KEIL SOFTWARE

C51 Compiler

Optimizing 8051 C Compiler and Library Reference

User's Guide 01.97

Preface

This manual describes how to use the C51 Optimizing C Compiler to compile C programs for your target 8051 environment. The C51 Compiler package can be used on all 8051 family processors and is executable under MS-DOS. This manual assumes that you are familiar with the MS-DOS operating system, know how to program 8051 processors, and have a working knowledge of the C language.

NOTE
MS-DOS and PC-DOS are, in essence, the same operating system. This manual uses MS-DOS or just DOS when referring to either system.

If you have questions about programming in C, or if you would like more information about the C programming language, refer to "Books About the C Language" on page 2.

Many of the examples and descriptions in this manual discuss invoking the compiler from the DOS command prompt. While this may not be applicable to you if you are running C51 within an integrated development environment, examples in this manual are universal in that they apply to all programming environments.

Manual Organization

This user's guide is divided into eight chapters and six appendices:

"Chapter 1. Introduction," describes the C51 compiler.

"Chapter 2. Compiling with C51," explains how to compile a source file using the C51 cross compiler. This chapter describes the command-line directives that control file processing, compiling, and output.

"Chapter 3. Language Extensions," describes the C language extensions required to support the 8051 system architecture. This chapter provides a detailed list of commands, functions, and controls not found in ANSI C compilers.

"Chapter 4. Preprocessor," describes the components of the C51 preprocessor and includes examples.

"Chapter 5. 8051 Derivatives," describes the 8051 family derivatives that the C51 compiler supports. This chapter also includes tips for improving target performance.

"Chapter 6. Advanced Programming Techniques," lists important information for the experienced developer. This chapter includes customization file descriptions, and optimizer and segment names. This chapter also discusses how to interface C51 with other 8051 programming languages.

"Chapter 7. Error Messages," lists the fatal errors, syntax errors, and warnings that you may encounter while using C51.

"Chapter 8. Library Reference," provides you with extensive C51 library routine reference material. The library routines are listed by category and include file. An alphabetical reference section, which includes example code for each of the library routines, concludes the chapter.

The Appendix includes information on the differences between compiler versions, writing code, and other items of interest.

Document Conventions

This document uses the following conventions:

Examples	Description
README.TXT	Bold capital text is used for the names of executable programs, data files, source files, environment variables, and commands you enter at the MS-DOS command prompt. This text usually represents commands that you must type in literally. For example: **CLS** **DIR** **BL51.EXE** Note that you are not required to enter these commands using all capital letters.
Language Elements	Elements of the C language are presented in bold type. This includes keywords, operators, and library functions. For example: **if** **!=** **long** **isdigit** **main** **>>**
`Courier`	Text in this typeface is used to represent information that displays on screen or prints at the printer. This typeface is also used within the text when discussing or describing command line items.
Variables	Text in italics represents information that you must provide. For example, *projectfile* in a syntax string means that you must supply the actual project file name. Occasionally, italics are also used to emphasize words in the text.
Elements that repeat…	Ellipses (…) are used in examples to indicate an item that may be repeated.

Omitted code
.
.
.
	Vertical ellipses are used in source code examples to indicate that a fragment of the program is omitted. For example:
	`void main (void) {`
	`.`
	`.`
	`.`
	`while (1);`

⟦*Optional Items*⟧	Optional arguments in command-line and option fields are indicated by double brackets. For example: C51 TEST.C PRINT ⟦(*filename*)⟧
{ *opt1* | *opt2* }	Text contained within braces, separated by a vertical bar represents a group of items from which one must be chosen. The braces enclose all of the choices and the vertical bars separate the choices. One item in the list must be selected.
Keys	Text in this sans serif typeface represents actual keys on the keyboard. For example, "Press **Enter** to continue."

Contents

Chapter 1. Introduction

The C programming language is a general-purpose, programming language that provides code efficiency, elements of structured programming, and a rich set of operators. C is not a *big* language and is not designed for any one particular area of application. Its generality, combined with its absence of restrictions, makes C a convenient and effective programming solution for a wide variety of software tasks. Many applications can be solved more easily and efficiently with C than with other more specialized languages.

The C51 Optimizing C Compiler for the MS-DOS operating system is a complete implementation of the American National Standards Institute (ANSI) standard for the C language. C51 is not a universal C compiler adapted for the 8051 target. It is a ground-up implementation dedicated to generating extremely fast and compact code for the 8051 microprocessor. C51 provides you the flexibility of programming in C and the code efficiency and speed of assembly language.

The C language on its own is not capable of performing operations (such as input and output) that would normally require intervention from the operating system. Instead, these capabilities are provided as part of the standard library. Because these functions are separate from the language itself, C is especially suited for producing code that is portable across a wide number of platforms.

Since C51 is a cross compiler, some aspects of the C programming language and standard libraries are altered or enhanced to address the peculiarities of an embedded target processor. Refer to "Chapter 3. Language Extensions" on page 57 for more detailed information.

Books About the C Language

1

There are a number of books that provide an introduction to the C programming language. There are even more books that detail specific tasks using C. The following list is by no means a complete list of books on the subject. The list is provided only as a reference for those who wish more information.

The C Programming Language, Second Edition
Kernighan & Ritchie
Prentice-Hall, Inc.
ISBN 0-13-110370-9

C: A Reference Manual, Second Edition
Harbison & Steel
Prentice-Hall Software Series
ISBN 0-13-109810-1

C and the 8051: Programming and Multitasking
Schultz
P T R Prentice-Hall, Inc.
ISBN 0-13-753815-4

Chapter 2. Compiling with C51

This chapter explains how to use C51 to compile C source files and discusses the control directives you may specify. These directives allow you to:

- Direct C51 to generate a listing file
- Define manifest constants on the command line
- Control the amount of information included in the object file
- Specify the level of optimization to use
- Specify the memory models
- Specify the memory space for variables

Environment Settings

To run the compiler and the utilities, you must create new entries in the DOS environment table. In addition, you must specify a **PATH** for the compiler directory. The following table lists the environment variables, their default paths, and a brief description.

Variable	Path	Description
PATH	**\C51\BIN**	This environment variable specifies the path of the C51 executable programs.
TMP		This environment variable specifies which path to use for temporary files generated by the compiler. For best performance, specify a RAM disk. If the specified path does not exist, the compiler generates an error and aborts compilation.
C51INC	**\C51\INC**	This environment variable specifies the location of the standard C51 include files.
C51LIB	**\C51\LIB**	This environment variable specifies the location of the standard C51 library files.

Typically, these environment settings are automatically placed in your **AUTOEXEC.BAT** file when you install the compiler. However, to put these settings in a separate batch file, use the following:

```
PATH = C:\C51\BIN
SET TMP = D:\
SET C51INC = C:\C51\INC
SET C51LIB = C:\C51\LIB
```

Running C51

To invoke the C51 compiler, type **C51** at the DOS prompt. On this command line, you must include the name of the C source file to be compiled, as well as any other necessary control directives required to compile your source file. The format for the C51 command line is:

2

C51 *sourcefile* [*directives…*]

where:

sourcefile	is the name of the source program you want to compile.
directives	are the directives you want to use to control the function of the compiler. Refer to “Control Directives” on page 6 for a detailed list of the available directives.

The following command line example invokes C51, specifies the source file **SAMPLE.C**, and uses the controls **DEBUG**, **CODE**, and **PREPRINT**.

```
C51 SAMPLE.C DEBUG CODE PREPRINT
```

The C51 compiler displays the following information upon successful invocation and compilation.

```
MS-DOS C51 COMPILER V5.0

C51 COMPILATION COMPLETE.  0 WARNING(S),  0 ERROR(S)
```

DOS ERRORLEVEL

After compilation, the number of errors and warnings detected is output to the screen. C51 then sets the DOS **ERRORLEVEL** to indicate the status of the compilation. Values are listed in the following table:

ERRORLEVEL	Meaning
0	No errors or warnings
1	Warnings only
2	Errors and possibly warnings
3	Fatal errors

You can access the **ERRORLEVEL** variable in DOS batch files. Refer to your DOS user's guide for more information on **ERRORLEVEL** or batch files.

C51 Output Files

C51 generates a number of output files during compilation. By default, each of these output files shares the same *basename* as the source file. However, each has a different file extension. The following table lists the files and gives a brief description of each.

File Extension	Description
***basename*.LST**	Files with this extension are listing files that contain the formatted source text along with any errors detected by the compiler. Listing files may optionally contain the used symbols and the generated assembly code. See the **PRINT** directive in the following sections for more information.
***basename*.OBJ**	Files with this extension are object modules that contain relocatable object code. Object modules may be linked to an absolute object module by the BL51 Linker/Locator.
***basename*.I**	Files with this extension contain the source text as expanded by the preprocessor. All macros are expanded and all comments are deleted in this listing. See the **PREPRINT** directive in the following sections for more information.
***basename*.SRC**	Files with this extension are assembly source files generated from your C source code. These files can be assembled with the A51 assembler. See the **SRC** directive in the following sections for more information.

Control Directives

C51 offers a number of control directives that you can use to control the operation of the compiler. Directives are composed of one or more letters or digits and, unless otherwise specified, can be specified after the filename on the command line or within a source file using the **#pragma** directive.

Example

```
C51  testfile.c  SYMBOLS  CODE  DEBUG

#pragma SYMBOLS CODE DEBUG
```

In the above examples, **SYMBOLS**, **CODE**, and **DEBUG** are all control directives. **testfile.c** is the source file to be compiled.

NOTE
*The syntax is the same for the command line and the **#pragma** directive. Multiple options, however, may be specified on the **#pragma** line.*

Typically, each control directive may be specified only once at the beginning of a source file. If a directive is specified more than once, the compiler generates a fatal error and aborts compilation. Directives that may be specified more than once are so noted in the following sections.

Directive Categories

Control directives can be divided into three groups: source controls, object controls, and listing controls.

- Source controls define macros on the command line and determine the name of the file to be compiled.
- Object controls affect the form and content of the generated object module (***.OBJ**). These directives allow you to specify the optimizing level or include debugging information in the object file.
- Listing controls govern various aspects of the listing file (***.LST**), in particular its format and specific content.

The following table is an alphabetical list of the control directives. This list shows each directive's abbreviation, class, and description.

Directive and (Abbreviation)	Class	Description
AREGS (AR), NOAREGS (NOAR)	Object	Enable or disable absolute register (ARn) addressing.
ASM, ENDASM	Object	Marks the beginning and the end of an inline assembly block.
CODE (CD) †	Listing	Add an assembly listing to the listing file.
COMPACT (CP) †	Object	Select COMPACT memory model.
COND (CO), NOCOND (NOCO) †	Listing	Include or exclude source lines skipped from the preprocessor.
DEBUG (DB) †	Object	Include debugging information in the object file.
DEFINE (DF)	Source	Define preprocessor names in the C51 invocation line.
DISABLE	Object	Disables interrupts for the duration of a function.
EJECT (EJ)	Listing	Inserts a form feed character into the listing file.
FLOATFUZZY (FF)	Object	Specify number of bits rounded during floating compare.
INTERVAL †	Object	Specify the interval for interrupt vectors for SIECO derivatives.
INTPROMOTE (IP), NOINTPROMOTE (NOIP) †	Object	Enable or disable ANSI integer promotion.
INTVECTOR (IV), NOINTVECTOR (NOIV) †	Object	Specify base address for interrupt vectors or disable vectors.
LARGE (LA) †	Object	Select LARGE memory model.
LISTINCLUDE (LC)	Listing	Display contents of include files in the listing file.
MAXARGS (MA) †	Object	Specify size of variable argument lists.
MOD517, NOMOD517	Object	Enable or disable code to support the additional hardware features of the 80C517 and derivatives.
MODDP2, NOMODDP2	Object	Enable or disable code to support the additional hardware features of the Dallas Semiconductor 320 and AMD derivatives.
NOAMAKE (NOAM) †	Object	Disable information records for AutoMAKE.
NOEXTEND †	Source	Disable C51 extensions to ANSI C.
OBJECT (OJ), NOOBJECT (NOOJ) †	Object	Enable object file and optionally specify name or suppress the object file.
OBJECTEXTEND (OE) †	Object	Include additional variable type information in the object file.
OPTIMIZE (OT)	Object	Specify the level of optimization performed by the compiler.
ORDER (OR) †	Object	Variables are allocated in the order in which they appear in the source file.
PAGELENGTH (PL) †	Listing	Specify number of rows on the page.
PAGEWIDTH (PW) †	Listing	Specify number of columns on the page.

Directive and (Abbreviation)	Class	Description
PREPRINT (PP) †	Listing	Produce a preprocessor listing file where all macros are expanded.
PRINT (PR), NOPRINT (NOPR) †	Listing	Specify a name for the listing file or disable the listing file.
REGFILE (RF) †	Object	Specify a register definition file for global register optimization.
REGISTERBANK (RB)	Object	Select the register bank that is used for absolute register accesses.
REGPARMS, NOREGPARMS	Object	Enable or disable register parameter passing.
ROM †	Object	Control generation of AJMP/ACALL instructions.
SAVE, RESTORE	Object	Saves and restores settings for AREGS, REGPARMS, and OPTIMIZE directives.
SMALL (SM) †	Object	Select SMALL memory model. (Default.)
SRC †	Object	Create an assembler source file instead of an object module.
SYMBOLS (SB) †	Listing	Include a list of all symbols used within the module in the listing file.
WARNINGLEVEL (WL) †	Listing	Selects the level of Warning detection.

† These directives may be specified only once on the command line or at the beginning of a source file using in the #pragma statement. They may not be used more than once in a source file.

Control directives and their arguments, with the exception of arguments specified with the **DEFINE** directive, are case insensitive.

Reference

The remainder of this chapter is devoted to describing each of the available C51 compiler control directives. The directives are listed in alphabetical order, and each is divided into the following sections:

Abbreviation: Gives any abbreviations that may be substituted for the directive name.

Arguments: Describes and lists optional and required directive arguments.

Default: Shows the directive's default setting.

Description: Provides a detailed description of the directive and how to use it.

See Also: Names related directives.

Example: Shows you an example of how to use and, sometimes, the effects of the directive.

AREGS / NOAREGS

Abbreviation: None.

Arguments: None.

Default: AREGS

Description: The **AREGS** control causes the compiler to use absolute register addressing for registers R0 through R7. Absolute addressing improves the efficiency of the generated code. For example, **PUSH** and **POP** instructions function only with direct or absolute addresses. Using the **AREGS** directive, allows you to directly push and pop registers.

You may use the **REGISTERBANK** directive to define which register bank to use.

The **NOAREGS** directive disables absolute register addressing for registers R0 through R7. Functions which are compiled with **NOAREGS** are not dependent on the register bank and may use all 8051 register banks. This directive may be used for functions that are called from other functions using different register banks.

NOTE
*Though it may be defined several times in a program, the **AREGS / NOAREGS** option is valid only when defined outside of a function declaration.*

Example: The following is a source and code listing which uses both **NOAREGS** and **AREGS**.

```
stmt level    source
   1          extern char func ();
   2          char k;
   3
   4          #pragma NOAREGS
   5          noaregfunc ()  {
   6   1        k = func () + func ();
   7   1      }
   8
   9          #pragma AREGS
  10          aregfunc ()  {
  11   1        k = func () + func ();
  12   1      }

     ; FUNCTION noaregfunc (BEGIN)
                        ; SOURCE LINE # 6
0000 120000 E   LCALL func
0003 EF         MOV   A,R7
0004 C0E0       PUSH  ACC
0006 120000 E   LCALL func
0009 D0E0       POP   ACC
000B 2F         ADD   A,R7
000C F500   R   MOV   k,A
                        ; SOURCE LINE # 7
000E 22         RET
      ; FUNCTION noaregfunc (END)

      ; FUNCTION aregfunc (BEGIN)
                        ; SOURCE LINE # 11
0000 120000  E  LCALL func
0003 C007       PUSH  AR7
0005 120000  E  LCALL func
0008 D0E0       POP   ACC
000A 2F         ADD   A,R7
000B F500    R  MOV   k,A
                        ; SOURCE LINE # 12
000D 22         RET
      ; FUNCTION aregfunc (END)
```

Note the different methods of saving R7 on the stack. The code generated for the function **`noaregfunc`** is:

```
MOV  A,R7
PUSH ACC
```

while the code for the **`aregfunc`** function is:

```
PUSH AR7
```

ASM / ENDASM

Abbreviation: None.

Arguments: None.

Default: None.

Description: The **ASM** directive signals the beginning merge of a block of source text into the **.SRC** file generated using the **SRC** directive.

This source text can be thought of as in-line assembly. However, it is output to the source file generated only when using the **SRC** directive. The source text is not assembled and output to the object file.

The **ENDASM** directive is used to signal the end of the source text block.

NOTE
*The **ASM / ENDASM** directive can occur only in the source file, as part of a **#pragma** directive.*

Example:

```
#pragma asm / #pragma endasm
```

The following C source file:

```
.
.
.
stmt level  source
   1        extern void test ();
   2
   3        main ()  {
   4   1      test ();
   5   1
   6   1    #pragma asm
   7   1      JMP   $  ; endless loop
   8   1    #pragma endasm
   9   1    }
.
.
.
```

generates the following **.SRC** file.

```
; ASM.SRC generated from: ASM.C
NAME  ASM
?PR?main?ASM          SEGMENT CODE
EXTRN  CODE (test)
EXTRN  CODE (?C_STARTUP)
PUBLIC main
; extern void test ();
;
; main ()  {
        RSEG  ?PR?main?ASM
        USING 0
main:
                      ; SOURCE LINE # 3
;   test ();
                      ; SOURCE LINE # 4
                LCALL test
;
; #pragma asm
                JMP   $  ; endless loop
; #pragma endasm
; }
                      ; SOURCE LINE # 9
                RET   ; END OF main
        END
```

CODE

Abbreviation: CD

Arguments: None.

Default: No assembly code listing is generated.

Description: The CODE directive appends an assembly mnemonics list to the listing file. The assembler code is represented for each function contained in the source program. By default, no assembly code listing is included in the listing file.

Example:

```
C51 SAMPLE.C CD

#pragma code
```

The following example shows the C source followed by the resulting object code and its representative mnemonics. The line number of each statement that produced the code is displayed between the assembly lines. The characters **R** and **E** stand for Relocatable and External, respectively.

```
stmt level  source
   1        extern unsigned char a, b;
   2               unsigned char    c;
   3
   4        main()
   5        {
   6   1      c = 14 + 15 * ((b / c) + 252);
   7   1    }
.
.
.
ASSEMBLY LISTING OF GENERATED OBJECT CODE

      ; FUNCTION main (BEGIN)
                       ; SOURCE LINE # 5
                       ; SOURCE LINE # 6
0000 E500   E   MOV    A,b
0002 8500F0 R   MOV    B,c
0005 84         DIV    AB
0006 75F00F     MOV    B,#0FH
0009 A4         MUL    AB
000A 24D2       ADD    A,#0D2H
000C F500   R   MOV    c,A
                       ; SOURCE LINE # 7
000E 22         RET
      ; FUNCTION main (END)
```

COMPACT

Abbreviation: CP

Arguments: None.

Default: SMALL

Description: This directive implements the COMPACT memory model.

In the COMPACT memory model, all function and procedure variables and local data segments reside in the external data memory of the 8051 system. This external data memory may be up to 256 bytes (one page) long. With this model, the short form of addressing the external data memory through @R0/R1 is used.

Regardless of memory model type, you may declare variables in any of the 8051 memory ranges. However, placing frequently used variables (such as loop counters and array indices) in internal data memory significantly improves system performance.

NOTE
The stack required for function calls is always placed in IDATA memory.

See Also: SMALL, LARGE, ROM

Example:

```
C51 SAMPLE.C COMPACT

#pragma compact
```

COND / NOCOND

Abbreviation: CO

Arguments: None.

Default: COND

Description: This directive determines whether or not those portions of the source file affected by conditional compilation are displayed in the listing file.

The **COND** directive forces lines omitted from compilation to appear in the listing file. Line numbers and nesting levels are not output. This allows for easier visual identification.

The effect of this directive takes place one line after it is detected by the preprocessor.

The **NOCOND** directive determines whether or not those portions of the source file affected by conditional compilation are displayed in the listing file.

This directive also prevents lines omitted from compilation from appearing in the listing file.

Example: The following example shows the listing file for a source file compiled with the **COND** directive.

```
.
.
.
stmt level  source
   1        extern unsigned char  a, b;
   2               unsigned char     c;
   3
   4        main()
   5        {
   6   1    #if defined (VAX)
              c = 13;
            #elif defined (_ _TIME_ _)
   9   1      b = 14;
  10   1      a = 15;
  11   1    #endif
  12   1    }
.
.
.
```

The following example shows the listing file for a source file compiled with the **NOCOND** directive.

```
.
.
.
stmt level  source
   1        extern unsigned char  a, b;
   2               unsigned char     c;
   3
   4        main()
   5        {
   6   1    #if defined (VAX)
   9   1      b = 14;
  10   1      a = 15;
  11   1    #endif
  12   1    }
.
.
.
```

DEBUG

Abbreviation: DB

Arguments: None.

Default: No Debug information is generated.

Description: The **DEBUG** directive instructs the compiler to include debugging information in the object file. By default, debugging information is excluded from the generated object file.

Debug information is necessary for the symbolic testing of programs. This information contains both global and local variable definitions and their addresses, as well as function names and their line numbers. Debug information contained in each object module remains valid through the BL51 Link/Locate procedure. This information can be used by DS51 or by any of the Intel-compatible emulators.

NOTE
*The **OBJECTEXTEND** directive can be used to instruct the compiler to include additional variable type definition information in the object file.*

See Also: OBJECTEXTEND

Example:

```
C51 SAMPLE.C DEBUG

#pragma db
```

DEFINE

Abbreviation: DF

Arguments: One or more names separated by commas, in accordance with the naming conventions of the C language. An optional argument can be specified for each name given in the define directive.

Default: None.

Description: The **DEFINE** directive defines names on the invocation line which may be queried by the preprocessor with **#if**, **#ifdef**, and **#ifndef**. The defined names are copied exactly as they are entered. This command is case-sensitive. As an option each name can be followed by an argument string.

NOTE
*The **DEFINE** directive can be specified only on the command line. Use the C preprocessor #define directive for use inside a C source.*

Example:

```
C51 SAMPLE.C DEFINE (check, NoExtRam)

C51 MYPROG.C DF (X1="1+5",iofunc="getkey ()")
```

DISABLE

Abbreviation: None.

Arguments: None.

Default: None.

Description: The **DISABLE** directive instructs the compiler to generate code that disables all interrupts for the duration of a function. **DISABLE** must be specified with a **#pragma** directive immediately before a function declaration. The **DISABLE** control applies to one function only and must be re-specified for each new function.

NOTE
DISABLE *may be specified using the* ***#pragma*** *directive only, and may not be specified at the command line.*

DISABLE *can be specified more than once in a source file and must be specified once for each function that is to execute with interrupts disabled.*

A function with disabled interrupts cannot return a bit value to the caller.

Example: The following example is a source and code listing of a function using the **DISABLE** directive. Note that the **EA** special function register is cleared at the beginning of the function **(`JBC EA,?C0002`)** and restored at the end **(`MOV EA,C`).**

```
.
.
.
stmt level    source
   1          typedef unsigned char  uchar;
   2
   3          #pragma disable   /* Disable Interrupts */
   4          uchar dfunc (uchar p1, uchar p2)  {
   5   1        return (p1 * p2 + p2 * p1);
   6   1      }

      ; FUNCTION _dfunc (BEGIN)
0000 D3         SETB  C
0001 10AF01     JBC   EA,?C0002
0004 C3         CLR   C
0005  ?C0002:
0005 C0D0       PUSH  PSW
;---- Variable 'p1' assigned to register 'R7' ----
;---- Variable 'p2' assigned to register 'R5' ----
                      ; SOURCE LINE # 4
                      ; SOURCE LINE # 5
0007 ED         MOV   A,R5
0008 8FF0       MOV   B,R7
000A A4         MUL   AB
000B 25E0       ADD   A,ACC
000D FF         MOV   R7,A
                      ; SOURCE LINE # 6
000E  ?C0001:
000E D0D0       POP   PSW
0010 92AF       MOV   EA,C
0012 22         RET
      ; FUNCTION _dfunc (END)
.
.
.
```

EJECT

Abbreviation: EJ

Arguments: None.

Default: None.

Description: The **EJECT** directive causes a form feed character to be inserted into the listing file.

NOTE
*The **EJECT** directive occurs only in the source file, and must be part of a **#pragma** directive.*

Example:

```
#pragma eject
```

FLOATFUZZY

Abbreviation: FF

Arguments: A number between 0 and 7.

Default: FLOATFUZZY (3)

Description: The **FLOATFUZZY** directive determines the number of bits rounded before a floating-point compare is executed. The default value of 3 specifies that the three least significant bits of a float value are rounded before the floating-point compare is executed.

