal*'.

The task to be executed by the function is implemented within the body of the function. In certain situations, you may have a single source ('*.c*') file containing the entire variable list, user defined functions, function *main()*, etc. There are two methods for writing user defined functions if the source code is a single '*.c*' file. The first method is writing all user defined functions on top of the *main* function and other user defined functions calling the user defined function.

```
E.g.

Int xyz (int i)
{
……..;
}

void abc (void)
{
……..;
xyz (a)
}

void main ()
  {
     abc ();
  }
```

If you are writing the user defined functions after the entry function *main()* and calling the same inside *main*, you should specify the function prototype (function declaration) of each user defined functions before the function *main()*. Otherwise the compiler assumes that the user defined function is an extern function returning integer value and if you define the function after *main()* without using a function declaration/prototype, compiler will generate error complaining the user defined function is redefining (Compiler already assumed the function which is used inside *main()* without a function prototype as an extern function returning an integer value). The function declaration informs the compiler about the format and existence of a function prior to its use. Implicit declaration of functions is not allowed: every function must be explicitly declared before it is called. The general form of function declaration is

```
Linkage Type Return type  function name (arguments);
E.g. static int add(int a, int b);
```

The '*Linkage Type*' specifies the linkage for the function. It can be either '*external*' or '*internal*'. The '*static*' keyword for the '*Linkage Type*' specifies the linkage of the function as internal whereas the '*extern*' '*Linkage Type*' specifies 'external' linkage for the function. It is not mandatory to specify the name of the argument along with its type in the argument list of the function declaration. The declarations can simply specify the types of parameters in the argument list. This is called function *prototyping*. A function prototype consists of the function return type, the name of the function, and the parameter list. The usage is illustrated below.

```
static int add(int, int);
```

Let us have a look at the examples for the different scenarios explained above on user defined functions.

```
//#####################################################################
//Example for Source code with function prototype for user defined-
//functions. Source file test.c
//#####################################################################
```

```
#include <stdio.h>
//function prototype for user defined function test
void test (void);

       void main ( )
       {
       test ();       //Calling user defined function from main
       }
//Implementation of user defined function test after function main
void test (void)
{
printf ("Hello World!");
return;
}

//#####################################################################
//Example for Source code without function prototype for user defined
//functions. Source file test.c
//#####################################################################
#include <stdio.h>
//function prototype for user defined function test not declared
       void main ()
       {
       test ();       //Calling user defined function from main
       }

//Implementation of user defined function test after function main
       void test (void)
       {
       printf ("Hello World!");
       return;
       }
```

Compiler Output:

Compiling…
test.c
test.c(5): warning c4013: 'test' undefined; assuming extern returning int
test.c(9): error C2371: 'test': redefinition; different basic types Error executing cl.exe.
test.exe-1 error(s), 1 warning(s)

There is another convenient method for declaring variables and functions involved in a source file. This technique is a header ('*.h*') file and source ('*.c*') file based approach. In this approach, corresponding to each 'c' source file there will be a header file with same name (not necessarily) as that of the 'c' source file. All functions and global/extern variables for the source file are declared in the header file instead of declaring the same in the corresponding source file. Include the header file where the functions are declared to the source file using the "*#include*" pre-processor directive. Functions declared in a file can be either global (extern access) in scope or static in scope depending on the declaration of the

function. By default all functions are global in scope (accessible from outside the file where the function is declared). If you want to limit the scope (accessibility) of the function within the file where it is declared, use the keyword '*static*' before the return type in the function declaration.

9.3.3.12 Function Pointers A function pointer is a pointer variable pointing to a function. When an application is compiled, the functions which are part of the application are also get converted into corresponding processor/compiler specific codes. When the application is loaded in primary memory for execution, the code corresponding to the function is also loaded into the memory and it resides at a memory address provided by the application loader. The function name maps to an address where the first instruction (machine code) of the function is present. A function pointer points to this address.

The general form of declaration of a function pointer is given below.

```
return_type (*pointer_name) (argument list)
```

where, '*return_type*' represents the return type of the function, '*pointer_name*' represents the name of the pointer and 'argument list' represents the data type of the arguments of the function. If the function contains multiple arguments, the data types are separated using ','. Typical declarations of function pointer are given below.

```
//Function pointer to a function returning int and takes no parameter
int (*fptr)();
//Function pointer to a function returning int and takes 1 parameter
int (*fptr)(int)
```

The parentheses () around the function pointer variable differentiates it as a function pointer variable. If no parentheses are used, the declaration will look like

```
int *fptr();
```

The cross compiler interprets it as a function declaration for function with name '*fptr*' whose argument list is void and return value is a pointer to an integer. Now we have declared a function pointer, the next step is 'How to assign a function to a function pointer?'.

Let us assume that there is a function with signature

```
int function1(void);
```

and a function pointer with declaration

```
int (*fptr)();
```

We can assign the address of the function '*function1()*' to our function pointer variable '*fptr*' with the following assignment statement:

```
fptr = &function1;
```

The '*&*' operator gets the address of function '*function1*' and it is assigned to the pointer variable '*fptr*' with the assignment operator '='. The address of operator '*&*' is optional when the name of the function is used. Hence the assignment operation can be re-written as:

```
fptr = function1;
```

Once the address of the right sort of function is assigned to the function pointer, the function can be invoked by any one of the following methods.

```
(*fptr)();
fptr();
```

Function pointers can also be declared using *typedef*. Declaration and usage of a function pointer with *typedef* is illustrated below.

```
//Function pointer to a function returning int and takes no parameter
      typedef int (*funcptr)();
      funcptr fptr;
```

The following sample code illustrates the declaration, definition and usage of function pointer.

```
#include <stdio.h>
void square(int x);
void main()
{
      //Declare a function pointer
      void (*fptr)(int);
      //Define the function pointer to function square
      fptr = square;
      //Style 1: Invoke the function through function pointer
      fptr(2);
      //Style 2: Invoke the function through function pointer
      (*fptr)(2);
}
//#####################################################################
//Function for printing the square of a number
void square(int x)
{
      printf("Square of %d = %d\n", x, x*x);
}
```

Function pointer is a helpful feature in late binding. Based on the situational need in the application you can invoke the required function by binding the function with the right sort of function pointer (The function signature and function pointer signature should be matching). This reduces the usage of 'if' and 'switch – case' statements with function names. Function pointers are extremely useful for handling situations which demand passing of a function as argument to another function. Function pointers are often used within functions where the function should be able to work with a number of functions whose names are not known until the program is running. A typical example for this is callback functions, which requires the information about the function which needs to be called. The following sample piece of code illustrates the usage of function pointer as parameter to a function.

```
#include <stdio.h>
//#####################################################################
//Function prototype declaration
void square(int x);
void cube(int x);
void power(void (*fptr)(int), int x);
```

```
void main()
{
      //Declare a function pointer
      void (*fptr)(int);
      //Define the function pointer to function square
      fptr = square;
      //Invoke the function 'square' through function pointer
      power (fptr,2);
      //Define the function pointer to function cube
      fptr = cube;
      //Invoke the function 'cube' through function pointer
      power (fptr,2);
}
//#####################################################################
//Interface function for invoking functions through function pointer

void power(void (*fptr)(int), int x)
{
      fptr(x);

}
//#####################################################################
//Function for printing the square of a number

void square(int x)
{
      printf("Square of %d = %d\n", x, x*x);

}
//#####################################################################
//Function for printing the third power (cube) of a number
void cube(int x)
{
      printf("Cube of %d = %d\n", x, x*x*x);
}
```

Arrays of Function Pointers An array of function pointers holds pointers to functions with same type of signature. This offers the flexibility to select a function using an index. Arrays of function pointers can be defined using either direct function pointers or the '*typedef*' qualifier.

```
//Declare an array of pointers to functions, which returns int and
//takes no parameters, using direct function pointer declaration
int (*fnptrarr[5])();

//Declare and initialise an array of pointers to functions, which
//return int and takes no parameters, using direct function pointer-
//declaration
int (*fnptrarr[])()= {/*initialisation*/};
```

```
//Declare and initialize to 'NULL' an array of pointers to functions,
//which return int and takes no parameters, using direct function
//pointer declaration
int (*fnptrarr[5])()= {NULL};

//Declare an array of pointers to functions, which returns int and
//takes no parameters, using typedef function pointer declaration
typedef int (*fncptr)();
fncptr fnptrarr[5]();

//Declare and initialize an array of pointers to functions, which
//return int and takes no parameters, using typedef function pointer-
//declaration
typedef int (*fncptr)();
fncptr fnptrarr[]()= {/*initialisation*/};

//Declare and initialise to 'NULL' an array of pointers to functions,
//which return int and takes no parameters, using typedef function
//pointer declaration
typedef int (*fncptr)();
fncptr fnptrarr[5]()= {NULL};
```

The following piece of code illustrates the usage of function pointer arrays:

```
#include <stdio.h>
//#####################################################################
//Function prototype definition
void square(int x);
void cube(int x);

void main()
{
      //Declare a function pointer array of size 2 and initialize
      void (*fptr[2])(int)= {NULL};
      //Define the function pointer 0 to function square
      fptr[0] = square;
      //Invoke the function square
      fptr[0](2);
      //Define the function pointer 1 to function cube
      fptr[1] = cube;
      //Invoke the function cube through function pointer
      fptr[1](2);
}
//#####################################################################
//Function for printing the square of a number
void square(int x)
{
      printf("Square of %d = %d\n", x, x*x);
}
```

```
//#################################################################
//Function for printing the third power (cube) of a number

void cube(int x)
{
      printf("Cube of %d = %d\n", x, x*x*x);
}
```

9.3.3.13 Structures and Unions *'structure'* is a variable holding a collection of data types (int, float, char, long etc). The data types can be either unique or distinct. The tag '*struct*' is used for declaring a structure. The general form of a structure declaration is given below.

```
struct   struct_name
         {
                  //variable 1 declaration
                  //variable 2 to declaration
                  //.....
                  //variable n declaration
         };
```

struct_name is the name of the structure and the choice of the name is left to the programmer.

Let us examine the details kept in an employee record for an employee in the employee database of an organisation as example. A typical employee record contains information like '*Employee Name*', '*Employee Code*', and '*Date of Birth*'. This information can be represented in 'C' using a structure as given below.


```
struct  employee
        {
        char emp_name [20]; // Allowed maximum length for name = 20
        int emp_code;
        char DOB [10]; // DD-MM-YYYY Format (10 character)
        };
```

Hence in 'C', the employee record is represented by two string variables (character arrays) and an integer variable. Since these three variables are relevant to each employee, they are grouped together in the form of a structure. Literally structure can be viewed as a collection of related data. The above structure declaration does not allocate memory for the variables declared inside the structure. It's a mere representation of the data inside a structure. To allocate memory for the structure we need to create a variable of the structure. The structure variable creation is illustrated below.

```
struct employee emp1;
```

Keyword '*struct*' informs the compiler that the variable '*emp1*' is a structure of type '*employee*'. The name of the structure is referred as '*structure tag*' and the variables declared inside the structure are called '*structure elements*'. The definition of the structure variable only allocates the memory for the different elements and will not initialise the members. The members need to be initialised explicitly. Members of the structure variable can be initialised altogether at the time of declaration of the structure variable itself or can be initialised (modified) independently using the '.' operator (member operator).

```
struct employee emp1= {"SHIBU K V", 42170, "11-11-1977"};
```

This statement declares a structure variable with name 'emp1' of type employee and each elements of the structure is initialised as

```
emp_name = "SHIBU K V"
emp_code = 42170
DOB= "11-11-1977"
```

It should be noted that the variables should be initialised in the order as per their declaration in the structure variable. The selective method of initialisation/modification is used for initialising /modifying each element independently.

```
E.g.    struct employee emp1;
        emp1.emp_code = 42170;
```

All members of a structure, except character string variables can be assigned values using '.' Operator and assignment ('=') operator (character strings are character arrays and character arrays cannot be initialised altogether using the assignment operator). Character string variables can be assigned using string copy function (*strcpy*).

```
strcpy (emp1.emp_name,"SHIBU K V");
strcpy (emp1.DOB,"11-11-1977");
```

Declaration of a structure variable requires the keyword '*structure*' as the first word and it may sound awkward from a programmer point of view. This can be eliminated by using '*typedef*' in defining the structure. The above 'employee' structure example can be re-written using typedef as

```
typedef struct
      {
      char emp_name [20];// Allowed maximum length for name = 20
      int emp_code;
      char DOB [10];     // DD-MM-YYYY Format (10 character)
      } employee;
employee emp1; //No need to add struct before employee
```

This approach eliminates the need for adding the keyword '*struct*' each time a structure variable is declared.

