SC51

C Compiler for 8051

User's Manual

SPJ Systems

101, Beaver Grandeur

Baner Road

Pune - 411045

Tel. (91) (20) 7293002

Fax. (91) (20) 7293003

http://www.spjsystems.com

Terms and Conditions for use of the software

For the purposes of this document, the term THE PRODUCT shall be used to refer to **SC51 (C compiler for 8051).**

Terms and Conditions :

1) You may make a backup copy of THE PRODUCT, and you may install THE PRODUCT on your computer systems hard disk, but only one copy of THE PRODUCT may be in use at any one time.
2) If you make any modifications to THE PRODUCT, the modified code shall, regardless of the extent of modification, remain the property of SPJ Systems. You may not remove or alter any copyright notices contained in the source files, documentation, executables or any part of THE PRODUCT.
3) You may not re-distribute any part of THE PRODUCT, or any materials which are based on, or otherwise derived from any portion of THE PRODUCT. Any and all copies of THE PRODUCT must be retained in your possession at all times.
4) You are granted permission to distribute any programs that you develop with THE PRODUCT, provided that they are not based on, or otherwise derived from any source code contained in THE PRODUCT.
5) You agree to use THE PRODUCT entirely at your own risk, and assume all responsibility for such use. The author and distributors of THE PRODUCT do not warrant it fit or suitable for any particular purpose.

===*===*===

CONTENTS :

===*===*===

1. Introduction

Thank you for purchasing SC51. Now you can write 8051 applications in C and debug (simulate) those without the target hardware.

How to use this manual:
This manual is intended to be read from start to end, so you learn about all the features of this compiler. However, the library functions' description may be skipped initially, and it can be referred to when you actually start writing applications.

The chapter 2 provides instructions to install the software. Chapter 3 gives you a general idea of the features of the software. After reading chapter 3, you may start actually using this software. Chapter 4 contains detail information about 'Configuration'. Chapter 5 contains information about 'Run' menu options.

For any questions or comments about the product, please feel free to contact us at :

SPJ SYSTEMS
101, Beaver Grandeur
Baner Road
Pune. Pin - 411 045.
Tel. +91-20-7293002 Fax. +91-20-7293003
E-mail : spj@spjsystems.com
Web-site : http://www.spjsystems.com

2. Installation

The SC51 software CD includes an automatic installation program called SETUP.EXE.

Insert the SC51 software CD in your CD-ROM drive and run SETUP.EXE from the CD. This will start the installation and the setup program will ask you some questions. e.g. where do you want to install the software (in the default path or somewhere else). Please supply the answers to all the questions and complete installation.

Please read this manual carefully before you start using the software.

3. Language Reference

This chapter describes the general syntax rules for the C language as implemented by this compiler. In order to keep your programs portable, we have tried to stick to ANSI standard C as far as possible. However, we have added certain extentions to ANSI C in our compiler – especially to take advantage of the special features of 8051 architecture, such as bit addressing capability. In this implementation of C Compiler, there are also some minor variations and limitations – in many cases, these are due to the limitations of 8051 architecture. This chapter provides information about such additional features. (Please also read the appendix “Limitations of the compiler” which lists certain limitations of this C compiler and some known bugs.) Please note, that this is NOT an authoritative document describing the C language in general. Many books from reputed publishers are available to learn more about C language. This chapter describes syntax related information applicable to this implementation of C Compiler – and not necessarily of the C language in general.

3.1 : Comments :

The character string "/*" marks the beginning of a comment and the string "*/" marks the end of it. e.g.

```
printf("Some message\n") ;
/*  This is a comment  */
/*  Comments may be spread over
multiple lines
*/
```

However nested comments are not allowed. e.g.

```
/*
comment line 1
/*  line 2  */
last line
*/
```

This will produce an error, because the string "*/" after 'line 2' marks the end of comment, and hence the string "*/" appearing again after 'last line' will generate an error.

C++ style of comments (//) is also supported. i.e. all stuff followed by "//" up to the end of the line is treated as comment.

e.g.

```
printf("Some message\n") ; // This is C++ style comment
```

3.2 : Identifier :

An identifier is just the name you give to a variable, function, or other user defined objects. An identifier can contain letters (A...Z, a...z) and digits (0...9) as well as the underscore character (_). However an identifier can only start with a letter or an underscore. Case is significant. i.e. var1 is not the same as Var1. The maximum length of an identifier is 20 characters.

3.3 : Keywords :

Following is a list of keywords reserved by the compiler. These can not be used as identifier names.

```
break
case
char
continue
default
do
double
else
void
float
for
goto
if
int
interrupt
long
return
short
```

```
unsigned
asm
struct
union
sizeof
SFR
bit
BIT
data
idata
bdata
xdata
pdata
code
using
extern
```

Although `double` and `sizeof` are not supported in this version, still these are reserved keyword, because these may be supported in later versions.

3.4 : Data types :

The following table lists all the data types which are supported by this C compiler, their range of possible values, and their size (in bits) :

Type	**Size (Bits)**	**Range**
bit	1	0 or 1
unsigned char	8	0 to 255
char	8	-128 to 127
unsigned int	16	0 to 65535
int	16	-32768 to 32767
unsigned long int	32	0 to 4294967295
long int	32	-2147483648 to 2147483647
float	32	3.4E-38 to 3.4E+38

Apart from these basic data types, you can declare one dimensional arrays of any of these types except the **bit** type. There are some more restrictions on the use of **bit** data type as follows :

- You can not declare arrays of bit type variables.
- bit type variables can not be passed as parameters.
- A bit type variable must be a global variable. It can not be declared as a local variable or a parameter.
- There can be maximum 120 variables of bit type in any program.
- bit type variables always reside in Internal RAM (bit addressable memory area – 0x20 to 0x2f), regardless of which memory model you are using.
- While assigning a constant value to any bit variable, you can assign only one of the two possible values : 0 or 1.
- In expression, you can combine variables of different data types. e.g. you can add an integer variable to a floating point variable. However, the same is not applicable to bit data type. i.e. if one of the operands is a bit type variable, then the other operand also must be another bit type variable (not even a constant of some other type)
- Only a certain operations can be performed on bit type variables. See the details below in section **3.7 Operators**.
- You can not declare a pointer to bit.

However, pointers (up to one level) to other data types can be declared.

3.5 : Type Conversions :

In an expression, if the two operands of a binary operator are of different types, then the compiler will attempt to convert one of the operands into the type of the other. The compiler uses some rules to do this as follows :

If either of the operands is of type ‘bit’, then compiler prints an error “type mis-match”, since it can not convert bit type to any other type or vice versa.

Otherwise, if either of the operands is of type ‘float’, then the other operand is converted to float type.

Otherwise, if either of the operands is of type ‘long int’ (or ‘unsigned long int’) then the other operand is converted to the same type.

Otherwise, if either of the operands is of type ‘int’ (or ‘unsigned int’) then the other operand is converted to the same type.

Thus ‘char’ type (or ‘unsigned char’ type) gets the lowest priority.

However, the compiler behaves differently for some operators i.e.

```
+=          -=          *=          /=
%=          &=          |=          ^=
<<=         >>=
```

For these operators, the result is to be written back onto the left hand side operand (which must be a variable). Hence, the compiler will ALWAYS convert the right hand side operand into the type of left hand side operand. e.g. if you write “`x *= f`” where x is of type int and f is of type float, then compiler will generate code to convert the value of ‘f’ into integer and then multiply the value of ‘x’ by it and finally store the result (which will be naturally int type) back into the variable x.

3.6 : Constants :

Integer constants may be written in decimal form (e.g. 123 or 30945) or in hexadecimal form with '0x' prefix (e.g. 0xff). Character constants must be enclosed in single quotation marks. e.g. 'A' or '?'. String constants must be enclosed in double quotation marks. e.g. "This is a string constant". Floating point constants are always considered to be of type float. Also floating point constants must be written with a decimal point, otherwise they may be considered as integer constants. e.g. "123" will be considered as "the integer value 123" whereas "123.0" will be considered as "the floating point value 123".
The suffix ‘L’ can be attached to a constant to force it to “long int” type. For example, 123L is “the long int value 123”. Without the suffix, it would have meant “the int value 123”.
Similarly, the suffix ‘UL’ can attached to a constant to force it to “unsigned long int” type.
Similarly, the suffix ‘S’ can attached to a constant to force it to “unsigned char” type. For example, 123S means “the unsigned char value 123 – a single byte value’. However, without the ‘S’ suffix, 123 would have meant “the int value 123 – two bytes”.

3.7 : Operators :

Following operators are supported by the compiler :

```
+           -           *           /
%           ++          --          =
```

```
==          ~           !          !=
<           >          <=          >=
&          &&           |          ||
^          >>          <<          +=
-=         *=          /=          %=
&=         |=          ^=         <<=
>>=
```

However, on bit type variables, some of the operators can not be used. Here is the list of operators which **can not be used with bit type variables** :

```
+           -           *           /
%          ++          --          >>
<<         +=          -=          *=
/=         %=         <<=         >>=
```

3.8 : Function prototypes :

You may write function prototypes to declare a function. These declarators include information about the function parameters.
e.g.

```
int func1 (char par1, int par2, long int par3) ;
```

The actual function definition may be written somewhere else as :

```
int func1 (char par1, int par2, long int par3) {
/*
write some code here
*/
}  /* this is a valid function definition */
```

The old style of writing function definitions is **NOT** supported.
e.g. this is not allowed :

```
int func1 (par1, par2, par3)
char par1 ;
int par2 ;
long int par3 ;
{
/*
```

```
    some statements here
*/
}  /* this is not a valid function definition */
```

3.9 : Compiler control lines :

`#include` directive may be used to include another file. However nesting of include files is not allowed. i.e. if file2 is #included in file1, then there may not be any `#include` statement in file2. The syntax is:

```
#include <file_name>
```

or

```
#include "file_name"
```

If the 'file_name' is enclosed in <>, then that file will be looked for in the folder ‘INC’ in the path where SC51 is installed. If 'file_name' is enclosed in "", then it will be looked for in the current folder.

`#define` and `#undef` directives may be used to define or undefine a macro. A macro may or may not have parameters. e.g.

```
#define  max_output  2048
```

This statement defines the symbol 'max_output' to the value 2048. i.e. wherever you write the word 'max_output', it will be replaced by 2048 by the pre-processor before compiling. A macro with parameters can be defined as :

```
#define  product(x,y)  (x*y)
```

Here x and y are parameters. While using the macro you can substitute the x and y with anything else. e.g. if you write :

```
value = product(factor,123.45) ;
```

it will be replaced by

```
value = factor*123.45 ;
```

`#ifdef` , `#ifndef` , `#else` and `#endif` directives may be used for conditional compiling. The syntax is :

```
#ifdef symbol_name
set of statements 1
#else
set of statements 2
#endif
```

If symbol_name is a defined macro name, then the `#ifdef` expression evaluates to true and the set of statements 1 will be compiled. Otherwise the set of statements 2 will be compiled . The `#else` and set of statements 2 is optional. The `#ifndef` expression evaluates to true, if symbol_name is not defined. Rest of the syntax is same as that for `#ifdef`.

3.10 : Additional features :

3.10.1 : Using Special Function Registers (SFR) :

You can access the SFRs of the micro-controller, by using the SFR declaration. 'SFR' is an additional keyword. It allows you to define a SFR. The syntax is :
SFR sfr_name sfr_addr
where 'sfr_name' is the name of the SFR you want to define (e.g. TMOD) and 'sfr_addr' is it's address (e.g. 0x89). There is no need to write a semicolon (;) after a SFR declaration. Thus the statement :

```
SFR  TMOD  0x89
```

defines the SFR 'TMOD' with an address 89 Hex. Once a SFR is declared, you can use it in your C program as if it were a variable of type unsigned char. e.g. you may write `TMOD = 0x11 ;` This statement will generate code to write the value 11 (Hex) in TMOD. In fact we have already declared most of the SFRs for you in different header files. e.g. the file SFR31.H contains the declarations for all 8031/8051/8751 SFRs. If you include the file SFR31.H in your C program, you can use all 8031 SFRs as variables (of unsigned char type). Similarly there are other header files, which define the SFRs for the corresponding CPUs. We have supplied number of header files which contain SFR declarations for many different 8051 derivatives. These header files are in the folder INC. There is 1 folder for each manufacturer, which contains the header

files for all 8051 derivatives manufactured by that particular manufacturer. You may include the appropriate header file in your C program. If you are using any other 8051 derivative (for which, no header file is supplied), you can create your own header file and include it in your C program.