Example:

```
C51 MYFILE.C FLOATFUZZY (2)

#pragma FF (0)
```

INTERVAL

Abbreviation: None

Arguments: An optional interval, in parentheses, for the interrupt vector table.

Default: INTERVAL (8)

Description: The **INTERVAL** directive specifies an interval for interrupt vectors. The interval specification is required for SIECO-51 derivatives which define interrupt vectors in 3-byte intervals. Using this directive, the compiler locates interrupt vectors at the absolute address calculated by:

(*interval* × *n*) + *offset* + 3,

where:

interval	is the argument of the **INTERVAL** directive (default 8).
n	is the interrupt number.
offset	is the argument of the **INTVECTOR** directive (default 0).

See Also: INTVECTOR / NOINTVECTOR

Example:

```
C51 SAMPLE.C INTERVAL(3)

#pragma interval(3)
```

INTPROMOTE / NOINTPROMOTE

Abbreviation: IP / NOIP

Arguments: None.

Default: INTPROMOTE

Description: The **INTPROMOTE** directive enables ANSI integer promotion rules. Expressions used in if statements are promoted from smaller types to integer expressions before comparison. This allows Microsoft C and Borland C programs to be ported to C51 with fewer modifications.

Because the 8051 is an 8-bit processor, use of the **INTPROMOTE** directive generates inefficient code in some applications.

The **NOINTPROMOTE** directive disables automatic integer promotions. Integer promotions are normally enabled to provide the greatest compatibility between C51 and other ANSI compilers. However, integer promotions can yield inefficient code on the 8051.

Example:

```
C51 SAMPLE.C INTPROMOTE

#pragma intpromote

C51 SAMPLE.C NOINTPROMOTE
```

The following example demonstrates code generated using the **INTPROMOTE**, and the **NOINTPROMOTE** control directive.

```
stmt lvl  source

   1        char c;
   2        unsigned char  c1,c2;
   3        int  i;
   4
   5        main ()  {
   6  1       if (c == 0xff) c = 0;    /* never true! */
   7  1       if (c == -1) c = 1;      /* works */
   8  1       i = c + 5;
   9  1       if (c1 < c2 +4) c1 = 0;
  10  1     }
```

2

Code generated with INTPROMOTE

```
      ; FUNCTION main (BEGIN)
            ; SOURCE LINE # 6
0000 AF00         MOV  R7,c
0002 EF           MOV  A,R7
0003 33           RLC  A
0004 95E0         SUBB A,ACC
0006 FE           MOV  R6,A
0007 EF           MOV  A,R7
0008 F4           CPL  A
0009 4E           ORL  A,R6
000A 7002         JNZ  ?C0001
000C F500         MOV  c,A
000E           ?C0001:
            ; SOURCE LINE # 7
000E E500         MOV  A,c
0010 B4FF03       CJNE A,#0FFH,?C0002
0013 750001       MOV  c,#01H
0016           ?C0002:
            ; SOURCE LINE # 8
0016 AF00         MOV  R7,c
0018 EF           MOV  A,R7
0019 33           RLC  A
001A 95E0         SUBB A,ACC
001C FE           MOV  R6,A
001D EF           MOV  A,R7
001E 2405         ADD  A,#05H
0020 F500         MOV  i+01H,A
0022 E4           CLR  A
0023 3E           ADDC A,R6
0024 F500         MOV  i,A
            ; SOURCE LINE # 9
0026 E500         MOV  A,c2
0028 2404         ADD  A,#04H
002A FF           MOV  R7,A
002B E4           CLR  A
002C 33           RLC  A
002D FE           MOV  R6,A
002E C3           CLR  C
002F E500         MOV  A,c1
0031 9F           SUBB A,R7
0032 EE           MOV  A,R6
0033 6480         XRL  A,#080H
0035 F8           MOV  R0,A
0036 7480         MOV  A,#080H
0038 98           SUBB A,R0
0039 5003         JNC  ?C0004
003B E4           CLR  A
003C F500         MOV  c1,A
            ; SOURCE LINE # 10
003E           ?C0004:
003E 22           RET
            ; FUNCTION main (END)
```

CODE SIZE = 63 Bytes

Code generated with NOINTPROMOTE

```
            ; FUNCTION main (BEGIN)
            ; SOURCE LINE # 6
0000 AF00         MOV  R7,c
0002 EF           MOV  A,R7
0003 33           RLC  A
0004 95E0         SUBB A,ACC
0006 FE           MOV  R6,A
0007 EF           MOV  A,R7
0008 F4           CPL  A
0009 4E           ORL  A,R6
000A 7002         JNZ  ?C0001
000C F500         MOV  c,A
000E           ?C0001:
            ; SOURCE LINE # 7
000E E500         MOV  A,c
0010 B4FF03       CJNE A,#0FFH,?C0002
0013 750001       MOV  c,#01H
0016
            ; SOURCE LINE # 8
0016 E500         MOV  A,c
0018 2405         ADD  A,#05H
001A FF           MOV  R7,A
001B 33           RLC  A
001C 95E0         SUBB A,ACC
001E F500         MOV  i,A
0020 8F00         MOV  i+01H,R7
            ; SOURCE LINE # 9
0022 E500         MOV  A,c2
0024 2404         ADD  A,#04H
0026 FF           MOV  R7,A
0027 E500         MOV  A,c1
0029 C3           CLR  C
002A 9F           SUBB A,R7
002B 5003         JNC  ?C0004
002D E4           CLR  A
002E F500         MOV  c1,A
            ; SOURCE LINE # 10
0030           ?C0004:
0030 22           RET
            ; FUNCTION main (END)
```

CODE SIZE = 49 Bytes

INTVECTOR / NOINTVECTOR

Abbreviation: IV / NOIV

Arguments: An optional offset, in parentheses, for the interrupt vector table.

Default: INTVECTOR (0)

Description: The **INTVECTOR** directive instructs the compiler to generate interrupt vectors for functions which require them. An offset may be entered if the vector table starts at an address other than 0.

Using this directive, the compiler generates an interrupt vector entry using either an **AJMP** or **LJMP** instruction depending upon the size of the program memory specified with the **ROM** directive.

The NOINTVECTOR directive prevents the generation of an interrupt vector table. This flexibility allows the user to provide interrupt vectors with other programming tools.

The compiler normally generates an interrupt vector entry using a 3-byte jump instruction (**LJMP**). Vectors are located starting at absolute address:

(*interval* × *n*) + *offset* + 3,

where:

n	is the interrupt number.
interval	is the argument of the **INTERVAL** directive (default 8).
offset	is the argument of the **INTVECTOR** directive (default 0).

See Also: INTERVAL

Example:

```
C51 SAMPLE.C INTVECTOR(0x8000)

#pragma iv(0x8000)

C51 SAMPLE.C NOINTVECTOR

#pragma noiv
```

LARGE

Abbreviation: LA

Arguments: None.

Default: SMALL

Description: This directive implements the **LARGE** memory model. In the **LARGE** memory model, all variables and local data segments of functions and procedures reside (as defined) in the external data memory of the 8051 system. Up to 64 KBytes of external data memory may be accessed. This, however, requires the long and therefore inefficient form of data access through the data pointer (**DPTR**).

Regardless of memory model type, you may declare variables in any of the 8051 memory ranges. However, placing frequently used variables (such as loop counters and array indices) in internal data memory significantly improves system performance.

NOTE
The stack required for function calls is always placed in IDATA memory.

See Also: SMALL, COMPACT, ROM

Example:

```
C51 SAMPLE.C LARGE

#pragma large
```

LISTINCLUDE

Abbreviation: LC

Arguments: None.

Default: NOLISTINCLUDE

Description: The **LISTINCLUDE** directive displays the contents of the include files in the listing file. By default, include files are not listed in the listing file.

Example:

```
C51 SAMPLE.C LISTINCLUDE

#pragma listinclude
```

MAXARGS

Abbreviation: None.

Arguments: Number of bytes compiler reserves for variable-length argument lists.

Default: **MAXARGS(15)** for small and compact models.

MAXARGS(40) for large model.

Description: With the **MAXARGS** directive, you specify the buffer size for parameters passed in variable-length argument lists. MAXARGS defines the maximum number of parameters. The MAXARGS directive must be applied before the C function. This directive has no impact on the maximum number of arguments that may be passed to reentrant functions.

Example:

```
C51 SAMPLE.C MAXARGS(20)
```

```
#pragma maxaregs (4)  /* allow 4 bytes for parameters */

#include <stdarg.h>

void func (char typ, ...)  {
  va_list ptr;
  char c;
  int i;

  va_start (ptr, typ);
  switch *typ)  {
    case 0:                      /* a CHAR is passed */
      c = va_arg (ptr, char);  break;

    case 1:                      /* an INT is passed */
      i = va_arg (ptr, int);  break;
  }
}


void testfunc (void)  {
  func (0, 'c');             /* pass a char variable */
  func (1, 0x1234);          /* pass an int variable */
}
```

MOD517 / NOMOD517

Abbreviation: None.

Arguments: Optional parameters, enclosed in parentheses, to control support for individual components of the 80C517.

Default: NOMOD517

Description: The **MOD517** directive instructs the C51 compiler to produce code for the additional hardware components (the arithmetic processor and the additional data pointers) of the Siemens 80C517. This feature dramatically impacts the execution of integer, long, and floating-point math operations as well as functions that make use of the additional data pointers.

The following library functions take advantage of the extra data pointers: **memcpy**, **memmove**, **memcmp**, **strcpy**, and **strcmp**.

Library functions which take advantage of the arithmetic processor are so indicated by a **517** suffix. (Refer to "Chapter 8. Library Reference" on page 175 for details on these functions.)

Additional parameters may be specified with **MOD517** to control C51 support of the individual components of the 80C517. When specified, the parameters must appear within parentheses immediately following the **MOD517** directive. Parentheses are not required if none of these additional parameters is specified.

Directive	Description
NOAU	When specified, C51 uses only the additional data pointers of the 80C517. The arithmetic processor is not used. The NOAU parameter is useful for functions that are called by an interrupt while the arithmetic processor is already being used.
NODP8	When specified, C51 uses only the arithmetic processor. The additional data pointers are not used. The NODP8 parameter is useful for interrupt functions declared without the using function attribute. In this case, the extra data pointers are not used and, therefore, do not need to be saved on the stack during the interrupt.

Specifying both of these additional parameters with **MOD517** has the same effect as using the **NOMOD517** directive.

The **NOMOD517** directive disables generation of code that utilizes the additional hardware components of the 80C517.

NOTE
*Though it may be defined several times in a program, the **MOD517** directive is valid only when defined outside of a function declaration.*

See Also: MODDP2

Example:

```
C51 SAMPL517.C  MOD517

#pragma MOD517 (NOAU)

#pragma MOD517 (NODP8)

#pragma MOD517 (NODP8, NOAU)

C51 SAMPL517.C  NOMOD517

#pragma NOMOD517
```

MODDP2 / NOMODDP2

Abbreviation: None.

Arguments: MODDP2

Default: NOMODDP2

Description: The **MODDP2** directive instructs the C51 compiler to produce code for the additional hardware components (specifically, the additional CPU data pointers) available in the Dallas 80C320, C520, C530, AMD 80C521, and compatible derivatives. Using additional data pointers can improve the performance of the following library functions: **memcpy**, **memmove**, **memcmp**, **strcpy**, and **strcmp**.

The **NOMODDP2** directive disables generation of code that utilizes the additional CPU data pointers.

See Also: MOD517 / NOMOD517

Example:

```
C51 SAMPL517.C  MODDP2

#pragma moddp2

C51 SAMPL517.C  NOMODDP2

#pragma nomoddp2
```

NOAMAKE

Abbreviation: NOAM

Arguments: None.

Default: AutoMAKE information is generated.

Description: **NOAMAKE** disables the project information records produced by the C51 compiler for use with AutoMAKE. This option also disables the register optimization information. Use **NOAMAKE** to generate object files that can be used with older versions of the 8051 development tool chain.

Example:

```
C51 SAMPLE.C  NOAMAKE

#pragma NOAM
```

NOEXTEND

Abbreviation: None.

Arguments: None.

Default: All language extensions are enabled.

Description: The **NOEXTEND** control instructs the compiler to process only ANSI C language constructs. The C51 language extensions are disabled. Reserved keywords such as **bit**, **reentrant**, and **using** are not recognized and generate compilation errors or warnings.

Example:

```
C51 SAMPLE.C NOEXTEND

#pragma NOEXTEND
```

OBJECT / NOOBJECT

Abbreviation: OJ / NOOJ

Arguments: An optional filename enclosed in parentheses.

Default: OBJECT (*basename*.OBJ)

Description: The **OBJECT**(*filename*) directive changes the name of the object file to the name provided. By default, the name and path of the source file with the extension **.OBJ** is used.

The **NOOBJECT** control disables the generation of an object file.

Example:

```
C51 SAMPLE.C OBJECT(sample1.obj)

#pragma oj(sample_1.obj)

C51 SAMPLE.C NOOBJECT

#pragma nooj
```

OBJECTEXTEND

Abbreviation: OE

Arguments: None.

Default: None.

Description: The **OBJECTEXTEND** directive instructs the compiler to include additional variable-type, definition information in the generated object file. This additional information is used to identify objects within different scopes that have the same names so that they may be correctly differentiated by various emulators and simulators.

NOTE
Object files generated using this directive contain a superset of the OMF-51 specification for relocatable object formats. Emulators or simulators must provide enhanced object loaders to use this feature. If in doubt, do not use ***OBJECTEXTEND****.*

See Also: DEBUG

Example:

```
C51 SAMPLE.C OBJECTEXTEND DEBUG

#pragma oe db
```

OPTIMIZE

Abbreviation: OT

Arguments: A decimal number between 0 and 6 enclosed in parentheses. In addition, **OPTIMIZE (SIZE)** or **OPTIMIZE (SPEED)** may be used to select whether the optimization emphasis should be placed on code size or on execution speed.

Default: OPTIMIZE (6, SPEED)

Description: The **OPTIMIZE** directive sets the optimization level and emphasis.

NOTE
Each higher optimization level contains all of the characteristics of the preceding lower optimization level.

Level	Description
0	**Constant Folding:** The compiler performs calculations that reduce expressions to numeric constants, where possible. This includes calculations of run-time addresses. **Simple Access Optimizing:** The compiler optimizes access of internal data and bit addresses in the 8051 system. **Jump Optimizing:** The compiler always extends jumps to the final target. Jumps to jumps are deleted.
1	**Dead Code Elimination:** Unused code fragments and artifacts are eliminated. **Jump Negation:** Conditional jumps are closely examined to see if they can be streamlined or eliminated by the inversion of the test logic.
2	**Data Overlaying:** Data and bit segments suitable for static overlay are identified and internally marked. The BL51 Linker/Locator has the capability, through global data flow analysis, of selecting segments which can then be overlaid.
3	**Peephole Optimizing:** Redundant MOV instructions are removed. This includes unnecessary loading of objects from the memory as well as load operations with constants. Complex operations are replaced by simple operations when memory space or execution time can be saved.

Level	Description
4	**Register Variables:** Automatic variables and function arguments are located in registers when possible. Reservation of data memory for these variables is omitted. **Extended Access Optimizing:** Variables from the IDATA, XDATA, PDATA and CODE areas are directly included in operations. The use of intermediate registers is not necessary most of the time. **Local Common Subexpression Elimination:** If the same calculations are performed repetitively in an expression, the result of the first calculation is saved and used further whenever possible. Superfluous calculations are eliminated from the code. **Case/Switch Optimizing:** Code involving switch and case statements is optimized as jump tables or jump strings.
5	**Global Common Subexpression Elimination:** Identical sub expressions within a function are calculated only once when possible. The intermediate result is stored in a register and used instead of a new calculation. **Simple Loop Optimizing:** Program loops that fill a memory range with a constant are converted and optimized.
6	**Loop Rotation:** Program loops are rotated if the resulting program code is faster and more efficient.

OPTIMIZE level 6 includes all optimizations of levels 0 to 5.

NOTE

The global optimizations beginning with level 4 are performed by the compiler completely in memory and do not utilize temporary disk files. If there is not enough memory available to complete the optimization, global optimization is only partially completed, if at all. In this event, the following error message displays:

```
*** can't optimize function filename, no memory available
```

The code produced is less optimal, but nonetheless correct. To resolve this problem, try to either write smaller C functions or increase the amount of available memory for the compiler.

Example:

```
C51 SAMPLE.C OPTIMIZE (4)

C51 SAMPLE.C OPTIMIZE (0)

#pragma ot(6, SIZE)

#pragma ot(size)
```

ORDER

Abbreviation: OR

Arguments: None.

Default: The variables are not ordered.

Description: The **ORDER** directive instructs C51 to order all variables in memory according to their order of definition in the C source file. **ORDER** disables the hash algorithm used by the C compiler. The C51 compiler runs a little slower.

Example:

```
C51 SAMPLE.C ORDER

#pragma OR
```

PAGELENGTH

Abbreviation: PL

Arguments: A decimal number up to 65535 enclosed in parentheses.

Default: PAGELENGTH (60)

Description: The **PAGELENGTH** directive specifies the number of lines printed per page in the listing file. The default is 60 lines per page. This includes headers and empty lines.

See Also: PAGEWIDTH

Example:

```
C51 SAMPLE.C PAGELENGTH (70)

#pragma pl (70)
```

PAGEWIDTH

Abbreviation: PW

Arguments: A decimal number in range 78 to 132 enclosed in parentheses.

Default: PAGEWIDTH (132)

Description: The **PAGEWIDTH** directive specifies the number of characters per line that can be printed to the listing file. Lines containing more than the specified number of characters are broken into two or more lines.

See Also: PAGELENGTH

Example:

```
C51 SAMPLE.C PAGEWIDTH(79)

#pragma pw(79)
```

PREPRINT

Abbreviation: PP

Arguments: An optional filename enclosed in parentheses.

Default: No preprocessor listing is generated.

Description: The **PREPRINT** directive instructs the compiler to produce a preprocessor listing. Macro calls are expanded and comments are deleted. If **PREPRINT** is used without an argument, the source filename with the extension **.I** is defined as the list filename. If this is not desired, you must specify a filename. By default, C51 does not generate a preprocessor output file.

NOTE
*The **PREPRINT** directive may be specified only on the command line. It may not be specified in the C source file by means of the **#pragma** directive.*

Example:

```
C51 SAMPLE.C PREPRINT

C51 SAMPLE.C PP (PREPRO.LSI)
```

PRINT / NOPRINT

Abbreviation: PR / NOPR

Arguments: An optional filename enclosed in parentheses.

Default: PRINT (*basename*.LST)

Description: The compiler produces a listing of each compiled program using the extension **.LST**. Using the **PRINT** directive, you may redefine the name of the listing file.

The **NOPRINT** directive prevents the compiler from generating a listing file.

Example:

```
C51 SAMPLE.C PRINT(CON:)

#pragma pr (\usr\list\sample.lst)

C51 SAMPLE.C NOPRINT

#pragma nopr
```

REGFILE

Abbreviation: RF

Arguments: A file name enclosed in parentheses.

Default: None.

Description: With **REGFILE**, the C51 compiler reads a register definition file for global register optimization. The register definition file specifies the register usage of external functions. With this information the C51 compiler *knows* about the register utilization of external functions. This enables global program-wide register optimization.

Example:

```
C51 SAMPLE.C REGFILE(sample.reg)

#pragma REGFILE(sample.reg)
```

REGISTERBANK

Abbreviation: RB

Arguments: A number between 0 and 3 enclosed in parentheses.

Default: REGISTERBANK (0)

Description: The **REGISTERBANK** directive selects which register bank to use for subsequent functions declared in the source file. Resulting code may use the absolute form of register access when the absolute register number can be computed. The **using** function attribute supersedes the effects of the **REGISTERBANK** directive.

NOTE
*Unlike the **using** function attribute, the **REGISTERBANK** control does not switch the register bank.*

Functions that return a value to the caller, must always use the same register bank as the caller. If the register banks are not the same, return values may be returned in registers of the wrong register bank.

*The **REGISTERBANK** directive may appear more than once in a source program; however, the directive is ignored if used within a function declaration.*

Example:

```
C51 SAMPLE.C REGISTERBANK(1)

#pragma rb(3)
```

REGPARMS / NOREGPARMS

Abbreviation: None.

Arguments: None.

Default: REGPARMS

Description: The **REGPARMS** directive enables the compiler to generate code that passes up to three function arguments in registers. This type of parameter passing is similar to what you would use when writing in assembly and is significantly faster than storing function arguments in memory. Parameters that cannot be located in registers are passed using fixed memory areas.

The **NOREGPARMS** directive forces all function arguments to be passed in fixed memory areas. This directive generates parameter passing code which is compatible with C51, Version 2 and Version 1.

NOTE
*You may specify both the **REGPARMS** and **NOREGPARMS** directives several times within a source program. This allows you to create some program sections with register parameters and other sections using the old style of parameter passing. Use **NOREGPARMS** to access existing older assembler functions or library files without having to reassemble or recompile them. This is illustrated in the following example program.*

```
#pragma NOREGPARMS          /* Parm passing-old method */
extern int old_func (int, char);

#pragma REGPARMS            /* Parm passing-new method */
extern int new_func (int, char);

main  ()  {
        char a;
        int  x1, x2;
        x1 = old_func (x2, a);
        x1 = new_func (x2, a);
}
```

Example:

```
C51 SAMPLE.C NOREGPARMS
```

ROM

Abbreviation: None.

Arguments: (SMALL), (COMPACT), or (LARGE)

Default: ROM (LARGE)

Description: You use the **ROM** directive to specify the size of the program memory. This directive affects the coding of the **JMP** and **CALL** instructions.

Memory Size	Description
SMALL	**CALL** and **JMP** instructions are coded as **ACALL** and **AJMP**. The maximum program size may be 2 KBytes. The entire program must be allocated within the 2 KByte program memory space.
COMPACT	**CALL** instructions are coded as **LCALL**. **JMP** instructions are coded as **AJMP** within a function. The size of a function must not exceed 2 KBytes. The entire program may, however, comprise a maximum of 64 KBytes. The type of application determines whether or not **ROM (COMPACT)** is more advantageous than **ROM (LARGE)**. Any code space saving advantages in using **ROM (COMPACT)** must be empirically determined.
LARGE	**CALL** and **JMP** instructions are coded as **LCALL** and **LJMP**. This allows you to use the entire address space without any restrictions. Program size is limited to 64 KBytes. Function size is also limited to 64 KBytes.

See Also: SMALL, COMPACT, LARGE

Example:

```
C51 SAMPLE.C ROM (SMALL)

#pragma ROM (SMALL)
```

SAVE / RESTORE

Abbreviation: None.

Arguments: None.

Default: None.

Description: The **SAVE** directive stores the current settings of **AREGS** and **REGPARMS**, and the current **OPTIMIZE** level and emphasis. These settings are saved, for example, before an **#include** directive and restored afterwards using **RESTORE**.

The **RESTORE** directive fetches the values of the last **SAVE** directive from the save stack.

The maximum nesting depth for **SAVE** directives is eight levels.

NOTE
***SAVE** and **RESTORE** may be specified only as an argument of a **#pragma** statement. You may not specify this control option in the command line.*

Example:

```
#pragma save
#pragma noregparms

extern void test1 (char c, int i);
extern char test2 (long l, float f);

#pragma restore
```

In the above example, parameter passing in registers is disabled for the two external functions, **`test1`** and **`test2`**. The settings at the time of the **SAVE** directive are restored by the **RESTORE** directive.

SMALL

Abbreviation: SM

Arguments: None.

Default: SMALL

Description: This directive implements the **SMALL** memory model. The **SMALL** memory model places all function variables and local data segments in the internal data memory of the 8051 system. This allows for very efficient access to data objects. The address space of the **SMALL** memory model, however, is limited.

Regardless of memory model type, you may declare variables in any of the 8051 memory ranges. However, placing frequently used directives (such as loop counters and array indices) in internal data memory significantly improves system performance.

NOTE
The stack required for function calls is always placed in IDATA memory.

*Always start by using the **SMALL** memory model. Then, as your application grows, you can place large variables and data in other memory areas by explicitly declaring the memory area with the variable declaration.*

See Also: COMPACT, LARGE, ROM

Example:

```
C51 SAMPLE.C SMALL

#pragma small
```

SRC

Abbreviation: None.

Arguments: An optional filename in parentheses.

Default: None.

Description: Use the **SRC** directive to create an assembler source file instead of an object file. This source file may be assembled with the A51 assembler. If a filename is not specified in parentheses, the base name and path of the C source file are used with the **.SRC** extension.

NOTE
The compiler cannot simultaneously produce a source file and an object file.

See Also: ASM, ENDASM

Example:

```
C51 SAMPLE.C SRC

C51 SAMPLE.C SRC(SML.A51)
```

SYMBOLS

Abbreviation: SB

Arguments: None.

Default: No list of symbols is generated.

Description: The **SYMBOLS** directive instructs the compiler to generate a list of all symbols used in and by the program module being compiled. This list is included in the listing file. The memory category, memory type, offset, and size are listed for each symbolic object.