Structure operations The following table lists the various operations supported by structures

Operator	Operation	Example

= (Assignment)	Assigns the values of one structure to another structure of same type	employee emp1,emp2;
		emp1.emp_code = 42170;
		strcpy(emp1.emp_name,"SHIBU");
		strcpy(emp1.DOB,"11/11/1977");
		emp2=emp1;

== (Checking the equality of all members of two structures)	Compare individual members of two structures of same type for equality. Return 1 if all members are identical in both structures else return 0	employee emp1,emp2;
		emp1.emp_code = 42170;
		strcpy(emp1.emp_name,"SHIBU");
		strcpy(emp1.DOB,"11/11/1977");
		if (emp2==emp1)

!= (Checking the equality of all members of two structures)	Compare individual members of two structures of same type for non equality. Return 1 if all members are not identical in both structures else return 0	employee emp1,emp2;
		emp1.emp_code = 42170;
		strcpy(emp1.emp_name,"SHIBU");
		strcpy(emp1.DOB,"11/11/1977");
		if (emp2!=emp1)

sizeof()	Returns the size of the structure (memory allocated for the structure variable in bytes)	employee emp1; sizeof (emp1);

Note:
1. The assignment and comparison operation is valid only if both structure variables are of the same type
2. Some compilers may not support the direct assignment and comparison operation. In such situation the individual members of structure should be assigned or compared separately.

Structure pointers Structure pointers are pointers to structures. It is easy to modify the memory held by structure variables if pointers are used. Functions with structure variable as parameter is a very good example for it. The structure variable can be passed to the function by two methods; either as a copy of the structure variable (pass by value) or as a pointer to a structure variable (pass by reference/address). Pass by value method is slower in operation compared to pass by pointers since the execution time for copying the structure parameter also needs to be accounted. Pass by pointers is also helpful if the structure which is given as input parameter to a function need to be modified and returned on execution of the calling function. Structure pointers can be declared by prefixing the structure variable name with '*'. The following structure pointer declaration illustrates the same.

```
struct employee *emp1;    //structure defined using the structure tag
employee *emp1;           //structure defined with typedef structure
```

For structure variables declared as pointers to a structure, the member selection of the structure is performed using the '→' operator.

```
E.g.    struct employee *emp1, emp2;
        emp1 = &emp2; //Obtain a pointer
        emp1→ emp_code = 42170;
        strcpy (emp1→DOB, "11-11-1977");
```

Structure pointers at absolute address Most of the time structures are used in Embedded C applications for representing the memory locations or registers of chip whose memory address is fixed at the time of hardware designing. Typical example is a Real Time Clock (RTC) which is memory mapped at a specific address and the memory address of the registers of the RTC is also fixed. If we use a structure to define the different registers of the RTC, the structure should be placed at an absolute address corresponding to the memory mapped address of the first register given as the member variable of the structure. Consider the structure describing RTC registers.

```
typedef struct
{
//RTC Control register (8bit) memory mapped at 0x4000
unsigned char control;
//RTC Seconds register (8bit) memory mapped at 0x4001
unsigned char seconds;
```

```
//RTC Minutes register (8bit) memory mapped at 0x4002
unsigned char minutes;
//RTC Hours register (8bit) memory mapped at 0x4003
unsigned char hours;
//RTC Day of week register (8bit) memory mapped at 0x4004
unsigned char day;
//RTC Date register (8bit) memory mapped at 0x4005
unsigned char date;
//RTC Month register (8bit) memory mapped at 0x4006
unsigned char month;
//RTC Year register (8bit) memory mapped at 0x4007
unsigned char year;
} RTC;
```

To read and write to these hardware registers using structure member variable manipulation, we need to place the RTC structure variable at the absolute address 0x4000, which is the address of the RTC hardware register, represented by the first member of structure RTC.

The implementation of structures using pointers to absolute memory location is cross-compiler dependent. Desktop applications may not give permission to explicitly place structures or pointers at an absolute address since we are not sure about the address allocated to general purpose RAM by the linker. Any attempt to explicitly allocate a structure at an absolute address may result in *access violation exception*. More over there is no need for a general purpose application to explicitly place structure or pointers at an absolute address, whereas embedded systems most often requires it if different hardware registers are memory mapped to the processor/controller memory map. The C51 cross compiler for *8051* microcontroller supports a specific Embedded C keyword '*xdata*' for external memory access and the structure absolute memory placement can be done using the following method.

```
RTC xdata *rtc_registers = (void xdata *) 0x4000;
```

Structure Arrays In the above employee record example, three important data is associated with each employee, namely; employee name (emp_name), employee code (emp_code) and Date of Birth (DOB). This information is held together as a structure for associating the same with an employee. Suppose the organisation contains 100 employees then we need 100 such structures to hold the data related to the 100 employees. This is achieved by array of structures. For the above employee structure example a structure array with 100 structure variables can be declared as

```
struct employee emp [100]; //structure declared using struct keyword
      or
employee emp [100]; //structure declared using typedef struct
```

emp [0] holds the structure variable data for the first employee and emp [99] holds the structure variable data corresponding to the 100th employee. The variables corresponding to each structure in an array can be initialised altogether at the time of defining the structure array or can be initialised/modified using the corresponding array subscript for the structure and the '.' Operator as explained below.

```
typedef struct
    {
     char emp_name [20]; // Allowed maximum length for name = 20
     int emp_code;
```

```
    char DOB [10]; // DD-MM-YYYY Format (10 character)
    } employee;

//Initialisation at the time of defining variable
employee emp [3] = {{"JOHN", 1, "01-01-1988"}, {"ALEX", 2, "10-01-1976"},
{"SMITH", 3, "07-05-1985"}};
//Selective initialization
emp [0].emp_code = 1;
strcpy (emp [0].emp_name, "JOHN");
strcpy (emp [0].DOB, "01-01-1988");
```

Structure in Embedded 'C' programming Simple structure variables and array of structures are widely used in 'embedded 'C' based firmware development. Structures and structure arrays are used for holding various configuration data, exchanging information between Interrupt Service Routine (ISR) and application etc. A typical example for the structure usage is for holding the various register configuration data for setting up a particular baudrate for serial communication. An array of such configuration data holding structures can be used for setting different baudrates according to user needs. It is interesting to note that more than 90% of the embedded C programmers use '*typedef*' to define the structures with meaningful names.

Structure padding (Packed structure) Structure variables are always stored in the memory of the target system with structure member variables in the same order as they are declared in the structure definition. But it is not necessary that the variables should be placed in continuous physical memory locations. The choice on this is left to the compiler/cross-compiler. For multi byte processors (processors with word length greater than 1 byte (8 bits)), if the structure elements are arranged in memory in such a way that they can be accessed with lesser number of memory fetches, it definitely speeds up the operation. The process of arranging the structure elements in memory in a way facilitating increased execution speed is called *structure padding*. As an example consider the following structure:

```
typedef struct
        {
        char x;
        int y;
        } exmpl;
```

Let us assume that the storage space for '***int***' is 4 bytes (32 bits) and for '***char***' it is 1 byte (8 bits) for the target embedded system under consideration. Suppose the target processor for embedded application, where the above structure is making use is with a 4 byte (32 bit) data bus and the memory is byte accessible. Every memory fetch operation retrieves four bytes from the memory locations $4x$, $4x + 1$, $4x + 2$ and $4x + 3$, where $x = 0, 1, 2$, etc. for successive memory read operations. Hence a 4 byte (32 bit) variable can be fully retrieved in a single memory fetch if it is stored at memory locations with starting address $4x$ ($x = 0, 1, 2$, etc.). If it is stored at any other memory location, two memory fetches are required to retrieve the variable and hence the speed of operation is reduced by a factor of 2.

Let us analyse the various possibilities of storing the above structure within the memory.

Method-1 member variables of structure stored in consecutive data memory locations.

In this model the member variables are stored in consecutive data memory locations (***Note:*** the member variable storage need not start at the address mentioned here, the address given here is only

Memory Address	4x + 3	4x + 2	4x + 1	4x
Data	Byte 2 of exmpl.y	Byte 1 of exmpl.y	Byte 0 of exmpl.y	exmpl.x
Data				Byte 3 of exmpl.y
Memory Address	4(x + 1) + 3	4(x + 1) + 2	4(x + 1) + 1	4(x + 1)

Fig. 9.8 **Memory representation for structure without padding**

for illustration) and if we want to access the character variable exmp1.x of structure exmp1, it can be achieved with a single memory fetch. But accessing the integer member variable exmp1.y requires two memory fetches.

Method-2 member variables of structure stored in data memory with padding

In this approach, a structure variable with storage size same as that of the word length (or an integer multiple of word length) of the processor is always placed at memory locations with starting address as multiple of the word length so that the variable can be retrieved with a single data memory fetch. The memory locations coming in between the first variable and the second variable of the structure are filled with extra bytes by the compiler and these bytes are called 'padding bytes' (Fig. 9.9). The structure padding technique is solely dependent on the cross-compiler in use. You can turn ON or OFF the padding of structure by using the cross-compiler supported padding settings. Structure padding is applicable only for processors with word size more than 8bit (1 byte). It is not applicable to processors/controllers with 8bit bus architecture.

Memory Address	4x + 3	4x + 2	4x + 1	4x
Data	Padding	Padding	Padding	exmpl.x
Data	Byte 3 of exmpl.y	Byte 2 of exmpl.y	Byte 1 of exmpl.y	Byte 0 of exmpl.y
Memory Address	4(x + 1) + 3	4(x + 1) + 2	4(x + 1) + 1	4(x + 1)

Fig. 9.9 **Memory representation for structure with padding**

Structure and Bit fields Bit field is a useful feature supported by structures for bit manipulation operation and flag settings in embedded applications. Most of the processors/controllers used in embedded application development provides extensive Boolean (bit) operation support and a similar support in the firmware environment can directly be used to explore the Boolean processing capability of the target device.