Thus you can use declared SFRs as global variables of unsigned char type. You can use these in any expressions, except that you can not use the *&* (address of) operator with these variables, because these are not really variables, but are pseudo-variables, and these are not indirectly accessible.

3.10.2 : Assembly language interface :

Parts of a program may be very time critical, e.g. you may like an interrupt service routine to finish in as small time as is possible. For this reason, you may like to write part of the program in assembly language. This is made possible by providing assembly language patch-up facility. You may write part of the program in assembly language in a **SEPARATE FILE.** When you invoke the compiler, use the /a option to specify the .ASM file name. i.e. include `/Aasm_file_name` in the command line before the C file name. You may not use ORG statement in assembly language patch file. If you must use, the ORG statement (e.g. to ljmp to ISR), use these at the end of the assembly language file. You can access global variables declared in C program from assembly language. e.g. if 'var1' is declared as global variable in C program, then you can use the symbol '_var1' as the address of that variable. i.e. you must attach an underscore (_) to the left of the variable name. Please note that all variables reside in external data memory in case of large model (unless memory area specifier or @ notation is used). On the contrary, all variables reside in internal RAM in case of small model (unless memory area specifier or @ notation is used).

3.10.3 : The *asm* keyword :

This is an additional keyword, which allows you to insert assembly language statements directly into your C program. This keyword when used, must appear as the first word of a line. Everything following this keyword - up to the end of line - is directly copied into the .ASM output file. Please note that, no syntax checking is done by the compiler on this assembly language statement, it is done later on by the assembler.

3.10.4 : The *BIT* keyword :

The BIT keyword is an additional (non - ANSI) keyword. It can be used to define pseudo variables of bit type. e.g. suppose you are using bit 0 of P1 for some purpose - say to switch on a motor. Then you might like to assign a symbolic name to P1.0. Exactly this can be done using the BIT keyword. The syntax for using it is :

BIT *identifier_name value*

Please note that a BIT statement is **NOT** followed by a semicolon. The *identifier* can be any legal identifier name - which is not previously declared. The *value* must take this forms :

sfr_name.bit_addr

where **sfr_name** can be the name of any **bit accessible** SFR of 8031 and **bit_addr** may be any value between 0 and 7.
Here are some examples of correct and incorrect BIT statements :

```
BIT     motor     p1.0
/* correct  */

BIT     limit_switch     p3.6
/* correct  */

BIT     motor     p3
/* wrong, because the bit_addr is missing  */

BIT     something     dpl.2
/* wring, because "dpl" is not a bit accessible SFR  */
```

Once you have defined a symbol using the BIT keyword, then onwards, you can use it in your program as if it were a variable of "bit" type. e.g. if you have defined "motor" and "limit_switch" as above, you may write :

```
if (limit_switch) motor = 0 ;
else motor = 1 ;
```

The above two statements are equivalent to checking bit 6 of P3 and accordingly either setting or clearing bit 0 of P1.

3.10.5 : Memory area specifiers (data, idata, bdata, xdata, pdata, code):

The 8051 architecture uses number of logically separate memory areas. The different memory areas have been assigned names as below:

Memory Area name	Memory area description	Address range
data	Internal data memory, directly accessible (access using direct and indirect addressing mode)	00H-7FH
idata	Internal data memory, indirectly accessible (access only using indirect addressing mode)	00H-FFH
bdata	Internal data memory, bit addressable (access using direct and indirect addressing mode, as well as bit addressing capability)	20H-2FH
xdata	External data memory (access using indirect addressing, using **movx @dptr** instruction).	0000H-FFFFH
pdata	Paged External data memory (access to only one 256 byte page of external data memory, using **movx @r0** or **movx @r1** instruction).	PP00H-PPFFH where PP is page address, output on P2.
code	(Internal or External) program memory, read only access using **movc** instruction.	0000H-FFFFH

In your target hardware, the actual amount of memory available may be less than the ranges mentioned above. In some cases, some memory areas maybe entirely missing (for example, some designs do not use external data memory at all, which means **xdata** and **pdata** is missing).

The compiler and linker generally assign appropriate address to all variables (unless the address is specified by user, with @ notation). By default, all global variables are assigned addresses in **data** memory area if small memory model is used; or in **xdata** memory area if large memory model is used. It is possible to override this default and force the compiler to keep variables in other memory areas – regardless of the memory model selected. The keywords **data**, **idata**, **bdata**, **xdata**, **pdata**, **code** can be used in a variable declaration, to specify the memory area to be used for that variable. Here are some examples:

```
xdata int j ;  // this will be stored in
               // xdata memory area
idata char ch ;     // idata memory area
pdata char arr[10] ;     // pdata memory area
data unsigned char uch ; // data memory area
```

```
bdata char bch ;    // bdata memory area
code float multipliers[3] = { 1.0, 2.0, 3.0 } ;
// code memory area
long int l ;   // default memory area
```

The usage of memory area specifiers must follow some rules:

1. Only one of these memory area specifiers maybe used: **data**, **idata**, **bdata**, **xdata**, **pdata**, **code**
2. Since **code** memory area is read only, the variables declared with code memory area specifier, can not be assigned any value. The only way to assign a value to **code** variables is to initialize them during declaration (as in the above example).
3. Using the keyword **const** has exactly the same effect as using the **code** memory area specifier. Although **const** is not a memory area name, the use of **const** informs the compiler that this is a "read only" variable, and hence compiler automatically keeps it in code memory area.
4. The memory area specifier must be placed before the data type. For example, "xdata int j ;" is a valid variable declaration, but "int xdata j ;" is not valid. Generally the variable declarations should be of the form:

   [extern] [memory_area_specifier/const] data_type var_name [= value] ;

   where,

   extern is optional keyword, indicating that variable is defined in another module.

   memory_area_specifier is (optional) one of the valid memory area specifiers.

   data_type must be a valid data type.

   var_name must be a valid identifier name.

   Optionally, the variable maybe initialized.
5. Use of memory area specifier is allowed only on global variables and pointer variables. (Please see next section for more about memory area specifiers on pointers).

3.10.6 : Memory specific and generic pointers:

Use of memory area specifier on a pointer variable has different effect than non-pointer variables. As explained above, use of memory area specifier on non-pointer variables forces the variable to reside in the specified memory area. However, pointers always reside in the default memory area, even if memory area specifier is used. If a pointer is declared with memory area specifier, it becomes a memory specific pointer – which means, it can point only to the specified memory area. Such memory specific pointers are 1 byte or 2 byte pointers. If a pointer is pointing to data, idata, bdata or pdata area, it is 1 byte pointer. If a pointer is pointing to xdata or code area, it is 2 byte pointer. If a pointer is declared without

any memory area specifier, it becomes a generic pointer – which means it can point to any memory area. Generic pointers are 3 byte pointers. Here are some examples of pointer variable declarations:

```
char *p_ch ;   // generic pointer (3 byte length)
idata char *p_ch2 ; // pointer to idata memory area
                    // (1 byte length)
xdata int *p_xi ;   // pointer to xdata memory area
                    // (2 byte length)
```

Local variables and parameters are always pushed on stack – which means, they are always stored in internal data memory (regardless of memory model). Hence, memory area specifier can not be used on local variables or parameters, unless it is a pointer variable. In case of pointer variables, the memory area specifier does not affect the location of pointer variable itself, but it restricts the pointer to point only to the specified memory area. Hence, use of memory area specifier on pointer variables is always allowed – even if the pointer variable is local, global or a parameter. All library functions of SC51, which take pointer parameter(s), use generic pointers. User defined functions may use memory specific or generic pointers.

3.10.7 : Declaring variables at specific address (in Internal / External RAM) :

This compiler allows the programmer to declare a variable at a specific address. The syntax for doing so is as follows :

```
var_type    @addr   var_name ;
```

or

```
var_type    @Iaddr  var_name ;
```

where ‘var_type’ is any legal data type (except `bit`) and ‘var_name’ is any legal identifier name and *`addr`* is any hexadecimal constant. In the first case, the variable ‘var_name’ is assigned an address = *`addr`* in external RAM, and in the later case, the variable ‘var_name’ is assigned an address = *`addr`* in internal RAM. Thus in a variable declaration, if the variable type is followed by @ followed by a hexadecimal constant address, then the variable will reside at the specified address in external RAM. On the other hand, if the variable type is followed by @I (there should not be any space between @ and I) followed by a hexadecimal constant address, then the variable will reside at the specified address in internal RAM. This is further illustrated in following example :

```
unsigned char @0x6000 dat_8279 ;
/* i.e. &dat_8279 is 0x6000 in external RAM */
unsigned char @0x6001 cmd_8279 ;
/* i.e. &cmd_8279 is 0x6001 in external RAM*/
int @I0x2d var1 ;
```

```
  /* i.e. &var1 is 0x2d in Internal RAM */
  unsigned char ch ;
  /* this is ordinary variable. It's address will be
     decided by the compiler. Also, whether it will
     reside in internal RAM or external RAM will
depend
     on the memory model used */
  void main () {
     cmd_8279 = ch ;
     /* an alternative without using @ would be
        outportb(0x6001,ch) ; */
     var1 = dat_8279 ;
     /* an alternative without using @ would be
        var1 = inportb(0x6000) ; */
  }
```

Thus declaring variables at a specific address results in a great amount of saving in code size and also the clock cycles required. However using 'outportb' and 'inportb' functions is also perfectly legal.

By using this feature, you can declare variables at any location in internal or external RAM. Thus, even if you are using small memory model, still you can forcefully keep some variables in external RAM at any address. Also, even if you are using large memory model, you can forcefully keep some variables in internal RAM at any address.

However, the compiler does NOT check whether the specified address is used by some other variable or temporary variable or stack or registers. Thus, it is the PROGRAMMER's RESPONSIBILITY to specify a valid address, which is not used for **any other purpose**.

3.10.8 : *interrupt* functions and the *using* directive:

The SC51 compiler supports a special type of functions – **interrupt** functions. As the name implies, these are Interrupt Service Routines (ISRs). They are different from ordinary functions in many respects:

- Interrupt functions are not called by user program, but are automatically invoked by hardware, when the corresponding "interrupt event" occurs. As a result, interrupt functions must not have any parameters.
- For the same reason, interrupt functions can not return a value.
- All interrupt functions are associated with a vector location. A jump instruction must be placed at the vector location, such that it will transfer control to the interrupt function.

- Interrupt functions must end with a RETI instruction, rather than RET instruction.
- At the beginning of interrupt functions, important registers such as accumulator or flags must be saved; and these must be restored just before the end of interrupt function. It is also usual to change register bank. In case of SC51 programs, all non-interrupt functions use register bank0. One another register bank must be reserved for each priority level of interrupts.

Due to the above differences, it is necessary to use somewhat different syntax to declare an interrupt function:

```
interrupt (INTR_NUM) FuncName () using N
{
        // write your code here
}
```

where:

interrupt is a keyword that informs the compiler, that this is an interrupt function rather than ordinary function.