Example:

```
C51 SAMPLE.C SYMBOLS

#pragma SYMBOLS
```

The following excerpt from a listing file shows the symbol listing:

```
NAME              CLASS   MSPACE  TYPE    OFFSET  SIZE
====              =====   ======  ====    ======  ====
EA . . . . . . . . ABSBIT  -----   BIT     00AFH   1
update . . . . . . PUBLIC  CODE    PROC    -----   -----
  dtime. . . . . . PARAM   DATA    PTR     0000H   3
setime . . . . . . PUBLIC  CODE    PROC    -----   -----
  mode . . . . . . PARAM   DATA    PTR     0000H   3
  dtime. . . . . . PARAM   DATA    PTR     0003H   3
  setuptime. . . . AUTO    DATA    STRUCT  0006H   3
time . . . . . . . * TAG * -----   STRUCT  -----   3
  hour . . . . . . MEMBER  DATA    U_CHAR  0000H   1
  min. . . . . . . MEMBER  DATA    U_CHAR  0001H   1
  sec. . . . . . . MEMBER  DATA    U_CHAR  0002H   1
SBUF . . . . . . . SFR     DATA    U_CHAR  0099H   1
ring . . . . . . . PUBLIC  DATA    BIT     0001H   1
SCON . . . . . . . SFR     DATA    U_CHAR  0098H   1
TMOD . . . . . . . SFR     DATA    U_CHAR  0089H   1
TCON . . . . . . . SFR     DATA    U_CHAR  0088H   1
mnu. . . . . . . . PUBLIC  CODE    ARRAY   00FDH   119
```

WARNINGLEVEL

Abbreviation: WL

Arguments: A number from 0-2.

Default: WARNINGLEVEL (2)

Description: The **WARNINGLEVEL** directive allows you to suppress compiler warnings. Refer to "Chapter 7. Error Messages" on page 155 for a full list of the compiler warnings.

Warning Level	Description
0	Disables almost all compiler warnings.
1	Lists only those warnings which may generate incorrect code.
2 (Default)	Lists all WARNING messages including warnings about unused variables, expressions, or labels.

Example:

```
C51 SAMPLE.C WL (1)

#pragma WARNINGLEVEL (0)
```

2

Chapter 3. Language Extensions

C51 provides a number of extensions for ANSI Standard C. Most of these provide direct support for elements of the 8051 architecture. C51 includes extensions for:

- Memory Types and Areas on the 8051
- Memory Models
- Memory Type Specifiers
- Variable Data Type Specifiers
- Bit variables and bit-addressable data
- Special Function Registers
- Pointers
- Function Attributes

The following sections describe each of these in detail.

Keywords

To facilitate many of the features of the 8051, C51 adds a number of new keywords to the scope of the C language. The following is a list of the keywords available in C51, Version 4:

_at_	**idata**	**sfr**
alien	**interrupt**	**sfr16**
bdata	**large**	**small**
bit	**pdata**	**_task_**
code	**_priority_**	**using**
compact	**reentrant**	**xdata**
data	**sbit**	

You can disable these extensions using the **NOEXTEND** control directive. Refer to "Chapter 2. Compiling with C51" on page 3 for more information.

8051 Memory Areas

The 8051 architecture supports a number of physically separate memory areas or memory spaces for program and data. Each memory area offers certain advantages and disadvantages. There are memory spaces that can be read from but not written to, memory spaces that can be read from or written to, and memory spaces that can be read from or written to more quickly than other memory spaces. This wide variety of memory space is quite different from most mainframe, minicomputer, and microcomputer architectures where the program, data, and constants are all loaded into the same physical memory space within the computer. Refer to the *Intel 8-Bit Embedded Controllers* handbook or other 8051 data books for more information about the 8051 memory architecture.

Program Memory

Program (CODE) memory is read only; it cannot be written to. Program memory may reside within the 8051 CPU, it may be external, or it may be both, depending upon the 8051 derivative and the hardware design. There may be up to 64 KBytes of program memory. Program code including all functions and library routines are stored in program memory. Constant variables may be stored in program memory, as well. The 8051 executes programs stored in program memory only.

Program memory can be accessed by using the **code** memory type specifier in C51.

Internal Data Memory

Internal data memory resides within the 8051 CPU and can be read from and written to. Up to 256 bytes of internal data memory are available depending upon the 8051 derivative. The first 128 bytes of internal data memory are both directly addressable and indirectly addressable. The upper 128 bytes of data memory (from 0x80 to 0xFF) can be addressed only indirectly. There is also a 16 byte area starting at 20h that is bit-addressable.

Access to internal data memory is very fast because it can be accessed using an 8-bit address. However, internal data memory is limited to a maximum of 256 bytes.

Internal data can be broken down into three distinct data types when using C51: **data**, **idata**, and **bdata**.

The **data** memory specifier always refers to the first 128 bytes of internal data memory. Variables stored here are accessed using direct addressing.

The **idata** memory specifier refers to all 256 bytes of internal data memory; however, this memory type specifier code is generated by indirect addressing which is slower than direct addressing.

The **bdata** memory specifier refers to the 16 bytes of bit-addressable memory in the internal data area (20h to 2Fh). This memory type specifier allows you to declare data types that can also be accessed at the bit level.

External Data Memory

External data memory can be read from and written to and is physically located externally from the 8051 CPU. Access to external data is very slow when compared to access to internal data. This is because external data memory is accessed indirectly through the data pointer (**DPTR**) register which must be loaded with a 16-bit address before accessing the external memory.

There may be up to 64 KBytes of external data memory; though, this address space does not necessarily have to be used as memory. Your hardware design may map peripheral devices into the memory space. If this is the case, your program would access external data memory to program and control the peripheral. This technique is referred to as memory-mapped I/O.

There are two different data types in C51 with which you may access external data: **xdata** and **pdata**.

The **xdata** memory specifier refers to any location in the 64 KByte address space of external data memory.

The **pdata** memory type specifier refers to only 1 page or 256 bytes of external data memory. See “Compact Model” on page 62 for more information on **pdata**.

Special Function Register Memory

The 8051 also provides 128 bytes of memory for Special Function Registers (SFRs). SFRs are bit, byte, or word-sized registers that are used to control timers, counters, serial I/O, port I/O, and peripherals. Refer to "Special Function Registers" on page 68 for more information on SFRs.

Memory Models

The memory model determines which default memory type to use for function arguments, automatic variables, and declarations with no explicit memory type specifier. You specify the memory model on the C51 command line using the **SMALL**, **COMPACT**, and **LARGE** control directives. Refer to "Control Directives" on page 6 for more information about these directives.

NOTE
*Except in very special selected applications, always use the default **SMALL** memory model. It generates the fastest, most efficient code.*

By explicitly declaring a variable with a memory type specifier, you may override the default memory type imposed by the memory model . Refer to "Memory Types" on page 62 for more information.

Small Model

In this model, all variables, by default, reside in the internal data memory of the 8051 system. (This is the same as if they were declared explicitly using the **data** memory type specifier.) In this memory model, variable access is very efficient. However, all objects, as well as the stack must fit into the internal RAM. Stack size is critical because the real stack size depends upon the nesting depth of the various functions. Typically, if the linker/locator is configured to overlay variables in the internal data memory, the small model is the best model to use.

Compact Model

Using the compact model, all variables, by default, reside in one page of external data memory. (This is as if they were explicitly declared using the **pdata** memory type specifier.) This memory model can accommodate a maximum of 256 bytes of variables. The limitation is due to the addressing scheme used, which is indirect through registers R0 and R1 (@R0, @R1). This memory model is not as efficient as the small model, therefore, variable access is not as fast. However, the compact model is faster than the large model.

When using the compact model, C51 accesses external memory with instructions that utilize the @R0 and @R1 operands. R0 and R1 are byte registers and provide only the low-order byte of the address. If the compact model is used with more than 256 bytes of external memory, the high-order address byte (or page) is provided by Port 2 on the 8051. In this case, you must initialize Port 2 with the proper external memory page to use. This can be done in the startup code. You must also specify the starting address for **PDATA** to the linker. Refer to “STARTUP.A51” on page 114 for more information on using the compact model.

Large Model

In the large model, all variables, by default, reside in external data memory (up to 64 KBytes). (This is the same as if they were explicitly declared using the **xdata** memory type specifier.) The data pointer (**DPTR**) is used for addressing. Memory access through this data pointer is inefficient, especially on variables with a length of two or more bytes. This type of data access mechanism generates more code than the small or compact models.

Memory Types

The C51 compiler explicitly supports the architecture of the 8051 and its derivatives and provides access to all memory areas of the 8051. Each variable may be explicitly assigned to a specific memory space.

Accessing the internal data memory is considerably faster than accessing the external data memory. For this reason, place frequently used variables in internal data memory. Place larger, less frequently used variables in external data memory.

Explicitly Declared Memory Types

By including a memory type specifier in the variable declaration, you may specify where variables are stored.

The following table summarizes the available memory type specifiers.

Memory Type	Description
code	Program memory (64 KBytes); accessed by opcode MOVC @A+DPTR.
data	Directly addressable internal data memory; fastest access to variables (128 bytes).
idata	Indirectly addressable internal data memory; accessed across the full internal address space (256 bytes).
bdata	Bit-addressable internal data memory; allows mixed bit and byte access (16 bytes).
xdata	External data memory (64 KBytes); accessed by opcode MOVX @DPTR.
pdata	Paged (256 bytes) external data memory; accessed by opcode MOVX @R*n*.

As with the **signed** and **unsigned** attributes, you may include memory type specifiers in the variable declaration.

Example:

```
char data var1;
char code text[] = "ENTER PARAMETER:";
unsigned long xdata array[100];
float idata x,y,z;
unsigned int pdata dimension;
unsigned char xdata vector[10][4][4];
char bdata flags;
```

NOTE

For compatibility with previous versions of the C51 compiler, you may specify the memory area before the data type. For example, the following declaration

```
data char x;
```

is equivalent to

```
char data x;
```

Nonetheless, this feature should not be used in new programs because it may not be supported in future versions of the C51 compiler.

Implicit Memory Types

If the memory type specifier is omitted in a variable declaration, the default or implicit memory type is automatically selected. Function arguments and automatic variables which cannot be located in registers are also stored in the default memory area.

The default memory type is determined by the **SMALL**, **COMPACT** and **LARGE** compiler control directives. Refer to "Memory Models" on page 61 for more information.

3

Data Types

C51 provides you with a number of basic data types to use in your C programs. C51 offers you the standard C data types and also supports several data types that are unique to the 8051 platform. The following table lists the available C51 data types.

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

† The **bit**, **sbit**, **sfr**, and **sfr16** data types are not provided in ANSI C and are unique to C51. These data types are described in detail in the following sections.

Bit Types

C51 provides you with a **bit** data type which may be used for variable declarations, argument lists, and function return values. A **bit** variable is declared just as other C data types are declared.

Example:

```
static bit done_flag = 0;    /* bit variable */

bit testfunc (               /* bit function */
  bit flag1,                 /* bit arguments */
  bit flag2)
{
.
.
.
return (0);                  /* bit return value */
}
```

All **bit** variables are stored in a bit segment located in the internal memory area of the 8051. Because this area is only 16 bytes long, a maximum of 128 **bit** variables may be declared within any one scope.

Memory types may be included in the declaration of a **bit** variable. However, because **bit** variables are stored in the internal data area of the 8051, the **data** and **idata** memory types only may be included in the declaration. Any other memory types are invalid.

The following restrictions apply to **bit** variables and **bit** declarations:

- Functions which use disabled interrupts (**#pragma disable**), and functions that are declared using an explicit register bank (**using** *n*) cannot return a bit value. The C51 compiler generates an error message for functions of this type that attempt to return a **bit** type.
- A bit cannot be declared as a pointer. For example:

```
bit *ptr;                                                    /* invalid */
```

- An array of type **bit** is invalid. For example:

```
bit ware [5];                                                /* invalid */
```

Bit-addressable Objects

Bit-addressable objects are objects which can be addressed as bytes or as bits. Only data objects that occupy the bit-addressable area of the 8051 internal memory fall into this category. The C51 compiler places variables declared with the **bdata** memory type into this bit-addressable area. You may declare these variables as shown below:

```
int bdata ibase;        /* Bit-addressable int */

char bdata bary [4];    /* Bit-addressable array */
```


The variables **ibase** and **bary** are bit-addressable. Therefore, the individual bits of these variables may be directly accessed and modified. Use the **sbit** keyword to declare new variables that access the bits of variables declared using **bdata**. For example:

```
sbit mybit0 = ibase ^ 0;      /* bit 0 of ibase */
sbit mybit15 = ibase ^ 15;    /* bit 15 of ibase */

sbit Ary07 = bary[0] ^ 7;     /* bit 7 of bary[0] */
sbit Ary37 = bary[3] ^ 7;     /* bit 7 of bary[3] */
```

The above example represents declarations, not assignments to the bits of the **ibase** and **bary** variables declared above. The expression following the carat symbol (**'^'**) in the example, specifies the position of the bit to access with this declaration. This expression must be a constant value. The range depends on the type of the base variable included in the declaration. The range is 0 to 7 for **char** and **unsigned char**, 0 to 15 for **int**, **unsigned int**, **short**, and **unsigned short**, and 0 to 31 for **long** and **unsigned long**.

You may provide external variable declarations for the **sbit** type to access these types in other modules. For example:

```
extern bit mybit0;      /* bit 0 of ibase */
extern bit mybit15;     /* bit 15 of ibase */

extern bit Ary07;       /* bit 7 of bary[0] */
extern bit Ary37;       /* bit 7 of bary[3] */
```

Declarations involving the **sbit** type require that the base object be declared with the memory type **bdata**. The only exceptions are the variants for special function bits. Refer to "Special Function Registers" on page 68 for more information.

The following example shows how to change the **ibase** and **bary** bits using the above declarations.

```
Ary37 = 0;        /* clear bit 7 in bary[3] */
bary[3] = 'a';    /* Byte addressing */
ibase = -1;       /* Word addressing */
mybit15 = 1;      /* set bit 15 in ibase */
```

The **bdata** memory type is handled like the **data** memory type except that variables declared with **bdata** reside in the bit-addressable portion of the internal data memory. Note that the total size of this area of memory may not exceed 16 bytes.

In addition to declaring **sbit** variables for scalar types, you may also declare **sbit** variables for structures and unions. For example:

```
union lft
  {
  float mf;
  long ml;
  };

bdata struct bad
  {
  char m1;
  union lft u;
  } tcp;

sbit tcpf31 = tcp.u.ml ^ 31;        /* bit 31 of float */
sbit tcpm10 = tcp.m1 ^ 0;
sbit tcpm17 = tcp.m1 ^ 7;
```

NOTE

*You may not specify **bit** variables for the bit positions of a **float**. However, you may include the **float** and a **long** in a **union**. Then, you may declare **bit** variables to access the bits in the **long** type.*

*The **sbit** data type uses the specified variable as a base address and adds the bit position to obtain a physical bit address. Physical bit addresses are not equivalent to logical bit positions for certain data types. Physical bit position 0 refers to bit position 0 of the first byte. Physical bit position 8 refers to bit position 0 of the second byte. Because **int** variables are stored high-byte first, bit 0 of the integer is located in bit position 0 of the second byte. This is physical bit position 8 when accessed using an **sbit** data type.*

Special Function Registers

The 8051 family of microprocessors provides you with a distinct memory area for accessing Special Function Registers (SFRs). SFRs are used in your program to control timers, counters, serial I/Os, port I/Os, and peripherals. SFRs reside from address 0x80 to 0xFF and can be accessed as bits, bytes, and words. For more information about special function registers, refer to the *Intel 8-Bit Embedded Controllers* handbook or other 8051 data books.

Within the 8051 family, the number and type of SFRs vary. Note that no SFR names are predefined by the C51 compiler. However, declarations for SFRs are provided in include files.

C51 provides you with a number of include files for various 8051 derivatives. Each file contains declarations for the SFRs available on that derivative. See "8051 Special Function Register Include Files" on page 191 for more information about include files.

C51 provides access to SFRs with the **sfr**, **sfr16**, and **sbit** data types. The following sections describe each of these data types.

sfr

SFRs are declared in the same fashion as other C variables. The only difference is that the data type specified is **sfr** rather than **char** or **int**. For example:

```
sfr P0 = 0x80;    /* Port-0, address 80h */
sfr P1 = 0x90;    /* Port-1, address 90h */
sfr P2 = 0xA0;    /* Port-2, address 0A0h */
sfr P3 = 0xB0;    /* Port-3, address 0B0h */
```

P0, P1, P2, and **P3** are the SFR name declarations. Names for **sfr** variables are defined just like other C variable declarations. Any symbolic name may be used in an **sfr** declaration.

The address specification after the equal sign (=) must be a numeric constant. (Expressions with operators are not allowed.) This constant expression must lie in the SFR address range (0x80 to 0xFF).

sfr16

Many of the newer 8051 derivatives use two SFRs with consecutive addresses to specify 16-bit values. For example, the 8052 uses addresses 0xCC and 0xCD for the low and high bytes of timer/counter 2. C51 provides the **sfr16** data type to access 2 SFRs as a 16-bit SFR.

Access to 16-bit SFRs is possible only when the low byte immediately precedes the high byte. The low byte is used as the address in the **sfr16** declaration. For example:

```
sfr16 T2 = 0xCC;     /* Timer 2: T2L 0CCh, T2H 0CDh */
sfr16 RCAP2 = 0xCA;  /* RCAP2L 0CAh, RCAP2H 0CBh */
```


In this example, **T2** and **RCAP2** are declared as 16-bit special function registers.

The **sfr16** declarations follow the same rules as outlined for **sfr** declarations. Any symbolic name can be used in an **sfr16** declaration. The address specification after the equal sign (**'='**) must be a numeric constant. Expressions with operators are not allowed. The address must be the low byte of the SFR low-byte, high-byte pair.

sbit

With typical 8051 applications, it is often necessary to access individual bits within an SFR. The C51 compiler makes this possible with the **sbit** data type. The **sbit** data type allows you to access bit-addressable SFRs. For example:

```
sbit EA = 0xAF;
```

This declaration defines **EA** to be the SFR bit at address **0xAF**. On the 8051, this is the *enable all* bit in the interrupt enable register.

NOTE
Not all SFRs are bit-addressable. Only those SFRs whose address is evenly divisible by 8 are bit-addressable. The lower nibble of the SFR's address must be 0 or 8. For example, SFRs at 0xA8 and 0xD0 are bit-addressable, whereas SFRs at 0xC7 and 0xEB are not. To calculate an SFR bit address, add the bit position to the SFR byte address. So, to access bit 6 in the SFR at 0xC8, the SFR bit address would be 0xCE (0xC8 + 6).

Any symbolic name can be used in an **sbit** declaration. The expression to the right of the equal sign (=) specifies an absolute bit address for the symbolic name. There are three variants for specifying the address:

Variant 1: **sfr_name ^ int_constant**

This variant uses a previously declared **sfr** (*sfr_name*) as the base address for the **sbit**. The address of the existing SFR must be evenly divisible by 8. The expression following the carat symbol (^) specifies the position of the bit to access with this declaration. The bit position must be a number in the 0 to 7 range. For example:

```
sfr  PSW = 0xD0;
sfr  IE = 0xA8;
sbit OV = PSW ^ 2;
sbit CY = PSW ^ 7;
sbit EA = IE ^ 7;
```

Variant 2: **int_constant ^ int_constant**

This variant uses an integer constant as the base address for the **sbit**. The base address value must be evenly divisible by 8. The expression following the carat symbol (**'^'**) specifies the position of the bit to access with this declaration. The bit position must be a number in the 0 to 7 range. For example:

```
sbit OV = 0xD0 ^ 2;
sbit CY = 0xD0 ^ 7;
sbit EA = 0xA8 ^ 7;
```

Variant 3: ***int_constant***

This variant uses an absolute bit address for the **sbit**. For example:

```
sbit OV = 0xD2;
sbit CY = 0xD7;
sbit EA = 0xAF;
```

NOTE

Special function bits represent an independent declaration class that may not be interchangeable with other bit declarations or bit fields.

*The **sbit** data type declaration may be used to access individual bits of variables declared with the **bdata** memory type specifier. Refer to "Bit-addressable Objects" on page 66 for more information.*

Absolute Variable Location

Variables may be located at absolute memory locations in your C program source modules using the **_at_** keyword. The usage for this feature is:

⟦*memory_space*⟧ *type variable_name* **_at_** *constant*;

where:

memory_space	is the memory space for the variable. If missing from the declaration, the default memory space is used. Refer to "Memory Models" on page 61 for more information about the default memory space.
type	is the variable type.
variable_name	is the variable name.
constant	is the address at which to locate the variable.

The absolute address following **_at_** must conform to the physical boundaries of the memory space for the variable. C51 checks for invalid address specifications.

The following restrictions apply to absolute variable location:

1. Absolute variables cannot be initialized.

2. Functions and variables of type **bit** cannot be located at an absolute address.

The following example demonstrates how to locate several different variable types using the **_at_** keyword.

```
struct link
  {
  struct link idata *next;
  char        code  *test;
  };

idata struct link list _at_ 0x40;    /* list at idata 0x40 */
xdata char text[256]   _at_ 0xE000;  /* array at xdata 0xE000 */
xdata int i1           _at_ 0x8000;  /* int at xdata 0x8000 */

void main ( void ) {
   link.next = (void *) 0;
   i1        = 0x1234;
   text [0]  = 'a';
}
```


Often, you may wish to declare your variables in one source module and access them in another. Use the following external declarations to access the **_at_** variables defined above in another source file.

```
struct link
  {
  struct link idata *next;
  char        code  *test;
  };

extern idata struct link list;    /* list at idata 0x40 */
extern xdata char text[256];      /* array at xdata 0xE000 */
extern xdata int i1;              /* int at xdata 0x8000 */
```

Pointers

C51 supports the declaration of variable pointers using the ***** character. C51 pointers can be used to perform all operations available in standard C. However, because of the unique architecture of the 8051 and its derivatives, C51 provides two different types of pointers: memory-specific pointers and generic pointers. Each of these pointer types, as well as conversion methods are discussed in the following sections.

Generic Pointers

Generic pointers are declared in the same fashion as standard C pointers. For example:

```
char *s;       /* string ptr */
int *numptr;   /* int ptr */
long *state;   /* Texas */
```

Generic pointers are always stored using three bytes. The first byte is for the memory type, the second is for the high-order byte of the offset, and the third is for the low-order byte of the offset. The following table contains the memory type byte values and their associated memory type.

Memory Type	idata / data / bdata	xdata	pdata	code
Value	0x00	0x01	0xFE	0xFF

Generic pointers may be used to access any variable regardless of its location in 8051 memory space. Many of the C51 library routines use these pointer types for this reason. By using these generic pointers, a function can access data regardless of the memory in which it is stored.

NOTE
The code generated for a generic pointer executes more slowly than the equivalent code generated for a memory-specific pointer. This is because the memory area is not known until run-time. The compiler cannot optimize memory accesses and must generate generic code that can access any memory area. If execution speed is a priority, you should use memory-specific pointers instead of generic pointers wherever possible.