For most of the application development scenarios we use integer and character to hold data items even though the variable is expected to vary in a range far below the upper limit of the data type used for holding the variable. A typical example is flags. Flags are used for indicating a 'TRUE' or 'FALSE' condition. Practically this can be done using a single bit and the two states of the bit (1 and 0) can be used for representing 'TRUE' or 'FALSE' condition. Unfortunately 'C' does not support a built in data type for holding a single bit and it forces us to use the minimum sized data type (quite often char (8bit data type) and short int) for representing the flag. This will definitely lead to the wastage of memory. Since memory is a big constraint in embedded applications we cannot tolerate the memory wastage. 'C' indirectly supports bit data types through '***Bit fields***' of structures. Structure bit fields can be directly accessed and manipulated. A set of bits in the structure bit field forms a '*char*' or '*int*' variable. The general format of declaration of bit fields within structure is illustrated below.

```
struct  struct_name
        {
        data type (char or int) bit_var 1_name : bit_size,
        bit_var 2_name : bit_size,
        ..........:,
        ..........:,
        bit_var n_name : bit_size;
        } ;
```

'*struct_name*' represents the name of the bit field structure. '*data type*' is the data type which will be formed by packing several bits. Only character (char) and integer (int/short int) data types are allowed in bit field structures in Embedded C applications. Some compilers may not support the 'char' data type. However our illustrative cross-compiler C51 supports 'char' data type as data type. '*bit_var 1_name*' denotes the bit variable and '*bit_size*' gives the number of bits required by the variable '*bit_var 1_name'* and so. The operator ':' associates the number of bits required with the bit variable. Bit variables that are packed to form a data type should be separated inside the structure using ',' operator. A real picture of bit fields in structures for embedded applications can be provided by the Program Status Word (PSW) register representation of *8051* controller. The PSW register of 8051 is bit addressable and its bit level representation is given below.

PSW.7	PSW.6	PSW.5	PSW.4	PSW.3	PSW.2	PSW.1	PSW.0
CY	AC	F0	RS1	RS0	OV		P

Using structure and bit fields the same can be represented as

```
struct PSW
        {
                char    P:1,    /* Bit 0 of PSW  : Parity Flag */
                        :1,     /* Bit 1 of PSW  : Unused */
                        OV:1,   /* Bit 2 of PSW  : Overflow Flag */
                        RS0:1,  /* Bit 3 of PSW  : Register Bank Select 0 bit */
                        RS1 :1, /* Bit 4 of PSW  : Register Bank Select 1 bit */
                        F0:1,   /* Bit 5  of PSW : User definable Flag */
                        AC :1,  /* Bit 6 of PSW  : Auxiliary Carry Flag */
                        C:1;    /* Bit 7 of PSW  : Carry  Flag */
```

```
    /*Note that the operator ';' is used after the last bit field to
    indicate end of bit field*/

};
```

In the above structure declaration, each bit field variable is defined with a name and an associated bit size representation. If some of the bits are unused in a packed fashion, the same can be skipped by merely giving the number of bytes to be skipped without giving a name for that bit variables. In the above example Bit 1 of PSW is skipped since it is an unused bit in the PSW register.

It should be noted that the total number of bits used by the bit field variables defined for a specific data type should not exceed the maximum number of allocated bits for that specific data type. In the above example the bit field data type is '*char*' (8 bits) and 7 bit field variables each of size 1 are declared and one bit variable is declared as unused bit. Hence the total number of bits consumed by all the bit variables including the non declared bit variable sums to 8, which is same as the bit size for the data type '*char*'. The internal representation of structure bit fields depends on the size supported by the cross-compiler data type (char/int) and the ordering of bits in memory. Some processors/controllers store the bits from left to right while others store from right to left (Here comes the significance of endianness).

Unions Union is a concept derived from structure and *union* declarations follow the same syntax as that of structures (*structure* tag is replaced by *union* tag). Though *union* looks similar to structure in declaration, it differs from structure in the memory allocation technique for the member variables. Whenever a *union* variable is created, memory is allocated only to the member variable of *union* requiring the maximum storage size. But for structures, memory is allocated to each member variables. This helps in efficient data memory usage. Even if a *union* variable contains different member variables of different data types, existence is only for a single variable at a time. The size of a *union* returns the storage size of its member variable occupying the maximum storage size. The syntax for declaring *union* is given below

```
union   union_name
        {
                //variable 1 declaration
                //variable 2 to declaration
                //.....
                //variable n declaration
        };
```

or

```
typedef union
        {
        //variable 1 declaration
        //variable 2 to declaration
        //.....
        //variable n declaration
        } union_name;
```

'*union_name*' is the name of the *union* and programmers can use any name according to their programming style. As an illustrative example let's declare a *union* variable consisting of an integer member variable and a character member variable.

```
typedef union
        {
        int y;      //Integer variable
        char z;     //Character variable
        } example;
example ex1;
```

Assuming the storage location required for '*int*' as 4 bytes and for '*char*' as 1 byte, the memory allocated to the *union* variable ex1 will be as shown below

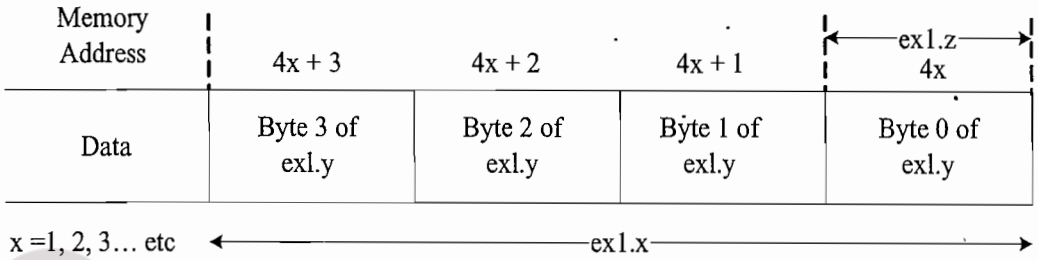

Fig. 9.10 **Memory representation for union**

Note: The start address is chosen arbitrarily for illustration, it can be any data memory. It is obvious from the figure that by using *union* the same physical address can be accessed with different data type references. Hence *union* is a convenient way of '*variant access*'.

In Embedded C applications, *union* may be used for fast accessing of individual bytes of 'long' or 'int' variables, eliminating the need for masking the other bytes of 'long' or 'int' variables which are of no interest, for checking some conditions. Typical example is extracting the individual bytes of 16 or 32 bit counters.

A few important points on Structure and Union

1. The *offsetof()* macro returns the offset of a member variable (in bytes) from the beginning of its parent structure. The usage is *offsetof (structName, memberName);* where '*structName*' is the name of the parent structure and '*memberName*' is the name of the member in the parent data structure whose offset is to be determined. For using this macro use the header file '*stddef.h*'
2. If you declare a *structure* just before the *main ()* function in your source file, ensure that the structure is terminated with the structure definition termination indicator ';'. Otherwise function *main ()* will be treated as a structure and the application may crash with exceptions.
3. A *union* variable can be initialised at the time of creation with the first member variable value only.

9.3.3.14 Pre-processors and Macros Pre-processor in 'C' is compiler/cross-compiler directives used by compiler/cross-compiler to filter the source code before compilation/cross-compilation. The pre-processor directives are mere directions to the compilers/cross compilers on how the source file should be compiled/cross compiled. No executable code is generated for pre-processor directives on compilation. They are same as pseudo ops in assembly language. Pre-processors are very useful for selecting target processor/controller dependent pieces of code for different target systems and allow a single source code to be compiled and run on several different target boards. The syntax for pre-processor directives is different from the syntax of 'C' language. Each pre-processor directive starts with the

'#' symbol and ends without a semicolon (;). Pre-processor directives are normally placed before the entry point function *main()* in a source file. Pre-processor directives are grouped into three categories; namely

1. File inclusion pre-processor directives
2. Compile control pre-processor directives
3. Macro substitution pre-processor directives

File inclusion pre processor directives The file inclusion pre processor directives include external files containing macro definitions, function declarations, constant definitions etc to the current source file. '***#include***' is the pre processor directive used for file inclusion. '*#include*' pre processor instruction reads the entire contents of the file specified by the '*#include*' directive and inserts it to the current source file at a location where the '*#include*' statement is invoked. This is generally used for reading header files for library functions and user defined header files or source files. Header files contain details of functions and types used within the library and user created files. They must be included before the program uses the library functions and other user defined functions. Library header file names are always enclosed in angle brackets, < >. This instructs the pre-processor to look the standard include directory for the header file, which needs to be inserted to the current source file.

e.g. *#include <stdio.h>* is the directive to insert the standard library file '*stdio.h*'. This file is available in the standard include directory of the development environment. (Normally inside the folder '*inc*'). If the file to be included is given in double quotes (" "), the pre-processor searches the file first in the current directory where the current source code file is present and if not found will search the standard include directory. Usually user defined files kept in the project folder are included using this technique. E.g. *#include "constants.h"* where 'constants' is a user defined header file with name *constants.h*, kept in the local project folder. An include file can include another include file in it (Nesting of include files). An include file is not allowed to include the same file itself. Source files (.c) file can also be used as include files.

Compile control pre-processor directives Compile control pre-processor directives are used for controlling the compilation process such as skipping compilation of a portion of code, adding debug features, etc. The conditional control pre-processor directives are similar to the conditional statement if else in 'C'. *#ifdef*, *#ifndef*, *#else*, *#endif*, *#undef*, etc are the compile control pre-processor directives.

#ifdef uses a name as argument and returns true if the argument (name) is already defined. *#define* is used for defining the argument (name).
#else is used for two way branching when combined with *#ifdef* (same as *if else* in 'C').
#endif is used for indicating the end of a block following *#ifdef* or *#else*

Usage of *#ifdef*, *#else* and *#endif* is given below.

```
#ifdef

#else  (optional)

#endif
```

The pre-processor directive *#ifndef* is complementary to *#ifdef*. It is used for checking whether an argument (e.g. macro) is not defined. Pre-processor directive *#undef* is used for disabling the definition of the argument or macro if it is defined. It is complementary to *#define*. Pre-processor directives are a powerful option in source code debugging. If you want to ensure the execution of a code block, for

debug purpose you can define a debug variable and define it. Within the code wherever you want to ensure the execution, use the *#ifdef* and *#endif* pre-processors. The argument to *#ifdef* should be the debug variable. Insert a *printf()* function within the *#ifdef #endif* block. If no debugging is required comment or remove the definition of debug variable.

E.g.

```
#define DEBUG 1
//Inside code block
#ifdef DEBUG
printf ("Debug Enabled");
#endif
```

The #error pre-processor directive The *#error* pre-processor generates error message in case of an error and stops the compilation on accounting an error condition. The syntax of *#error* directive is given below

```
#error  error message
```

Macro substitution pre-processor directives *Macros* are a means of creating portable inline code. '*Inline*' means wherever the macro is called in a source file it is replaced directly with the code defined by the macro. In-line code improves the performance in terms of execution speed. Macros are similar to subroutines in functioning but they differ in the way in which they are coded in the source code. Functions are normally written only once with arguments. Whenever a function needs to be executed, a call to the function code is made with the required parameters at the point where it needs to be invoked. If a macro is used instead of functions, the compiler inserts the code for macro wherever it is called. From a code size viewpoint macros are non-optimised compared to functions. Macros are generally used for coding small functions requiring execution without latency. The '*#define*' pre-processor directive is used for coding macros. Macros can be either simple definitions or functions with arguments.

#define PI 3.1415 is an example for a simple macro definition.

Macro definition can contain arithmetic operations also. Proper care should be taken in giving right syntax while defining macros, keeping an eye on their usage in the source code. Consider the following example

```
#define A 12+25
#define B 45-10
```

Suppose the source contains a statement *multiplier = A*B;* the pre-processor directly replaces the macros A and B and the statement becomes

```
multiplier = 12+25*45-10;
```

Due to the operator precedence criteria, this won't give the expected result. Expected result can be obtained by a simple re-writing of the macro with necessary parentheses as illustrated below.

```
#define A (12+25)
#define B (45-10)
```

Proper care should be given to parentheses the macro arguments. As mentioned earlier macros can also be defined with arguments. Consider the following example

```
#define CIRCLE_AREA(a) (3.14 * a*a)
```

This defines a macro for calculating the area of a circle. It takes an argument and returns the area. It should be noted that there is no space between the name of the macro (macro identifier) and the left bracket parenthesis.