INTR_NUM can be a constant in the range 1 to 32, which identifies the interrupt number – and consequently, the vector location associated with it. The vector location address associated is (((INTR_NUM – 1) * 8) + 3). Thus INTR_NUM = 1 corresponds to External Interrupt0 (EXT0) whose vector address is 0003H. INTR_NUM = 2 corresponds to Timer0 (T0) interrupt whose vector address is 000BH. Symbolic constants (such as INT_EXT0, INT_TMR0) are defined in the file STANDARD.H. We strongly recommend that you should #include <standard.h> in your program and use the appropriate symbolic constant as INTR_NUM.

FuncName is the name of the interrupt function – it must follow the usual rules for identifier name.

using is the keyword required to specify the register bank to be used for this interrupt function.

N is the register bank number, to be used for this interrupt function. It is legal to specify any value in the range 0 to 3 here. However, it is strongly recommended to use only values 1 to 3, because register bank0 is used by all non-interrupt functions (hence interrupt function must not use it). <u>If incorrect register bank is specified, the entire program may malfunction</u>.

The SC51 compiler treats an interrupt function, differently than ordinary functions:

- The SC51 compiler will automatically insert a jump instruction at the vector location, which points to this interrupt function.
- At the beginning of interrupt function, the compiler automatically generates code to save important “context” (for example accumulator, flags and so on).

The compiler also generates instruction to switch register bank to the specified value.

- At the end of interrupt function, the compiler automatically generates code to restore the saved context (and switch back to original register bank).
- The compiler uses RETI instruction (rather than RET) to terminate the interrupt function.

Please note, an interrupt function can not have any parameters.

3.10.9 : Absolute Register Addressing:

The SC51 compiler generates Assembler instructions that often use registers R0…R7. These can be accessed as "registers" (with the names R0…R7) or as "direct addresses in data memory". In the later case, the compiler uses pre-defined register address symbols (AR0…AR7). Depending on the currently selected register bank, the ARn symbol translates to appropriate address. For example, if bank0 is selected, then AR4 translates into 04H. If bank1 is selected, AR2 translates into 0AH. In this case, the function will work correctly, only if the same register bank is selected at runtime. This maybe a problem in case of some functions. For example, consider a function f1() which is called by main() as well as an Interrupt function. By default, register bank0 is selected for all non-interrupt functions. But Interrupt functions must select another register bank (with the **using** keyword). Thus when function f1 is called, sometimes register bank0 will be selected, but at some other times, another register bank maybe selected. Obviously, if "absolute register addressing" is used during function f1, it will result in malfunctioning. The SC51 provides a way to disable or enable absolute register addressing. The controls NOAREGS and AREGS can be used in conjunction with #pragma. The syntax is as follows:

```
#pragma     NOAREGS
// this point onwards, absolute register addressing will be disabled
// write functions like f1 here
// so that absolute register addressing will not be used
#pragma     AREGS
// this point onwards, absolute register addressing will be enabled
```

By default, absolute register addressing is enabled.

4. SC51 Library Functions Reference

List of functions in alphabetical order :

1) abs
2) acos
3) asin
4) atof
5) atoi
6) atol
7) bcd2int
8) ceil
9) clr_double_baud
10) clr_ri
11) clr_ti
12) cos
13) cosh
14) delay
15) delay_ms
16) disable
17) disable_all
18) disable_ex0
19) disable_ex1
20) disable_rx
21) disable_ser
22) disable_t0
23) disable_t1
24) disable_t2
25) disp_lcd
26) enable
27) enable_all
28) enable_ex0
29) enable_ex1
30) enable_rx
31) enable_ser
32) enable_t0
33) enable_t1

34) enable_t2
35) ex0_edge
36) ex0_level
37) ex1_edge
38) ex1_level
39) exp
40) floor
41) fmod
42) frexp
43) getbyte
44) getch
45) getchar
46) getchare
47) go_idle
48) hi_nibb
49) init_ser
50) inportb
51) int2bcd
52) isalnum
53) isalpha
54) isascii
55) isdigit
56) islower
57) isspace
58) isupper
59) isxdigit
60) itoa_c31
61) labs
62) ldexp
63) log
64) log10
65) long2bcd
66) lo_nibb
67) ltoa_c31
68) memset
69) modf
70) movmem
71) outportb
72) peekb
73) pokeb

74) pow
75) powerdown
76) printf
77) putchar
78) puts
79) scanf
80) sendbyte
81) ser_rdy
82) set_com_mode
83) set_double_baud
84) set_hi_ex0
85) set_hi_ex1
86) set_hi_ser
87) set_hi_t0
88) set_hi_t1
89) set_hi_t2
90) set_lo_ex0
91) set_lo_ex1
92) set_lo_ser
93) set_lo_t0
94) set_lo_t1
95) set_lo_t2
96) set_t0_mode
97) set_t1_mode
98) set_tcnt
99) sin
100)sinh
101)sprintf
102)sqrt
103)sscanf
104)start_timer0
105)start_timer1
106)stop_timer0
107)stop_timer1
108)strcat
109)strcmp
110)strcpy
111)strlen
112)strlwr
113)strncpy

114)strupr
115)tan
116)tanh
117)tolower
118)toupper
119)ui2a_c31
120)ui2bcd

---

abs

Function : Returns the absolute value of an integer
Syntax : `#include <stdlib.h>`
`int abs(int x) ;`
Prototype in : math.h
Remarks : abs returns the absolute value of an integer argument x.
Return value : abs returns an integer in the range of 0 to 32767, with the exception that an argument of 32768 is returned as -32768.
See also : labs

---

acos

Function : Returns the arc cosine of the argument.
Syntax : `#include <math.h>`
`float acos(float x) ;`
Prototype in : math.h
Remarks : This function is NOT included in the library, but it is given in the form of source code. It's source code can be found in SOURCE\CFILES\MATH.C
Return value : acos returns the arc cosine of x.
See also : cos

---

asin

Function : Returns the arc sine of the argument.
Syntax : `#include <math.h>`
`float asin(float x) ;`
Prototype in : math.h
Remarks : This function is NOT included in the library, but it is given in the form of source code. It's source code can be found in SOURCE\CFILES\MATH.C
Return value : asin returns the arc sine of x.
See also : sin

---

atof

Function : Converts ASCII string to floating point number

Syntax :
```
#include <stdlib.h>
float atof(char *str) ;
```

Prototype in : stdlib.h

Remarks : atof converts the ASCII string str into a floating point number.

Return value : atof returns the converted floating point value. If the string can not be converted, returns 0.

See also : atol, atoi

---

atoi

Function : Converts ASCII string to integer number

Syntax :
```
#include <stdlib.h>
int atoi(char *str) ;
```

Prototype in : stdlib.h

Remarks : atoi converts the ASCII string str into a integer number.

Return value : atoi returns the converted integer value. If the string can not be converted, returns 0.

See also : atol, atof

---

atol

Function : Converts ASCII string to long integer number

Syntax :
```
#include <stdlib.h>
long int atol(char *str) ;
```

Prototype in : stdlib.h

Remarks : atol converts the ASCII string str into a long integer number.

Return value : atol returns the long integer value. If the string can not be converted, returns 0.

See also : atoi, atof

---

bcd2int

Function : Converts BCD string to integer number
Syntax : `#include <etc.h>`
`int bcd2int (char str[10], int ndigits) ;`
Prototype in : etc.h
Remarks : bcd2int converts the unpacked BCD string str into an integer number. It considers the first 'ndigits' digits of str.
Return value : bcd2int returns the integer value.
See also : int2bcd

---

ceil

Function : Rounds up.
Syntax : `#include <math.h>`
`float ceil (float x) ;`
Prototype in : math.h
Return value : ceil returns the smallest integer greater that or equal to x.
See also : floor

---

clr_double_baud

Function : Clears 'double baud rate'
Syntax : `#include <macros31.h>`
`clr_double_baud() ;`
Prototype in : macros31.h
Remarks : Clears bit 7 of PCON so that baud rate is not doubled.
Return value : None.
See also : set_double_baud

---

clr_ri

Function : Clears receiver interrupt

Syntax : `#include <macros31.h>`
`clr_ri() ;`
Prototype in : macros31.h
Remarks : Clears RI bit in SCON, so that serial port's pending receiver interrupt is cleared.
Return value : None.
See also : clr_ti

---

clr_ti

Function : Clears transmitter interrupt
Syntax : `#include <macros31.h>`
`clr_ti() ;`
Prototype in : macros31.h
Remarks : Clears TI bit in SCON, so that serial port's pending transmitter interrupt is cleared.
Return value : None.
See also : clr_ri

---

cos

Function : Returns the arc cosine of the argument.
Syntax : `#include <math.h>`
`float cos(float x) ;`
Prototype in : math.h
Remarks : This function is NOT included in the library, but it is given in the form of source code. It's source code can be found in SOURCE\CFILES\MATH.C
Return value : cos returns the cosine of x.
See also : acos

---

cosh

Function : Returns the hyperbolic cosine of the argument.

Syntax : `#include <math.h>`
`float cosh(float x) ;`
Prototype in : math.h
Remarks : This function is NOT included in the library, but it is given in the form of source code. It's source code can be found in SOURCE\CFILES\MATH.C
Return value : cosh returns the hyperbolic cosine of x.
See also : cos

---

delay

Function : Produces a delay
Syntax : `#include <standard.h>`
`void delay (int count) ;`
Prototype in : standard.h
Remarks : Produces a certain delay. Since the crystal frequency in the target system is not known, this function passes 'count' number of clock cycles only. Thus if you are using 8031 with 12 MHz crystal, you may say, that this function produces a delay of 'count' micro-seconds where 'count' is the integer parameter you pass to it. However if you are using a different crystal frequency and / or different CPU, it may not be so. The accuracy of this function is plus or minus 50 cycles. i.e. if you call delay function with count parameter = 20000, then it will return after 'n' clock cycles where 'n' will be > 19050 and < 20050.
Also please note, that this functions produces a software delay. Thus, if interrupts are enabled, then this function may produce more delay than is expected. Because, if interrupt occurs during this function, then it will take some more time to execute but that time will not be counted by this function. Thus, if interrupts are enabled, the acuracy and repeatability of this function will be poor.
Return value : None.

---

delay_ms

Function : Produces a delay
Syntax : `#include <standard.h>`
`void delay_ms (int count) ;`
Prototype in : standard.h

Remarks : Produces a certain delay. Since the crystal frequency in the target system is not known, this function passes 'count * 1000' number of clock cycles only. Thus if you are using 8031 with 12 MHz crystal, you may say, that this function produces a delay of 'count' milli-seconds where 'count' is the integer parameter you pass to it. However if you are using a different crystal frequency and / or different CPU, it may not be so. The accuracy of this function is plus or minus 2 cycles. i.e. if you call delay function with count parameter = 20, then it will return after 'n' clock cycles where 'n' will be >= 19998 and <= 20002.
Also please note, that this functions produces a software delay. Thus, if interrupts are enabled, then this function may produce more delay than is expected. Because, if interrupt occurs during this function, then it will take some more time to execute but that time will not be counted by this function. Thus, if interrupts are enabled, the acuracy and repeatability of this function will be poor.
Return value : None.

---

disable

Function : Disables interrupts
Syntax :
```
#include <macros31.h>
disable() ;
```
Prototype in : macros31.h
Remarks : Clears only bit 7 of IE, so that all interrupts remain disabled.
Return value : None.
See also : disable_all, enable

---

disable_all

Function : Disables interrupts
Syntax :
```
#include <macros31.h>
disable_all() ;
```
Prototype in : macros31.h
Remarks : Clears all bits of IE so that all interrupts remain disabled. Please note the subtle difference between disable() and disable_all().
Return value : None.