The following code and assembly listing shows the values assigned to generic pointers for variables in different memory areas. Note that the first value is the memory space followed by the high-order byte and low-order byte of the address.

```
stmt level  source
   1        char *c_ptr;          /* char ptr */
   2        int  *i_ptr;          /* int ptr */
   3        long *l_ptr;          /* long ptr */
   4
   5        void main (void)
   6        {
   7   1    char data dj;         /* data vars */
   8   1    int  data dk;
   9   1    long data dl;
  10   1
  11   1    char xdata xj;        /* xdata vars */
  12   1    int  xdata xk;
  13   1    long xdata xl;
  14   1
  15   1    char code cj = 9;     /* code vars */
  16   1    int  code ck = 357;
  17   1    long code cl = 123456789;
  18   1
  19   1
  20   1    c_ptr = &dj;          /* data ptrs */
  21   1    i_ptr = &dk;
  22   1    l_ptr = &dl;
  23   1
  24   1    c_ptr = &xj;          /* xdata ptrs */
  25   1    i_ptr = &xk;
  26   1    l_ptr = &xl;
  27   1
  28   1    c_ptr = &cj;          /* code ptrs */
  29   1    i_ptr = &ck;
  30   1    l_ptr = &cl;
  31   1    }


ASSEMBLY LISTING OF GENERATED OBJECT CODE

      ; FUNCTION main (BEGIN)
                        ; SOURCE LINE # 5
                        ; SOURCE LINE # 6
                        ; SOURCE LINE # 20
0000 750000 R   MOV   c_ptr,#00H
0003 750000 R   MOV   c_ptr+01H,#HIGH dj
0006 750000 R   MOV   c_ptr+02H,#LOW dj
                        ; SOURCE LINE # 21
0009 750000 R   MOV   i_ptr,#00H
000C 750000 R   MOV   i_ptr+01H,#HIGH dk
000F 750000 R   MOV   i_ptr+02H,#LOW dk
                        ; SOURCE LINE # 22
0012 750000 R   MOV   l_ptr,#00H
0015 750000 R   MOV   l_ptr+01H,#HIGH dl
0018 750000 R   MOV   l_ptr+02H,#LOW dl
                        ; SOURCE LINE # 24
001B 750001 R   MOV   c_ptr,#01H
001E 750000 R   MOV   c_ptr+01H,#HIGH xj
```

```
0021 750000 R   MOV   c_ptr+02H,#LOW xj
                      ; SOURCE LINE # 25
0024 750001 R   MOV   i_ptr,#01H
0027 750000 R   MOV   i_ptr+01H,#HIGH xk
002A 750000 R   MOV   i_ptr+02H,#LOW xk
                      ; SOURCE LINE # 26
002D 750001 R   MOV   l_ptr,#01H
0030 750000 R   MOV   l_ptr+01H,#HIGH xl
0033 750000 R   MOV   l_ptr+02H,#LOW xl
                      ; SOURCE LINE # 28
0036 7500FF R   MOV   c_ptr,#0FFH
0039 750000 R   MOV   c_ptr+01H,#HIGH cj
003C 750000 R   MOV   c_ptr+02H,#LOW cj
                      ; SOURCE LINE # 29
003F 7500FF R   MOV   i_ptr,#0FFH
0042 750000 R   MOV   i_ptr+01H,#HIGH ck
0045 750000 R   MOV   i_ptr+02H,#LOW ck
                      ; SOURCE LINE # 30
0048 7500FF R   MOV   l_ptr,#0FFH
004B 750000 R   MOV   l_ptr+01H,#HIGH cl
004E 750000 R   MOV   l_ptr+02H,#LOW cl
                      ; SOURCE LINE # 31
0051 22               RET
     ; FUNCTION main (END)
```

In the above example listing, the generic pointers **c_ptr**, **i_ptr**, and **l_ptr** are all stored in the internal data memory of the 8051. However, you may specify the memory area in which a generic pointer is stored by using a memory type specifier. For example:

```
char * xdata strptr;     /* generic ptr stored in xdata */
int * data numptr;       /* generic ptr stored in data */
long * idata varptr;     /* generic ptr stored in idata */
```

These examples are pointers to variables that may be stored in any memory area. The pointers, however, are stored in **xdata**, **data**, and **idata** respectively.

Memory-specific Pointers

Memory-specific pointers always include a memory type specification in the pointer declaration and always refer to a specific memory area. For example:

```
char data *str;         /* ptr to string in data */
int xdata *numtab;      /* ptr to int(s) in xdata */
long code *powtab;      /* ptr to long(s) in code */
```

Because the memory type is specified at compile-time, the memory type byte required by generic pointers is not needed by memory-specific pointers. Memory-specific pointers can be stored using only one byte (**idata**, **data**, **bdata**, and **pdata** pointers) or two bytes (**code** and **xdata** pointers).

NOTE

The code generated for a memory-specific pointer executes more quickly than the equivalent code generated for a generic pointer. This is because the memory area is known at compile-time rather than at run-time. The compiler can use this information to optimize memory accesses. If execution speed is a priority, you should use memory-specific pointers instead of generic pointers wherever possible.

Like generic pointers, you may specify the memory area in which a memory-specific pointer is stored. To do so, prefix the pointer declaration with a memory type specifier. For example:

```
char data * xdata str;          /* ptr in xdata to data char */
int xdata * data numtab;        /* ptr in data to xdata int */
long code * idata powtab;       /* ptr in idata to code long */
```

Memory-specific pointers may be used to access variables in the declared 8051 memory area only. Memory-specific pointers provide the most efficient method of accessing data objects, but at the cost of reduced flexibility.

The following code and assembly listing shows how pointer values are assigned to memory-specific pointers. Note that the code generated for these pointers is much less involved than the code generated in the generic pointers example listing in the previous section.

```
stmt level  source

   1        char data  *c_ptr;      /* memory-specific char ptr */
   2        int  xdata *i_ptr;      /* memory-specific int ptr */
   3        long code  *l_ptr;      /* memory-specific long ptr */
   4
   5        long code powers_of_ten [] =
   6          {
   7          1L,
   8          10L,
   9          100L,
  10          1000L,
  11          10000L,
  12          100000L,
  13          1000000L,
  14          10000000L,
  15          100000000L
  16          };
  17
  18        void main (void)
  19        {
  20   1    char data strbuf [10];
  21   1    int xdata ringbuf [1000];
  22   1
  23   1    c_ptr = &strbuf [0];
  24   1    i_ptr = &ringbuf [0];
  25   1    l_ptr = &powers_of_ten [0];
  26   1    }


ASSEMBLY LISTING OF GENERATED OBJECT CODE

      ; FUNCTION main (BEGIN)
                        ; SOURCE LINE # 18
                        ; SOURCE LINE # 19
                        ; SOURCE LINE # 23
0000 750000 R     MOV   c_ptr,#LOW strbuf
                        ; SOURCE LINE # 24
0003 750000 R     MOV   i_ptr,#HIGH ringbuf
0006 750000 R     MOV   i_ptr+01H,#LOW ringbuf
                        ; SOURCE LINE # 25
0009 750000 R     MOV   l_ptr,#HIGH powers_of_ten
000C 750000 R     MOV   l_ptr+01H,#LOW powers_of_ten
                        ; SOURCE LINE # 26
000F 22             RET
      ; FUNCTION main (END)
```

Pointer Conversions

C51 can convert between memory-specific pointers and generic pointers. Pointer conversions can be forced by explicit program code using type casts or can be coerced by the compiler.

The C51 compiler coerces a memory-specific pointer into a generic pointer when the memory-specific pointer is passed as an argument to a function which requires a generic pointer. This is the case for functions such as **printf**, **sprintf**, and **gets** which use generic pointers as arguments. For example:


```
extern int printf (void *format, ...);

extern int myfunc (void code *p, int xdata *pq);

int xdata *px;
char code *fmt = "value = %d | %04XH\n";

void debug_print (void)  {
  printf (fmt, *px, *px);          /* fmt is converted */
  myfunc (fmt, px);                /* no conversions */
}
```

In the call to **printf**, the argument `fmt` which represents a 2-byte **code** pointer is automatically converted or coerced into a 3-byte generic pointer. This is done because the prototype for **printf** requires a generic pointer as the first argument.

NOTE

*A memory-specific pointer used as an argument to a function is always converted into a generic pointer if no function prototype is present. This can cause errors if the called function actually expects a shorter pointer as an argument. In order to avoid these kinds of errors in programs, use **#include** files, and prototype all external functions. This guarantees conversion of the necessary types by the compiler and increases the likelihood that the compiler detects type conversion errors.*

The following table details the process involved in converting generic pointers (generic *****) to memory-specific pointers (**code ***, **xdata ***, **idata ***, **data ***, **pdata ***).

Conversion Type	Description
generic * to **code** *	The offset section (2 bytes) of the generic pointer is used.
generic * to **xdata** *	The offset section (2 bytes) of the generic pointer is used.
generic * to **data ***	The low-order byte of the generic pointer offset is used. The high-order byte is discarded.
generic * to i**data** *	The low-order byte of the generic pointer offset is used. The high-order byte is discarded.
generic * to **pdata** *	The low-order byte of the generic pointer offset is used. The high-order byte is discarded.

The following table describes the process involved in converting memory-specific pointers (**code ***, **xdata ***, **idata ***, **data ***, **pdata ***) to generic pointers (generic *****).

Conversion Type	Description
xdata * to generic *	The memory type of the generic pointer is set to 0x01 for **xdata**. The 2-byte offset of the **xdata** * is used.
code * to generic *	The memory type of the generic pointer is set to 0xFF for **code**. The 2-byte offset of the **code** * is used.
idata * to generic * **data** * to generic *	The memory type of the generic pointer is set to 0x00 for **idata** / **data**. The 1-byte offset of the **idata *** / **data *** is converted to an **unsigned int** and used as the offset.
pdata * to generic *	The memory type of the generic pointer is set to 0xFE for **pdata**. The 1-byte offset of the **pdata *** is converted to an **unsigned int** and used as the offset.

The following listing illustrates a few pointer conversions and the resulting code:

```
stmt level  source
   1        int *p1;           /* generic ptr (3 bytes) */
   2        int xdata *p2;     /* xdata ptr (2 bytes) */
   3        int idata *p3;     /* idata ptr (1 byte) */
   4        int code *p4;      /* code ptr (2 bytes */
   5
   6        void pconvert (void) {
   7   1    p1 = p2;           /* xdata* to generic* */
   8   1    p1 = p3;           /* idata* to generic* */
   9   1    p1 = p4;           /* code*  to generic* */
  10   1
  11   1    p4 = p1;           /* generic* to code* */
  12   1    p3 = p1;           /* generic* to idata* */
  13   1    p2 = p1;           /* generic* to xdata* */
  14   1
  15   1    p2 = p3;           /* idata* to xdata* (WARN) */
*** WARNING 259 IN LINE 15 OF P.C: pointer: different mspace
  16   1    p3 = p4;           /* code*  to idata* (WARN) */
*** WARNING 259 IN LINE 16 OF P.C: pointer: different mspace
  17   1    }

ASSEMBLY LISTING OF GENERATED OBJECT CODE
      ; FUNCTION pconvert (BEGIN)
                        ; SOURCE LINE # 7
0000 750001 R   MOV  p1,#01H
0003 850000 R   MOV  p1+01H,p2
0006 850000 R   MOV  p1+02H,p2+01H
                        ; SOURCE LINE # 8
0009 750000 R   MOV  p1,#00H
000C 750000 R   MOV  p1+01H,#00H
000F 850000 R   MOV  p1+02H,p3
                        ; SOURCE LINE # 9
0012 7B05       MOV  R3,#0FFH
0014 AA00   R   MOV  R2,p4
0016 A900   R   MOV  R1,p4+01H
0018 8B00   R   MOV  p1,R3
001A 8A00   R   MOV  p1+01H,R2
001C 8900   R   MOV  p1+02H,R1
                        ; SOURCE LINE # 11
001E AE02       MOV  R6,AR2
0020 AF01       MOV  R7,AR1
0022 8E00   R   MOV  p4,R6
0024 8F00   R   MOV  p4+01H,R7
                        ; SOURCE LINE # 12
0026 AF01       MOV  R7,AR1
0028 8F00   R   MOV  p3,R7
                        ; SOURCE LINE # 13
002A AE02       MOV  R6,AR2
002C 8E00   R   MOV  p2,R6
002E 8F00   R   MOV  p2+01H,R7
                        ; SOURCE LINE # 15
0030 750000 R   MOV  p2,#00H
0033 8F00   R   MOV  p2+01H,R7
                        ; SOURCE LINE # 16
0035 850000 R   MOV  p3,p4+01H
                        ; SOURCE LINE # 17
0038 22         RET
      ; FUNCTION pconvert (END)
```

Abstract Pointers

Abstract pointer types let you access fixed memory locations in any memory area. You may also use abstract pointers to call functions located at absolute or fixed addresses.

Abstract pointer types are described here through code examples which use the following variables.

```
char xdata *px;       /* ptr to xdata */
char idata *pi;       /* ptr to idata */
char code  *pc;       /* ptr to code  */

char c;               /* char variable in data space */
int  i;               /* int variable in data space */
```


The following example assigns the address of the **main** C function to a pointer (stored in **data** memory) to a **char** stored in **code** memory.

Source	`pc = (void *) main;`
Object	`0000 750000  R     MOV     pc,#HIGH main` `0003 750000  R     MOV     pc+01H,#LOW main`

The following example casts the address of the variable **i** (which is an **int data** *) to a pointer to a **char** in **idata**. Since **i** is stored in **data** and since indirectly accessed **data** is **idata**, this pointer conversion is valid.

Source	`pi = (char idata *) &i;`
Object	`0000 750000  R     MOV     pi,#LOW i`

The following example casts a pointer to a **char** in **xdata** to a pointer to a **char** in **idata**. Since **xdata** pointers occupy 2 bytes and **idata** pointers occupy 1 byte, this pointer conversion may not yield the desired results since the upper byte of the **xdata** pointer is ignored. Refer to "Pointer Conversions" on page 78 for more information about converting between different pointer types.

Source	`pi = (char idata *) px;`
Object	`0000 850000  R     MOV     pi,px+01H`

The following example casts 0x1234 as a pointer to a **char** in **code** memory.

Source	`pc = (char code *) 0x1234;`
Object	`0000 750012  R     MOV     pc,#012H` `0003 750034  R     MOV     pc+01H,#034H`

The following example casts 0xFF00 as a function pointer that takes no arguments and returns an **int**, invokes the function, and assigns the return value to the variable **i**. The portion of this example that performs the function pointer type cast is: **`((int (code *)(void)) 0xFF00)`**. By adding the argument list to the end of the function pointer, the compiler can correctly invoke the function.

Source	`i = ((int (code *)(void)) 0xFF00) ();`

Object	`0000 12FF00 LCALL 0FF00H`
	`0003 8E00 R MOV i,R6`
	`0005 8F00 R MOV i+01H,R7`

The following example casts 0x8000 as a pointer to a **char** in **code** memory, extracts the **char** pointed to, and assigns it to the variable **c**.

Source	`c = *((char code *) 0x8000);`

Object	`0000 908000 MOV DPTR,#08000H`
	`0003 E4 CLR A`
	`0004 93 MOVC A,@A+DPTR`
	`0005 F500 R MOV c,A`

The following example casts 0xFF00 as a pointer to a **char** in **xdata** memory, extracts the **char** pointed to, and adds it to the variable **c**.

Source	`c += *((char xdata *) 0xFF00);`

Object	`0000 90FF00 MOV DPTR,#0FF00H`
	`0003 E0 MOVX A,@DPTR`
	`0004 2500 R ADD A,c`
	`0006 F500 R MOV c,A`

The following example casts 0xF0 as a pointer to a **char** in **idata** memory, extracts the **char** pointed to, and adds it to the variable **c**.

Source	`c += *((char idata *) 0xF0);`

Object	`0000 78F0 MOV R0,#0F0H`
	`0002 E6 MOV A,@R0`
	`0003 2500 R ADD A,c`
	`0005 F500 R MOV c,A`

The following example casts 0xE8 as a pointer to a **char** in **pdata** memory, extracts the **char** pointed to, and adds it to the variable **c**.

Source	`c += *((char pdata *) 0xE8);`

Object	`0000 78E8 MOV R0,#0E8H`
	`0002 E2 MOVX A,@R0`
	`0003 2500 R ADD A,c`
	`0005 F500 R MOV c,A`

The following example casts 0x2100 as a pointer to an **int** in **code** memory, extracts the **int** pointed to, and assigns it to the variable **i**.

Source	`i = *((int code *) 0x2100);`

Object	`0000 902100 MOV DPTR,#02100H`
	`0003 E4 CLR A`
	`0004 93 MOVC A,@A+DPTR`
	`0005 FE MOV R6,A`
	`0006 7401 MOV A,#01H`
	`0008 93 MOVC A,@A+DPTR`
	`0009 8E00 R MOV i,R6`
	`000B F500 R MOV i+01H,A`

The following example casts 0x4000 as a pointer to a pointer in **xdata** that points to a **char** in **xdata**. The assignment extracts the pointer stored in **xdata** that points to the **char** which is also stored in **xdata**.

Source	`px = *((char xdata * xdata *) 0x4000);`

Object	`0000 904000 MOV DPTR,#04000H`
	`0003 E0 MOVX A,@DPTR`
	`0004 FE MOV R6,A`
	`0005 A3 INC DPTR`
	`0006 E0 MOVX A,@DPTR`
	`0007 8E00 R MOV px,R6`
	`0009 F500 R MOV px+01H,A`

Like the previous example, this example casts 0x4000 as a pointer to a pointer in **xdata** that points to a **char** in **xdata**. However, the pointer is accessed as an array of pointers in **xdata**. The assignment accesses array element 0 (which is stored at 0x4000 in **xdata**) and extracts the pointer there that points to the **char** stored in **xdata**.

Source	`px = ((char xdata * xdata *) 0x4000) [0];`

Object	`0000 904000 MOV DPTR,#04000H`
	`0003 E0 MOVX A,@DPTR`
	`0004 FE MOV R6,A`
	`0005 A3 INC DPTR`
	`0006 E0 MOVX A,@DPTR`
	`0007 8E00 R MOV px,R6`
	`0009 F500 R MOV px+01H,A`

The following example is identical to the previous one except that the assignment accesses element 1 from the array. Since the object pointed to is a pointer in **xdata** (to a **char**), the size of each element in the array is 2 bytes. The assignment accesses array element 1 (which is stored at 0x4002 in **xdata**) and extracts the pointer there that points to the **char** stored in **xdata**.

Source	`px = ((char xdata * xdata *) 0x4000) [1];`

Object	`0000 904002 MOV DPTR,#04002H`
	`0003 E0 MOVX A,@DPTR`
	`0004 FE MOV R6,A`
	`0005 A3 INC DPTR`
	`0006 E0 MOVX A,@DPTR`
	`0007 8E00 R MOV px,R6`
	`0009 F500 R MOV px+01H,A`

Function Declarations

C51 provides you with a number of extensions for standard C function declarations. These extensions allow you to:

- Specify a function as an interrupt procedure
- Choose the register bank used
- Select the memory model
- Specify reentrancy
- Specify alien (PL/M-51) functions

You include these extensions or attributes (many of which may be combined) in the function declaration. Use the following standard format for your C51 function declarations.

⟦*return_type*⟧ *funcname* (⟦*args*⟧) ⟦{**small** | **compact** | **large**}⟧
⟦**reentrant**⟧ ⟦**interrupt** ***n***⟧ ⟦**using** ***n***⟧

where:

return_type	is the type of the value returned from the function. If no type is specified, **int** is assumed.
funcname	is the name of the function.
args	is the argument list for the function.
small, **compact**, or **large**	is the explicit memory model for the function.
reentrant	indicates that the function is recursive or reentrant.
interrupt	indicates that the function is an interrupt function.
using	specifies which register bank the function uses.

Descriptions of these attributes and other features are described in detail in the following sections.

Function Parameters and the Stack

The stack pointer on the 8051 accesses internal data memory only. C51 locates the stack area immediately following all variables in the internal data memory. The stack pointer accesses internal memory indirectly and can use all of the internal data memory up to the 0xFF limit.

The total stack space is quite limited: only 256 bytes maximum. Rather than consume stack space with function parameters or arguments, C51 assigns a fixed memory location for each function parameter. When a function is called, the caller must copy the arguments into the assigned memory locations before transferring control to the desired function. The function then extracts its parameters, as needed, from these fixed memory locations. Only the return address is stored on the stack during this process. Interrupt functions require more stack space because they must switch register banks and save the values of a few registers on the stack.

By default, the C51 compiler passes up to three function arguments in registers. This enhances speed performance. For more information, refer to “Passing Parameters in Registers” on page 87.

NOTE
Some 8051 derivatives provide as little as 64 bytes of internal memory. The 8051 provides 128 and the 8052 provides 256. Take this into consideration when determining which memory model to use, because the amount of internal data memory directly affects the amount of stack space.

Passing Parameters in Registers

The C51 compiler allows up to three function arguments to be passed in CPU registers. This mechanism significantly improves system performance as arguments do not have to be written to and read from memory. Argument or parameter passing can be controlled by the **REGPARMS** and **NOREGPARMS** control directives defined in the previous chapter.

The following table details the registers used for different argument positions and data types.

Argument Number	char, 1-byte ptr	int, 2-byte ptr	long, float	generic ptr
1	R7	R6 & R7	R4—R7	R1—R3
2	R5	R4 & R5	R4—R7	R1—R3
3	R3	R2 & R3		R1—R3

If no registers are available for argument passing, fixed memory locations are used for function parameters.

Function Return Values

CPU registers are always used for function return values. The following table lists the return types and the registers used for each.

Return Type	Register	Description
bit	Carry Flag	
char, **unsigned char**, 1-byte ptr	R7	
int, **unsigned int**, 2-byte ptr	R6 & R7	MSB in R6, LSB in R7
long, **unsigned long**	R4-R7	MSB in R4, LSB in R7
float	R4-R7	32-Bit IEEE format
generic ptr	R1-R3	Memory type in R3, MSB R2, LSB R1

NOTE
*If the first parameter of a function is of type **bit**, other parameters are not passed in registers. This is because the parameters that can be passed in registers are out of sequence with the numbering scheme shown above. For this reason, **bit** parameters should be declared at the end of the argument list.*

Specifying the Memory Model for a Function

C51 functions normally use the default memory model to determine which memory space to use for function arguments and local variables. Refer to "Memory Models" on page 61 for more information.

You may, however, specify which memory model to use for a single function by including the **small**, **compact**, or **large** function attribute in the function declaration. For example:

```
#pragma small         /* Default to small model */

extern int calc (char i, int b) large reentrant;
extern int func (int i, float f) large;
extern void *tcp (char xdata *xp, int ndx) small;


int mtest (int i, int y)            /* Small model */
  {
  return (i * y + y * i + func(-1, 4.75));
  }


int large_func (int i, int k) large /* Large model */
  {
  return (mtest (i, k) + 2);
  }
```

The advantage of functions using the **SMALL** memory model is that the local data and function argument parameters are stored in the internal 8051 RAM. Therefore, data access is very efficient. The internal memory is limited, however. Occasionally, the limited amount of internal data memory available when using the small model cannot satisfy the requirements of a very large program, and other memory models must be used. In this situation, you may declare that a function use a different memory model, as shown above.

By specifying the function model attribute in the function declaration, you can select which of the three possible reentrant stacks and frame pointers are used. Stack access in the **SMALL** model is more efficient than in the **LARGE** model.

Specifying the Register Bank for a Function

The lowest 32 bytes of all members of the 8051 family are grouped into 4 banks of 8 registers each. Programs can access these registers as R0 through R7. The register bank is selected by two bits of the program status word (**PSW**). Register banks are useful when processing interrupts or when using a real-time operating system. Rather than saving the 8 registers, the CPU can switch to a different register bank for the duration of the interrupt service routine.

The **using** function attribute is used to specify which register bank a function uses. For example:

```
void rb_function (void) using 3
  {
  .
  .
  .
  }
```


The **using** attribute takes as an argument an integer constant in the 0 to 3 range value. Expressions with operators are not allowed, and the **using** attribute is not allowed in function prototypes. The **using** attribute affects the object code of the function as follows:

- The currently selected register bank is saved on the stack at function entry.
- The specified register bank is set.
- The former register bank is restored before the function is exited.

The following example shows how to specify the **using** function attribute and what the generated assembly code for the function entry and exit looks like.

```
stmt level  source

   1
   2         extern bit alarm;
   3         int alarm_count;
   4         extern void alfunc (bit b0);
   5
   6         void falarm (void) using 3  {
   7   1           alarm_count++;
   8   1           alfunc (alarm = 1);
   9   1         }


ASSEMBLY LISTING OF GENERATED OBJECT CODE

      ; FUNCTION falarm (BEGIN)
0000 C0D0       PUSH  PSW
0002 75D018     MOV   PSW,#018H
                      ; SOURCE LINE # 6
                      ; SOURCE LINE # 7
0005 0500   R   INC   alarm_count+01H
0007 E500   R   MOV   A,alarm_count+01H
0009 7002       JNZ   ?C0002
000B 0500   R   INC   alarm_count
000D  ?C0002:
                      ; SOURCE LINE # 8
000D D3         SETB  C
000E 9200   E   MOV   alarm,C
0010 9200   E   MOV   ?alfunc?BIT,C
0012 120000 E   LCALL alfunc
                      ; SOURCE LINE # 9
0015 D0D0       POP   PSW
0017 22         RET
      ; FUNCTION falarm (END)
```

In the previous example, the code starting at offset **0000h** saves the initial **PSW** on the stack and sets the new register bank. The code starting at offset **0015h** restores the original register bank by popping the original **PSW** from the stack.

The **using** attribute may not be used in functions that return a value in registers. You must exercise extreme care to ensure that register bank switches are performed only in carefully controlled areas. Failure to do so may yield incorrect function results. Even when you use the same register bank, functions declared with the **using** attribute cannot return a bit value.

Typically, the **using** attribute is most useful in functions that also specify the **interrupt** attribute. It is most common to specify a different register bank for each interrupt priority level. Therefore, you could use one register bank for all non-interrupt code, one for the high-level interrupt, and one for the low-level interrupt.

Register Bank Access

The C51 compiler allows you to define the default register bank in a function. The **REGISTERBANK** control directive allows you to specify which default register bank to use for all functions in a source file. This directive, however, does not generate code to switch the register bank.