Suppose the source code contains a statement like *area=CIRCLE_AREA(5);* it will be replaced as

```
area = (3.14*5*5);
```

Suppose the call is like this, *area=CIRCLE_AREA(2+5);* the pre-processor will translate the same as

```
area = (3.14*2+5*2+5);
```

Will it produce the expected result? Obviously no. This shortcoming in macro definition can be eliminated by using parenthesis to each occurrence of the argument. Hence the ideal solution will be;

```
#define CIRCLE_AREA(a) (3.14 *(a)*(a))
```

9.3.3.15 Constant Declarations in Embedded 'C' In embedded applications the qualifier (keyword) '*const*' represents a 'Read only' variable. Use of the keyword '*const*' in front of a variable declares that the value of the variable cannot be altered by the program. This is a kind of defensive programming in which any attempt to modify a variable declared as '*const*' is reported as an access violation by the cross-compiler. The different types of constant variables used in embedded 'C' application are explained below.

Constant data Constant data informs that the data held by a variable is always constant and cannot be modified by the application. Constants used as scaling variables, ratio factors, various scientific computing constants (e.g. Plank's constant), etc. are represented as constant data. The general form of declaring a constant variable is given below.

```
const data type variable name;
```

or

```
data type const variable name;
```

'*const*' is the keyword informing compiler/cross compiler that the variable is constant. '*data type*' gives the data type of the variable. It can be 'int', 'char', 'float', etc. '*variable name*' is the constant variable.

```
E.g.  const float PI = 3.1417;
      float const PI = 3.1417;
```

Both of these statements declare PI as floating point constant and assign the value 3.1417 to it. Constant variable can also be defined using the *#define* pre-processor directive as given below.

```
#define PI 3.1417
/*No assignment using = operator and no ';' at end*/
```

The difference between both approaches is that the first one defines a constant of a particular data type (int, char, float, etc.) whereas in the second method the constant is represented using a mere symbol (text) and there is no implicit declaration about the data type of the constant. Both approaches are used in declaring constants in embedded C applications. The choice is left to the programmer.

Pointer to constant data Pointer to constant data is a pointer which points to a data which is read only. The pointer pointing to the data can be changed but the data is non-modifiable. Example of pointer to constant data

```
const int* x;    //Integer pointer x to constant data
int const* x     //Same meaning as above definition
```

Constant pointer to data Constant pointer has significant importance in Embedded C applications. An embedded system under consideration may have various external chips like, data memory, Real Time Clock (RTC), etc interfaced to the target processor/controller, using memory mapped technique. The range of address assigned to each chips and their registers are dependent on the hardware design. At the time of designing the hardware itself, address ranges are allocated to the different chips and the target hardware is developed according to these address allocations. For example, assume we have an RTC chip which is memory mapped at address range 0x3000 to 0x3010 and the memory mapped address of register holding the time information is at 0x3007. This memory address is fixed and to get the time information we have to look at the register residing at location 0x3007. But the content of the register located at address 0x3007 is subject to change according to the change in time. We can access this data using a constant pointer. The declaration of a constant pointer is given below.

```
// Constant character pointer x to constant/variable data
char *const x;

/*Explicit declaration of character pointer pointing to 8bit memory location,
mapped at location 0x3007; RTC example illustrated above*/

char *const x= (char*) 0x3007;
```

Constant pointer to constant data Constant pointers pointing to constant data are widely used in embedded programming applications. Typical uses are reading configuration data held at ROM chips which are memory mapped at a specified address range, Read only status registers of different chips, memory mapped at a fixed address. Syntax of declaring a constant pointer to constant data is given below.

```
/*Constant character pointer x pointing to constant data*/
const char *const x;
char const* const x;    //Equivalent to above declaration

/*Explicit declaration of constant character pointer* pointing to constant
data/
char const* const x = (char*) 0x3007;
```

9.3.3.16 The 'Volatile' Type Qualifier in Embedded 'C' The keyword '*volatile*' prefixed with any variable as qualifier informs the cross-compiler that the value of the variable is subject to change at any point of time (subject to asynchronous modification) other than the current statement of code where the control is at present.

Examples of variables which are subject to asynchronous modifications are

1. Variables common to Interrupt Service Routines (ISR) and other functions of a file
2. Memory mapped hardware registers

3. Variables shared by different threads in a multi threaded application (Not applicable to Super loop Firmware development approach)

The '***volatile***' keyword informs the cross-compiler that the variable with '*volatile*' qualifier is subject to asynchronous change and there by the cross compiler turns off any optimisation (assumptions on the variable) for these variables. The general form of declaring a volatile variable is given below.

```
volatile data type variable name; or
data type volatile variable name;
```

'data type' refers to the data type of the variable. It can be *int*, *char*, *float,* etc. 'variable name' is the user defined name for the *volatile* variable of the specified type.

```
E.g.    volatile unsigned char x;
        unsigned char volatile x;
```

What is the catch in using 'volatile' variable? Let's examine the following code snippet.

```
//Declare a memory mapped register
char* status_reg = (char*) 0x3000;

while (*status_reg!=0x01);    //Wait till status_reg = 0x01
```

On cross-compiling the code snippet, the cross-compiler converts the code to a read operation from the memory location mapped at address 0x3000 and it will assume that there is no point where the variable is going to modify (sort of over smartness) and may keep the data in a register to speed up the execution. The actual intention of the programmer, with the above code snippet is to read a memory mapped hardware status register and halt the execution of the rest of the code till the status register shows a ready status. Unfortunately the program will not produce the expected result due to the oversmartness of the cross-compiler in optimising the code for execution speed. Re-writing the code as given below serves the intended purpose.

```
//Declares volatile variable
volatile char *status_reg = (char *) 0x3000;

while (*status_reg!=0x01);    //Wait till status_reg = 0x01
```

In embedded applications all memory mapped device registers which are subject to asynchronous modifications (e.g. status, control and general purpose registers of memory mapped external devices) should be declared with '*volatile*' keyword to inform the cross-compiler that the variables representing these registers/locations are subject to asynchronous changes and do not optimise them. Another area which needs utmost care in embedded applications is variables shared between ISR and functions (variables which can be modified by both Interrupt Sub Routines and functions). These include structure variable, union variable and other ordinary variables. To avoid unexpected behaviour of the application, always declare such variables using '*volatile*' keyword.

The 'constant volatile' Variable Some variables used in embedded applications can be both '*constant*' and '*volatile*'. A '***Read only***' status register of a memory mapped device is a typical example for this. From a user point of view the '***Read only***' status registers can only be read but cannot modify. Hence it is a constant variable. From the device point the contents can be modified at any time by the device. So it is a volatile variable. Typical declarations are given ahead.

```
volatile const int a;      // Constant volatile integer
volatile const int* a;     //Pointer to a Constant volatile integer
```

Volatile pointer Volatile pointers is subject to change at any point after they are initialised. Typical examples are pointer to arrays or buffers modifiable by Interrupt Service Routines and pointers in dynamic memory allocation. Pointers used in dynamic memory allocation can be modified by the *realloc()* function. The general form of declaration of a volatile pointer to a non-volatile variable is given below.

```
data type* volatile variable name;
e.g.   unsigned char* volatile a;
```

Volatile pointer to a volatile variable is declared with the following syntax.

```
       data type volatile* volatile variable name;
e.g.   unsigned char volatile* volatile a;
```

9.3.3.17 Delay Generation and Infinite Loops in Embedded C Almost every embedded application involves delay programming. Embedded applications employ delay programming for waiting for a fixed time interval till a device is ready, for inserting delay between displays updating to give the user sufficient time to view the contents displayed, delays involved in bit transmission and reception in asynchronous serial transmissions like I2C, 1-Wire data transfer, delay for key de-bouncing etc. Some delay requirements in embedded application may be critical, meaning delay accuracy should be within a very narrow tolerance band. Typical example is delay used in bit data transmission. If the delay employed is not accurate, the bits may lost while transmission or reception. Certain delay requirements in embedded application may not be much critical, e.g. display updating delay.

It is easy to code delays in desktop applications under DOS or Windows operating systems. The library function *delay()* in DOS and *Sleep ()* in Windows provides delays in milliseconds with reasonable accuracy. Coding delay routines in embedded applications is bit difficult. The major reason is delay is dependent on target system's clock frequency. So we need to have a trial and error approach to code delays demanding reasonably good accuracy. Refer to the code snippet given for 'Performance Analyser' in *Chapter 14* for getting a handle on how to code delay routine in embedded applications using IDEs. Delay codes are generally non-portable. Delay routine requires a complete re-work if the target clock frequency is changed. Normally '*for loops*' are used for coding delays. Infinite loops are created using various loop control instructions like while (), do while (), for and goto labels. The super loop created by while (1) instruction in a traditional super loop based embedded firmware design is a typical example for infinite loop in embedded application development.

Infinite loop using while The following code snippet illustrates '*while*' for infinite loop implementation.

```
while (1)
{
}
```

Infinite loop using do while

```
do
{
} while (1);
```

Infinite loop using for

```
for (; ; ;)
{
}
```

Infinite loop using goto '*goto*' when combined with a '*label*' can create infinite loops.

```
label: //Task to be repeated
       //.....
       //.....
       goto label;
```

Which technique is the best? According to all experienced Embedded 'C' programmers *while()* loop is the best choice for creating infinite loops. There is no technical reason for this. The clean syntax of while loop entitles it for the same. The syntax of *for* loop for infinite loop is little puzzling and it is not capable of conveying its intended use. '*goto*' is the favorite choice of programmers migrating from Assembly to Embedded C ☺.

break; statement is used for coming out of an infinite loop. You may think why we implement an infinite loop and then quitting it? Answer – There may be instructions checking some condition inside the infinite loop. If the condition is met the program control may have to transfer to some other location.

9.3.3.18 Bit Manipulation Operations Though Embedded 'C' does not support a built in Boolean variable (Bit variable) for holding a 'TRUE (Logic 1)' or 'FALSE (Logic 0)' condition, it provides extensive support for Bit manipulation operations. Boolean variables required in embedded application are quite often stored as variables with least storage memory requirement (obviously *char* variable). Indeed it is wastage of memory if the application contains large number of Boolean variables and each variable is stored as a *char* variable. Only one bit (Least Significant bit) in a *char* variable is used for storing Boolean information. Rest 7 bits are left unused. This will definitely lead to serious memory bottle neck. Considerable amount of memory can be saved if different Boolean variables in an application are packed into a single variable in 'C' which requires less memory storage bytes. A character variable can accommodate 8 Boolean variables. If the Boolean variables are packed for saving memory, depending upon the program requirement each variable may have to be extracted and some manipulation (setting, clearing, inverting, etc.) needs to be performed on the bits. The following Bit manipulation operations are employed for the same.

Bitwise AND Operator '&' performs Bitwise AND operations. Please note that the Bitwise AND operator '&' is entirely different from the Logical AND operator '&&'. The '&' operator acts on individual bits of the operands. Bitwise AND operations are usually performed for selective clearing of bits and testing the present state of a bit (Bitwise ANDing with '1').