See also : disable, enable_all

---

disable_ex0

Function : Disables INT0 external interrupt.
Syntax : `#include <macros31.h>`
`disable_ex0() ;`
Prototype in : macros31.h
Remarks : Clears bit 0 of IE, so that external interrupt INT0 remains disabled.
Return value : None.
See also : enable_ex0

---

disable_ex1

Function : Disables INT1 external interrupt.
Syntax : `#include <macros31.h>`
`disable_ex1() ;`
Prototype in : macros31.h
Remarks : Clears bit 2 of IE, so that external interrupt INT1 remains disabled.
Return value : None.
See also : enable_ex1

---

disable_rx

Function : Disables receiver of serial port
Syntax : `#include <macros31.h>`
`disable_rx() ;`
Prototype in : macros31.h
Remarks : Clears bit 4 of SCON so that serial port's receiver remains disabled.
Return value : None.
See also : enable_rx

---

disable_ser

Function : Disables serial port interrupt.
Syntax : `#include <macros31.h>`
`disable_ser() ;`
Prototype in : macros31.h
Remarks : Clears bit 4 of IE, so that serial port interrupt remains disabled.
Return value : None.
See also : enable_ser

---

disable_t0

Function : Disables timer 0 interrupt.
Syntax : `#include <macros31.h>`
`disable_t0() ;`
Prototype in : macros31.h
Remarks : Clears bit 1 of IE, so that Timer 0 interrupt remains disabled.
Return value : None.
See also : enable_t0

---

disable_t1

Function : Disables timer 1 interrupt.
Syntax : `#include <macros31.h>`
`disable_t1() ;`
Prototype in : macros31.h
Remarks : Clears bit 3 of IE, so that Timer 1 interrupt remains disabled.
Return value : None.
See also : enable_t1

---

disable_t2

Function : Disables timer 2 interrupt.
Syntax : `#include <macros31.h>`
`disable_t2() ;`
Prototype in : macros31.h
Remarks : Clears bit 5 of IE, so that Timer 2 interrupt remains disabled.
Return value : None.
See also : enable_t2

---

enable

Function : Enables interrupts
Syntax : `#include <macros31.h>`
`enable() ;`
Prototype in : macros31.h
Remarks : Sets only bit 7 of IE, so that all interrupts may be enabled.
Return value : None.
See also : enable_all, disable

---

enable_all

Function : Enables interrupts
Syntax : `#include <macros31.h>`
`enable_all() ;`
Prototype in : macros31.h
Remarks : Sets all bits of IE so that all interrupts are enabled. Please note the subtle difference between enable() and enable_all().
Return value : None.
See also : enable, disable_all

---

enable_ex0

Function : Enables INT0 external interrupt.
Syntax : `#include <macros31.h>`
`enable_ex0() ;`
Prototype in : macros31.h
Remarks : Sets bit 0 of IE, so that external interrupt INT0 gets enabled.
Return value : None.
See also : disable_ex0

---

enable_ex1

Function : Enables INT1 external interrupt.
Syntax : `#include <macros31.h>`
`enable_ex1() ;`
Prototype in : macros31.h
Remarks : Sets bit 2 of IE, so that external interrupt INT1 gets enabled.
Return value : None.
See also : disable_ex1

---

enable_rx

Function : Enables receiver of serial port
Syntax : `#include <macros31.h>`
`enable_rx() ;`
Prototype in : macros31.h
Remarks : Sets bit 4 of SCON so that serial port's receiver gets enabled.
Return value : None.
See also : disable_rx

---

enable_ser

Function : Enables serial port interrupt.
Syntax : `#include <macros31.h>`
`enable_ser() ;`
Prototype in : macros31.h
Remarks : Sets bit 4 of IE, so that serial port interrupt gets enabled.
Return value : None.
See also : disable_ser

---

enable_t0

Function : Enables timer 0 interrupt.
Syntax : `#include <macros31.h>`
`enable_t0() ;`
Prototype in : macros31.h
Remarks : Sets bit 1 of IE, so that Timer 0 interrupt gets enabled.
Return value : None.
See also : disable_t0

---

enable_t1

Function : Enables timer 1 interrupt.
Syntax : `#include <macros31.h>`
`enable_t1() ;`
Prototype in : macros31.h
Remarks : Sets bit 3 of IE, so that Timer 1 interrupt gets enabled.
Return value : None.
See also : disable_t1

---

enable_t2

Function : Enables timer 2 interrupt.
Syntax : `#include <macros31.h>`
`enable_t2() ;`
Prototype in : macros31.h
Remarks : Sets bit 5 of IE, so that Timer 2 interrupt gets enabled.
Return value : None.
See also : disable_t2

---

ex0_edge

Function : Makes INT0 edge sensitive.
Syntax : `#include <macros31.h>`
`ex0_edge() ;`
Prototype in : macros31.h
Remarks : Sets bit 0 of TCON, so that external interrupt INT0 becomes edge sensitive.
Return value : None.
See also : ex0_level

---

ex0_level

Function : Makes INT0 level sensitive.
Syntax : `#include <macros31.h>`
`ex0_level() ;`
Prototype in : macros31.h
Remarks : Clears bit 0 of TCON, so that external interrupt INT0 becomes level sensitive.
Return value : None.
See also : ex0_edge

---

ex1_edge

Function : Makes INT1 edge sensitive.
Syntax :
```
#include <macros31.h>
ex1_edge() ;
```
Prototype in : macros31.h
Remarks : Sets bit 2 of TCON, so that external interrupt INT1 becomes edge sensitive.
Return value : None.
See also : ex1_level

---

ex1_level

Function : Makes INT1 level sensitive.
Syntax :
```
#include <macros31.h>
ex1_level() ;
```
Prototype in : macros31.h
Remarks : Clears bit 2 of TCON, so that external interrupt INT1 becomes level sensitive.
Return value : None.
See also : ex1_edge

---

exp

Function : exponential function
Syntax :
```
#include <math.h>
float exp (float x) ;
```
Related Functions :
```
float frexp (float value, int *eptr) ;
float ldexp (float value, int exp) ;
float log (float x) ;
float log10 (float x) ;
float pow (float x, float y) ;
float sqrt (float x) ;
```
Prototype in : math.h
Remarks : **exp** calculates the exponential function e^x.

frexp calculates the mantissa x (a float less than 1) and n (an integer) such that *value = x . 2^n*. **frexp** stores n in the integer that eptr points to.
ldexp calculates *value . 2^{exp}*.
log calculates natural logarithm of x.
log10 calculates the base 10 logarithm of x.
pow calculates x^y.
sqrt calculates $+\ddot{O}x$.

Return value : All these functions on success, return the value they calculated.
exp returns e^x.
frexp returns x (< 1) where *value = x . 2^n*.
ldexp returns *value . 2^{exp}*.
log returns natural logarithm of x.
log10 returns the base 10 logarithm of x.
pow returns x^y.
sqrt returns $+\ddot{O}x$.

---

floor

Function : rounds down
Syntax : `float floor (float value) ;`
Prototype in : math.h
Return value : **floor** finds the largest integer not greater than x. It returns this integer (as a float value).
See also : ceil

---

fmod

Function : Calculates x modulo y, the remainder of x / y.
Syntax : `float fmod (float x, float y) ;`
Related functions :

```
float modf (float x, float *ipart) ;
```

Prototype in : math.h
Remarks : **fmod** calculates x modulo y (the remainder f where x = iy + f for some integer I and 0 <= f < y.

modf breaks x into two parts: the integer and the fraction. It stores the integer in ipart and returns the fraction.
Return value : **fmod** returns the remainder f where x = iy + f (as described)
modf returns the fractional part of x.

---

frexp

Function : Splits a float number into mantissa and exponent.
Syntax : `float frexp (float number, int *power) ;`
Prototype in : math.h
Remarks : See **exp.**

---

getbyte

Function : Reads a byte from the serial port.
Syntax :
```
#include <standard.h>
int getbyte() ;
```
Prototype in : standard.h
Remarks : Waits for some time or until a byte is available from the on-chip serial port of the 8031.
Return value : If a byte was available, returns the received byte. Otherwise returns -1.
See also : ser_rdy, sendbyte

---

getch

Function : Reads a byte from the 8279 based keyboard.
Syntax :
```
#include <hardware.h>
int getch() ;
```
Prototype in : hardware.h
Remarks : Waits till a byte is available from the keyboard which is connected to 8279. When a key is pressed, reads it's scancode from 8279. If you want to use this function, you must declare the address of 8279 using the `peripheral` keyword. e.g.
```
peripheral    8279  =  0x4000
```

Return value : Returns the scancode read from 8279.
See also : kbhit

---

getchar

Function : Reads a character from standard input device.
Syntax :
```
#include <stdio.h>
char getchar () ;
```
Prototype in : stdio.h
Remarks : Waits till a character is available from the standard input device (which is on-chip serial port of 8051). When a character is available, reads it. This function is called by *scanf* function to get input.
Return value : Returns the character read from standard input device.
See also : scanf, getchare

---

getchare

Function : Reads a character from standard input device.
Syntax :
```
#include <standard.h>
char getchare () ;
```
Prototype in : stdio.h
Remarks : Waits till a character is available from the standard input device (which is on-chip serial port of 8051). When a character is available, reads it and also echoes it back to the standard output device (which is again on-chip serial port of 8051, unless the user redirects it by writing his own *putchar* function).
Return value : Returns the character read from standard input device.
See also : scanf, getchar

---

go_idle

Function : Puts the CPU in idle mode.
Syntax :
```
#include <macros31.h>
go_idle() ;
```
Prototype in : macros31.h

Remarks : Sets bit 0 of PCON, so that the CPU goes to idle mode in which it consumes less power.
Return value : None.
See also : powerdown

---

hi_nibb

Function : Separates the high order nibble
Syntax : unsigned char hi_nibb (unsigned char ch) ;
Prototype in : standard.h
Return value : Returns the high order nibble of ch.
See also : lo_nibb

---

init_ser

Function : Initializes the on-chip serial port
Syntax : void init_lcd() ;
Prototype in : standard.h
Remarks : It initializes the serial port for a certain mode. If you are going to use `printf` function, you must first initialize the serial port. Calling this `init_ser` function is just one method of initializing the serial port.
This function puts the Timer1 in 8 bit auto-reload mode, and it puts the serial port in mode 1 with receiver enabled. It sets the baudrate at 2400 assuming 12 Mhz crystal. The source code of this function is available to the users.
Return value : None.
See also : printf

---

inportb

Function : Reads a byte from a hardware port.
Syntax : unsigned char inportb(int portid) ;
Prototype in : standard.h
Remarks : inportb reads a byte from a location in data memory whose address is specified by portid.
Return value : inportb returns the value read.

See also : peekb, outportb

---

int2bcd

Function : Converts an integer number into a BCD string.
Syntax : void int2bcd (int value, char *dest, int ndigits) ;
Prototype in : standard.h
Remarks : Converts value into an unpacked BCD string which is 'ndigits' digits long and places the string in 'dest'
Return value : None
See also : bcd2int

---

isalnum

Function : Character classification function
Syntax :
```
#include <ctype.h>
bit isalnum (char c) ;
```
Prototype in : ctype.h
Remarks : isalnum is a function that classifies ASCII-coded character values. It returns 1 for true and 0 for false.
Return value : isalnum returns 1 if c is a letter (A-Z or a-z) or a digit (0-9).

---

isalpha

Function : Character classification function
Syntax :
```
#include <ctype.h>
bit isalpha (char c) ;
```
Prototype in : ctype.h
Remarks : isalpha is a function that classifies ASCII-coded integer values. It returns 1 for true and 0 for false.
Return value : isalpha returns 1 if c is a letter (A-Z or a-z).

---

isascii

Function : Character classification function
Syntax : `#include <ctype.h>`
`bit isascii (char c) ;`
Prototype in : ctype.h
Remarks : isascii is a function that classifies ASCII-coded character values. It returns 1 for true and 0 for false.
Return value : isascii returns 1 if c is in the range 32-127 (0x20-0x7F).