Upon reset, the 8051 loads the PSW with 00h which selects register bank 0. By default, all non-interrupt functions use register bank 0. To change this, you must:

- Modify the startup code to select a different register bank
- Specify the **REGISTERBANK** control directive along with the new register bank number

By default, the C51 compiler generates code that accesses the registers R0—R7 using absolute addresses. This is done for maximum performance. Absolute register accesses are controlled by the **AREGS** and **NOAREGS** control directives. Functions which employ absolute register accesses must not be called from another function that uses a different register bank. Doing so causes unpredictable results because the called function assumes that a different register bank is selected. To make a function insensitive to the current register bank, the function must be compiled using the **NOAREGS** control directive. This would be useful for a function that was called from the main program and also from an interrupt function that uses a different register bank.

NOTE
The C51 compiler does not and cannot detect a register bank mismatch between functions. Therefore, make sure that functions using alternate register banks call only other functions that do not assume a default register bank.

Refer to "Chapter 2. Compiling with C51" on page 3 for more information regarding the **REGISTERBANK**, **AREGS**, and **NOARGES** directives.

Interrupt Functions

The 8051 and its derivatives provide a number of hardware interrupts that may be used for counting, timing, detecting external events, and sending and receiving data using the serial interface. The standard interrupts found on an 8051 are listed in the following table:

Interrupt Number	Interrupt Description	Address
0	EXTERNAL INT 0	0003h
1	TIMER/COUNTER 0	000Bh
2	EXTERNAL INT 1	0013h
3	TIMER/COUNTER 1	001Bh
4	SERIAL PORT	0023h

As 8051 vendors created new parts, more interrupts were added. The Keil C51 compiler supports interrupt functions for 32 interrupts (0-31). Use the interrupt vector address in the following table to determine the interrupt number.

Interrupt Number	Address
0	**0003h**
1	**000Bh**
2	**0013h**
3	**001Bh**
4	**0023h**
5	**002Bh**
6	**0033h**
7	**003Bh**
8	**0043h**
9	**004Bh**
10	**0053h**
11	**005Bh**
12	**0063h**
13	**006Bh**
14	**0073h**
15	**007Bh**

Interrupt Number	Address
16	**0083h**
17	**008Bh**
18	**0093h**
19	**009Bh**
20	**00A3h**
21	**00ABh**
22	**00B3h**
23	**00BBh**
24	**00C3h**
25	**00CBh**
26	**00D3h**
27	**00DBh**
28	**00E3h**
29	**00EBh**
30	**00F3h**
31	**00FBh**

The C51 compiler provides you with a method of calling a C function when an interrupt occurs. This support lets you create interrupt service routines in C. You need only be concerned with the interrupt number and register bank selection. The compiler automatically generates the interrupt vector and entry and exit code for the interrupt routine. The **interrupt** function attribute, when included in a declaration, specifies that the associated function is an interrupt function. For example:

```
unsigned int  interruptcnt;
unsigned char second;

void timer0 (void) interrupt 1 using 2  {
  if (++interruptcnt == 4000)  {    /* count to 4000 */
    second++;                       /* second counter    */
    interruptcnt = 0;               /* clear int counter */
  }
}
```


The **interrupt** attribute takes as an argument an integer constant in the 0 to 31 value range. Expressions with operators are not allowed, and the **interrupt** attribute is not allowed in function prototypes. The **interrupt** attribute affects the object code of the function as follows:

- The contents of the SFR **ACC**, **B**, **DPH**, **DPL**, and **PSW**, when required, are saved on the stack at the function invocation time.
- All working registers that are used in the interrupt function are stored on the stack if a register bank is not specified with the **using** attribute.
- The working registers and special registers that were saved on the stack are restored before exiting the function.
- The function is terminated by the 8051 **RETI** instruction.

The following sample program shows you how to use the **interrupt** attribute. The program also shows you what the code generated to enter and exit the interrupt function looks like. The **using** function attribute is also used in the example to select a register bank different from that of the non-interrupt program code. However, because no working registers are needed in this function, the code generated to switch the register bank is eliminated by the optimizer.

```
stmt level  source

   1        extern bit alarm;
   2        int alarm_count;
   3
   4
   5        void falarm (void) interrupt 1 using 3  {
   6   1      alarm_count *= 2;
   7   1      alarm = 1;
   8   1      }


ASSEMBLY LISTING OF GENERATED OBJECT CODE

      ; FUNCTION falarm (BEGIN)
0000 C0E0       PUSH  ACC
0002 C0D0       PUSH  PSW
                      ; SOURCE LINE # 5
                      ; SOURCE LINE # 6
0004 E500   R   MOV   A,alarm_count+01H
0006 25E0       ADD   A,ACC
0008 F500   R   MOV   alarm_count+01H,A
000A E500   R   MOV   A,alarm_count
000C 33         RLC   A
000D F500   R   MOV   alarm_count,A
                      ; SOURCE LINE # 7
000F D200   E   SETB   alarm
                      ; SOURCE LINE # 8
0011 D0D0       POP   PSW
0013 D0E0       POP   ACC
0015 32         RETI
      ; FUNCTION falarm (END)
```

In the example above, note that the **ACC** and **PSW** registers are saved at offset **0000h** and restored at offset **0011h.** Note also the **RETI** instruction generated to exit the interrupt.

The following rules apply to interrupt functions.

- No function arguments may be specified for an interrupt function. The compiler emits an error message if an interrupt function is declared with any arguments.
- Interrupt function declarations may not include a return value. They must be declared as void (see the above examples). The compiler emits an error message if any attempt is made to define a return value for the interrupt function. The implicit **int** return value, however, is ignored by the compiler.
- The compiler recognizes direct invocations of interrupt functions and summarily rejects them. It is pointless to invoke interrupt procedures directly, because exiting the procedure causes execution of the **RETI** instruction which affects the hardware interrupt system of the 8051 chip. Because no interrupt request on the part of the hardware existed, the effect of this instruction is indeterminate and usually fatal. Do not call an interrupt function indirectly through a function pointer.
- The compiler generates an interrupt vector for each interrupt function. The code generated for the vector is a jump to the beginning of the interrupt function. Generation of interrupt vectors can be suppressed by including the **NOINTVECTOR** control directive in the C51 command line. In this case, you must provide interrupt vectors from separate assembly modules. Refer to the **INTVECTOR** and **INTERVAL** control directives for more information about the interrupt vector table.
- The C51 compiler allows **interrupt** numbers within the 0 to 31 range. Refer to your 8051 derivative document to determine which interrupts are available.
- Functions that are invoked from an interrupt procedure must function with the same register bank as the interrupt procedure. When the **NOAREGS** directive is not explicitly specified, the compiler may generate absolute register accesses using the register bank selected (by the **using** attribute or by the **REGISTERBANK** control) for that function. Unpredictable results may occur when a function assumes a register bank other than the one currently selected. Refer to "Register Bank Access" on page 91 for more information.

Reentrant Functions

A reentrant function can be shared by several processes at the same time. When a reentrant function is executing, another process can interrupt the execution and then begin to execute that same reentrant function. Normally, functions in C51 cannot be called recursively or in a fashion which causes reentrancy. The reason for this limitation is that function arguments and local variables are stored in fixed memory locations. The **reentrant** function attribute allows you to declare functions that may be reentrant and, therefore, may be called recursively. For example:


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

Reentrant functions can be called recursively and can be called *simultaneously* by two or more processes. Reentrant functions are often required in real-time applications or in situations where interrupt code and non-interrupt code must share a function.

As in the above example, you may selectively define (using the **reentrant** attribute) functions as being reentrant. For each reentrant function, a reentrant stack area is simulated in internal or external memory depending upon the memory model used, as follows:

- Small model reentrant functions simulate the reentrant stack in **idata** memory.
- Compact model reentrant functions simulate the reentrant stack in **pdata** memory.
- Large model reentrant functions simulate the reentrant stack in **xdata** memory.

Reentrant functions use the default memory model to determine which memory space to use for the reentrant stack. You may specify (with the **small**, **compact**, and **large** function attributes) which memory model to use for a function. Refer to "Specifying the Memory Model for a Function" on page 88 for more information about memory models and function declarations.

The following rules apply to functions declared with the **reentrant** attribute.

- **bit** type function arguments may not be used. Local **bit** scalars are also not available. The reentrant capability does not support bit-addressable variables.
- Reentrant functions must not be called from **alien** functions.
- Reentrant function cannot use the **alien** attribute specifier to enable PL/M-51 argument passing conventions.
- A reentrant function may simultaneously have other attributes like **using** and **interrupt** and may include an explicit memory model attribute (**small**, **compact**, **large**).
- Return addresses are stored in the 8051 hardware stack. Any other required **PUSH** and **POP** operations also affect the 8051 hardware stack.
- Reentrant functions using different memory models may be intermixed. However, each reentrant function must be properly prototyped and must include its memory model attribute in the prototype. This is necessary for calling routines to place the function arguments in the proper reentrant stack.
- Each of the three possible reentrant models contains its own reentrant stack area and stack pointer. For example, if **small** and **large** reentrant functions are declared in a module, both small and large reentrant stacks are created along with two associated stack pointers (one for small and one for large).

The reentrant stack simulation architecture is inefficient, but necessary due to a lack of suitable addressing methods available on the 8051. For this reason, use reentrant functions sparingly.

The simulated stack used by reentrant functions has its own stack pointer which is independent of the 8051 stack and stack pointer. The stack and stack pointer are defined and initialized in the **STARTUP.A51** file.

The following table details the stack pointer assembler variable name, data area, and size for each of the three memory models.

Model	Stack Pointer	Stack Area
SMALL	**?C_IBP** (1 Byte)	Indirectly accessible internal memory (idata). 256 bytes maximum stack area.
COMPACT	**?C_PBP** (1 Byte)	Page-addressable external memory (pdata). 256 bytes maximum stack area.
LARGE	**?C_XBP** (2 Bytes)	Externally accessible memory (xdata). 64 KBytes maximum stack area.

The simulated stack area for reentrant functions is organized from top to bottom. The 8051 hardware stack is just the opposite and is organized bottom to top. When using the **SMALL** memory model, both the simulated stack and the 8051 hardware stack share the same memory area but from opposite directions.

The simulated stack and stack pointers are declared and initialized in the C51 startup code in **STARTUP.A51** which can be found in the **LIB** subdirectory. You must modify the startup code to specify which simulated stack(s) to initialize in order to use reentrant functions. You can also modify the starting address for the top of the simulated stack(s) in the startup code. Refer to “STARTUP.A51” on page 114 for more information on reentrant function stack areas.

Alien Function (PL/M-51 Interface)

C51 lets you call routines written in PL/M-51 from your C programs. You can access PL/M-51 routines from C by declaring them external along with the **alien** function type specifier. For example:

```
extern alien char plm_func (int, char);

char c_func (void)  {
  int i;
  char c;

  for (i = 0; i < 100; i++) {
    c = plm_func (i, c);          /* call PL/M func */
  }
  return (c);
}
```


You may also create functions in C that can be invoked by PL/M-51 routines. To do this, use the **alien** function type specifier in the C function declaration. For example:

```
alien char c_func (char a,  int b)  {
  return (a * b);
}
```

Parameters and return values of PL/M-51 functions may be any of the following types: **bit**, **char**, **unsigned char**, **int**, and **unsigned int**. Other types, including **long**, **float**, and all types of pointers, can be declared in C functions with the **alien** type specifier. However, use these types with care because PL/M-51 does not directly support 32-bit binary integers or floating-point numbers.

Public variables declared in the PL/M-51 module are available to your C programs by declaring them external like you would for any C variable.

Real-time Function Tasks

The C51 compiler provides support for the **RTX51 Full** and **RTX51 Tiny** real-time multitasking operating systems through use of the **_task_** and **_priority_** keywords. The **_task_** keyword lets you define a function as a real-time task. The **_priority_** keyword lets you specify the priority for the task.

For example:

```
void func (void) _task_ num _priority_ pri
```

where:

num is a task ID number from 0 to 255 for **RTX51 Full** or 0 to 15 for **RTX51 Tiny**.

pri is the priority for the task. Refer to the *RTX51 User's Guide* or the *RTX51 Tiny User's Guide* for more information.

Task functions must be declared with a void return type and a void argument list.

Chapter 4. Preprocessor

The preprocessor built into the C51 compiler handles directives found in the source file. C51 supports all of the ANSI Standard C directives. This chapter gives a brief overview of the directives and elements provided by the preprocessor.

Directives

Preprocessor directives must be the first non-whitespace text specified on a line. All directives are prefixed with the pound or number-sign character (**'#'**). For example:

```
#pragma
#include <stdio.h>
#define DEBUG 1
```

The following table lists the preprocessor directives and gives a brief description of each.

Directive	Description
define	Defines a preprocessor macro or constant.
elif	Initiates an alternative branch of the if condition, when the previous if, ifdef, ifndef, or elif branch was not taken.
else	Initiates an alternative branch when the previous if, ifdef, or ifndef branch was not taken.
endif	Ends an **if**, **ifdef**, **ifndef**, **elif**, or **else** block.
error	Outputs an error message defined by the user. This directive instructs the compiler to emit the specified error message.
ifdef	Evaluates an expression for conditional compilation. The argument to be evaluated is the name of a definition.
ifndef	Same as ifdef but the evaluation succeeds if the definition is not defined.
if	Evaluates an expression for conditional compilation.
include	Reads source text from an external file. The notation sequence determines the search sequence of the included files. C51 searches for include files specified with less-than/greater-than symbols ('<' '>') in the include file directory. C51 searches for include files specified with double-quotes (" ") in the current directory.
line	Specifies a line number together with an optional filename. These specifications are used in error messages to identify the error position.
pragma	Allows you to specify control directives that may be included on the C51 command line. Pragmas may contain the same control directives that are specified on the command line.
undef	Deletes a preprocessor macro or constant definition.

Stringize Operator

The stringize or number-sign operator (**'#'**), when used within a macro definition, converts a macro parameter into a string constant. This operator may be used only in a macro that has a specified argument or parameter list.

When the stringize operator immediately precedes the name of one of the macro parameters, the parameter passed to the macro is enclosed within quotation marks and is treated as a string literal. For example:

```
#define stringer(x)  printf (#x "\n")

stringer (text)
```

results in the following actual output from the preprocessor.

```
printf ("text\n")
```


The expansion shows that the parameter is converted literally as if it were a string. When the preprocessor stringizes the **x** parameter, the resulting line is:

```
printf ("text" "\n")
```

Because strings separated by whitespace are concatenated at compile time, these two strings are combined into **"text\n".**

If the string passed as a parameter contains characters that should normally be literalized or escaped (for example, " and \), the required \ character is automatically added.

Token-pasting Operator

The token-pasting operator (##) within a macro definition combines two arguments. It permits two separate tokens in the macro definition to be joined into a single token.

If the name of a macro parameter used in the macro definition is immediately preceded or followed by the token-pasting operator, the macro parameter and the token-pasting operator are replaced by the value of the passed parameter. Text that is adjacent to the token-pasting operator that is not the name of a macro parameter is not affected. For example:

```
#define paster(n) printf ("token" #n " = %d", token##n)

paster (9);
```

results in the following actual output from the preprocessor.

```
printf ("token9 = %d", token9);
```

This example shows the concatenation of **token##n** into **token9**. Both the stringize and the token-pasting operators are used in this example.

Predefined Macro Constants

C51 provides you with predefined constants to use in preprocessor directives and C code for more portable programs. The following table lists and describes each one.

Constant	Description
_ _**C51**_ _	Version number of the C51 compiler (for example, 300 for version 3.00).
_ _**DATE**_ _	Date when the compilation was started.
_ _**FILE**_ _	Name of the file being compiled.
_ _**LINE**_ _	Current line number in the file being compiled.
_ _**MODEL**_ _	Memory model selected (0 for small, 1 for compact, 2 for large).
_ _**TIME**_ _	Time when the compilation was started.
__**STDC**_ _	Defined to 1 to indicate full conformance with the ANSI C Standard.

Chapter 5. 8051 Derivatives

A number of 8051 derivatives are available that provide enhanced performance while remaining compatible with the 8051 core. These derivatives provide additional data pointers, very fast math operations, and reduced instruction sets.

The C51 compiler directly supports the enhanced features of the following 8051-based microcontrollers:

- AMD 80C321, 80C521, and 80C541 (2 data pointers).
- Dallas 80C320, 80C520, and 80C530 (2 data pointers).
- Phillips/Signetics 8xC750, 8xC751, and 8xC752 (maximum code space of 2 KBytes, no **LCALL** or **LJMP** instructions, 64 bytes internal, no external data memory).
- Siemens 80C517 and 80C537 (high-speed 32-bit and 16-bit binary arithmetic operations, 8 data pointers).

The C51 compiler provides you with support for these CPUs through the use of special libraries, library routines, and the command-line directives **MODDP2** and **MOD517**. These directives enable C51 to generate object code that takes advantage of the enhancements mentioned above. Refer to "Chapter 3. Language Extensions" on page 57 for more information about these directives.

AMD 80C321, 80C521, and 80C541

The AMD 80C321, 80C521, and 80C541 provide 2 data pointers which can be used for memory access. Using multiple data pointers can improve the speed of library functions like **memcpy**, **memmove**, **memcmp**, **strcpy**, and **strcmp**.

The **MODDP2** control directive instructs the C51 compiler to generate code that uses both data pointers in your program.

The C51 compiler uses at least one data pointer in an interrupt function. If an interrupt function is compiled using the **MODDP2** directive, both data pointers are saved on the stack. This happens even if the interrupt function uses only one data pointer.

To conserve stack space, you may compile interrupt functions with the **NOMODDP2** directive. The C51 compiler does not use the second data pointer when this directive is used.

Dallas 80C320, 80C520, and 80C530

The Dallas Semiconductor 80C320, 80C520, and 80C530 provide 2 data pointers which can be used for memory access. Using multiple data pointers can improve the speed of library functions like **memcpy**, **memmove**, **memcmp**, **strcpy**, and **strcmp**.

The **MODDP2** control directive instructs the C51 compiler to generate code that uses both data pointers in your program.

The C51 compiler uses at least one data pointer in an interrupt function. If an interrupt function is compiled using the **MODDP2** directive, both data pointers are saved on the stack. This happens even if the interrupt function uses only one data pointer.

To conserve stack space, you may compile interrupt functions with the **NOMODDP2** directive. The C51 compiler does not use the second data pointer when this directive is used.

Siemens 80C517 and 80C537

The Siemens 80C517 and 80C537 provide high-speed 32-bit and 16-bit arithmetic operations as well as 8 data pointers which can be used for memory access. Using the high-speed arithmetic unit improves the performance of many **int**, **long**, and **float** operations.

The **MOD517** control directive instructs the C51 compiler to generate code that utilizes the advanced features of these CPUs.

Data Pointers

The Siemens 80C517 and 80C537 provide 8 data pointers which can be used to improve memory accesses. Using multiple data pointers can improve the execution of library functions such as: **memcpy**, **memmove**, **memcmp**, **strcpy**, and **strcmp**. The 8 data pointers of the 80C517 and 80C537 can also reduce the stack load of interrupt functions.

C51 uses only 2 of the 8 data pointers of the 80C517 at a time. In order to keep the stack load in the interrupt routines low, C51 switches to 2 unused data pointers when switching the register bank. In this case, the contents of the register **DPSEL** are saved on the stack, and a new pair of data pointers is selected. Saving the data pointers on the stack is no longer required.

If an interrupt routine does not switch to another register bank (for example, the function is declared without the **using** attribute), the data pointers must be saved on the stack (using 4 bytes of stack space). To keep the size of the stack as small as possible, use the **MOD517(NODP8)** directive to compile the interrupt routine and the functions called from within the interrupt. This generates code for the interrupt that uses only one data pointer and, therefore, only 2 bytes of stack space.

High-speed Arithmetic

C51 uses the 32-bit and 16-bit arithmetic operations of the 80C517 to improve performance of a number of math-intensive operations. C language programs execute considerably faster when using either of these CPUs.

The following tables show execution times for various arithmetic operations and compare the performance of the standard 8051 to that of the 80C517 CPU.

16-bit Binary Integer Operations

Operation	CPU	Routine	Min.	Avg.	Max.
Signed/unsigned multiplication	8051	IMUL	29	29	29
	80517	intrinsic	17	17	17
Unsigned division	8051	UIDIV	16	128	153
	80517	UIDIV517	22	22	22
Signed division	8051	SIDIV	53	141	181
	80517	SIDIV517	35	52	60

Times are shown in CPU cycles.

32-bit Binary Integer Operations

Operation	CPU	Routine	Min.	Avg.	Max.
Signed/unsigned multiplication	8051	LMUL	106	106	106
	80517	LMUL517	62	62	62
Unsigned division	8051	ULDIV	227	497	650
	80517	ULDIV517	36	52	101
Signed division	8051	SLDIV	267	564	709
	80517	SLDIV517	49	75	141
Left shift	8051	LSHL	5	237	470
	80517	LSHL517	5	28	29
Unsigned right shift	8051	ULSHR	5	237	470
	80517	ULSHR517	5	29	30
Signed right shift	8051	SLSHR	5	237	470
	80517	—	—	—	—

Times are shown in CPU cycles.

Floating-point Operations

Operation	CPU	Routine	Min.	Avg.	Max.
Addition	8051	FPADD	8	107	202
	80517	FPADD517	8	107	202
Subtraction	8051	FPSUB	11	113	214
	80517	FPSUB517	11	113	214
Multiplication	8051	FPMUL	13	114	198
	80517	FPMUL517	13	86	141
Division	8051	FPDIV	48	687	999
	80517	FPDIV517	48	165	209
Comparison	8051	FPCMP	42	54	59
	80517	FPCMP517	42	54	59
Square root	8051	SQRT	12	1936	2360
	80517	SQRT517	12	755	882
Sine	8051	SIN	1565	2928	3476
	80517	SIN517	1422	2519	3048
Cosine	8051	COS	1601	2921	3665
	80517	COS517	1458	2514	3180
Tangent	8051	TAN	1982	4966	5699
	80517	TAN517	1839	3753	4329
Arcsine	8051	ASIN	912	6991	8554
	80517	ASIN517	912	3984	4717
Arccosine	8051	ACOS	796	7578	8579
	80517	ACOS517	796	4255	4871
Arctangent	8051	ATAN	1069	3320	3712
	80517	ATAN517	1037	2444	2737
Exponential	8051	EXP	233	3314	5308
	80517	EXP517	176	2879	4724
Natural Logarithm	8051	LOG	32	3432	4128
	80517	LOG517	32	2405	2926
Common Logarithm	8051	LOG10	34	3607	4328
	80517	LOG10517	34	2530	3069
ASCII to float conversion	8051	FPATOF	960	3006	5611
	80517	FPATOF517	722	2202	4144

Times are shown in CPU cycles.

NOTES
The execution times specified in the preceding tables do not take access times for variables or stack operations into consideration. Actual processing times may consume up to 100 additional cycles depending on the stack load and address space used.

When using the arithmetic features of the 80C517 and 80C537, note that operations involving the arithmetic processor are exclusive and may not be interrupted. Do not use the arithmetic extensions in both the main program and an interrupt service routine.

Use the following suggestions to help guarantee that only one thread of execution uses the arithmetic processor:

- Use the **MOD517** directive to compile functions which are guaranteed to execute only in the main program or functions used by one interrupt service routine, but not both.
- Compile all remaining functions with the **MOD517(NOAU)** directive.

Library Routines

The extra features of the 80C517 and 80C537 are used in several library routines to enhance performance. These routines are listed below and are described in detail in “Chapter 8. Library Reference” on page 175.

acos517
asin517
atan517
atof517
cos517
exp517
log10517
log517
printf517
scanf517
sin517
sprintf517
sqrt517
sscanf517
tan517

Philips/Signetics 8xC750, 8xC751, and 8xC752

The Philips/Signetics 8xC750, 8xC751, and 8xC752 derivatives support a maximum of 2 KBytes of internal program memory. The CPU cannot execute **LCALL** and **LJMP** instructions. The following must be considered when using these devices:

- A special library, **80C751.LIB**, which does not use these instructions is necessary for these devices.
- The C51 compiler must be set to avoid using **LJMP** and **LCALL** instructions. This is accomplished using the **ROM(SMALL)** directive.

Note that the following restrictions apply when creating programs for the 8xC750, 8xC751, and 8xC752:

- Stream functions such as **printf** and **putchar** may not be used. These functions are usually not necessary for this chip because it is only equipped with a maximum of 2 KBytes and has no serial interface.
- Floating-point operations may not be used. Only operations using **char**, **unsigned char**, **int**, **unsigned int**, **long**, **unsigned long**, and **bit** data types are allowed.
- The C51 compiler must be invoked with the **ROM(SMALL)** control directive. This control statement instructs the C51 compiler to use only **AJMP** and **ACALL** instructions.
- The library file **80C751.LIB** must be included in the input module list of the linker. For example:

```
BL51 myprog.obj, startup751.obj, 80C751.LIB
```

- A special startup module, **START751.A51**, is required. This file contains startup code that is comparable to that found in **STARTUP.A51**, but contains no **LJMP** or **LCALL** instructions. Refer to "Customization Files" on page 113 for more information.