Bitwise OR Operator '|' performs Bitwise OR operations. Logical OR operator '||' is in no way related to the Bitwise OR operator '|'. Bitwise OR operation is performed on individual bits of the operands. Bitwise OR operation is usually performed for selectively setting of bits and testing the current state of a bit (Bitwise ORing with '0').

Bitwise Exclusive OR- XOR Bitwise XOR operator '^' acts on individual operand bits and performs an 'Excusive OR' operation on the bits. Bitwise XOR operation is used for toggling bits in embedded applications.

Bitwise NOT Bitwise NOT operations negates (inverts) the state of a bit. The operator '~' (tilde) is used as the Bitwise NOT operator in C.

Setting and Clearing and Bits Setting the value of a bit to '1' is achieved by a Bitwise OR operation. For example consider a character variable (8bit variable) flag. The following instruction sets its 0th bit always 1.

```
flag = flag | 1;
```

Brief explanation about the above operation is given below.
Using 8 bits, 1 is represented as 00000001. Upon a Bitwise OR each bit is ORed with the corresponding bit of the operand as illustrated below.

Bit 0 of flag is ORed with 1 and Resulting o/p bit=1
Bit 1 of flag is ORed with 0 and Resulting o/p bit= Bit 1 of flag
Bit 2 of flag is ORed with 0 and Resulting o/p bit= Bit 2 of flag
Bit 3 of flag is ORed with 0 and Resulting o/p bit= Bit 3 of flag
Bit 4 of flag is ORed with 0 and Resulting o/p bit= Bit 4 of flag
Bit 5 of flag is ORed with 0 and Resulting o/p bit= Bit 5 of flag
Bit 6 of flag is ORed with 0 and Resulting o/p bit= Bit 6 of flag
Bit 7 of flag is ORed with 0 and Resulting o/p bit= Bit 7 of flag

Bitwise OR operation combined with left shift operation of '1' is used for selectively setting any bit in a variable. For example the following operation will set bit 6 of *char* variable flag.

```
//Sets 6th bit of flag. Bit numbering starts with 0.
flag = flag | (1<<6);
```

Re-writing the above code for a neat syntax will give

```
flag |= (1<<6);     //Equivalent to flag = flag | (1<<6);
```

The same can also be achieved by bitwise ORing the variable flag with a mask with 6th bit '1' and all other bits '0', i.e. mask with 01000000 in Binary representation and 0x40 in hex representation.

```
flag |= 0x40;  //Equivalent to flag = flag | (1<<6);
```

Clearing a desired bit is achieved by Bitwise ANDing the bit with '0'. Bitwise AND operation combined with left shifting of '1' can be used for clearing any desired bit in a variable.

```
Example:
      flag &= ~ (1<<6);
```

The above instruction will clear the 6th bit of the character variable *flag*. The operation is illustrated below.

Execution of (1<<6) shifts '1' to six positions left and the resulting output in binary will be 01000000. Inverting this using the Bitwise NOT operation (~ (1<<6)) inverts the bits and give 10111111 as output. When *flag* is Bitwise ANDed with 10111111, the 6th bit of *flag* is cleared (set to '0') and all other bits of *flag* remain unchanged.

From the above illustration it can be inferred that the same operation can also be achieved by a direct Bitwise ANDing of the variable *flag* and a mask with binary representation 10111111 or hex representation 0xBF.

```
flag &= 0xBF;      //Equivalent to flag = flag & ~(1<<6);
```

Shifting the mask '1' for setting or clearing a desired bit works perfectly, if the operand on which these operations are performed is 8bit wide. If the operand is greater than 8 bits in size, care should be taken to adjust the mask as wide as the operand. As an example let us assume *flag* as a 32bit operand. For clearing the 6th bit of *flag* as illustrated in the previous example, the mask 1 should be re-written as '1L' and the instruction becomes

```
flag &= ~ (1L<<6);
```

Toggling Bits

Toggling a bit is performed to negate (toggle) the current state of a bit. If current state of a specified bit is '1', after toggling it becomes '0' and vice versa. Toggling is also known as inverting bits. The Bitwise XOR operator is used for toggling the state of a desired bit in an operand.

```
flag ^= (1<<6);      //Toggle bit 6 of flag
```

The above instruction toggles bit 6 of *flag*.

Adding '1' to the desired bit position (In the above example 0x40 for 6th bit) will also toggle the current state of the desired bit. This approach has the following drawback. If the current state of the bit which is to be inverted is '1', adding a '1' to that bit position inverts the state of that bit and at the same time a carry is generated and it is propagated to the next most significant bit (MSB) and change the status of some of the other bits falling to the left of the bit which is toggled.

Extracting and Inserting Bits

Quite often it is meaningful to store related information as the bits of a single variable instead of saving them as separate variables. This saves considerable amount of memory in an embedded system where data memory is a big bottleneck. Typical scenario in embedded applications where information can be stored using the bits of single variable is, information provided by Real Time Clock (RTC). RTC chip provides various data like current date/month/year, day of the week, current time in hours/minutes/seconds, etc. If an application demands the storage of all these information in the data memory of the embedded system for some reason, it can be achieved in two ways;

1. Use separate variables for storing each data (date, month, year, etc.)
2. Club the related data and store them as the bits of a single variable

As an example assume that we need to store the date information of the RTC in the data memory in D/M/Y format. Where 'D' stands for date varying from 1 to 31, 'M' stands for month varying from 1 to 12 and 'Y' stands for year varying from 0 to 99. If we follow the first approach we need 3 character variables to store the date information separately. In the second approach, we need only 5 bits for date (With 5 bits we can represent 0 to $2^5 - 1$ numbers (0 to 31), 4 bits for month and 7 bits for year. Hence the total number of bits required to represent the date information is 16. All these bits can be fitted into a 16 bit variable as shown in Fig. 9.11.

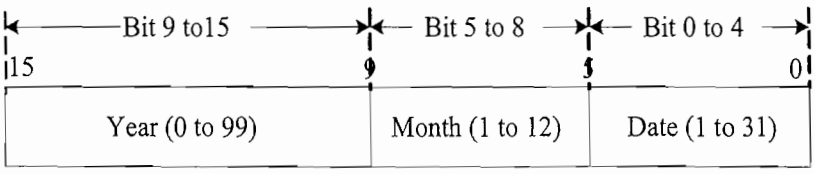

Fig. 9.11 **Packed Bits for data representation**

Suppose this is arranged in a 16bit integer variable date, for any calculation requiring 'Year', 'Month' or 'Date', each should be extracted from the single variable date. The following code snippet illustrates the extraction of 'Year'

```
char year = (date>>9) & 0x7F;
```

(*date*>>9) shifts the contents 9 bits to the left and now 'Year' is stored in the variable *date* in bits 0 to 6 (including both). The contents of other bits (7 to 15) are not relevant to us and we need only the first 7 bits (0 to 6) for getting the 'Year' value. ANDing with the mask 0x7F (01111111 in Binary) retains the contents of bits 0 to 6 and now the variable year contains the 'Year' value extracted from variable '*date*'.

Similar to the extracting of bits, we may have to insert bits to change the value of certain data. In the above example if the program is written in such a way that after each 1 minute, the RTC's year register is read and the contents of bit fields 9 to 15 in variable '*date*' needs to be updated with the new value. This can be achieved by inserting the bits corresponding to the data into the desired bit position. Following code snippet illustrates the updating of 'Year' value for the above example. To set the new 'Year' value to the variable '*date*' all the corresponding bits in the variable for 'Year' should be cleared first. It is illustrated below.

```
date = date & ~ (0x7F<<9);
```

The mask for 7 bits is 0x7F (01111111 in Binary). Shifting the mask 9 times left aligns the mask to that of the bits for storing 'Year'. The ~ operator inverts the 7 bits aligning to the bits corresponding to 'Year'. Bitwise ANDing with this negated mask clears the current bits for 'Year' in the variable '*date*'. Now shift the new value which needs to be put at the 'Year' location, 9 times for bitwise alignment for the bits corresponding to 'Year'. Bitwise OR the bit aligned new value with the '*date*' variable whose bits corresponding to 'Year' is already cleared. The following instruction performs this.

```
date |= (new_year <<9);
```

where 'new_year' is the new value for 'Year'.

In order to ensure the inserted bits are not exceeding the number of bits assigned for the particular bit variable, bitwise AND the new value with the mask corresponding to the number of bits allocated to the corresponding variable (Above example 7 bits for 'Year' and the mask is 0x7F). Hence the above instruction can be written more precisely as

```
date |= (new_year & 0x7F) <<9;
```

If all the bits corresponding to the bit field to be modified are not set to zero prior to Bitwise ORing with the new value, any existing bit with value '1' will remain as such and expected result will not be obtained.

Testing Bits So far we discussed the Bitwise operators for changing the status of a selected bit. Bitwise operators can also be used for checking the present status of a bit without modifying it for decision making operations.

```
if (flag & (1<<6))    //Checks whether 6th bit of flag is '1'
```

This instruction examines the status of the 6th bit of variable *flag*. The same can also be achieved by using a constant bit mask as

```
if (flag & 0x40)    //Checks whether 6th bit of flag is '1'
```

9.3.3.19 Coding Interrupt Service Routines (ISR) Interrupt is an event that stops the current execution of a process (task) in the CPU and transfers the program execution to an address in code memory where the service routine for the event is located. The event which stops the current execution can be an internal event or an external event or a proper combination of the external and internal event. Any trigger signal coming from an externally interfaced device demanding immediate attentions is an example for external event whereas the trigger indicating the overflow of an internal timer is an example for internal event. Reception of serial data through the serial line is a combination of internal and external events. The number of interrupts supported, their priority levels, interrupt trigger type and the structure of interrupts are processor/controller architecture dependent and it varies from processor to processor. Interrupts are generally classified into two: Maskable Interrupts and Non-maskable Interrupts (NMI). Maskable interrupt can be ignored by the CPU if the interrupt is internally not enabled or if the CPU is currently engaged in processing another interrupt which is at high priority. Non-maskable interrupts are interrupts which require urgent attention and cannot be ignored by the CPU. Reset (RST) interrupt and TRAP interrupt of *8085* processor are examples for Non-maskable interrupts.

Interrupts are considered as boon for programmers and their main motto is "*give real time behaviour to applications*". In a processor/controller based simple embedded system where the application is designed in a super loop model, each interrupt supported by the processor/controller will have a fixed memory location assigned in the code memory for writing its corresponding service routine and this address is referred as *Interrupt Vector Address*. The vector address for each interrupt will be pre-defined and the number of code memory bytes allocated for each Interrupt Service Routine, starting from the Interrupt Vector Address may also be fixed. For example the interrupt vector address for interrupt 'INT0' of *8051* microcontroller is '0003H' and the number of code memory bytes allowed for writing its Service routine is 8 bytes.

The function written for serving an Interrupt is known as Interrupt Service Routine (ISR). ISR for each interrupt may be different and they are placed at the Interrupt Vector Address of corresponding Interrupt. ISR is essentially a function that takes no parameters and returns no results. But, unlike a regular function, the ISR can be active at any time since the triggering of interrupts need not be in sync with the internal program execution (e.g. An external device connected to the external interrupt line can assert external interrupt at any time regardless at what stage the program execution is currently). Hence special care must be taken in writing these functions keeping in mind; they are not going to be executed in a pre-defined order. What all special care should be taken in writing an ISR? The following section answers this query.