---

isdigit

Function : Character classification function
Syntax : `#include <ctype.h>`
`bit isdigit(char c) ;`
Prototype in : ctype.h
Remarks : isdigit is a function that classifies ASCII- coded character values. It returns 1 for true and 0 for false.
Return value : isdigit returns 1 if c is a digit ('0'- 9').

---

islower

Function : Character classification function
Syntax : `#include <ctype.h>`
`bit islower (char c) ;`
Prototype in : ctype.h
Remarks : islower is a function that classifies ASCII-coded character values. It returns 1 for true and 0 for false.
Return value : islower returns 1 if c is a lower-case letter (‘a’-‘z’).

---

isspace

Function : Character classification function
Syntax : `#include <ctype.h>`

```
bit isspace (char  c)  ;
```
Prototype in : ctype.h
Remarks : isspace is a function that classifies ASCII-coded character values. It returns 1 for true and 0 for false.
Return value : isspace returns 1 if c is a space, tab, carriage return, or newline.

---

isupper

Function : Character classification function
Syntax :
```
#include  <ctype.h>
bit isupper (char  c)  ;
```
Prototype in : ctype.h
Remarks : isupper is a function that classifies ASCII-coded character values. It returns 1 for true and 0 for false.
Return value : isupper returns 1 if c is an uppercase letter (‘A’-‘Z’).

---

isxdigit

Function : Character classification function
Syntax :
```
#include  <ctype.h>
bit isxdigit (char  c)  ;
```
Prototype in : ctype.h
Remarks : isxdigit is a function that classifies ASCII-coded character values. It returns 1 for true and 0 for false.
Return value : isxdigit returns 1 if c is a hexadecimal digit (‘0’- 9’,‘A’-‘F’,‘a’-‘f’).

---

itoa_c31

Function : Converts an integer into ASCII string
Syntax :
```
#include  <stdlib.h>
void itoa_c31(int val, char *dest, int ndigits) ;
```
Prototype in : stdlib.h
Remarks : itoa_c31 converts an integer number 'val' into ASCII string 'dest' containing exactly ndigits digits.

Return value : None.
See also : ltoa_c31

---

labs

Function : Gives long absolute value.
Syntax : `#include <stdlib.h>`
`long int labs(long int x) ;`
Prototype in : stdlib.h
Remarks : labs computes the absolute value of the parameter x.
Return value : On success, labs returns the absolute value of x. There is no error return.
See also : abs

---

ldexp

Function : Calculates value * 2^{power}.
Syntax : `float ldexp (float value, int power) ;`
Prototype in : math.h
Remarks : See **exp.**

---

log

Function : Calculates natural logarithm of value.
Syntax : `float log (float value) ;`
Prototype in : math.h
Remarks : See **exp.**

---

log10

Function : Calculates base 10 logarithm of value.

Syntax : `float log10 (float value) ;`
Prototype in : math.h
Remarks : See **exp.**

---

long2bcd

Function : Converts a long int number into BCD string.
Syntax : `#include <stdlib.h>`
`void long2bcd (long int value, char *dest, int ndigits) ;`
Prototype in : stdlib.h
Remarks : long2bcd converts value into an unpacked BCD string having ‘ndigits’ digits. It puts the BCD string in ‘dest’.
Return value : None
See also : int2bcd

---

lo_nibb

Function : Separates the low order nibble
Syntax : unsigned char lo_nibb (unsigned char ch) ;
Prototype in : standard.h
Return value : Returns the low order nibble of ch.
See also : hi_nibb

---

ltoa_c31

Function : Converts a long integer into ASCII string
Syntax : `#include <stdlib.h>`
`void ltoa_c31(long int val, char *dest, int ndigits) ;`
Prototype in : stdlib.h
Remarks : ltoa_c31 converts a long integer number 'val' into ASCII string 'dest' containing exactly ‘ndigits’ digits.
Return value : None.
See also : itoa_c31

---

memset

Function : Sets n bytes a block of memory to byte c.
Syntax : `#include <mem.h>`
`void *memset (void *s, int c, int n) ;`
Prototype in : mem.h, string.h
Remarks : memset sets the first n bytes of the array s to the character c.
Return value : memset returns s.

---

modf

Function : Splits into mantissa and exponent.
Syntax : `float modf (float x, float *ipart) ;`
Prototype in : math.h
Remarks : See **fmod.**

---

movmem

Function : Copies a block of length bytes.
Syntax : `#include <mem.h>`
`void movmem (void *src, void *dest, unsigned length) ;`
Prototype in : mem.h
Remarks : movmem copies a block of length bytes from src to dest.
Return value : None.

---

outportb

Function : Outputs a byte to a hardware port.
Syntax : `void outportb (int portid, unsigned int value) ;`
Prototype in : standard.h
Remarks : outportb is a function that writes the byte given by value to the location in data memory whose address is specified by portid.
Return value : None.
See also : inportb, pokeb

---

peekb

Function : Returns a byte of memory.
Syntax : `char peekb (unsigned addr) ;`
Prototype in : standard.h
Remarks : peekb returns the byte at the location in program memory whose address is specified by addr. Please note the difference between inportb and peekb functions. inportb reads from data memory whereas peekb reads from program memory.
Return value : peekb returns the value read.
See also : pokeb, inportb

---

pokeb

Function : Writes a byte to a memory location
Syntax : `void pokeb(int addr, unsigned int value) ;`
Prototype in : standard.h
Remarks : pokeb is a function that writes the byte given by value to the location in data memory whose address is specified by addr. Actually the functions outportb and pokeb are exactly equivalent.
Return value : None.
See also : peekb, outportb

---

pow

Function : Computes x to the power of y.
Syntax :
```
#include <math.h>
float pow (float x, float y) ;
```
Prototype in : math.h
Remarks : See **exp**.

---

powerdown

Function : Puts the CPU in powerdown mode.
Syntax : `#include <macros31.h>`
`powerdown() ;`
Prototype in : macros31.h
Remarks : Sets bit 1 of PCON, so that the CPU goes to powerdown mode in which it consumes much less power.
Return value : None.
See also : go_idle

---

printf

Function : Sends formatted output to the standard output device
Syntax : `int printf(char *format, ...) ;`
Related Functions :
`int sprintf (char *dest, char *format, ...) ;`
Prototype in : stdio.h
Remarks : The `printf` function sends formatted output to the standard output device (which is the 8051's on-chip serial port in case of this compiler). The *sprintf* function is same as *printf* function, except, it sends the output to the argument dest instead of to the standard output device. The parameter `format` specifies how the output is to be formatted. This parameter is mandatory. After that, there may be a variable number of parameters.
Format specifications have the following form :

```
% [width] [.prec] [l/b] type
```

where

`[flags]` is an optional sequence of flag characters
`[width]` is an optional width specifier
`[.prec]` is an optional precision specifier
`[l/b]` is an optional input size modifier
`type` is the conversion type character

Here is a list of conversion type characters, the type of input argument accepted by each, and in what format the output will appear (assuming no flag characters, width specifiers, precision specifiers, or input size modifiers were included in the format specification). The effect of optional characters and modifiers is described later.

Type Character	Input Argument	Format of output
d	integer(int / unsigned int or long int / unsigned long int)	signed decimal integer
i	integer(int / unsigned int or long int / unsigned long int)	signed decimal integer
u	integer(int / unsigned int or long int / unsigned long int)	unsigned decimal integer
f	floating point (float)	signed value in the form dddd.ddd
c	character (char)	single character
s	string pointer	outputs characters until a null terminator is found
%	none	the '%' character is printed

The optional flag character **-** can be used to get left justified output. If this is not specified, the output is right justified.

How the width specifier affects the output :

Width Specifier	How output width is affected
n	n characters are always printed. If the output value has less than n characters, the output is padded with blank. If the output value has more than n characters, then it is truncated (not rounded) to first n characters only.
0n	n characters are always printed. If the output value has less than n characters, the output is padded with zero. If the output value has more than n characters, then it is truncated (not rounded) to first n characters only.

However, please note that in this implementation of the compiler, the maximum allowed value of the width specifier is 10.

How the precision specifier affects the output : The precision specifier affects the output of only floating point values. It determines, how many digits will be printed after the decimal point. If the precision is not specified at all, then a floating point value will be printed with 3 digits after the decimal point. If the precision is specified as .n then n digits will be printed after the decimal point. If the output value has more than n digits after the decimal point, then the output will be truncated (NOT rounded). If it has less than n digits, then the output is padded with 0. However, please note that in this implementation of the compiler, the maximum allowed value of the precision specifier is 6.

The optional input size modifier – `l` or `b` – applies only if the type character is `d` or `i` or `u`. When input size modifier is not used, these type characters need a 2 byte argument and it is interpreted as signed or unsigned integer. If input size modifier `l` is specified (as in "`%ld`" or "`%lu`") then a 4 byte argument (`long int` or `unsigned long int`) is needed. On the other hand, if input size modifier `b` is used (as in "`%bd`" or "`%bu`") then single byte (`char` or `unsigned char`) argument is needed.

In this implementation of the compiler, the on-chip serial port of 8051 is considered as the standard output device. However, the user can change this with little effort. The `printf` function actually calls the function `putchar` to output each character. The source code of the `putchar` function is available to the users. If you want the standard output device to be something else (say LCD), you simply need to rewrite the `putchar` function. If you have written a function named as `putchar` somewhere in your C program, then the compiler will not link the default `putchar` function from the library. Then if you call `printf` function, it eventually calls `putchar` to send the output. In this case ***your*** `putchar` function will be called rather than the default `putchar` function (which is there in the library).

Return value : The printf functions return the number of bytes sent to the standard output device / to destination string.
See also : sprintf, putchar. Also, please look at the file PRINT.C in EXAMPLES\PRINT folder.

---

putchar

Function : Sends a byte to the standard output device (on-chip serial port.)
Syntax :
```
#include  <stdio.h>
char putchar (char ch) ;
```

Prototype in : stdio.h
Remarks : Sends the character 'ch' to standard output device - i.e. on-chip serial port of 8051 (i.e. SBUF). It also waits till the transmission is complete. That is, it will wait till the TI bit in SCON is set. Before returning, it clears the TI bit, so that you are ready to send the next byte.
Please note, that as per conventions, it checks whether the character being sent is Line Feed character (0x0a). If so, it also sends the Carriage Return character (0x0d) to the standard output device.
The functions `printf` and `puts` call this function to send out each character. The source code of this function (in assembly language) is available to the users. The users may change this function as per their requirement, so as to direct the standard output to something else rather than the serial port.
Return value : Returns the character 'ch'.
See also : printf, puts

---

puts

Function : Sends a character string to standard output device
Syntax :
```
#include <stdio.h>
char puts (char arr[]) ;
```
Prototype in : stdio.h
Remarks : Sends each character of the string 'arr' to standard output device (i.e. normally the on-chip serial port), until a null terminator character is found, (the null character is NOT output), and then output a newline character. Please note, that it calls the `putchar` function to output each character. Thus, if `putchar` function is re-written by the user to send the output to some other device, the output of `puts` will also go to the new device.
Return value : Returns the last character written.
See also : putchar

---

scanf

Function : Performs formatted input.
Syntax :
```
#include <stdio.h>
int scanf (char *format, ...) ;
```
Related functions :

```
int sscanf (char *src, char *format, ...) ;
```

Prototype in : stdio.h

Remarks : The scanf family of functions scan input fields, one character at a time, and convert them according to a given format. They accept a format string (i.e. the *format* parameter) that determines how the input fields are to be interpreted. They apply the format string to a variable number of input fields in order to format the input. Then they store the formatted input in the addresses given in arguments after the *format*.

When it encounters it's first format specification in the format string, it scans and converts the first input field according to that specification, then stores the result in the location given by the first address argument; it then scans, converts and stores the second input field; then the third etc.

The **scanf** function gets input from the standard input device i.e. normally the on-chip serial port. It actually calls the getchar function to read each character from the standard input device. If you write your own getchar function to read one character from some other device, then scanf takes all it's input from this new device.