Chapter 6. Advanced Programming Techniques

This chapter describes advanced programming information that the experienced software engineer will find invaluable. Knowledge of most of these topics is not necessary to successfully create an embedded 8051 target program using the C51 compiler. However, the following sections provide insight into how many non-standard procedures can be accomplished (for example, interfacing to PL/M-51). This chapter discusses the following topics:

- Files you can alter to customize the startup procedures or run-time execution of several library routines in your target program
- The conventions C51 uses to name code and data segments
- How to interface C51 functions to assembly and PL/M-51 routines
- Data storage formats for the different C51 data types
- Different optimizing features of the C51 optimizing compiler

Customization Files

The C51 compiler includes a number of source files you can modify to adapt your target program to a specific hardware platform. These files contain: code that is executed upon startup (**STARTUP.A51**), code that is used to initialize static variables (**INIT.A51**), and code that is used to perform low-level stream I/O (**GETKEY.C** and **PUTCHAR.C**). Source code for the memory allocation routines is also included in the files **CALLOC.C**, **FREE.C**, **INIT_MEM.C**, **MALLOC.C**, and **REALLOC.C**. All of these source files are described in detail in the sections that follow.

The code contained in these files is already compiled or assembled and included in the C library. When you link, the code from the library is automatically included.

To include custom startup or initialization routines, you must include them in the linker command line. The following example shows you how to include custom replacement files for **STARTUP.A51** and **PUTCHAR.C.**

```
BL51 MYMODUL1.OBJ, MYMODUL2.OBJ, STARTUP.OBJ, PUTCHAR.OBJ
```

STARTUP.A51

The **STARTUP.A51** file contains the startup code for a C51 target program. This source file is located in the **LIB** directory. Include a copy of this file in each 8051 project that needs custom startup code.

This code is executed immediately upon reset of the target system and optionally performs the following operations, in order:

- Clears internal data memory
- Clears external data memory
- Clears paged external data memory
- Initializes the small model reentrant stack and pointer
- Initializes the large model reentrant stack and pointer
- Initializes the compact model reentrant stack and pointer
- Initializes the 8051 hardware stack pointer
- Transfers control to the main C function

The **STARTUP.A51** file provides you with assembly constants that you may change to control the actions taken at startup. These are defined in the following table.

Constant Name	Description
IDATALEN	Indicates the number of bytes of idata that are to be initialized to 0. The default is 80h because most 8051 derivatives contain at least 128 bytes of internal data memory. Use a value of 100h for the 8052 and other derivatives that have 256 bytes of internal data memory.
XDATASTART	Specifies the xdata address to start initializing to 0.
XDATALEN	Indicates the number of bytes of xdata to be initialized to 0. The default is 0.
PDATASTART	Specifies the pdata address to start initializing to 0.
PDATALEN	Indicates the number of bytes of pdata to be initialized to 0. The default is 0.
IBPSTACK	Indicates whether or not the small model reentrant stack pointer (**?C_IBP**) should be initialized. A value of 1 causes this pointer to be initialized. A value of 0 prevents initialization of this pointer. The default is 0.

Constant Name	Description
IBPSTACKTOP	Specifies the top start address of the small model reentrant stack area. The default is 0xFF in idata memory. C51 does not check to see if the stack area available satisfies the requirements of the applications. It is your responsibility to perform such a test.
XBPSTACK	Indicates whether or not the large model reentrant stack pointer (**?C_XBP**) should be initialized. A value of 1 causes this pointer to be initialized. A value of 0 prevents initialization of this pointer. The default is 0.
XBPSTACKTOP	Specifies the top start address of the large model reentrant stack area. The default is 0xFFFF in xdata memory. C51 does not check to see if the available stack area satisfies the requirements of the applications. It is your responsibility to perform such a test.
PBPSTACK	Indicates whether the compact model reentrant stack pointer (**?C_PBP**) should be initialized. A value of 1 causes this pointer to be initialized. A value of 0 prevents initialization of this pointer. The default is 0.
PBPSTACKTOP	Specifies the top start address of the compact model reentrant stack area. The default is 0xFF in pdata memory. C51 does not check to see if the available stack area satisfies the requirements of the applications. It is your responsibility to perform such a test.
PPAGEENABLE	Enables (a value of 1) or disables (a value of 0) the initialization of port 2 of the 8051 device. The default is 0. The addressing of port 2 allows the mapping of 256 byte variable memory in any arbitrary xdata page.
PPAGE	Specifies the value to write to Port 2 of the 8051 for pdata memory access. This value represents the xdata memory page to use for pdata. This is the upper 8 bits of the absolute address range to use for pdata. For example, if the pdata area begins at address 1000h (page 10h) in the xdata memory, **PPAGEENABLE** should be set to 1, and **PPAGE** should be set to 10h. The BL51 Linker/Locator must contain a value between 1000h and 10FFh in the PDATA control directive. For example: **BL51 <input modules> PDATA (1050H)** Neither BL51 nor C51 checks to see if the **PDATA** control directive and the **PPAGE** assembler constant are correctly specified. You must ensure that these parameters contain suitable values.

The following is a listing of **STARTUP.A51.**

```
;------------------------------------------------------------------
;  This file is part of the C51 Compiler package
;------------------------------------------------------------------
;  STARTUP.A51:  This code is executed after processor reset.
;
;  To translate this file use A51 with the following invocation:
;
;     A51 STARTUP.A51
```

```
;
;  To link the modified STARTUP.OBJ file to your application use
;  the following BL51 invocation:
;
;     BL51 <your object file list>, STARTUP.OBJ <controls>
;
;------------------------------------------------------------------
;  User-defined Power-On Initialization of Memory
;
;  With the following EQU statements the initialization of memory
;  at processor reset can be defined:
;
; the absolute start-address of IDATA memory is always 0
IDATALEN      EQU 80H  ; the length of IDATA memory in bytes.
;
XDATASTART    EQU 0H   ; the absolute start-address of XDATA memory
XDATALEN      EQU 0H   ; the length of XDATA memory in bytes.
;
PDATASTART    EQU 0H   ; the absolute start-address of PDATA memory
PDATALEN      EQU 0H   ; the length of PDATA memory in bytes.
;
;  Notes:  The IDATA space overlaps physically the DATA and BIT
;  areas of the 8051 CPU. At minimum the memory space occupied from
;  the C-51 run-time routines must be set to zero.
;------------------------------------------------------------------
;  Reentrant Stack Initialization
;
;  The following EQU statements define the stack pointer for
;  reentrant functions and initialized it:
;
;  Stack Space for reentrant functions in the SMALL model.
IBPSTACK      EQU 0        ; set to 1 if small reentrant is used.
IBPSTACKTOP   EQU 0FFH+1   ; set top of stack to highest location+1.
;
;  Stack Space for reentrant functions in the LARGE model.
XBPSTACK      EQU 0        ; set to 1 if large reentrant is used.
XBPSTACKTOP   EQU 0FFFFH+1 ; set top of stack to highest location+1.
;
;  Stack Space for reentrant functions in the COMPACT model.
PBPSTACK      EQU 0        ; set to 1 if compact reentrant is used.
PBPSTACKTOP   EQU 0FFFFH+1 ; set top of stack to highest location+1.
;------------------------------------------------------------------
;  Page Definition for Using the Compact Model with 64 KByte xdata
;  RAM
;
;  The following EQU statements define the xdata page used for pdata
;  variables. The EQU PPAGE must conform with the PPAGE control used
;  in the linker invocation.
;
PPAGEENABLE   EQU 0    ; set to 1 if pdata object are used.
PPAGE         EQU 0    ; define PPAGE number.
;------------------------------------------------------------------
               NAME    ?C_STARTUP

?C_C51STARTUP  SEGMENT   CODE
?STACK         SEGMENT   IDATA

               RSEG  ?STACK
               DS    1

               EXTRN CODE (?C_START)
```

```
                PUBLIC  ?C_STARTUP

                CSEG  AT  0
?C_STARTUP:     LJMP  STARTUP1

                RSEG  ?C_C51STARTUP
STARTUP1:

IF IDATALEN <> 0
                MOV   R0,#IDATALEN - 1
                CLR   A
IDATALOOP:      MOV   @R0,A
                DJNZ  R0,IDATALOOP
ENDIF

IF XDATALEN <> 0
                MOV   DPTR,#XDATASTART
                MOV   R7,#LOW (XDATALEN)
  IF (LOW (XDATALEN)) <> 0
                MOV   R6,#(HIGH XDATALEN) +1
  ELSE
                MOV   R6,#HIGH (XDATALEN)
  ENDIF
                CLR   A
XDATALOOP:      MOVX  @DPTR,A
                INC   DPTR
                DJNZ  R7,XDATALOOP
                DJNZ  R6,XDATALOOP
ENDIF

IF PPAGEENABLE <> 0
                MOV   P2,#PPAGE
ENDIF

IF PDATALEN <> 0
                MOV   R0,#PDATASTART
                MOV   R7,LOW (PDATALEN)
                CLR   A
PDATALOOP:      MOVX  @R0,A
                INC   R0
                DJNZ   R7,PDATALOOP
ENDIF

IF IBPSTACK <> 0
EXTRN DATA (?C_IBP)

                MOV   ?C_IBP,#LOW IBPSTACKTOP
ENDIF

IF XBPSTACK <> 0
EXTRN DATA (?C_XBP)

                MOV   ?C_XBP,#HIGH XBPSTACKTOP
                MOV   ?C_XBP+1,#LOW XBPSTACKTOP
ENDIF

IF PBPSTACK <> 0
EXTRN DATA (?C_PBP)
                MOV   ?C_PBP,#LOW PBPSTACKTOP
ENDIF
```

```
                MOV   SP,#?STACK-1
                LJMP  ?C_START

                END
```

START751.A51

The **START751.A51** file contains the startup code for a C51 target program that is to run on the Signetics 8xC751 CPU. This source file is located in the **LIB** directory. To use this file, follow the instructions on how to use **STARTUP.A51** in the previous section. The only difference between the two files is that **START751.A51** is specifically used for the 8xC751 which cannot access more than 2 KBytes of code space and can access no external data memory. For these reasons, there are no assembler constants that can affect **xdata** and **pdata** memory.

The following is a listing of **START751.A51**.

```
;------------------------------------------------------------------------
;  This file is part of the C51 Compiler package
;
;------------------------------------------------------------------------
;  START751.A51:  This code is executed after processor reset.
;
;  To translate this file use A51 with the following invocation:
;
;     A51 START751.A51
;
;  To link the modified START751.OBJ file to your application use the
;  following BL51 invocation:
;
;     BL51 <your object file list>, START751.OBJ <controls>
;
;------------------------------------------------------------------------
;
;  User-defined Power-On Initialization of Memory
;
;  With the following EQU statements the initialization of memory
;  at processor reset can be defined:
;
;           the absolute start-address of IDATA memory is always 0
IDATALEN    EQU     40H  ; the length of IDATA memory in bytes.
;
;  Notes:  The IDATA space physically overlaps the DATA and BIT areas of
;          the 80751 CPU.  At minimum the memory space occupied by C51
;          run-time routines must be set to zero.
;------------------------------------------------------------------------
;
;  Reentrant Stack Initialization
;
;  The following EQU statements define the stack pointer for reentrant
;  functions and initialized it:
```

```
;
;  Stack Space for reentrant functions in the SMALL model.
IBPSTACK    EQU     0        ; set to 1 if small reentrant is used.
IBPSTACKTOP EQU     0FFH+1   ; set top of stack to highest location+1.
;
;------------------------------------------------------------------------

                NAME     ?C_STARTUP


?C_C51STARTUP   SEGMENT   CODE
?STACK          SEGMENT   IDATA

                RSEG      ?STACK
                DS        1

                EXTRN CODE (?C_START)
                PUBLIC    ?C_STARTUP

                CSEG      AT   0
?C_STARTUP:     AJMP  STARTUP1

                RSEG      ?C_C51STARTUP

STARTUP1:

IF IDATALEN <> 0
                MOV   R0,#IDATALEN - 1
                CLR   A
IDATALOOP:      MOV   @R0,A
                DJNZ  R0,IDATALOOP
ENDIF


IF IBPSTACK <> 0
EXTRN DATA (?C_IBP)

                MOV   ?C_IBP,#LOW IBPSTACKTOP
ENDIF

                MOV   SP,#?STACK-1
                AJMP  ?C_START

                END
```

INIT.A51

The **INIT.A51** file contains the initialization routine for variables that were explicitly initialized. If your system is equipped with a watchdog timer, you can integrate a watchdog refresh into the initialization code using the **watchdog** macro. This macro need be defined only if the initialization takes longer than the watchdog cycle time. If you are using an 80515, the macro could be defined as follows:

```
WATCHDOG    MACRO
            SETB   WDT
            SETB   SWDT
            ENDM
```

The following is a partial listing of **INIT.A51**.

```
;------------------------------------------------------------------------
; This file is part of the C51 Compiler package
;------------------------------------------------------------------------
; INIT.A51:  This code is executed, if the application program contains
;            initialized variables at file level.
;
; To translate this file use A51 with the following invocation:
;
;    A51 INIT.A51
;
; To link the modified INIT.OBJ file to your application use the following
; BL51 invocation:
;
;    BL51 <your object file list>, INIT.OBJ <controls>
;
;------------------------------------------------------------------------
; User-defined Watch-Dog Refresh.
;
; If the C application contains many initialized variables & uses a
; watchdog it might be possible that the user has to include a watchdog
; refresh into the initialization process. The watchdog refresh routine
; can be defined in the following MACRO and can alter all CPU registers
; except DPTR.

WATCHDOG        MACRO
      ; Include any Watchdog refresh code here
      ENDM
;------------------------------------------------------------------------
?C_START:
                MOV   DPTR,#?C_INITSEG
LOOP:
                WATCHDOG
                CLR   A
                MOV   R6,#1
                MOVC  A,@A+DPTR
                JZ    INITEND
.
.
.
```

INIT751.A51

The **INIT751.A51** file contains the initialization routine for variables that were explicitly initialized. Use this initialization routine for the Signetics 8xC751. The following is a listing of the **INIT751.A51** file.

```
;------------------------------------------------------------------------
;  This file is part of the C51 Compiler package
;
;------------------------------------------------------------------------
;  INIT751.A51:  This code is executed, if the application program
;                contains initialized variables at file level.
;
;  To translate this file use A51 with the following invocation:
;
;     A51 INIT751.A51
;
;  To link the modified INIT.OBJ file to your application use the
;  following BL51 invocation:
;
;     BL51 <your object file list>, INIT751.OBJ <controls>
;
;------------------------------------------------------------------------

                NAME      ?C_INIT


?C_C51STARTUP   SEGMENT   CODE
?C_INITSEG      SEGMENT   CODE          ; Segment with Initializing Data


EXTRN CODE (?C_INITSEGSTART)

                EXTRN CODE (MAIN)
                PUBLIC    ?C_START

                RSEG      ?C_C51STARTUP
INITEND:        AJMP  MAIN

IorPData:                         ; If CY=1 PData Values
                CLR   A
                MOVC  A,@A+DPTR
                INC   DPTR
                MOV   R0,A        ; Start Address
IorPLoop:       CLR   A
                MOVC  A,@A+DPTR
                INC   DPTR
                MOV   @R0,A
Common:         INC   R0
                DJNZ  R7,IorPLoop
                SJMP  Loop

Bits:           CLR   A
                MOVC  A,@A+DPTR
                INC   DPTR
                MOV   R0,A
                ANL   A,#007H
                ADD   A,#Table-LoadTab
```

```
                XCH   A,R0
                CLR   C
                RLC   A          ; Bit Condition to Carry
                SWAP  A
                ANL   A,#00FH
                ORL   A,#20H     ; Bit Address
                XCH   A,R0       ; convert to Byte Address
                MOVC  A,@A+PC
LoadTab:        JC    SetIt
                CPL   A
                ANL   A,@R0
                SJMP  BitReady
SetIt:          ORL   A,@R0
BitReady:       MOV   @R0,A
                DJNZ  R7,Bits
                SJMP  Loop

Table:          DB   00000001B
                DB   00000010B
                DB   00000100B
                DB   00001000B
                DB   00010000B
                DB   00100000B
                DB   01000000B
                DB   10000000B


?C_START:       MOV   DPTR,#?C_INITSEGSTART
LOOP:           CLR   A
                MOV   R6,#1
                MOVC  A,@A+DPTR
                JZ    INITEND
                INC   DPTR
                MOV   R7,A
                ANL   A,#3FH
                JNB   ACC.5,NOBIG
                ANL   A,#01FH
                MOV   R6,A
                CLR   A
                MOVC  A,@A+DPTR
                INC   DPTR
                JZ    NOBIG
                INC   R6
NOBIG:          XCH   A,R7
                ANL   A,#0C0H    ; Typ is in Bit 6 and Bit 7
                ADD   A,ACC
                JZ    IorPDATA
                JC    Bits
                SJMP  $

                RSEG  ?C_INITSEG
                DB    0

                END
```

PUTCHAR.C

This file contains the **putchar** function which is the low-level character output routine for the stream I/O routines. All stream routines that output character data do so through this routine. You may adapt this routine to your individual hardware (for example, LCD or LED displays).

The default **PUTCHAR.C** file delivered with the C51 compiler outputs characters via the serial interface. An **XON**/**XOFF** protocol is used for flow control. Linefeed characters (**'\n'**) are automatically converted into carriage return/linefeed sequences (**'\r\n'**).

GETKEY.C

This file contains the **_getkey** function which is the low-level character input routine for the stream I/O routines. All stream routines that input character data do so through this routine. You may adapt this routine to your individual hardware (for example, for matrix keyboards). The default **GETKEY.C** file delivered with the C51 compiler reads a character via the serial interface. No data conversions are performed.

CALLOC.C

This file contains the source code for the **calloc** function. This routine allocates memory for an array from the memory pool.

FREE.C

This file contains the source code for the **free** function. This routine returns a previously allocated memory block to the memory pool.

INIT_MEM.C

This file contains the source code for the **init_mempool** function. This routine allows you to specify the location and size of a memory pool from which memory may be allocated using the **malloc**, **calloc**, and **realloc** functions.

MALLOC.C

This file contains the source code for the **malloc** function. This routine allocates memory from the memory pool.

REALLOC.C

This file contains the source code for the **realloc** function. This routine resizes a previously allocated memory block.

Optimizer

The C51 compiler is an optimizing compiler. This means that the compiler takes certain steps to ensure that the code that is generated and output to the object file is the most efficient (smaller and/or faster) code possible. The compiler analyzes the generated code to produce more efficient instruction sequences. This ensures that your C51 program runs as quickly as possible.

The C51 compiler provides six different levels of optimizing. Each increasing level includes the optimizations of the levels below it.

Level	Description
0	**Constant Folding:** The compiler performs calculations that reduce expressions to numeric constants, where possible. This includes calculations of run-time addresses. **Simple Access Optimizing:** The compiler optimizes access of internal data and bit addresses in the 8051 system. **Jump Optimizing:** The compiler always extends jumps to the final target. Jumps to jumps are deleted.
1	**Dead Code Elimination:** Unused code fragments and artifacts are eliminated. **Jump Negation:** Conditional jumps are closely examined to see if they can be streamlined or eliminated by the inversion of the test logic.
2	**Data Overlaying:** Data and bit segments suitable for static overlay are identified and internally marked. The BL51 Linker/Locator has the capability, through global data flow analysis, of selecting segments which can then be overlaid.
3	**Peephole Optimizing:** Redundant MOV instructions are removed. This includes unnecessary loading of objects from the memory as well as load operations with constants. Complex operations are replaced by simple operations when memory space or execution time can be saved.
4	**Register Variables:** Automatic variables and function arguments are located in registers when possible. Reservation of data memory for these variables is omitted. **Extended Access Optimizing:** Variables from the IDATA, XDATA, PDATA and CODE areas are directly included in operations. The use of intermediate registers is not necessary most of the time. **Local Common Subexpression Elimination:** If the same calculations are performed repetitively in an expression, the result of the first calculation is saved and used further whenever possible. Superfluous calculations are eliminated from the code. **Case/Switch Optimizing:** Code involving switch and case statements is optimized as jump tables or jump strings.
5	**Global Common Subexpression Elimination:** Identical sub expressions within a function are calculated only once when possible. The intermediate result is stored in a register and used instead of a new calculation. **Simple Loop Optimizing:** Program loops that fill a memory range with a constant are converted and optimized.

Level	Description
6	**Loop Rotation:** Program loops are rotated if the resulting program code is faster and more efficient.

General Optimizations

Optimization	Description
Constant Folding	Several constant values occurring in an expression or address calculation are combined as a constant.
Jump Optimizing	Jumps are inverted or extended to the final target address when the program efficiency is thereby increased.
Dead Code Elimination	Code which cannot be reached (dead code) is removed from the program.
Register Variables	Automatic variables and function arguments are located in registers when possible. Reservation of data memory for these variables is omitted.
Parameter Passing Via Registers	A maximum of three function arguments can be passed in registers.
Global Common Subexpression Elimination	Identical subexpressions or address calculations that occur multiple times in a function are recognized and calculated only once when possible.

8051-Specific Optimizations

Optimization	Description
Peephole Optimization	Complex operations are replaced by simplified operations when memory space or execution time can be saved as a result.
Extended Access Optimizing	Constants and variables are included directly in operations.
Data Overlaying	Data and bit segments of functions are identified as OVERLAYABLE and are overlaid with other data and bit segments by the BL51 Linker/Locator.
Case/Switch Optimizing	Any switch and case statements are optimized by using a jump table or string of jumps.

Options for Code Generation

Optimization	Description
OPTIMIZE(SIZE)	Common C operations are replaced by subprograms. Program code is thereby reduced.
NOAREGS	C51 no longer uses absolute register access. Program code is independent of the register bank.

Optimization	Description
NOREGPARMS	Parameter passing is always performed in local data segments. The program code is compatible to earlier versions of C51.

Segment Naming Conventions

Objects generated by the C51 compiler (program code, program data, and constant data) are stored in segments which are units of code or data memory. A segment may be relocatable or may be absolute. Each relocatable segment has a type and a name. This section describes the conventions used by C51 for naming these segments.

Segment names include a *module_name*. The *module_name* is the name of the source file in which the object is declared and excludes the drive letter, path specification, and file extension. In order to accommodate a wide variety of existing software and hardware tools, all segment names are converted and stored in uppercase.

Each segment name has a prefix that corresponds to the memory type used for the segment. The prefix is enclosed in question marks (?). The following is a list of the standard segment name prefixes:

Segment Prefix	Data Type	Description
?PR?	**code**	Executable program code
?CO?	**code**	Constant data in program memory
?XD?	**xdata**	External data memory
?DT?	**data**	Internal data memory
?ID?	**idata**	Indirectly-addressable internal data memory
?BI?	**bit**	Bit data in internal data memory
?BA?	**bdata**	Bit-addressable data in internal data memory
?PD?	**pdata**	Paged data in external data memory

Data Objects

Data objects are the variables and constants you declare in your C programs. C51 generates a separate segment for each memory type for which a variable is declared. The following table lists the segment names generated for different variable data objects.

Segment Name	Description
?CO?*module_name*	Constants (strings and initialized variables)
?XD?*module_name*	Objects declared in xdata
?DT?*module_name*	Objects declared in data
?ID?*module_name*	Objects declared in idata

Segment Name	Description
?BI?*module_name*	bit objects
?BA?*module_name*	Bit-addressable data objects
?PD?*module_name*	Objects declared in pdata

Program Objects

Program objects include the code generated for C program functions by the C51 compiler. Each function in a source module is assigned a separate code segment using the ?PR?*function_name*?*module_name* naming convention. For example, the function **error_check** in the file **SAMPLE.C** would result in a segment name of ?PR?ERROR_CHECK?SAMPLE.

Segments are also created for local variables that are declared within the body of a function. These segment names follow the above conventions and have a different prefix depending upon the memory area in which the local variables are stored.

Function arguments were historically passed using fixed memory locations. This is still true for routines written in PL/M-51. However, C51 can pass up to 3 function arguments in registers. Other arguments are passed using the traditional fixed memory areas. Memory space is reserved for all function arguments regardless of whether or not some of these arguments may be passed in registers. The parameter areas must be publicly known to any calling module. So, they are publicly defined using the following segment names:

```
?function_name?BYTE
?function_name?BIT
```

For example, if **func1** is a function that accepts both **bit** arguments as well as arguments of other data types, the **bit** arguments are passed starting at ?FUNC1?BIT, and all other parameters are passed starting at ?FUNC1?BYTE. Refer to "Interfacing C Programs to Assembler" on page 131 for examples of the function argument segments.

Functions that have parameters, local variables, or **bit** variables contain all additional segments for these variables. These segments can be overlaid by the BL51 Linker/Locator.

They are created as follows based on the memory model used.