Imagine a situation where the application is doing some operations and some registers are modified and an interrupt is triggered in between the operation. Indeed the program flow is diverted to the Interrupt Service Routine, if the interrupt is an enabled interrupt and the operation in progress is not a service routine of a high priority interrupt, and the ISR is executed. After completing the ISR, the program flow is re-directed to the point where it got interrupted and the interrupted operation is continued. What happens if the ISR modifies some of the registers used by the program? No doubt the application will produce unexpected results and may go for a toss. How can this situation be tackled? Such a situation can be avoided if the ISR is coded in such a way that it takes care of the following:

1. Save the current context (Important Registers which the ISR will modify)
2. Service the Interrupt
3. Retrieve the saved context (Retrieve the original contents of registers)
4. Return to the execution point where the execution is interrupted

Normal functions will not incorporate code for saving the current context before executing the function and retrieve the saved context before exiting the function. ISR should incorporate code for perform-

ing these operations. Also it is known to the programmer at what point a normal function is going to be invoked and the programmer can take necessary precautions before calling the function, it is not the case for an ISR. Interrupt Service Routines can be coded either in Assembly language or High level language in which the application is written. Assembly language is the best choice if the ISR code is very small and the program itself is written in Assembly. Assembly code generates optimised code for ISR and user will have a good control on deciding what all registers needs to be preserved from alteration. Most of the modern cross-compilers provide extensive support for efficient and optimised ISR code. The way in which a function is declared as ISR and what all context is saved within the function is cross-compiler dependent.

Keil C51 Cross-complier for 8051 microcontroller implements the Interrupt Service Routine using the keyword *interrupt* and its syntax is illustrated below.

```
void interrupt_name (void)    interrupt x using y
{
/*Process Interrupt*/
}
```

interrupt_name is the function name and programmers can choose any name according to his/her taste. The attribute '*interrupt*' instructs the cross compiler that the associated function is an interrupt service routine. The *interrupt* attribute takes an argument *x* which is an integer constant in the range 0 to 31 (supporting 32 interrupts). This number is the interrupt number and it is essential for placing the generated hex code corresponding to the ISR in the corresponding Interrupt Vector Address (e.g. For placing the ISR code at code memory location 0003H for Interrupt 0 – External Interrupt 0 for *8051* microcontroller). *using* is an optional keyword for indicating which register bank is used for the general purpose Registers R0 to R7 (For more details on register banks and general purpose registers, refer to the hardware description of *8051*). The argument *y* for *using* attribute can take values from 0 to 3 (corresponding to the register banks 0 to 3 of 8051). The *interrupt* attribute affects the object code of the function in the following way:

1. If required, the contents of registers ACC, B, DPH, DPL, and PSW are saved on the stack at function invocation time.
2. All working registers (R0 to R7) used in the interrupt function are stored on the stack if a register bank is not specified with the *using* attribute.
3. The working registers and special registers that were saved on the stack are restored before exiting the function.
4. The function is terminated by the 8051 RETI instruction.

Typical usage is illustrated below

```
void external_interrupt0 (void )    interrupt 0 using 0
{
adc_control = *adc_control_reg //Read memory mapped ADC
                               // control Register
}
```

If the cross-compiler you are using don't have a built in support for writing ISRs, What shall you do? Don't be panic you can implement the ISR feature with little tricky coding. If the cross-compiler provides support for mixing high level language-C and Assembly, write the ISR in Assembly and place the ISR code at the corresponding Interrupt Vector address using the cross compiler's support for placing the code in an absolute address location of code memory (Using keywords like *_at*. Refer to your

cross compiler's documentation for getting the exact keyword). If the ISR is too complicated, you can place the body of the ISR processing in a normal C function and call it from a simple assembly language wrapper. The assembly language wrapper should be installed as the ISR function at the absolute address corresponding to the Interrupt's Vector Address. It is responsible for executing the current context saving and retrieving instructions. Current context saving instructions are written on top of the call to the 'C' function and context retrieving instructions are written just below the 'C' function call. It is little puzzling to find answers to the following questions in this approach.

1. Which registers must be saved and restored since we are not sure which registers are used by the cross compiler for implementing the 'C' function?
2. How the assembly instructions can be interfaced with high-level language like 'C'?

Answers to these questions are cross compiler dependent and you need to find the answer by referring the documentation files of the cross-compiler in use. Context saving is done by 'Pushing' the registers (using PUSH instructions) to stack and retrieving the saved context is done by 'Poping' (using POP instructions) the pushed registers. Push and Pop operations usually follow the Last In First Out (LIFO) method. While writing an ISR, always keep in mind that the primary aim of an interrupt is to provide real time behaviour to the program. Saving the current context delays the execution of the original ISR function and it creates Interrupt latency (Refer to the section on Interrupt Latency for more details) and thereby adds lack of real time behaviour to an application. As a programmer, if you are responsible for saving the current context by Pushing the registers, avoid Pushing the registers which are not used by the ISR. If you deliberately avoid the saving of registers which are going to be modified by the ISR, your application may go for a toss. Hence Context saving is an unavoidable evil in Interrupt driven programming. If you go for saving unused registers, it will increase interrupt latency as well as stack memory usage. So always take a judicious decision on the context saving operation. If the cross compiler offers the context saving operation by supporting ISR functions, always rely on it. Because most modern cross compilers are smart and capable of taking a judicious decision on context saving. In case the cross compiler is not supporting ISR function and as a programmer you are the one writing ISR functions either in Assembly or by mixing 'C' and Assembly, ensure that the size of ISR is not crossing the size of code memory bytes allowed for an Interrupt's ISR (8 bytes for *8051*). If it exceeds, it will overlap with the location for the next interrupt and may create unexpected behaviour on servicing the Interrupt whose Vector address got overlapped.

9.3.3.20 Recursive Functions A function which calls itself repeatedly is called a Recursive Function. Using recursion, a complex problem is split into its single simplest form. The recursive function only knows how to solve that simplest case. Recursive functions are useful in evaluating certain types of mathematical function, creating and accessing dynamic data structures such as linked lists or binary trees. As an example let us consider the factorial calculation of a number.

By mathematical definition

n factorial $= 1 \times 2 \times \ldots\ldots (n-2) \times (n-1) \times n$; where $n = 1, 2,$ etc… and 0 factorial = 1

Using 'C' the function for finding the factorial of a number '*n*' is written as

```
int factorial (int n)
{
int count;
int factorial=1;
for (count=1; count<=n; count++)
    factorial*=count;
```

```
    return factorial;
    }
```

This code is based on iteration. The iteration of calculating the repeated products is performed until the count value exceeds the value of the number whose factorial is to be calculated. We can split up the task performed inside the function as *count * (count +1)* till count is one short of the number whose factorial is to be calculated. Using recursion we can re-write the above function as

```
int factorial (int n)
{
if (n==0)
   return 1;
else
   return (n*factorial (n-1));
}
```

Here the function *factorial (n)* calls itself, with a changed version of the parameter for each call, inside the same for calculating the factorial of a given number. The 'if' statement within the recursive function forces the function to stop recursion when a certain criterion is met. Without an 'if' statement a recursive function will never stop the recursion process. Recursive functions are very effective in implementing solutions expressed in terms of a successive application of the same solution to the solution subsets. Recursion is a powerful at the same time most dangerous feature which may lead to application crash. The local variables of a recursive function are stored in the stack memory and new copies of each variable are created for successive recursive calls of the function. As the recursion goes in a deep level, stack memory may overflow and the application may bounce.

Recursion vs. Iteration: A comparison Both recursion and iteration is used for implementing certain operations which are self repetitive in some form.

- Recursion involves a lot of call stack overhead and function calls. Hence it is slower in operation compared to the iterative method. Since recursion implements the functionality with repeated self function calls, more stack memory is required for storing the local variables and the function return address
- Recursion is the best method for implementing certain operations like certain mathematical operation, creating and accessing of dynamic data structures such as linked lists or binary trees
- A recursive solution implementation can always be replaced by iteration. The process of converting a recursive function to iterative method is called '*unrolling*'

Benefits of Recursion Recursion brings the following benefits in programming:

- Recursive code is very expressive compared to iterative code. It conveys the intended use.
- Recursion is the most appropriate method for certain operations like permutations, search trees, sorting, etc.

Drawbacks of Recursion Though recursion is an effective technique in implementing solutions expressed in terms of a successive application of the same solution to the solution subsets, it possesses the following drawbacks.

- Recursive operation is slow in operation due to call stack overheads. It involves lot of stack operations like local variable storage and retrieval, return address storage and retrieval, etc.
- Debugging of recursive functions are not so easy compared to iterative code debugging

9.3.3.21 Re-entrant Functions Functions which can be shared safely with several processes concurrently are called re-entrant functions. When a re-entrant function is executing, another process can interrupt the execution and can execute the same re-entrant function. The "*another process*" referred here can be a thread in a multithreaded application or can be an Interrupt Service Routine (ISR). Re-entrant function is also referred as '*pure*' function. Embedded applications extensively make use of re-entrant functions. Interrupt Service Routines (ISR) in Super loop based firmware systems and threads in RTOS based systems can change the program's control flow to alter the current context at any time. When an interrupt is asserted, the current operation is put on hold and the control is transferred to another function or task (ISR). Imagine a situation where an interrupt occurs while executing a *function* ***x*** and the ISR also contain the task of executing the *function* ***x***. What will happen? - Obviously the ISR will modify the shared variables of *function* ***x*** and when the control is switched back to the point where the execution got interrupted, the function will resume its execution with the data which is modified by the ISR. What will be the outcome of this action? - Unpredictable result causing data corruption and potential disaster like application break-down. Why it happens so? Due to the corruption of shared data in function which is unprotected. How this situation can be avoided? By carefully controlling the sharing of data in the function.

In embedded applications, a function/subroutine is considered re-entrant if and only if it satisfies the following criteria.

1. The function should not hold static data over successive calls, or return a pointer to static data.
2. All shared variables within the function are used in an atomic way.
3. The function does not call any other non-reentrant functions within it.
4. The function does not make use of any hardware in a non-atomic way

Rule# 1 deals with variable usage and function return value for a reentrant function. In an operating system based embedded system, the embedded application is managed by the operating system services and the '*memory manager*' service of the OS kernel is responsible for allocating and de-allocating the memory required by an application for its execution. The working memory required by an application is divided into three groups namely; stack memory, heap memory and data memory (Refer to *the Dynamic Memory Allocation* section of this chapter for more details). The life time of static variables is same as that of the life time of the application to which it belongs and they are usually held in the data memory area of the memory space allocated for the application. All static variables of a function are allocated in the data memory area space of the application and each instance of the function shares this, whereas local (auto) variables of a function are stored in the stack memory and each invocation of the function creates independent copies of these variables in the stack memory. For a function to be reentrant, it should not keep any data over successive invocations (the function should not contain any static storage). If a function needs some data to be kept over successive invocations, it should be provided through the caller function instead of storing it in the function in the form of static variables. If the function returns a pointer to a static data, each invocation of the function makes use of this pointer for returning the result. This can be tackled by using caller function provided storage for passing the data back to the caller function. The 'callee' function needs to be modified accordingly to make use of the caller function provided storage for passing the data back to the caller.