The **sscanf** function reads the input from the first argument *src*.

The format string :

The format string, present in both **scanf** and **sscanf**, controls how each function will scan, convert and store it's input fields. There must be enough address arguments for the given format specifications; if not the results are unpredictable and likely to disturb the entire system. Excess address arguments (more than required by the format) are ignored.

The format string contains format specifications which direct it to read and convert characters from the input field into specific types of values and store them in the locations given by the address arguments. The format specifications have the following form :

```
% [l] type_character
```

Each format specification begins with a percent character (%). After the percent character, there may be an optional type modifier character (i.e. 'l') followed by mandatory type_character. The following table lists valid type characters, the type of input expected by each, and in what format the input will be stored :

Type Character	Input	Type of argument
d	decimal integer	pointer to int (int *arg)
i	decimal integer	pointer to int (int *arg)
u	decimal integer	pointer to int (int *arg)
f	floating point (float)	pointer to float (float *arg)
c	character (char)	pointer to character (char *arg)

s	character string	pointer to character string (char *arg)
%	% character	No conversion is done, the '%' character is stored

The optional type modifier character *l* can be used with d, I or u type characters; when used, it forces the scanf function to convert the input field into long int value instead of int; so the corresponding address argument must be pointer to long int.

Return value : Returns the last character written.
See also : putchar

---

sendbyte

Function : Sends a byte to the on-chip serial port.
Syntax :
```
#include  <standard.h>
void sendbyte (unsigned char ch) ;
```
Prototype in : standard.h
Remarks : Sends the character 'ch' to on-chip serial port (i.e. SBUF). It also waits till the transmission is complete. That is, it will wait till the TI bit in SCON is set. Before returning, it clears the TI bit, so that you are ready to send the next byte.
Return value : None.
See also : getbyte, ser_rdy

---

ser_rdy

Function : Checks the on-chip serial port status.
Syntax :
```
#include  <standard.h>
int ser_rdy () ;
```
Prototype in : standard.h
Remarks : Checks whether a character is received on the on-chip serial port.
Return value : If a character is received, returns non-zero (but the character remains in SBUF, you may use 'getbyte' to read it). Otherwise returns zero.
See also : getbyte

---

set_com_mode

Function : Sets the mode of serial port
Syntax : `#include <macros31.h>`
`set_com_mode(mode,sm2,ren) ;`
Prototype in : macros31.h
Remarks : This macro writes sets and/or clears appropriate bits in SCON. In effect, it sets the serial port in the given 'mode', it sets the SM2 bit to 'sm2', and if 'ren' is 1, it enables the receiver.
Return value : None.

---

set_double_baud

Function : Sets 'double baud rate'
Syntax : `#include <macros31.h>`
`set_double_baud() ;`
Prototype in : macros31.h
Remarks : Sets bit 7 of PCON so that baud rate is doubled.
Return value : None.
See also : clr_double_baud

---

set_hi_ex0

Function : Sets high priority for INT0
Syntax : `#include <macros31.h>`
`set_hi_ex0() ;`
Prototype in : macros31.h
Remarks : Sets bit 0 of IP so that external interrupt INT0 is gets a high priority.
Return value : None.
See also : set_lo_ex0

---

set_hi_ex1

Function : Sets high priority for INT1
Syntax : `#include <macros31.h>`
`set_hi_ex1() ;`
Prototype in : macros31.h
Remarks : Sets bit 2 of IP so that external interrupt INT1 is gets a high priority.
Return value : None.
See also : set_lo_ex1

---

set_hi_ser

Function : Sets high priority for serial port interrupt
Syntax : `#include <macros31.h>`
`set_hi_ser() ;`
Prototype in : macros31.h
Remarks : Sets bit 4 of IP so that serial port interrupt gets a high priority.
Return value : None.
See also : set_lo_ser

---

set_hi_t0

Function : Sets high priority for timer 0 interrupt
Syntax : `#include <macros31.h>`
`set_hi_t0() ;`
Prototype in : macros31.h
Remarks : Sets bit 1 of IP so that timer 0 interrupt gets a high priority.
Return value : None.
See also : set_lo_t0

---

set_hi_t1

Function : Sets high priority for timer 1 interrupt
Syntax : `#include <macros31.h>`
`set_hi_t1() ;`
Prototype in : macros31.h
Remarks : Sets bit 3 of IP so that timer 1 interrupt gets a high priority.
Return value : None.
See also : set_lo_t1

---

set_hi_t2

Function : Sets high priority for timer 2 interrupt
Syntax : `#include <macros31.h>`
`set_hi_t2() ;`
Prototype in : macros31.h
Remarks : Sets bit 5 of IP so that timer 2 interrupt gets a high priority.
Return value : None.
See also : set_lo_t2

---

set_lo_ex0

Function : Sets low priority for INT0
Syntax : `#include <macros31.h>`
`set_lo_ex0() ;`
Prototype in : macros31.h
Remarks : Clears bit 0 of IP so that external interrupt INT0 is gets a low priority.
Return value : None.
See also : set_hi_ex0

---

set_lo_ex1

Function : Sets low priority for INT1
Syntax : `#include <macros31.h>`
`set_lo_ex1() ;`
Prototype in : macros31.h
Remarks : Clears bit 2 of IP so that external interrupt INT1 is gets a low priority.
Return value : None.
See also : set_hi_ex1

---

set_lo_ser

Function : Sets low priority for serial port interrupt
Syntax : `#include <macros31.h>`
`set_lo_ser() ;`
Prototype in : macros31.h
Remarks : Clears bit 4 of IP so that serial port interrupt gets a low priority.
Return value : None.
See also : set_hi_ser

---

set_lo_t0

Function : Sets low priority for timer 0 interrupt
Syntax : `#include <macros31.h>`
`set_lo_t0() ;`
Prototype in : macros31.h
Remarks : Clears bit 1 of IP so that timer 0 interrupt gets a low priority.
Return value : None.
See also : set_hi_t0

---

set_lo_t1

Function : Sets low priority for timer 1 interrupt
Syntax : `#include <macros31.h>`
`set_lo_t1() ;`
Prototype in : macros31.h
Remarks : Clears bit 3 of IP so that timer 1 interrupt gets a low priority.
Return value : None.
See also : set_hi_t1

---

set_lo_t2

Function : Sets low priority for timer 2 interrupt
Syntax : `#include <macros31.h>`
`set_lo_t2() ;`
Prototype in : macros31.h
Remarks : Clears bit 5 of IP so that timer 2 interrupt gets a low priority.
Return value : None.
See also : set_hi_t2

---

set_t0_mode

Function : Sets timer 0 mode
Syntax : `#include <macros31.h>`
`set_t0_mode(gate,c_t,mode) ;`
Prototype in : macros31.h
Remarks : This function writes the appropriate byte in TMOD. If 'gate' is 1, then the GATE bit in TMOD (for Timer 0) will be set, otherwise it will be cleared. If 'c_t' is 1, then Timer 0 will be used as a counter, otherwise it will be used as a timer. 'mode' can be either 0,1,2 or 3.
Return value : None.
See also : set_t1_mode

---

set_t1_mode

Function : Sets timer 1 mode

Syntax :
```
#include  <macros31.h>
set_t1_mode(gate,c_t,mode)  ;
```

Prototype in : macros31.h

Remarks : This function writes the appropriate byte in TMOD. If 'gate' is 1, then the GATE bit in TMOD (for Timer 1) will be set, otherwise it will be cleared. If 'c_t' is 1, then Timer 1 will be used as a counter, otherwise it will be used as a timer. 'mode' can be either 0,1,or 2.

Return value : None.

See also : set_t0_mode

---

set_tcnt

Function : Sets timer 0 or 1 count.

Syntax : void set_tcnt(int tnum, unsigned int count) ;

Prototype in : standard.h

Remarks : This function sets the count (high and low) for timer 0 or 1. 'tnum' can be 0 or 1. 'count' is the count which will be loaded in either th0 and tl0 or th1 and tl1.

Return value : None.

---

sin

Function : Returns the sine of the argument.

Syntax :
```
#include  <math.h>
float sin (float x) ;
```

Prototype in : math.h

Remarks : This function is NOT included in the library, but it is given in the form of source code. It's source code can be found in SOURCE\CFILES\MATH.C

Return value : sin returns the sine of x.

See also : cos, asin

---

sinh

Function : Returns the hyperbolic sine of the argument.
Syntax : `#include <math.h>`
`float sinh (float x) ;`
Prototype in : math.h
Remarks : This function is NOT included in the library, but it is given in the form of source code. It's source code can be found in SOURCE\CFILES\MATH.C
Return value : sinh returns the hyperbolic sine of x.
See also : sin

---

sprintf

Function : Sends formatted output to the destination string
Syntax : `int sprintf(char *dest, char *format, ...) ;`
Prototype in : stdio.h
Remarks : The `sprintf` function sends formatted output to the destination string (dest parameter). The second parameter *`format`* specifies how the output is to be formatted. The first and second parameters are mandatory.
The `sprintf` function beheaves almost the same as `printf` function, except the difference in output location. The output of `printf` goes to standard output device (which is generally the on-chip serial port), but the output of `sprintf` goes into a character string pointed by the first parameter 'dest'.
For details about the format parameter, please see the description of `printf`.

Return value : The sprintf function returns the number of bytes copied to the destination string.
See also : printf. Also, please look at the file PRINT.C in EXAMPLES\PRINT directory.

---

sqrt

Function : Returns the square root of the argument.
Syntax : `#include <math.h>`
`float sqrt (float x) ;`
Prototype in : math.h

Remarks : This function is NOT included in the library, but it is given in the form of source code. It's source code can be found in SOURCE\CFILES\MATH.C
Return value : sqrt returns the square root of x.

---

sscanf

Function : Reads formatted input.
Syntax : `#include <stdio.h>`
`int sscanf (char *dest, char format, ...) ;`
Prototype in : stdio.h
Remarks : See scanf.

---

start_timer0

Function : Starts timer 0.
Syntax : `#include <macros31.h>`
`start_timer0() ;`
Prototype in : macros31.h
Remarks : Sets bit 4 of TCON to start timer0.
Return value : None.
See also : stop_timer0

---

start_timer1

Function : Starts timer 1.
Syntax : `#include <macros31.h>`
`start_timer1() ;`
Prototype in : macros31.h
Remarks : Sets bit 6 of TCON to start timer1.
Return value : None.
See also : stop_timer1

---

stop_timer0

Function : Stops timer 0.
Syntax : `#include <macros31.h>`
`stop_timer0() ;`
Prototype in : macros31.h
Remarks : Clears bit 4 of TCON to stop timer0.
Return value : None.
See also : start_timer0

---

stop_timer1

Function : Stops timer 1.
Syntax : `#include <macros31.h>`
`stop_timer1() ;`
Prototype in : macros31.h
Remarks : Clears bit 6 of TCON to stop timer1.
Return value : None.
See also : start_timer1

---

strcat

Function : Appends one string to another.
Syntax : char *strcat(char *dest, char *src) ;
Prototype in : string.h
Remarks : strcat appends a copy of src to the end of dest. The length of the resulting string is strlen(dest) + strlen(src).
Return value : strcat returns a pointer to the concatenated strings.

---

strcmp

Function : Compares one string to another.
Syntax : int strcmp(char *s1, char *s2) ;
Prototype in : string.h
Remarks : strcmp performs an unsigned comparison of s1 to s2, starting with the first character in each string and continuing with subsequent characters until the corresponding characters differ or until the end of the strings is reached.
Return value : strcmp returns a value that is < 0 if s1 is less than s2, = 0 if s1 is the same as s2 > 0 if s1 is greater than s2

---

strcpy

Function : Copies one string into another.
Syntax : char* strcpy(char *dest, char *src) ;
Prototype in : string.h
Remarks : copies string src to dest, stopping after the terminating null character has been moved.
Return value : strcpy returns dest.