Small model segment naming conventions		
Information	**Segment Type**	**Segment Name**
Program code	**code**	?PR?*function_name*?*module_name*
Local variables	**data**	?DT?*function_name*?*module_name*
Local bit variables	**bit**	?BI?*function_name*?*module_name*

Compact model segment naming conventions		
Information	**Segment Type**	**Segment Name**
Program code	**code**	?PR?*function_name*?*module_name*
Local variables	**pdata**	?PD?*function_name*?*module_name*
Local bit variables	**bit**	?BI?*function_name*?*module_name*

Large model segment naming conventions		
Information	**Segment Type**	**Segment Name**
Program code	**code**	?PR?*function_name*?*module_name*
Local variables	**xdata**	?XD?*function_name*?*module_name*
Local bit variables	**bit**	?BI?*function_name*?*module_name*

The names for functions with register parameters and reentrant attributes are modified slightly to avoid run-time errors. The following table lists deviations from the standard segment names.

Declaration	Symbol	Description
void func (void) …	FUNC	Names of functions that have no arguments or whose arguments are not passed in registers are transferred to the object file without any changes. The function name is converted to uppercase.
void func1 (char) …	_FUNC1	For functions with arguments passed in registers, the underscore character ('_') is prefixed to the function name. This identifies those functions that transfer arguments in CPU registers.
void func2 (void) reentrant …	_?FUNC2	For functions that are reentrant, the string "_?" is prefixed to the function name. This is used to identify reentrant functions.

Interfacing C Programs to Assembler

You can easily interface C51 to routines written in 8051 Assembler. The A51 Assembler is an 8051 macro assembler that emits object modules in OMF-51 format. By observing a few programming rules, you can call assembly routines from C and vice versa. Public variables declared in the assembly module are available to your C programs.

There are several reasons why you might want to call an assembly routine from your C program. You may have assembly code already written that you wish to use, you may need to improve the speed of a particular function, or you may want to manipulate SFRs or memory-mapped I/O devices directly from assembly. This section describes how to write assembly routines that can be directly interfaced to C programs.

For an assembly routine to be called from C, it must be aware of the parameter passing and return value conventions used in C functions. For all practical purposes, it must appear to be a C function.

Function Parameters

By default, C functions pass up to three parameters in registers. The remaining parameters are passed in fixed memory locations. You may use the directive **NOREGPARMS** to disable parameter passing in registers. Parameters are passed in fixed memory locations if parameter passing in registers is disabled or if there are too many parameters to fit in registers. Functions that pass parameters in registers are flagged by C51 with an underscore character (**'_'**) prefixed to the function name at code generation time. Functions that pass parameters only in fixed memory locations are not prefixed with an underscore. Refer to "Using the SRC Directive" on page 134 for an example.

Parameter Passing in Registers

C functions may pass parameters in registers and fixed memory locations. A maximum of 3 parameters may be passed in registers. All other parameters are passed using fixed memory locations. The following tables define what registers are used for passing parameters.

Arg Number	char, 1-byte ptr	int, 2-byte ptr	long, float	generic ptr
1	R7	R6 & R7 (MSB in R6, LSB in R7)	R4—R7	R1—R3 (Mem type in R3, MSB in R2, LSB in R1)
2	R5	R4 & R5 (MSB in R4, LSB in R5)	R4—R7	R1—R3 (Mem type in R3, MSB in R2, LSB in R1)
3	R3	R2 & R3 (MSB in R2, LSB in R3)		R1—R3 (Mem type in R3, MSB in R2, LSB in R1)

The following examples clarify how registers are selected for parameter passing.

Declaration	Description
`func1 ( int a)`	The first and only argument, ***a***, is passed in registers R6 and R7.
`func2 ( int b, int c, int *d)`	The first argument, ***b***, is passed in registers R6 and R7. The second argument, ***c***, is passed in registers R4 and R5. The third argument, ***d***, is passed in registers R1, R2, and R3.
`func3 ( long e, long f)`	The first argument, ***e***, is passed in registers R4, R5, R6, and R7. The second argument, ***f***, cannot be located in registers since those available for a second parameter with a type of long are already used by the first argument. This parameter is passed using fixed memory locations.
`func4 ( float g, char h)`	The first argument, ***g***, passed in registers R4, R5, R6, and R7. The second parameter, ***h***, cannot be passed in registers and is passed in fixed memory locations.

Parameter Passing in Fixed Memory Locations

Parameters passed to assembly routines in fixed memory locations use segments named ?*function_name*?BYTE and ?*function_name*?BIT to hold the parameter values passed to the function *function_name*. Bit parameters are copied into the ?*function_name*?BIT segment prior to calling the function. All other parameters are copied into the ?*function_name*?BYTE segment. All parameters are assigned space in these segments even if they are passed using registers. Parameters are stored in the order in which they are declared in each respective segment.

The fixed memory locations used for parameter passing may be in internal data memory or external data memory depending upon the memory model used. The small memory model is the most efficient and uses internal data memory for parameter segments. The compact and large models use external data memory for the parameter passing segments.

Function Return Values

Function return values are always passed using CPU registers. The following table lists the possible return types and the registers used for each.

Return Type	Register	Description
bit	Carry Flag	Single bit returned in the carry flag
char / unsigned char, 1-byte pointer	R7	Single byte typed returned in R7
int / unsigned int, 2-byte ptr	R6 & R7	MSB in R6, LSB in R7
long / unsigned long	R4-R7	MSB in R4, LSB in R7
float	R4-R7	32-Bit IEEE format
generic pointer	R1-R3	Memory type in R3, MSB R2, LSB R1

Using the SRC Directive

You may use the C51 compiler to generate the shell for an assembly routine you want to write or to help determine the passing conventions your assembly routine should use. The **SRC** command-line directive specifies that C51 generate an assembly file instead of an object file. For example, the following C source file:

```
#pragma SRC
#pragma SMALL

unsigned int asmfunc1 (
  unsigned int arg)
{
return (1 + arg);
}
```

generates the following assembly output file when compiled using the **SRC** directive.

```
; ASM1.SRC generated from: ASM1.C

NAME    ASM1

?PR?_asmfunc1?ASM1   SEGMENT CODE
PUBLIC  _asmfunc1
; #pragma SRC
; #pragma SMALL
;
; unsigned int asmfunc1 (

                RSEG   ?PR?_asmfunc1?ASM1
                USING  0
_asmfunc1:
;---- Variable 'arg?00' assigned to Register 'R6/R7' ----
                      ; SOURCE LINE # 4
                      ; SOURCE LINE # 6
; return (1 + arg);
                      ; SOURCE LINE # 7
                MOV   A,R7
                ADD   A,#01H
                MOV   R7,A
                CLR   A
                ADDC  A,R6
                MOV   R6,A
; }
                      ; SOURCE LINE # 8
?C0001:
                RET
; END OF _asmfunc1

                END
```

In this example, note that the function name, **`asmfunc1`**, is prefixed with an underscore character signifying that arguments are passed in registers. The **`arg`** parameter is passed using R6 and R7.

The following example shows the assembly source generated for the same function; however, register parameter passing has been disabled using the **NOREGPARMS** directive.

```
; ASM2.SRC generated from: ASM2.C

NAME    ASM2

?PR?asmfunc1?ASM2    SEGMENT CODE
?DT?asmfunc1?ASM2    SEGMENT DATA
PUBLIC  ?asmfunc1?BYTE
PUBLIC  asmfunc1

                RSEG  ?DT?asmfunc1?ASM2
?asmfunc1?BYTE:
arg?00:          DS  2
; #pragma SRC
; #pragma SMALL
; #pragma NOREGPARMS
;
; unsigned int asmfunc1 (

                RSEG   ?PR?asmfunc1?ASM2
                USING  0
asmfunc1:
                       ; SOURCE LINE # 5
                       ; SOURCE LINE # 7
; return (1 + arg);
                       ; SOURCE LINE # 8
                MOV    A,arg?00+01H
                ADD    A,#01H
                MOV    R7,A
                CLR    A
                ADDC   A,arg?00
                MOV    R6,A
; }
                       ; SOURCE LINE # 9
?C0001:
                RET
; END OF asmfunc1

                END
```

Note in this example that the function name, **asmfunc1,** is not prefixed with an underscore character and that the **arg** parameter is passed in the **?asmfunc1?BYTE** segment.

Register Usage

Assembler functions can change all register contents in the current selected register bank as well as the contents of the registers **ACC**, **B**, **DPTR**, and **PSW**. When invoking a C function from assembly, assume that these registers may be destroyed by the C function that is called.

Overlaying Segments

If the overlay process is executed during program linking and locating, it is important that each assembler subroutine have a unique program segment. This is necessary so that during the overlay process, the references between the functions are calculated using the references of the individual segments. The data areas of the assembler subprograms may be included in the overlay analysis when the following points are observed:

- All segment names must be created using the C51 segment naming conventions.
- Each assembler function with local variables must be assigned its own data segment. This data segment may be accessed by other functions only for passing parameters. Parameters must be passed in order.

Example Routines

The following program examples show you how to pass parameters to and from assembly routines. The following C functions are used in all of these examples:

```
int function (
  int v_a,       /* passed in R6 & R7 */
  char v_b,      /* passed in R5 */
  bit v_c,       /* passed in fixed memory location */
  long v_d,      /* passed in fixed memory location */
  bit v_e);      /* passed in fixed memory location */
```

Small Model Example

In the small model, parameters passed in fixed memory locations are stored in internal data memory. The parameter passing segment for variables is located in the **data** area.

The following are two assembly code examples. The first shows how the example function is invoked from assembly. The second example displays the assembly code for the example function.

Function invocation from assembly.

```
.
.
.
EXTRN CODE    (_function)         ; Ext declarations for function names
EXTRN DATA    (?_function?BYTE)   ; Seg for local variables
EXTRN BIT     (?_function?BIT)    ; Seg for local bit variables
.
.
.
        MOV   R6,#HIGH intval               ; int  a
        MOV   R7,#LOW  intval               ; int  a
        MOV   R7,#charconst                 ; char b
        SETB  ?_function?BIT+0              ; bit  c
        MOV   ?_function?BYTE+3,longval+0   ; long d
        MOV   ?_function?BYTE+4,longval+1   ; long d
        MOV   ?_function?BYTE+5,longval+2   ; long d
        MOV   ?_function?BYTE+6,longval+3   ; long d
        MOV   C,bitvalue
        MOV   ?_function?BIT+1,C            ; bit  e
        LCALL _function
        MOV   intresult+0,R6                ; store int
        MOV   intresult+1,R7                ; retval
.
.
.
```

Function implementation in assembly.

```
NAME          MODULE                  ; Names of the program module
?PR?FUNCTION?MODULE  SEGMENT  CODE    ; Seg for prg code in 'function'
?DT?FUNCTION?MODULE  SEGMENT  DATA OVERLAYABLE
                                      ; Seg for local vars in 'function'
?BI?FUNCTION?MODULE  SEGMENT  BIT  OVERLAYABLE
                                      ; Seg for local bit vars in 'function'

PUBLIC        _function, ?_function?BYTE, ?_function?BIT
                                      ; Public symbols for 'C' function call

RSEG          ?PD?FUNCTION?MODULE     ; Segment for local variables
?_function?BYTE:                      ; Start of parameter passing segment
v_a:    DS   2                        ;   int variable: v_a
v_b:    DS   1                        ;   char variable: v_b
v_d:    DS   4                        ;   long variable: v_d
.
.                                     ; Additional local variables
.

RSEG          ?BI?FUNCTION?MODULE     ; Segment for local bit variables
?_function?BIT:                       ; Start of parameter passing segment
v_c:    DBIT  1                       ;   bit variable: v_c
v_e:    DBIT  1                       ;   bit variable: v_e
.
.                                     ; Additional local bit variables
.

RSEG          ?PR?FUNCTION?MODULE     ; Program segment
_function:    MOV  v_a,R6             ; A function prolog and epilog is
              MOV  v_a+1,R7           ; not necessary.  All variables can
              MOV  v_b,R5             ; immediately be accessed.
.
.
.
              MOV  R6,#HIGH retval    ; Return value
              MOV  R7,#LOW  retval    ; int constant
              RET                     ; Return
```

Compact Model Example

In the compact model, parameters passed in fixed memory locations are stored in external data memory. The parameter passing segment for variables is located in the **pdata** area.

The following are two assembly code examples. The first shows you how the example function is invoked from assembly. The second example displays the assembly code for the example function.

Function invocation from assembly.

```
EXTRN CODE    (_function)          ; Ext declarations for function names
EXTRN XDATA   (?_function?BYTE)    ; Seg for local variables
EXTRN BIT     (?_function?BIT)     ; Seg for local bit variables
.
.
.
        MOV   R6,#HIGH intval           ; int  a
        MOV   R7,#LOW  intval           ; int  a
        MOV   R5,#charconst             ; char b
        SETB  ?_function?BIT+0          ; bit  c
        MOV   R0,#?_function?BYTE+3     ; Addr of 'v_d' in the passing area
        MOV   A,longval+0               ; long d
        MOVX  @R0,A                     ; Store parameter byte
        INC   R0                        ; Inc parameter passing address
        MOV   A,longval+1               ; long d
        MOVX  @R0,A                     ; Store parameter byte
        INC   R0                        ; Inc parameter passing address
        MOV   A,longval+2               ; long d
        MOVX  @R0,A                     ; Store parameter byte
        INC   R0                        ; Inc parameter passing address
        MOV   A,longval+3               ; long d
        MOVX  @R0,A                     ; Store parameter byte
        MOV   C,bitvalue
        MOV   ?_function?BIT+1,C        ; bit  e
        LCALL _function
        MOV   intresult+0,R6            ; Store int
        MOV   intresult+1,R7            ; Retval
.
.
.
```

Function implementation in assembly.

```
NAME          MODULE                    ; Name of the program module
?PR?FUNCTION?MODULE  SEGMENT  CODE     ; Seg for program code in 'function';
?PD?FUNCTION?MODULE  SEGMENT  XDATA OVERLAYABLE IPAGE
                                        ; Seg for local vars in 'function'
?BI?FUNCTION?MODULE  SEGMENT  BIT   OVERLAYABLE
                                        ; Seg for local bit vars in
'function'

PUBLIC        _function, ?_function?BYTE, ?_function?BIT
                                        ; Public symbols for C function call

RSEG          ?PD?FUNCTION?MODULE       ; Segment for local variables
?_function?BYTE:                        ; Start of the parameter passing seg
v_a:    DS   2                          ;   int variable: v_a
v_b:    DS   1                          ;   char variable: v_b
v_d:    DS   4                          ;   long variable: v_d
.
.                                       ; Additional local variables
.

RSEG          ?BI?FUNCTION?MODULE       ; Segment for local bit variables
?_function?BIT:                         ; Start of the parameter passing seg
v_c:    DBIT  1                         ;   bit variable: v_c
v_e:    DBIT  1                         ;   bit variable: v_e
.
.                                       ; Additional local bit variables
.

RSEG          ?PR?FUNCTION?MODULE           ; Program segment
_function:    MOV   R0,#?_function?BYTE+0   ; Special function prolog
              MOV   A,R6                    ; and epilog is not
              MOVX  @R0,A                   ; necessary.  All
              INC   R0                      ; vars can immediately
              MOV   A,R7                    ; be accessed
              MOVX  @R0,A
              INC   R0
              MOV   A,R5
              MOVX  @R0,A
.
.
.
              MOV   R6,#HIGH retval         ; Return value
              MOV   R7,#LOW  retval         ; int constant
              RET                           ; Return
```

Large Model Example

In the large model, parameters passed in fixed memory locations are stored in external data memory. The parameter passing segment for variables is located in the **xdata** area.

The following are two assembly code examples. The first shows you how the example function is invoked from assembly. The second example displays the assembly code for the example function.

Function invocation from assembly

```
EXTRN CODE    (_function)           ; Ext declarations for function names
EXTRN XDATA   (?_function?BYTE)     ; Start of transfer for local vars
EXTRN BIT     (?_function?BIT)      ; Start of transfer for local bit vars
.
.
.
        MOV   R6,#HIGH intval       ; int  a
        MOV   R7,#LOW  intval       ; int  a
        MOV   R5,#charconst         ; char b
        SETB  ?_function?BIT+0      ; bit  c
        MOV   R0,#?_function?BYTE+3 ; Address of 'v_d' in the passing area
        MOV   A,longval+0           ; long d
        MOVX  @DPTR,A               ; Store parameter byte
        INC   DPTR                  ; Increment parameter passing address
        MOV   A,longval+1           ; long d
        MOVX  @DPTR,A               ; Store parameter byte
        INC   DPTR                  ; Increment parameter passing address
        MOV   A,longval+2           ; long d
        MOVX  @DPTR,A               ; Store parameter byte
        INC   DPTR                  ; Increment parameter passing address
        MOV   A,longval+3           ; long d
        MOVX  @DPTR,A               ; Store parameter byte
        MOV   C,bitvalue
        MOV   ?_function?BIT+1,C    ; bit  e
        LCALL _function
        MOV   intresult+0,R6        ; Store int
        MOV   intresult+1,R7        ; Retval
.
.
.
```

Function implementation in assembly

```
NAME          MODULE                  ; Name of the program module
?PR?FUNCTION?MODULE  SEGMENT  CODE    ; Seg for program code in 'functions'
?XD?FUNCTION?MODULE  SEGMENT  XDATA   OVERLAYABLE
                                      ; Seg for local vars in 'function'
?BI?FUNCTION?MODULE  SEGMENT  BIT     OVERLAYABLE
                                      ; Seg for local bit vars in 'function'

PUBLIC        _function, ?_function?BYTE, ?_function?BIT
                                      ; Public symbols for C function call

RSEG          ?XD?FUNCTION?MODULE     ; Segment for local variables
?_function?BYTE:                      ; Start of the parameter passing seg
v_a:    DS    2                       ;   int variable: v_a
v_b:    DS    1                       ;   char variable: v_b
v_d:    DS    4                       ;   long variable: v_l
.
.; Additional local variables from 'function'
.

RSEG          ?BI?FUNCTION?MODULE     ; Segment for local bit variables
?_function?BIT:                       ; Start of the parameter passing seg
v_c:    DBIT   1                      ;   bit variable: v_c
v_e:    DBIT   1                      ;   bit variable: v_e
.
.                                     ; Additional local bit variables
.

RSEG          ?PR?FUNCTION?MODULE             ; Program segment
_function:    MOV   DPTR,#?_function?BYTE+0   ; Special function prolog
              MOV   A,R6                      ; and epilog is not
              MOVX  @DPTR,A                   ; necessary.  All vars
              INC   R0                        ; can immediately be
              MOV   A,R7                      ; accessed.
              MOVX  @DPTR,A
              INC   R0
              MOV   A,R5
              MOVX  @DPTR,A
.
.
.
              MOV   R6,#HIGH retval           ; Return value
              MOV   R7,#LOW  retval           ; int constant
              RET                             ; Return
```

Interfacing C Programs to PL/M-51

You can easily interface C51 to routines written in PL/M-51. Intel's PL/M-51 is a popular programming language that is similar to C in many ways. The PL/M-51 compiler generates object files in the OMF-51 format. You can access PL/M-51 functions from C by declaring them with the **alien** function type specifier. Public variables declared in the PL/M-51 module are available to your C programs.

C51 can optionally operate with PL/M-51 parameter passing conventions. The **alien** function type specifier is used to declare public or external functions that are compatible with PL/M-51 in any memory model. For example:

```
extern alien char plm_func (int, char);

alien unsigned int c_func (unsigned char x, unsigned char y)  {
  return (x * y);
}
```

Parameters and return values of PL/M-51 functions may be any of the following types: **bit**, **char**, **unsigned char**, **int**, and **unsigned int**. Other types, including **long**, **float**, and all types of pointers, can be declared in C functions with the **alien** type specifier. However, use these types with care because PL/M-51 does not directly support 32-bit binary integers or floating-point numbers.

PL/M-51 does not support variable-length argument lists. Therefore, functions declared using the **alien** type specifier must have a fixed number of arguments. The ellipsis notation used for variable-length argument lists is not allowed for **alien** functions and causes C51 to generate an error message. For example:

```
extern alien unsigned int  plm_i (char, int, ...);

*** ERROR IN LINE 1 OF A.C: 'plm_i': Var_parms on alien function
```

Data Storage Formats

This section describes the storage formats of the data types available in C51. C51 provides you with a number of basic data types to use in your C programs. The following table lists these data types along with their size requirements and value ranges.

Data Type	Bits	Bytes	Value Range
bit	1	—	0 to 1
signed char	8	1	-128 to +127
unsigned char	8	1	0 to 255
enum	16	2	-32768 to +32767
signed short	16	2	-32768 to +32767
unsigned short	16	2	0 to 65535
signed int	16	2	-32768 to +32767
unsigned int	16	2	0 to 65535
signed long	32	4	-2147483648 to 2147483647
unsigned long	32	4	0 to 4294967295
float	32	4	±1.175494E-38 to ±3.402823E+38
data *, idata *, pdata *	8	1	0x00 to 0xFF
code*, xdata *	16	2	0x0000 to 0xFFFF
generic pointer	24	3	Memory type (1 byte); Offset (2 bytes) 0 to 0xFFFF

Other data types, like structures and unions, may contain scalars from this table. All elements of these data types are allocated sequentially and are byte-aligned due to the 8-bit architecture of the 8051 family.

Bit Variables

Scalars of type **bit** are stored using a single bit. Pointers to and arrays of **bit** are not allowed. Bit objects are always located in the bit-addressable internal memory space of the 8051 CPU. The BL51 Linker/Locator overlays bit objects if possible.

Signed and Unsigned Characters, Pointers to data, idata, and pdata

Scalars of type **char** are stored in a single byte (8 bits). Memory-specific pointers that reference **data**, **idata**, and **pdata** are also stored using a single byte (8 bits).

Signed and Unsigned Integers, Enumerations, Pointers to xdata and code

Scalars of type **int**, scalars of type **short**, **enum** types, and memory-specific pointers that reference **xdata** or **code** are all stored using 2 bytes (16 bits). The high-order byte is stored first, followed by the low-order byte. For example, an integer value of 0x1234 is stored in memory as follows:

Address	+0	+1
Contents	0x12	0x34

Signed and Unsigned Long Integers

Scalars of type **long** are stored using 4 bytes (32 bits). The bytes are stored in high to low order. For example, the long value 0x12345678 is stored in memory as follows:

Address	+0	+1	+2	+3
Contents	0x12	0x34	0x56	0x78

Generic Pointers

Generic pointers have no declared explicit memory type. They may point to any memory area on the 8051. These pointers are stored using 3 bytes (24 bits). The first byte contains a value that indicates the memory area or memory type. The remaining two bytes contain the address offset with the high-order byte first. The following memory format is used:

Address	+0	+1	+2
Contents	Memory Type	Offset; High-Order Byte	Offset; Low-Order Byte

The memory type byte may have one of the following values:

Memory Type	idata / data / bdata	xdata	pdata	code
Value	0x00	0x01	0xFE	0xFF

Use of any other memory type values may lead to unpredictable program behavior.

The following example shows the memory storage of a generic pointer that references address 0x1234 in the **xdata** memory area.

Address	+0	+1	+2
Contents	0x01	0x12	0x34

Floating-point Numbers

Scalars of type **float** are stored using 4 bytes (32 bits). The format used corresponds to that of the IEEE-754 standard.

There are two components of a floating-point number: the mantissa and the exponent. The mantissa stores the actual digits of the number. The exponent stores the power to which the mantissa must be raised. The exponent is an 8-bit value in the 0 to 255 range and is stored relative to 127. The actual value of the exponent is calculated by subtracting 127 from the stored value (0 to 255). The value of the exponent can be anywhere from +128 to -127. The mantissa is a 24-bit value whose most significant bit (MSB) is always 1 and is, therefore, not stored. There is also a sign bit which indicates if the floating-point number is positive or negative.

Floating-point numbers are stored in 8051 memory using the following format:

Address	+0	+1	+2	+3
Contents	**SEEE EEEE**	**EMMM MMMM**	**MMMM MMMM**	**MMMM MMMM**

where:

S represents the sign bit where 1 is negative and 0 is positive.

E is the two's complement exponent with an offset of 127.

M is the 23-bit normal mantissa. The highest bit is always 1 and, therefore, is not stored

Using the above format, the floating-point number -12.5 would be stored as a hexadecimal value of 0xC1480000. In memory, this appears as follows:

Address	+0	+1	+2	+3
Contents	0xC1	0x48	0x00	0x00

It is fairly simple to convert floating-point numbers to and from their hexadecimal storage equivalents. The following example demonstrates how this is done for the value -12.5 shown above.

The floating-point storage representation is not an intuitive format. To convert this to a floating-point number, the bits must be separated as specified in the storage format table above. For example:

Address	+0	+1	+2	+3
Format	SEEEEEEE	EMMMMMMM	MMMMMMMM	MMMMMMMM
Binary	11000001	01001000	00000000	00000000
Hex	C1	48	00	00

From this illustration, you can determine the following information:

- The sign bit is **1**, indicating a negative number.
- The exponent value is **10000010** binary or 130 decimal. Subtracting 127 from 130 leaves 3 which is the actual exponent.
- The mantissa appears as the following binary number:

```
10010000000000000000000
```

There is an understood decimal point at the left of the mantissa that is always preceded by a **1**. This digit is not stored in the hexadecimal representation of the floating-point number. Adding **1** and the decimal point to the beginning of the mantissa gives the following:

```
1.10010000000000000000000
```


Now, adjust the mantissa for the exponent. A negative exponent moves the decimal point to the left. A positive exponent moves the decimal point to the right. Because the exponent is 3, the mantissa is adjusted as follows:

```
1100.10000000000000000000
```

The result is now a binary floating-point number. Binary digits left of the decimal point represent the power of two corresponding to the position: **1100** represents $(1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (0 \times 2^0)$ which equals **12**.