Rule# 2 deals with '*atomic*' operations. So what does '*atomic*' mean in reentrancy context? Meaning the operation cannot be interrupted. If an embedded application contains variables which are shared between various threads of a multitasking system (Applicable to Operating System based embedded systems) or between the application and ISR (In Non-RTOS based embedded systems), and if the operation on the shared variable is non-atomic, there is a possibility for corruption of the variable due

to concurrent access of the variable by multiple threads or thread and ISR. As an example let us assume that the variable counter is shared between multiple threads or between a thread and ISR, the instruction,

```
counter++;
```

need not operate in an '*atomic*' way on the variable *counter*. The compiler/cross-compiler in use converts this high level instruction into machine dependent code (machine language) and the objective of incrementing the variable *counter* may be implemented using a number of low level instructions by the compiler/cross compiler. This violates the 'atomic' rule of operation. Imagine a situation where an execution switch (context switch) happens when one thread is in the middle of executing the low level instructions corresponding to the high level instruction *counter*++, of the function and another thread (which is currently in execution after the context switch) or an ISR calls the same function and executes the instruction *counter*++, this will result in inconsistent result. Refer to the section '*Racing*' under the topic '*Task Synchronisation Issues*' of the chapter '*Designing with Real Time Operating Systems*' for more details on this. Eliminating shared global variables and making them as variables local to the function solves the problem of modification of shared variables by concurrent execution of the function.

Rule# 3 is selfexplanatory. If a re-entrant function calls another function which is non-reentrant, it may create potential damages due to the unexpected modification of shared variables if any. What will happen if a reentrant function calls a standard library function at run time? By default most of the run time library is reentrant. If a standard library function is not reentrant the function will no longer be reentrant.

Rule# 4 deals with the atomic way of accessing hardware devices. The term '*atomic*' in hardware access refers to the number of steps involved in accessing a specific register of a hardware device. For the hardware access to be atomic the number of steps involved in hardware access should be one. If access is achieved through multiple steps, any interruption in between the steps may lead to erroneous results. A typical example is accessing the hardware register of an I/O device mapped to the host CPU using paged addressing technique. In order to Read/Write from/to any of the hardware registers of the device a minimum of two steps is required. First write the page address, corresponding to the page in which the hardware register belongs, to the page register and then read the register by giving the address of the register within that page. Now imagine a situation where an interrupt occurs at the moment executing the page setting instruction in progress. The ISR will be executed after finishing this instruction. Suppose ISR also involves a Read/Write to another hardware register belonging to a different page. Obviously it will modify the page register of the device with the required value. What will be its impact? On finishing the ISR, the interrupted code will try to read the hardware register with the page address which is modified by the ISR. This yields an erroneous result.

How to declare a function as Reentrant The way in which a function is declared reentrant is cross-compiler dependent. For Keil C51 cross-compiler for *8051* controller, the keyword '*reentrant*' added as function attribute treats the function as reentrant. For each reentrant function, a reentrant stack area is simulated in internal or external memory depending whether the data memory is internal or external to the processor/controller. A typical reentrant function implementation for C51 cross-compiler for *8051* controller is given below.

```
int multiply (char i, int b) reentrant
{
 int x;
 x = table [i];
return (x * b);
}
```

The simulated stack used by reentrant functions has its own stack pointer which is independent of the processor stack and stack pointer. For C51 cross compiler the simulated stack and stack pointers are declared and initialised in the startup code in STARTUP.A51 which can be found in the LIB subdirectory. You must modify the startup code to specify which simulated stack(s) to initialize in order to use re-entrant functions. You can also modify the starting address for the top of the simulated stack(s) in the startup code. When calling a function with a re-entrant stack, the cross-compiler must know that the function has a re-entrant stack. The cross-compiler figures this out from the function prototype which should include the reentrant keyword (just like the function definition). The cross compiler must also know which reentrant stack to stuff the arguments for it. For passing arguments, the cross-compiler generates code that decrements the stack pointer and then "pushes" arguments onto the stack by storing the argument indirectly through R0/R1 or DPTR (*8051* specific registers). Calling reentrant function decrements the stack pointer (for local variables) and access arguments using the stack pointer plus an offset (which corresponds to the number of bytes of local variables). On returning, reentrant function adjusts the stack pointer to the value before arguments were pushed. So the caller does not need to perform any stack adjustment after calling a reentrant function.

Reentrant vs. Recursive Functions The terms Reentrant and Recursive may sound little confusing. You may find some amount of similarity among them and ultimately it can lead to the thought are Reentrant functions and Recursive functions the same? The answer is—it depends on the usage context of the function. It is not necessary that all recursive functions are reentrant. But a Reentrant function can be invoked recursively by an application.

For *8051* Microcontroller, the internal data memory is very small in size (128 bytes) and the stack as well user data memory is allocated within it. Normally all variables local to a function and function arguments are stored in fixed memory locations of the user data memory and each invocation of the function will access the fixed data memory and any recursive calls to the function use the same memory locations. And, in this case, arguments and locals would get corrupted. Hence the scope for implementing recursive functions is limited. A reentrant function can be called recursively to a recursion level dependent on the simulated stack size for the same reentrant function.

C51 Cross-compiler does not support recursive calls if the functions are non-reentrant.

9.3.3.22 Dynamic Memory Allocation Every embedded application, regardless of whether it is running on an operating system based product or a non-operating system based product (Super loop based firmware Architecture) contains different types of variables and they fall into any one of the following storage types namely; *static*, *auto*, *global* or *constant* data. Regardless of the storage type each variable requires memory locations to hold them physically. The storage type determines in which area of the memory each variable needs to be kept. For an Assembly language programmer, handling memory for different variables is quite difficult. S/he needs to assign a particular memory location for each variable and should recollect where s/he kept the variable for operations involving that variable.

Certain category of embedded applications deal with fixed number of variables with fixed length and certain other applications deal with variables with fixed memory length as well as variable with total storage size determined only at the runtime of application (e.g. character array with variable size). If the number of variables are fixed in an application and if it doesn't require a variable size at run time, the cross compiler can determine the storage memory required by the application well in advance at the run time and can assign each variable an absolute address or relative (indirect) address within the data memory. Here the memory required is fixed and allocation is done before the execution of the application. This type of memory allocation is referred as '*Static Memory Allocation*'. The term '*Static*' mentioned

here refers '*fixed*' and it is no way related to the storage class static. As mentioned, some embedded applications require data memory which is a combination of fixed memory (Number of variables and variable size is known prior to cross compilation) and variable length data memory. As an example, let's take the scenario where an application deals with reading a stream of character data from an external environment and the length of the stream is variable. It can vary between any numbers (say 1 to 100 bytes). The application needs to store the data in data memory temporarily for some calculation and it can be ignored after the calculation. This scenario can be handled in two ways in the application program. In the first approach, allocate fixed memory with maximum size (say.100 bytes) for storing the incoming data bytes from the stream. In the second approach allocate memory at run time of the application and de-allocate (free) the memory once the data memory storage requirement is over. In the first approach if the memory is allocated fixedly, it is locked forever and cannot re-used by the application even if there is no requirement for the allocated number of bytes and it will definitely create memory bottleneck issues in embedded systems where memory is a big constraint. Hence it is not advised to go for fixed memory allocations for applications demanding variable memory size at run time. Allocating memory on demand and freeing the memory at run time is the most advised method for handling the storage of dynamic (changing) data and this memory allocation technique is known as '*Dynamic Memory Allocation*'.

Dynamic memory allocation technique is employed in Operating System (OS) based embedded systems. Operating system contains a '*Memory Management Unit*' and it is responsible for handling memory allocation related operations. The memory management unit allocates memory to hold the code for the application and the variables associated with the application. The conceptual view of storage of an application and the variables related to the application is represented in Fig. 9.12.

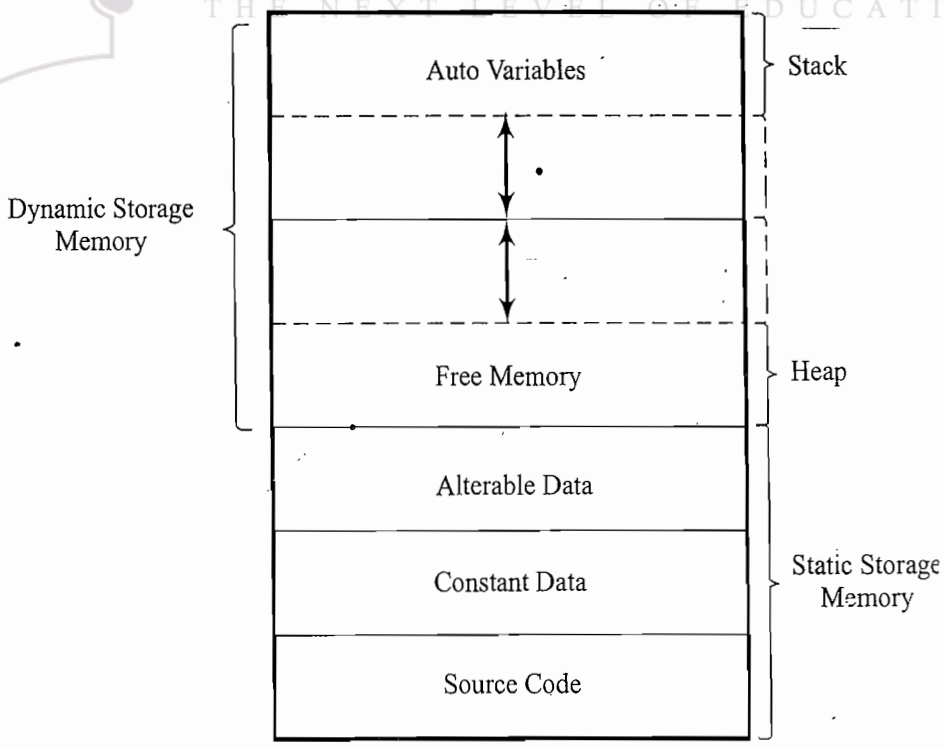

Fig. 9.12 **Static and Dynamic Memory Storage Allocation**

Memory manager allocates a memory segment to each application and the memory segment holds the source code of the application as well as data related to the application. The actual program code (executable machine code instructions) is stored at the lower address of the memory (at the beginning address of the memory segment and stores upward). Constant data related to the application (e.g. const *int x* = 10;) is stored at the memory area just above the executable code. The alterable data (global and static variables; e.g. static int *j* = 10; *int k* = 2; (global declaration), etc.) are stored at the '*Alterable Data*' memory area. The size of the executable code, the number of constant data and alterable data in an application is always fixed and hence they occupy only a fixed portion of the memory segment. Memory allocated for the executable code, constant data and alterable data together constitute the '*Static storage memory* (*Fixed storage memory*)'. The storage memory available within the memory segment excluding the fixed memory is the '*Dynamic storage memory*'. Dynamic storage memory area is divided into two separate entities namely '*stack*' and '*heap*'. '*Stack memory*' normally starts at the high memory area (At the top address) of the memory segment and grows down. Within a memory segment, fixed number of storage bytes are allocated to stack memory and the stack can grow up to this maximum allocated size. Stack memory is usually used for holding auto variables (variables local to a function), function parameters and function return values. Depending upon the function calls/return and auto variable storage, the size of the stack grows and shrinks dynamically. If at any point of execution the stack memory exceeds the allocated maximum storage size, the application may crash and this condition is called '*Stack overflow*'. The free memory region lying in between the stack and fixed memory area is called '*heap*'. Heap is the memory pool where storage for data is done dynamically as and when the application demands. Heap is located immediately above the fixed memory area and it grows upward. Any request for dynamic memory allocation by the program increases the size of *heap* (depending on the availability of free memory within heap) and the free memory request decrements the size of *heap* (size of already occupied memory within the heap area). *Heap* can be viewed as a 'bank' in real life application, where customers can demand for loan. Depending on the availability of money, bank may allot loan to the customer and customer re-pays the loan to the bank when he/she is through with his/her need. Bank uses the money repaid by a customer to allocate a loan to another customer—some kind of rolling mechanism. 'C' language establishes the dynamic memory allocation technique through a set of '*Memory management library routines*'. The most commonly used 'C' library functions for dynamic memory allocation are '*malloc*', '*calloc*', '*realloc*' and '*free*'. The following sections illustrate the use of these memory allocation library routines for allocating and de-allocating (freeing) memory dynamically.