---

strlen

Function : Calculates the length of a string.
Syntax :
```
#include<string.h>
int  strlen(char  *s);
```
Prototype in : string.h
Remarks : strlen calculates the length of s.
Return value : strlen returns the number of characters in s, not counting the null-terminating character.

---

strlwr

Function : Converts uppercase letters in a string to lower-case.
Syntax : char* strlwr(char*s);
Prototype in : string.h
Remarks : strlwr converts uppercase letters (A-Z) in string s to lower-case (a-z). No other characters are changed.
Return value : strlwr returns a pointer to the string s.
See also : strupr

---

strncpy

Function : Copies a given number of bytes from one string into another, truncating or padding as necessary.
Syntax :
```
#include<string.h>
char *strncpy(char*dest, char*src, int maxlen);
```
Prototype in : string.h
Remarks : strncpy copies up to maxlen characters from src into dest, truncating or null-padding dest. The target string, dest, might not be null-terminated if the length of src is maxlen or more.
Return value : strncpy returns dest.

---

strupr

Function : Converts lower-case letters in a string to uppercase.
Syntax : `char *strupr(char*s);`
Prototype in : string.h
Remarks : strupr converts lower-case letters (a-z) in string s to uppercase (A-Z). No other characters are changed.
Return value : strupr returns s.
See also : strlwr

---

tan

Function : Trigonometric tangent function.
Syntax : `#include <math.h>`
`float tan (float x) ;`
Prototype in : math.h
Remarks : This function is NOT included in the library, but it is given in the form of source code. It's source code can be found in SOURCE\CFILES\MATH.C
Return value : tan returns the tan of x.
See also : cos, sin, tanh

---

tanh

Function : Hyperbolic tangent function.
Syntax : `#include <math.h>`
`float tanh (float x) ;`
Prototype in : math.h
Remarks : This function is NOT included in the library, but it is given in the form of source code. It's source code can be found in SOURCE\CFILES\MATH.C
Return value : tanh returns the hyperbolic tan of x.
See also : cos, sin, tan

---

tolower

Function : Translates characters to lower-case.
Syntax : `char tolower (char ch);`
Prototype in : ctype.h
Remarks : tolower is a function that converts a character ch (in the range 0 to 255) to its lower-case ('a'-'z') value (if it was uppercase ('A'-'Z'); all others are left unchanged.
Return value : tolower returns the converted value of ch if it is uppercase; all others it returns unchanged.

---

toupper

Function : Translates characters to uppercase.
Syntax : `char toupper (char ch);`
Prototype in : ctype.h
Remarks : toupper is a function that converts a character ch (in the range 0 to 255) to its uppercase value (‘A’-‘Z’) if it was lower-case (‘a’-‘z’); all others are left unchanged.
Return value : toupper returns the converted value of ch if it is lower-case; it returns all others unchanged.

---

ui2a_c31

Function : Converts an unsigned integer into ASCII string
Syntax : `#include <stdlib.h>`
`void ui2a_c31(unsigned int val, char *dest, int ndigits) ;`
Prototype in : stdlib.h
Remarks : ui2a_c31 converts an unsigned integer number 'val' into ASCII string 'dest' containing exactly ndigits digits.
Return value : None.
See also : itoa_c31

---

ui2bcd

Function : Converts an unsigned integer number into a BCD string.
Syntax : `#include <stdlib.h>`
`void ui2bcd (unsigned int value, char *dest, int ndigits);`
Prototype in : stdlib.h
Remarks : Converts value into an unpacked BCD string which is ‘ndigits’ digits long and places the string in ‘dest’
Return value : None
See also : int2bcd

5. Advanced Programming Techniques

5.1 Writing Simple Assembly Language Sub-routines :

It may be sometimes very convenient to write small functions in assembly language and call these functions from C language program. This section describes exactly how to do it. Let us take a simple example - it may not be useful in real life, but it nonetheless illustrates the method to be followed. This is how our example function goes :

```
_exmpl_func:
; Example function written in assembly language
        mov   a,p0          ; read P0 data
        rlc   a             ; rotate left through Carry
        mov   p0,a          ; write back to P0
        ret
```

In your C program, you might call this function, simply by writing :

```
exmpl_func() ;
```

Writing assembly language functions is as simple as that. You only need to remember a few rules :

- An assembly language function must start with a label, which is same as the name of the function preceded by an underscore character ('_').
- This label name must be declared as PUBLIC (for example “PUBLIC _exmpl_func”).
- While calling such a function from C language program, you must omit the underscore character.
- The assembly language function may modify the contents of any of the registers (R0 through R7) and Acc.
- Never switch register bank.
- Never change the contents of Stack Pointer (SP). Make sure that the PUSHs and POPs are exactly matching.
- Never write into any location in internal or external data memory.

- Never access any location in internal / external data memory by using hardcoded addresses. Always define global variable in your C program, so that the required space gets allotted to them. Use only the names of these variables as internal / external data memory addresses. e.g. if you want to use a byte location in external data memory, you should declare a unsigned char type global variable in your C program as :

```
xdata unsigned char asm_var ;
```

In your assembly language function, you can use it as :

```
mov   dptr,#_asm_var  ; load the addr. in DPTR
movx  a,@dptr         ; read the value

;  do  some  processing
movx  @dptr,a         ; write back the value
```

If you stick to these rules, chances of "system crash" will be far too less!

5.2 Assembly Language Function's return value :

Suppose you want the assembly language function to return some value, follow these guidelines. In your C program you must put the function's declaration. e.g.

```
unsigned  char  some_func  ()  ;
```

Then you may call this function as and when required. e.g.

```
value  =  some_func()  +  25 ;
```

In assembly language you write,

```
_some_func:
;  some  code  here
;  finally  compute  the  return  value  in  Acc,
;  and  then  store
        mov   myacc,a
;  the  return  value  put  in  proper  place
        ret
```

Note that the return value should be always put in the internal RAM location - myacc. Depending on the type of return value, you may write 1 to 4 bytes from location onwards. e.g. in case of int type return value, you should write the LSByte of the return value at myacc and the MSByte should be written at (myacc + 1).

Please note that if a function's return value is of type 'bit' then you should use the bit location 'bit_acc' instead of internal RAM location 'myacc'. e.g. if you want to return 1 (bit type), then you must write :

```
setb    bit_acc
ret
```

at the end of the function.

5.3 Assembly Language Functions with parameters:

If you want the assembly language function to take parameters, you must start and end the function in a specific manner.

5.3.1 Accessing parameters in assembler program:

At the beginning, you must write these two lines:

```
push   bp
mov    bp,sp
```

At the end, you must write these three lines:

```
mov    sp,bp
pop    bp
ret
```

It is also necessary to declare "bp" as an external symbol, with the declaration:

```
EXTRN DATA (bp)
```

When a function is called from C language program, it's parameters are pushed onto stack. The first parameter is pushed first. To access these parameters, you should make use of bp. For example consider this C language statement:

```
some_func1(par1,par2) ;
```

Let us assume that par1 is of int type and par2 is of unsigned char type. The last byte of the first parameter can be always found at the address ((bp) - 3). e.g. to load the value of par1 in R3-R2, you may write:

```
mov    a,bp
clr    c
subb   a,#4
mov    r0,a   ; R0 = (bp) - 4 = addr of par1
mov    a,@r0  ; read LSByte of par1
mov    r2,a   ; put it in R2
inc    r0
mov    a,@r0  ; read MSByte of par1
mov    r3,a   ; put it in R3
```

Note that 'lower byte at lower address' philosophy (or the “little endian format”) is followed everywhere. The remaining parameters are stored at consecutive locations - backward. e.g. par2 can be found at (bp) - 5.

Please note that you must never modify the contents of bp or sp.

5.4 Calling C functions from assembly language:

You can simply call C language function from assembly language by name, but by preceding it by an underscore character. e.g. if your C function is defined as :

```
void  some_func  ()  ;
```

You can call it from assembly language as :

```
lcall   _some_func
```

Please note, it is also necessary to declare `_some_func` as an external symbol:

```
EXTRN CODE (_some_func)
```

5.4.1 Passing parameters:

If the C function takes any parameters, you should make use of sp to pass parameters. For example, consider the function:

```
void  some_func  (int  par1,  unsigned  char  par2)  ;
```

For simplicity, let us assume that value of par1 is in R3-R2 (MSByte in R3) and value of par2 is in R4. The last parameter (par2 in this case) is pushed first. i.e. parameters are pushed in the reverse sequence. First you should push the parameters onto stack, then you call the function, then you should adjust the stack. For the above example, this is how you do it:

```
mov    a,r4
push   acc   ; par2 is pushed
mov    a,r2
push   acc   ; low byte of par1 is pushed
mov    a,r3
push   acc   ; high byte of par1 is pushed
lcall  _some_func
mov    a,sp
clr    c
subb   a,#3    ;  decrement  sp  by  total  no.  of
mov    sp,a    ;  bytes  required  for  ALL  parameters
```

For more information about assembler language programming, please refer to the user's manual of SASM51 assembler (SASM51.PDF).

Appendix A : Error and Warning messages

This appendix describes all warning and error messages produced by the compiler. The description includes possible causes and remedy or suggestion.
An **Error** message is produced by the compiler when the severity of the error makes it impossible for the compiler to produce the assembly language or machine code output. Thus if you encounter an error message while compiling, it indicates that the compiler is unable to produce the ROMable code file (.BIN)
On the other hand, a **Warning** message is produced by the compiler, when it does not know whether it really is an error or not. It is expected that the programmer should decide it himself, and hence the compiler also prints sufficient information in the warning message. However, even if you encounter one or more warning messages, the compiler still completes the compile process and produces the ROMable code file (provided there are no Errors). The programmer may then analyze each warning message and then only decide whether to use the compiler output or to make changes in the source program and compile again.

How can I see all of the error / warning messages ?

When you run the compiler, it prints all of the warning and error messages on the Standard Output device i.e. generally the console. If there are too many warnings / errors, you may not be able to see all of them on the screen. In such a case, you may use output redirection to catch all the warning / error messages. i.e. these can be saved in a file - say SC51.ERR. To do this, invoke the compiler as shown below :

```
Sc51 [options] filename.c  >  sc51.err
```

When you do this, you will not see any warning / error messages on the screen. Instead these will go into the file “sc51.err”. When the compile process is over, you can view the file “sc51.err” using any text editor.
In most cases, you will use the SIDE51 Integrated Development Environment. In this case, the error/warning messages produced by the compiler/assembler/linker are displayed in the “error window”.

Warning Messages :

Parameter '*paraname*' never used :

Where *paraname* is the name of a parameter. This warning is usually displayed with a line number of the last line of a function in which the said parameter was defined.

Local variable '*localname*' declared but never used :

Where *localname* is the name of a local variable. This warning is usually displayed with a line number of the last line of a function in which the said local variable was defined.

No declaration for function '*funcname*' :

If a function is called in an expression and if it is not earlier defined or declared, this warning message will be printed.

Error Messages :

Almost all of the error messages also display the source file name and the line number which caused this error message. The meaning of all error messages are described below.

Declaration Syntax Error :

This error message is generally caused by an incorrect variable or function declaration or definition. Check the spellings and syntax of the given line number. Also see the list of supported data types.

Redeclaration of identifier OR function :

If a variable or function is defined more than once, then it causes this error message. However, 2 (or more) variables of the same name can exist, provided one of those is a global variable and the other one is a local variable or a parameter.

Too Many Identifiers :

If the compiler runs out of memory for the symbol table, it produces this error message. Check for any unused variables / functions and remove them if possible.

Unknown Identifier :

If you attempt to use a variable without declaring it, this error message will occur. Please check the spellings.

Invalid Identifier :

If an identifier is badly defined, this error will occur. Please see the rules of defining an identifier in “Language Reference” in this manual.

Too many errors :

If the compiler encounters too many error messages while compiling your program, it will print this message and stop the compile process. Please note, that this indicates that your program has not been completely compiled. i.e. there may be more errors which are not detected yet. Please correct the displayed errors and compile again.

Invalid Statement :

Syntax error ! :

If the general syntax rules of C language are not followed, one of the above two general error messages is printed. e.g. the “if” keyword must be followed by a valid expression enclosed in a pair of parenthesis. If this is missing, you may see this error message. This is just one example, there may be numerous situations under which you will see this message.

Mismatch brackets :

If in an expression, the parenthesis (‘(‘ and ‘)’) or the square brackets (‘[‘ and ‘]’) are not in matching pairs, this error will be printed. If, in your program the curly brackets are not in matching pairs, then also you will see the same error message.

Too Many Brackets :

An expression may contain any number of parenthesis or square brackets - provided they are in matching pairs. However, the more brackets in an expression, the more memory is required by the compiler. Thus when the compiler runs out of memory while sorting out an expression, this error message is printed.

Statement Missing ; :

In ‘C’ language, each statement must be followed by a semicolon. (barring a few exceptions like the SFR statement or the BIT statement etc.) If you forget to write the semicolon, you will see this error message.

Misplaced continue :

The ‘continue’ keyword must be used inside a loop. e.g. in a ‘for’ loop or a ‘do - while’ loop or in a ‘while’ loop. However, when the compiler encounters a ‘continue’ statement which is NOT inside the body of a loop, it will print this error message.

Misplaced else :

The else keyword must be used in conjunction with the ‘if’ keyword. Use of ‘else’ keyword without a corresponding ‘if’ will produce this error.

Misplaced break :

The ‘break’ keyword must be used inside a loop - e.g. in a ‘for’ loop or a ‘do - while’ loop or in a ‘while’ loop, or inside a ‘switch’ statement. However, when the compiler encounters a ‘break’ statement which is NOT inside the body of a loop or a ‘switch’ statement, it will print this error message.

do statement missing 'while' :

The ‘do’ keyword must have a corresponding ‘while’ at the end of the ‘do - while’ loop. If the ‘while’ keyword is missing, this error message is printed.

case outside of switch :

The ‘case’ keyword must be used inside a ‘switch’ statement. However, when the compiler encounters a ‘case’ statement which is NOT inside a ‘switch’ statement, it will print this error message.

Colon Missing :

The ‘case’ or ‘default’ keyword (inside a ‘switch’ statement) must be followed by a colon (‘:’) If this is missing, the compiler prints this error message.

Too Many Default Statements OR default May be Outside of switch :

The ‘default’ keyword must be used inside a ‘switch’ statement. However, when the compiler encounters a ‘default’ statement which is NOT inside a ‘switch’ statement, it will print this error message.
Also, you may use the ‘default’ keyword only once in a single ‘switch’ statement. If you tried to use more than one ‘default’ in a single ‘switch’ statement, then also the same error message is printed.

Type Mismatch :

If the types of two operands are not ‘matching’ with each other, this error message will be printed. However, this does not mean that both the operands must be of the SAME type. Even the types are not same, the compiler generates code to convert one of the operand into that of the other. However, this may not be always possible. e.g. a bit type variable can not be converted to any other type. Hence, for example, if you try to add ‘bit’ type variable into an ‘int’ type variable, it will produce this error.

Too many 'goto' statements ! (max. 10) :

In a single function, you may not write more than 10 ‘goto’ statements. If you attempt to use ‘goto’ more than 10 times in a function, this error message will be printed.

Undefined label :

If a label corresponding to a ‘goto’ statement is not defined till the end of the function, this error message will be printed.

Undefined macro :

If you try to undefine a macro (using #undefine) which is not defined at all, this error message will be printed.

Macro Parameters missing :

If you have defined a macro with parameters, you must pass the same number of parameters while using the macro. If in the macro call, the parameters are missing, this error message will be printed.

Too many/few parameters passed to macro :

If you have defined a macro with parameters, you must pass the same number of parameters while using the macro. If in the macro call, the no. of parameters passed is not same as the no. of parameters required, this error message will be printed.

Macro definition syntax error :

If a syntax error is detected in a macro definition, this error message will be printed. e.g. if you attempt to use a keyword as the macro name, this error message will be printed.

Unexpected #else or #endif :

The #else or #endif must have a corresponding #ifdef or #ifndef. If #else or #endif are found out of place, this error message will be printed.

Too many macro definitions :

As you keep defining macros, the compiler remembers all the macro definitions. But when the compiler runs out of memory to store any more macro definitions, this error message will be printed.

Too big macro definition :

If a macro definition is too big to fit in the available memory, this error message will be printed. Please note that, multi-line macro definitions are not supported in this version of the compiler.

Syntax error in include statement (or file not found) :

If the #include statement is not correctly written or if the compiler is unable to open the included file, this error message will be printed.

Macro has too many parameters :

A macro can not have more than 10 parameters. If you attempt to define a macro with more than 10 parameters, this error message will be printed.

Invalid Statement :

If an invalid preprocessor directive is encountered, this error message will be printed.

Line Too Long ! :

Any line in your program must not have more than 240 characters. If a line is found to have more than 240 characters, this error message will be printed.

Too many ifs :

The #ifdef or #ifndef statements can be nested. However, this version of the compiler does not support a nesting level of more than 10. If you attempt to nest more than 10 #ifdef or #ifndef statements, this error message will be printed.

Too many (> 128) SFR declarations :

Using the SFR keyword, you can define Special Function Registers (to be used as pseudo variables). However, you may not declare more than 128 SFRs. If you attempt to do so, this error message will be printed.

'bit' type local var/para/array : not allowed :

A ‘bit’ type variable must be declared as a global variable only. If you attempt to declare a bit variable as a local variable or as a parameter, this error message will be printed. Also, arrays of bit variables are not supported in this version of the compiler. If you attempt to declare an array of bit type variable, this error message will be printed.

pointer to bit variable : not allowed :

The current version of the compiler does not support pointer to bit type variable. If you attempt to declare a pointer to a bit type variable, this error message will be printed.

Appendix B : Limitations of the compiler

There are certain areas of the compiler that have known limitations or bugs. SPJ Systems will attempt repair these problems in future versions of the C Compiler. Until that version is released, you should be aware of these limitations. This appendix describes all such known limitations and known bugs which are reported and not corrected yet. You will also find some useful suggestions in order to use the compiler in a better and more efficient way, in Appendix C.

LIMITATIONS AND BUGS :

1) Only global variables can be initialized during declaration. In case, you attempt to initialize a local variable, the compiler will produce some "Assembler error" during the Assembler Pass #2. If you want to initialize a character array to a very long constant string, then you can write the string on multiple lines. e.g.

```
char arr[100] = “hello”\
                “ world !” ;
```

Please note the backslash at the end of first line. It indicates that string is continued on next line.

2) While passing parameters to a function, the compiler checks only first 12 parameters for type conversion.
3) Type casting is not supported.
4) Sometimes, the compiler does not recognize the unary minus operator correctly. It is recommended to use the expression "(0 - something)" instead of just writing "-something". e.g.

```
x = -10 ; /* may produce error */
x = 0 - 10 ; /* will not produce error */
x = (-z) * 15 ; /* may produce error */
x = (0-z) * 15 ; /* will not produce error */
```

5) Maximum 2 dimensional arrays are supported. Arrays with more than 2 dimensions are not supported.
6) Pointer to pointer is not supported.
7) Array of pointers is not supported.
8) A member of a `struct` / `union` may not be a pointer.
9) A member of a `struct` / `union` may not be a `struct` / `union`.

Appendix C : 10 ways to improve code efficiency

This appendix lists some useful suggestions in order to use the compiler in a better and more efficient way:

1) Small model produces smaller code, because all (or most) variables reside in internal RAM (**data** memory area). Manipulating internal RAM is much easier than external RAM. So, if you are able to fit all or most of your data in 128 bytes of RAM, it is advisable to compile your program using small model. If you need more data variables, however, you can either use large memory model or use the memory area specifiers to force some variables in idata, xdata, or pdata area.
2) Wherever possible, use `unsigned char` data type instead of int. e.g. the count variable in a for loop etc. If the count does not exceed 255, you may use `unsigned char` type variable instead of int. Since the 8051 has instructions only to manipulate 8 bit data (and bits), it is easier to handle 8 bit data elements rather than 16 bit or 32 bits.
3) For the same reason, use of 'S' suffix on constants can produce smaller code. As an example, consider the statement "x = y * 7 ;" where x and y are "unsigned char" variables. The constant 7 is considered as an int constant (0007H). Due to ANSI C integer promotion rules, the single byte (unsigned char) value of "y" is converted to a two byte (int) value (by appending a byte 00H to it's left side). After that, code is generated to perform multiplication of two 16 bit numbers. The 8051 does not provide any instruction for 16 bit multiplication – which means the compiler must call a library sub-routine or generate lot of in-line code. Did you really expect all that big code? In most cases, no. If you really expected a single byte multiplication, then you should use the 'S' suffix on 7. In other words, re-write the above statement as: "x = y * 7S ;". Now, 7S will be considered as a single byte (unsigned char) value (07H). Since the two operands ("y" and "07H") have same data types, no code is generated for conversion to int. Instead, the compiler will simply generate the "mul ab" instruction to perform the 8 bit X 8 bit multiplication, thus the generated code will be much smaller and it will run much faster.
4) Generally, use as small data type as is sufficient. Using "int" or "unsigned int" is better than using "long int" or "unsigned long int". Using "char" or "unsigned char" is even better.
5) Instead of using 'inportb' and 'outportb' functions, it is much efficient to use a variable declared at a specific address in external RAM. e.g. instead of writing :

```
outportb(0x6000,0) ;
```

it is better to write :

```
dat_8279 = 0 ;
```

where 'dat_8279' is a variable defined as follows :

```
unsigned char @0x6000 dat_8279 ;
```

6) Wherever possible, use unsigned data types instead of signed. i.e. `unsigned char` instead of `char`, `unsigned int` instead of `int` and `unsigned long int` instead of `long int`. Since the 8051 does not have any instructions to perform signed arithmetic, it is easier to perform arithmetic operations on unsigned number.
7) Whenever a variable is going to hold only two possible values (0 or 1, or TRUE or FALSE), it is efficient to use `bit` variable instead of `int` or `char`. Even a function's return value type could be `bit`.
8) To access a bit of a bit accessible SFR, you can make use of `BIT` declarations. e.g. to set bit 3 of P1, you can write either :

```
P1 |= 8 ;
```

or

```
BIT  p1_3   p1.3    /*   Define a BIT   */
p1_3 = 1 ;          /*   set the bit    */
```

The later method is much more efficient.

9) If you need a software delay, it is more convenient, efficient and accurate to use the library function ‘delay()’ or ‘delay_ms()’ instead of writing empty for loop etc. Please refer to the chapter “Library functions reference” for details about the ‘delay()’ and ‘delay_ms()’ functions.
10) Floating point numbers are fairly complex data types and the compiler needs to generate number of sub-routine calls for even basic arithmetic operations on floating point numbers. Hence, the code generated for floating point arithmetic is generally longer and runs slower. Whenever possible, avoid the use of float numbers. In many situations, “long int” data type can be used instead of “float” data type, to represent even fractional numbers (if they are “fixed point”). For example, if it is required to store only 3 digits after the decimal point and maximum 5 digits before the decimal point for a certain variable, then it may be possible to use the long int data type (instead of float). You only need to remember that the value of the variable = TheRealValue * 1000. Thus to display the value, you may use a suitable function to convert the long int value into ASCII string (such as sprintf or ltoa_c31), and insert a decimal point (‘.’) at 3 positions from the right. “long int” arithmetic produces much smaller and faster code, as compared to “float” arithmetic. Use the “float” data type only when you really need to store and process floating point numbers.

---X---X---