malloc() *malloc()* function allocates a block of memory dynamically. The *malloc()* function reserves a block of memory of size specified as parameter to the function, in the *heap* memory and returns a pointer of type void. This can be assigned to a pointer of any valid type. The general form of using *malloc()* function is given below.

```
pointer =  (pointer_type *) malloc (no. of bytes);
```

where '*pointer*' is a pointer of type 'pointer_type'. 'pointer_type' can be '*int*', '*char*', '*float*' etc. *malloc()* function returns a pointer of type 'pointer_type' to a block of memory with size 'no. of bytes'. A typical example is given below.

```
ptr= (char *) malloc(50);
```

This instruction allocates 50 bytes (If there is 50 bytes of memory available in the *heap* area) and the address of the first byte of the allocated memory in the *heap* area is assigned to the pointer *ptr* of type *char*. It should be noted that the *malloc()* function allocates only the requested number of bytes and it

will not allocate memory automatically for the units of the pointer in use. For example, if the programmer wants to allocate memory for 100 integers dynamically, the following code

```
x= (int *) malloc(100);
```

will not allocate memory size for 100 integers, instead it allocates memory for just 100 bytes. In order to make it reserving memory for 100 integer variables, the code should be re-written as

```
x= (int *) malloc(100 * sizeof (int));
```

malloc() function can also be used for allocating memory for complex data types such as structure pointers apart from the usual data types like 'int', 'char', 'float' etc. *malloc()* allocates the requested bytes of memory in *heap* in a continuous fashion and the allocation fails if there is no sufficient memory available in continuous fashion for the requested number of bytes by the *malloc()* functions. The return value of *malloc()* will be NULL (0) if it fails. Hence the return value of *malloc()* should be checked to ensure whether the memory allocation is successful before using the pointer returned by the *malloc()* function.

```
E.g.   int * ptr;
       if ((ptr= (int *) malloc(50 * sizeof (int))))
              printf ("Memory allocated successfully");
       else
              printf ("Memory allocation failed");
```

Remember malloc() only allocates required bytes of memory and will not initialise the allocated memory. The allocated memory contains random data.

calloc() The library function *calloc()* allocates multiple blocks of storage bytes and initialises each allocated byte to zero. Syntax of *calloc()* function is illustrated below.

```
pointer = (pointer_type *) calloc (n, size of block);
```

where 'pointer' is a pointer of type 'pointer_type'. 'pointer_type' can be 'int', 'char', 'float' etc. '*n*' stands for the number of blocks to be allocated and 'size of block' tells the size of bytes required per block. The *calloc*(*n*, size of block) function allocates continuous memory for '*n*' number of blocks with 'size of block' number of bytes per block and returns a pointer of type 'pointer_type' pointing to the first byte of the allocated block of memory. A typical example is given below.

```
ptr= (char *) calloc (50,1);
```

Above instruction allocates 50 contiguous blocks of memory each of size one byte in the *heap* memory and assign the address of the first byte of the allocated memory region to the character pointer '*ptr*'. Since *malloc()* is capable of allocating only fixed number of bytes in the *heap* area regardless of the storage type, *calloc()* can be used for overcomming this limitation as discussed below.

```
ptr= (int *) calloc(50,sizeof(int));
```

This instruction allocates 50 blocks of memory, each block representing an integer variable and initialises the allocated memory to zero. Similar to the *malloc()* function, *calloc()* also returns a 'NULL' pointer if there is not enough space in the *heap* area for allocating storage for the requested number of memory by the *calloc()* function. Hence it is advised to check the pointer to which the *calloc()* assigns

the address of the first byte in the allocated memory area. If *calloc()* function fails, the return value will be 'NULL' and the pointer also will be 'NULL'. Checking the pointer for 'NULL' immediately after assigning with pointer returned by *calloc()* function avoids possible errors and application crash.

Features provided by *calloc()* can be implemented using *malloc()* function as

```
pointer = (pointer_type *) malloc (n * size of block);
memset(pointer, 0, n * size of block);
```

For example

```
ptr= (int *) malloc (10 * sizeof (int));
memset(ptr, 0, n * size of block);
```

The function *memset (ptr, 0, n * size of block)* sets the memory block of size (*n* * size of block) with starting address pointed by '*ptr*', to zero.

free() The 'C' memory management library function *free()* is used for releasing or de-allocating the memory allocated in the *heap* memory by *malloc()* or *calloc()* functions. If memory is allocated dynamically, the programmer should release it if the dynamically allocated memory is no longer required for any operation. Releasing the dynamically allocated memory makes it ready to use for other dynamic allocations. The syntax of *free()* function is given below.

```
free (ptr);
```

'*ptr*' is the valid pointer returned by the *calloc()* or *malloc()* function on dynamic memory allocation. Use of an invalid pointer with function *free()* may result in the unexpected behaviour of the application.

Note:

1. *The dynamic memory allocated using malloc () or calloc() functions should be released (deallocated) using free () function .*
2. *Any use of a pointer which refers to freed memory space results in abnormal behaviour of application.*
3. *If the parameter to free () function is not a valid pointer, the application behaviour may be unexpected.*

realloc() *realloc()* function is used for changing the size of allocated bytes in a dynamically allocated memory block. You may come across situations where the allocated memory is not sufficient to hold the required data or it is surplus in terms of allocated memory bytes. Both of these situations are handled using *realloc()* function. The *realloc()* function changes the size of the block of memory pointed to, by the *pointer* parameter to the number of bytes specified by the *modified size* parameter and it returns a new pointer to the block. The pointer specified by the *pointer* parameter must have been created with the malloc, calloc, or realloc subroutines and not been de-allocated with the free or realloc subroutines. Function *realloc()* may shift the position of the already allocated block depending on the new size, with preserving the contents of the already allocated block and returns a pointer pointing to the first byte of the re-allocated memory block. *realloc()* returns a void pointer if there is sufficient memory available for allocation, else return a 'NULL' pointer. Syntax of *realloc* is given below.

```
realloc (pointer, modified size);
```

Example illustrating the use of *realloc()* function is given below.

```
char *p;
p= (char*) malloc (10); //Allocate 10 bytes of memory
```

```
p= realloc(p,15);       //Change the allocation to 15 bytes
```

realloc(p,0) is same as free(p), provided 'p' is a valid pointer.

Summary

- ✓ Embedded firmware can be developed to run with the help of an embedded operating system or without an OS. The non-OS based embedded firmware execution runs the tasks in an infinite loop and this approach is known as 'Super loop based' execution
- ✓ Low level language (Assembly code) or High Level language (C,C++ etc.) or a mix of both are used in embedded firmware development
- ✓ In Assembly language based design, the firmware is developed using the Assembly language Instructions specific to the target processor. The Assembly code is converted to machine specific code by an assembler.
- ✓ In High Level language based design, the firmware is developed using High Level language like 'C/C++' and it is converted to machine specific code by a compiler or cross-compiler
- ✓ Mixing of Assembly and High Level language can be done in three ways namely; mixing assembly routines with a high level language like 'C', mixing high level language functions like 'C' functions with application written in assembly code and invoking assembly instructions inline from the high level code
- ✓ Embedded 'C' can be considered as a subset of conventional 'C' language. Embedded C supports almost all 'C' instructions and incorporates a few target processor specific functions/instructions. The standard ANSI 'C' library implementation is always tailored to the target processor/controller library files in Embedded C
- ✓ Compiler is a tool for native platform application development, whereas cross-compiler is a tool for cross platform application development
- ✓ Embedded 'C' supports all the keywords, identifiers and data types, storage classes, arithmetic and logical operations, array and branching instructions supported by standard 'C'
- ✓ *structure* is a collection of data types. Arrays of structures are helpful in holding configuration data in embedded applications. The *bitfield* feature of structures helps in bit declaration and bit manipulation in embedded applications
- ✓ *Pre-processor* in 'C' is compiler/cross-compiler directives used by compiler/cross-compiler to filter the source code before compilation/cross-compilation. Preprocessor directives falls into one of the categories: file inclusion, compile control and macro substitution
- ✓ 'Read only' variable in embedded applications are represented with the keyword *const*. *Pointer to constant data* is a pointer which points to a data which is read only. A *constant pointer* is a pointer to a fixed memory location. *Constant pointer to constant data* is a pointer pointing to a fixed memory location which is read only
- ✓ A variable or memory location in embedded application, which is subject to asynchronous modification should be declared with the qualifier *volatile* to prevent the compiler optimisation on the variable
- ✓ A *constant volatile pointer* represents a Read only register (memory location) of a memory mapped device in embedded application
- ✓ *while(1) { }; do { }while (1); for (;;){}* are examples for infinite loop setting instructions in embedded 'C'.
- ✓ The ISR contains the code for saving the current context, code for performing the operation corresponding to the interrupt, code for retrieving the saved context and code for informing the processor that the processing of interrupt is completed
- ✓ *Recursive function* is a function which calls it repeatedly. Functions which can be shared safely with several processes concurrently are called *re-entrant function*.
- ✓ *Dynamic memory allocation* is the technique for allocating memory on a need basis for tasks. *malloc()*, *calloc()*, *realloc()* and *free()* are the 'C' library routines for dynamic memory management

Keywords

Super Loop Model	:	An embedded firmware design model which executes tasks in an infinite loop at a predefined order
Assembly Language	:	The human readable notation of 'machine language'
Machine Language	:	Processor understandable language made up of 1s and 0s
Mnemonics	:	Symbols used in Assembly language for representing the machine language
Assembler	:	A program to translate Assembly language program to object code
Library	:	Specially formatted, ordered program collections of object modules
Linker	:	A software for linking the various object files of a project
Hex File	:	ASCII representation of the machine code corresponding to an application
Inline Assembly	:	A technique for inserting assembly instructions in a C program
Embedded C	:	'C' Language for embedded firmware development. It supports 'C' instructions and incorporates a few target processor specific functions/instructions along with tailoring of the standard library functions for the target embedded system
Compiler	:	A software tool that converts a source code written in a high level language on top of a particular operating system running on a specific target processor architecture
Cross-compiler	:	The software tools used in cross-platform development applications. It converts the application to a target processor specific code, which is different from the processor architecture on which the compiler is running
Function	:	A self-contained and re-usable code snippet intended to perform a particular task
Function Pointer	:	Pointer variable pointing to a function
structure	:	Variable holding a collection of data types (int, float, char, long, etc.) in C language
structure Padding	:	The act of arranging the structure elements in memory in a way facilitating increased execution speed
union	:	A derived form of structure, which allocates memory only to the member variable of union requiring the maximum storage size on declaring a union variable
Pre-processor	:	A compiler/cross-compiler directives used by compiler/ cross-compiler to filter the source code before compilation/cross-compilation in 'C' language
Macro	:	The 'C' pre-processor for creating portable inline code
const	:	A keyword used in 'C' language for informing the compiler/cross-compiler that the variable is constant. It represents a 'Read only' variable
Dynamic Memory Allocation	:	The technique for allocating memory on a need basis at run time
Stack	:	The memory area for storing local variables, function parameters and function return values and program counter, in the memory model for an application/ task
Static Memory Area	:	The memory area holding the program code, constant variables, static and global variables, in the memory model for an application/task
Heap Memory	:	The free memory lying in between stack and static memory area, which is used for dynamic memory allocation