Binary digits that are right of the decimal point also represent the power of two corresponding to their position. However, the powers are negative: **.100**… represents $(1 \times 2^{-1)} + (0 \times 2^{-2}) + (0 \times 2^{-3}) + \ldots$ which equals **.5**.

Adding these values together gives **12.5** which must be negated since the sign bit is set. So, the floating-point hexadecimal value **0xC1480000** is **-12.5**.

Floating-point Errors

The 8051 does not contain an interrupt vector to trap floating-point errors; therefore, your software must appropriately respond to these error conditions. In addition to the normal floating-point values, a floating-point number may contain a binary error value. These values are defined as a part of the IEEE standard and are used whenever an error occurs during normal processing of floating-point operations. Your code should check for possible arithmetic errors at the end of each floating-point operation.

Name	Value	Meaning
NaN	0xFFFFFFF	Not a number
+INF	0x7F80000	Positive infinity (positive overflow)
-INF	0xFF80000	Negative infinity (negative overflow)

NOTE
*The C51 library function **_chkfloat_** lets you quickly check floating-point status.*

You can use the following **union** to store floating-point values.

```
union f  {
  float          f;          /* Floating-point value */
  unsigned long ul;          /* Unsigned long value */
};
```

This **union** contains a **float** and an **unsigned long** in order to perform floating-point math operations and to respond to the IEEE error states. For example:

```
#define NaN       0xFFFFFFFF    /* Not a number (error) */
#define plusINF   0x7F800000    /* Positive overflow    */
#define minusINF  0xFF800000    /* Negative overflow    */

union f {
  float           f;            /* Floating-point value */
  unsigned long   ul;           /* Unsigned long value */
};

void main (void)  {
  float a, b;
  union f x;

  x.f = a * b;
  if (x.ul == NaN  || x.ul == plusINF  || x.ul == minusINF)  {
    /* handle the error  */
  }
  else  {
    /* result is correct */
  }
}
```

Accessing Absolute Memory Locations

The C programming language does not support a method of explicitly specifying the memory location of a static or global variable. There are three ways to reference explicit memory location. You can use the:

- Absolute memory access macros
- Linker location controls
- The **_at_** keyword

Each of these three methods is described below.

Absolute Memory Access Macros

First, you may use the absolute memory access macros provided as part of the C51 library. Use the following macros to directly access the memory areas of the 8051.

CBYTE	**XBYTE**	**PWORD**
DBYTE	**CWORD**	**XWORD**
PBYTE	**DWORD**	

Refer to “Absolute Memory Access Macros” on page 178 for definitions of these macros.

Linker Location Controls

The second method of referencing explicit memory location is to declare the variables in a stand-alone C module, and use the location control directives of the BL51 Linker/Locator to specify an absolute memory address.

In the following example, assume that we have a structure called **alarm_control** that we want to reside at address 2000h in **xdata**. We start by entering a source file named **ALMCTRL.C** that contains only the declaration for this structure.

```
.
.
.
struct alarm_st  {
  unsigned int alarm_number;
  unsigned char enable flag;
  unsigned int time_delay;
  unsigned char status;
 };

xdata struct alarm_st alarm_control;
.
.
.
```

The C51 compiler generates an object file for **ALMCTRL.C** and includes a segment for variables in the **xdata** memory area. Because it is the only variable declared in this module, **alarm_control** is the only variable in that segment. The name of the segment is ?XD?ALMCTRL.

The BL51 Linker/Locator allows you to specify the base address of any segment by using the location control directives. Because the **alarm_control** variable was declared to reside in **xdata**, the **XDATA** BL51 directive must be used as follows:

```
BL51 … almctrl.obj XDATA(?XD?ALMCTRL(2000h)) …
```

This instructs the linker to locate the segment named ?XD?ALMCTRL at address 2000h in the **xdata** memory area.

There are linker directives for locating segments in the **code**, **xdata**, **pdata**, **idata**, and **data** memory areas. Refer to the *8051 Utilities User's Guide* for more information about the Linker/Locator.

The _at_ Keyword

The third method of accessing absolute memory locations is to use the **_at_** keyword when you declare variables in your C source files. The following example demonstrates how to locate several different variable types using the **_at_** keyword.

```
struct link  {
  struct link idata *next;
  char        code  *test;
};

idata struct link list _at_ 0x40;     /* list at idata 0x40 */
xdata char text[256]   _at_ 0xE000;   /* array at xdata 0xE000 */
xdata int i1           _at_ 0x8000;   /* int at xdata 0x8000 */

void main ( void ) {
  link.next = (void *) 0;
  i1        = 0x1234;
  text [0]  = 'a';
}
```

Refer to “Absolute Variable Location” on page 71 for more information about the **_at_** keyword.

Debugging

C51 uses the Intel Object Format (OMF-51) for object files and generates complete symbol information. All Intel compatible emulators may be used for program debugging. The **DEBUG** control directive embeds debugging information in the object file. In addition, the **OBJECTEXTEND** control directive embeds additional variable type information in the object file which allows type-specific display of variables and structures when using certain emulators.

Chapter 7. Error Messages

This chapter lists Fatal Error, Syntax Error, and Warning messages that you may encounter as you develop a program. Each section includes a brief description of the message as well as corrective actions you can take to eliminate the error or warning condition.

Fatal Errors

Fatal errors cause immediate termination of the compilation. These errors normally occur as the result of invalid options specified on the command line. Fatal errors are also generated when the compiler cannot access a specified source include file.

Fatal error messages conform to one of the following formats:

```
C51 FATAL-ERROR -
              ACTION:          <current action>
              LINE:            <line in which the error is detected>
              ERROR:           <corresponding error message>
C51 TERMINATED.

C51 FATAL-ERROR -
              ACTION:          <current action>
              FILE:            <file in which the error is detected>
              ERROR:           <corresponding error message>
C51 TERMINATED.
```

The following are descriptions of the possible text for the **Action** and **Error** fields in the above messages.

Actions

ALLOCATING MEMORY
The compiler could not allocate enough memory to compile the specified source file.

CREATING LIST-FILE / OBJECT-FILE / WORKFILE
The compiler could not create the list file, object file, or work file. This error may occur if the disk is full or write-protected, or if the file already exists and is read only.

GENERATING INTERMEDIATE CODE
The source file contains a function that is too large to be translated into pseudo-code by the compiler. Try breaking the function into smaller functions and re-compiling.

OPENING INPUT-FILE
The compiler failed to find or open the selected source or include file.

PARSING INVOKE-/#PRAGMA-LINE
An error was detected while evaluating arguments on the command line or while evaluating parameters in a **#pragma** statement.

PARSING SOURCE-FILE / ANALYZING DECLARATIONS
The source file contains too many external references. Reduce the number of external variables and functions accessed by the source file.

WRITING TO FILE
An error was encountered while writing to the list file, object file, or work file.

Errors

'(' AFTER CONTROL EXPECTED
Some control parameters need an argument enclosed in parentheses. This message is displayed when the left parenthesis is missing.

')' AFTER PARAMETER EXPECTED
This message indicates that the right parenthesis of the enclosed argument is missing.

BAD DIGIT IN NUMBER
The numerical argument of a control parameter contains invalid characters. Only decimal digits are acceptable.

CAN'T CREATE FILE
The filename defined on the **FILE** line cannot be created.

CAN'T HAVE GENERAL CONTROL IN INVOCATION LINE
General controls (for example, **EJECT**) cannot be included on the command line. Place these controls in the source file using the **#pragma** statement.

FILE DOES NOT EXIST
The filename defined on the **FILE** line, cannot be found.

FILE WRITE-ERROR
An error occurred while writing to the list, preprint, work, or object file because of insufficient disk space.

IDENTIFIER EXPECTED
This message is generated when the **DEFINE** control has no arguments. **DEFINE** requires an identifier as its argument. This is the same convention as in the C language.

MEMORY SPACE EXHAUSTED
The compiler could not allocate enough memory to compile the specified source file. If you receive this message consistently, you should break the source file into two or more smaller files and re-compile. Alternatively, you can add more memory to your PC because the C51 compiler uses a DOS extender to utilize all extended memory available.

MORE THAN 100 ERRORS IN SOURCE-FILE
During the compilation more than 100 errors were detected. This causes the termination of the compiler.

MORE THAN 256 SEGMENTS/EXTERNALS
More than 256 total references were encountered in a source file. A single source file cannot contain more than 256 functions or external references. This is a historical restriction mandated by the Intel Object Module Format

(OMF-51). Functions which contain scalar and/or **bit** declarations produce two and sometimes three segment definitions in the object file.

NON-NULL ARGUMENT EXPECTED

The selected control parameter needs an argument (for example, a filename or a number) enclosed in parentheses.

OUT OF RANGE NUMBER

The numerical argument of a control parameter is out of range. For instance, the **OPTIMIZE** control allows only the numbers 0 through 6. A value of 7 would generate this error message.

PARSE STACK OVERFLOW

The parse stack has overflowed. This can occur if the source program contains extremely complex expressions or if blocks are nested more than 31 levels deep.

PREPROCESSOR: LINE TOO LONG (32K)

An intermediate expansion exceeded 32K characters in length.

PREPROCESSOR: MACROS TOO NESTED

During macro expansion the stack consumption of the preprocessor grew too large to continue. This message usually indicates a recursive macro definition, but can also indicate a macro with too many levels of nesting.

RESPECIFIED OR CONFLICTING CONTROL

A command-line parameter was specified twice or conflicting command-line parameters were specified.

SOURCE MUST COME FROM A DISK-FILE

The source and include files must exist on either a hard disk or diskette. The console **CON:**, **:CI:**, or similar devices are not allowed as input files.

UNKNOWN CONTROL

The selected control parameter is unrecognized by the compiler.

Syntax and Semantic Errors

Syntax and semantic errors typically occur in the source program. They identify actual programming errors. When one of these errors is encountered, the compiler attempts to recover from the error and continue processing the source file. As more errors are encountered, the compiler outputs additional error messages. However, no object file is produced.

Syntax and semantic errors produce a message in the list file. These error messages are in the following format:

***** ERROR *number* IN LINE *line* OF *file*: *error message***

where:

number	is the error number.
line	corresponds to the line number in the source file or include file.
file	is the name of the source or include file in which the error was detected.
error message	is descriptive text and is dependent upon the type of error encountered.

The following table lists syntax and semantic errors by error number. The error message displayed is listed along with a brief description and possible cause and correction.

Number	Error Message and Description
100	**Unprintable character 0x?? skipped** An illegal character was found in the source file. (Note that characters inside a comment are not checked.)
101	**Unclosed string** A string is not terminated with a quote (**"**).
102	**String too long** A string may not contain more than 4096 characters. Use the concatenation symbol ('\') to logically continue strings longer than 4096 characters. Lines terminated in this fashion are concatenated during lexical analysis.
103	**Invalid character constant** A character constant has an invalid format. The notation **'\c'** is valid only when **c** is any printable ASCII character.
125	**Declarator too complex (20)** The declaration of an object may contain a maximum of 20 type modifiers ('[', ']', '*', '(', ')'). This error is almost always followed by error 126.

Number	Error Message and Description
126	**Type-stack underflow** The type declaration stack has underflowed. This error is usually a side-effect of error 125.
127	**Invalid storage class** An object was declared with an invalid memory space specification. This occurs if an object is declared with storage class of **auto** or **register** outside of a function.
129	**Missing ';' before 'token'** This error usually indicates that a semicolon is missing from the previous line. When this error occurs, the compiler may generate an excess of error messages.
130	**Value out of range** The numerical argument after a **using** or **interrupt** specifier is invalid. The **using** specifier requires a register bank number between 0 and 3. The **interrupt** specifier requires an interrupt vector number between 0 and 31.
131	**Duplicate function-parameter** A formal parameter name exists more than once within a function. The formal parameter names must be unique in function declarations.

132	**Not in formal parameter list**
	The parameter declarations inside a function use a name not present in the parameter name list. For example:
	`char function (v0, v1, v2)`
	`char *v0, *v1, *v5;`
	`/* 'v5' is unknown in the formal list */`
	`{`
	`  /* ... */`
	`}`

134	**xdata/idata/pdata/data on function not permitted** Functions always reside in **code** memory and cannot be executed out of other memory areas. Functions are implicitly defined as memory type **code**.
135	**Bad storage class for bit** Declarations of **bit** scalars may include one of the **static** or **extern** storage classes. The **register** or **alien** classes are invalid.
136	**'void' on variable** The type **void** is allowed only as a non-existent return value or an empty argument list for functions (**void func (void)**), or in combination with a pointer (**void ***).
138	**Interrupt() may not receive or return value(s)** An interrupt function was defined with one or more formal parameters or with a return value. Interrupt functions may not contain invocation parameters or return values.
140	**Bit in illegal memory-space** Definitions of **bit** scalars may contain the optional memory type **data**. If the memory type is missing then the type **data** is assumed, because bits always reside in the internal data memory. This error can occur when an attempt is made to use another data type with a **bit** scalar definition.
141	**Syntax error near *token*: expected *other_token*, …** The ***token*** seen by the compiler is wrong. Depending upon the context the expected token is displayed.
142	**Invalid base address** The base-address of an **sfr** or **sbit** declaration is in error. Valid bases are values in the 0x80 to 0xFF range. If the declaration uses the notation **base^pos**, then the base address must also be a multiple of eight.

Number	Error Message and Description
143	**Invalid absolute bit address** The absolute address in **sbit** declarations must be in the 0x80 to 0xFF range.
144	**Base^pos: invalid bit position** The definition of the bit position within an **sbit** declaration must be in the 0 to 7 range.
145 146	**Undeclared sfr** **Invalid sfr** The declaration of an absolute bit (base^pos) contains an invalid base-specification. The base must be the name of a previously declared **sfr**. Any other names are invalid.
147	**Object too large** The size of a single object may not exceed the absolute limit of 65535 (64 Kbytes - 1).
149	**Function member in struct/union** A struct or union may not contain a function-type member. However, pointers to functions are perfectly valid.
150	**Bit member in struct/union** A union-aggregate may not contain members of type **bit**. This restriction is imposed due to the architecture of the 8051.
151	**Self relative struct/union** A structure cannot contain an instance of itself.
152	**Bit-field type too small for number of bits** The number of bits specified in the bit-field declaration exceeds the number of bits in the given base type.
153	**Named bit-field cannot have zero width** The named field had a zero width. Only unnamed bit-fields are allowed to have zero width.
154	**Ptr to field** Pointers to bit-fields are not valid types.
155	**char/int required for fields** The base type for bit-fields requires one of the types **char** or **int**. **unsigned char** and **unsigned int** types are also valid.
156 157	**Alien permitted on functions only** **Var_parms on alien function** The storage class **alien** is allowed only for external PL/M-51 functions. The formal notation (**char ***, ...) is not legal on **alien** functions. PL/M-51 functions always require a fixed number of parameters.
158	**Function contains unnamed parameter** The parameter list of a function definition contains an unnamed abstract type definition. This notation is permitted only in function prototypes.
159	**Type follows void** Prototype declarations of functions may contain an empty parameter list (for example, **int func (void)**). This notation may not contain further type definitions after **void**.
160	**void invalid** The **void** type is legal only in combination with pointers or as the non-existent return value of a function.
161	**Formal parameter ignored** A declaration of an external function inside a function used a parameter name list without any type specification (for example, **extern yylex(a,b,c);**).

Number	Error Message and Description
162	**Duplicate function-parameter** The name of a defined object inside a function duplicates the name of a parameter.
163	**Unknown array size** In general, a formal size specifier is not required for external, single, or multi-dimensional arrays. Typically, the compiler calculates the size at initialization time. For external arrays, the size is of no great interest. This error is the result of attempting to use the **sizeof** operator on an undimensioned array or on a multi-dimensional array with undefined element sizes.
164	**Ptr to nul** This error is usually the result of a previous error for a pointer declaration.
165	**Ptr to bit** The type combination pointer to **bit** is not a legal type.
166	**Array of functions** Arrays cannot contain functions; however, they may contain pointers to functions.
167	**Array of fields** Bit-fields may not be arranged as arrays.
168	**Array of bit** An array may not have type **bit** as its basic type. This limitation is imposed by the architecture of the 8051.
169	**Function returns function** A function cannot return a function; however, a function may return a pointer to a function.
170	**Function returns array** A function cannot return an array; however, a pointer to an array is valid.
171	**Missing enclosing loop** A **break** or **continue** statement may occur only within a **for**, **while**, **do**, or **switch** statement.
172	**Missing enclosing switch** A **case** statement may occur only within a **switch** statement.
173	**Missing return-expression** A function which returns a value of any type but **int**, must contain a **return** statement including an expression. Because of compatibility to older programs, no check is done on functions which return an **int** value.
174	**Return-expression on void-function** A **void** function cannot return a value and thus may not contain a **return** statement.
175	**Duplicate case value** Each **case** statement must contain a constant expression as its argument. The value must not occur more than once in the given level of the **switch** statement.
176	**More than one 'default'** A **switch** statement may not contain more than one **default** statement.
177	**Different struct/union** Different types of structures are used in an assignment or as an argument to a function.
178	**Struct/union comparison illegal** The comparison of two structures or unions is not allowed according to ANSI.
179	**Illegal type conversation from/to 'void'** Type casts to or from **void** are invalid.

Number	Error Message and Description
180	**Can't cast to 'function'** Type casts to function types are invalid. Try casting to a pointer to a function.
181	**Incompatible operand** At least one operand type is not valid with the given operator (for example, ~**float_type**).
183	**Unmodifiable lvalue** The object to be changed resides in **code** memory or has **const** attribute and therefore cannot be modified.
184	**Sizeof: illegal operand** The **sizeof** operator cannot determine the size of a function or bit-field.
185	**Different memory space** The memory space of an object declaration differs from the memory space of a prior declaration for the same object.
186	**Invalid dereference** This error message may be caused by an internal compiler problem. Please contact technical support if this error is repeated.
187	**Not an lvalue** The needed argument must be the address of an object that can be modified.
188	**Unknown object size** The size of an object cannot be computed because of a missing dimension size on an array or indirection via a **void** pointer.
189	**'&' on bit/sfr illegal** The address-of operator (**'&'**) is not allowed on **bit** objects or special function registers (**sfr**).
190	**'&': not an lvalue** An attempt was made to construct a pointer to an anonymous object.
193 193 193 193	**Illegal op-type(s)** **Illegal add/sub on ptr** **Illegal operation on bit(s)** **Bad operand type** This error results when an expression uses illegal operand-types with the given operator. Examples of invalid expressions are **bit** * **bit**, ptr + ptr, or ptr * *anything*. The error message includes the operator which caused the error. The following operations may be executed with bit-type operands: ■ Assignment (**=**) ■ OR / Compound OR (**|**, **|=**) ■ AND / Compound AND (**&**, **&=**) ■ XOR / Compound XOR (**^**, **^=**) ■ Compare bit with bit or constant (**==**, **!=**) ■ Negation (**~**) **bit** operands may be used in expressions with other data types. In this case a type cast is automatically performed.
194	**'*' indirection to object of unknown size** The indirection operator ***** may not be used with **void** pointers because the object size, which the pointer refers to, is unknown.
195	**'*' illegal indirection** The ***** operator may not be applied on non-pointer arguments.

Number	Error Message and Description
196	**Mspace probably invalid** The conversion of a constant to a pointer constant yields an invalid memory space, for example char *p = 0x91234.
198	**Sizeof returns zero** The sizeof operator returns a zero value.
199	**Left side of '->' requires struct/union pointer** The argument on the left side of the **->** operator must be a **struct** pointer or a **union** pointer.
200	**Left side of '.' requires struct/union** The argument on the left side of the **.** operator must have type **struct** or **union**.
201	**Undefined struct/union tag** The given **struct** or **union** tag name is unknown.
202	**Undefined identifier** The given identifier is undefined.
203	**Bad storage class (nameref)** This error indicates a problem within the compiler. Please contact technical support if this error is repeated.
204	**Undefined member** The given member name in a **struct** or **union** reference is undefined.
205	**Can't call an interrupt function** An **interrupt** function should not be called like a normal function. The entry and exit code for these functions is specially coded for interrupts.
207	**Declared with 'void' parameter list** A function declared with a **void** parameter list cannot receive parameters from the caller.
208	**Too many actual parameters** The function call includes more parameters than previously declared.
209	**Too few actual parameters** Too few actual parameters were included in a function call.
210	**Too many nested calls** Function calls can be nested at most 10 levels deep.
211	**Call not to a function** The term of a function call does not evaluate to a function or pointer to function.
212	**Indirect call: parameters do not fit within registers** An indirect function call through a pointer cannot contain actual parameters. An exception to this rule is when all parameters can be passed in registers. This is due to the method of parameter passing employed by C51. The name of the called function must be known because parameters are written into the data segment of the called function. For indirect calls, however, the name of the called function is not known.
213	**Left side of asn-op not an lvalue** The address of a changeable object is required at the right side of the assignment operator.
214	**Illegal pointer conversion** Objects of type bit, float or aggregates cannot be converted to pointers.
215	**Illegal type conversion** Struct/union/void cannot be converted to any other types.

Number	Error Message and Description
216	**Subscript on non-array or too many dimensions** An array reference contained either too many dimension specifiers or the object was not an array.
217	**Non-integral index** The dimension expression of an array must be of the type **char**, **unsigned char**, **int**, or **unsigned int**. All other types are illegal.
218	**Void-type in controlling expression** The limit expression in a **while**, **for**, or **do** statement cannot be of type **void**.
219	**Long constant truncated to int** The value of a constant expression must be capable of being represented by an **int** type.
220	**Illegal constant expression** A constant expression is expected. Object names, variables or functions, are not allowed in constant expressions.
221	**Non-constant case/dim expression** A case value or a dimension specification (**[ ]**) must be a constant expression.
222 223	**Div by zero** **Mod by zero** The compiler detected a division or a modulo by zero.
225	**Expression too complex, simplify** An expression is too complex and must be broken into two or more sub expressions.
226	**Duplicate struct/union/enum tag** The name for a **struct**, **union**, or **enum** is already defined within current scope.
227	**Not a union tag** The name for a **union** is already defined as a different type.
228	**Not a struct tag** The name for a **struct** is already defined as a different type.
229	**Not an enum tag** The name for an **enum** is already defined as a different type.
230	**Unknown struct/union/enum tag** The specified **struct**, **union**, or **enum** name is undefined.
231	**Redefinition** The specified name is already defined and cannot be redefined.
232	**Duplicate label** The specified label is already defined.
233	**Undefined label** This message indicates a label that was accessed but was not defined. Sometimes this message appears several lines after the actual label reference. This is caused by the method used to search for undefined labels.
234	**'{', scope stack overflow(31)** The maximum of 31 nested blocks has been exceeded. Additional levels of nested blocks are ignored.
235	**Parameter <number>: different types** Parameter types in the function declaration are different from those in the function prototype.

Number	Error Message and Description
236	**Different length of parameter lists** The number of parameters in the function declaration is different from the number of parameters in the function prototype.
237	**Function already has a body** An attempt was made to declare a body for a function twice.
238 239	**Duplicate member** **Duplicate parameter** An attempt was made to define an already defined **struct** member or function parameter.
240	**More than 128 local bit's** No more than 128 **bit**-scalars may be defined inside a function.
241	**Auto segment too large** The required space for local objects exceeds the model-dependent maximum. The maximum segment sizes are defined as follows: **SMALL** 128 bytes **COMPACT** 256 bytes **LARGE** 65535 bytes
242	**Too many initializers** The number of initializers exceeded the number of objects to be initialized.
243	**String out of bounds** The number of characters in the string exceeds the number of characters required to initialize the array of characters.
244	**Can't initialize, bad type or class** An attempt was made to initialize a **bit** or an **sfr**.
245	**Unknown pragma, line ignored** The **#pragma** statement is unknown so, the entire line is ignored.
246	**Floating-point error** This error occurs when a floating-point argument lies outside of the valid range for 32-bit floating values. The numeric range of the 32-bit IEEE values is: ±1.175494E-38 to ±3.402823E+38.
247	**Non-address/constant initializer** A valid initializer expression must evaluate to a constant value or the name of an object plus or minus a constant.
248	**Aggregate initialization needs curly braces** The braces (**{ }**) around the given **struct** or **union** initializer were missing.
249	**Segment <name>: Segment too large** The compiler detected a data segment that was too large. The maximum size of a data segment depends on memory space.
250	**'\esc'; value exceeds 255** An escape sequence in a string constant exceeds the valid value range. The maximum value is 255.
251	**Illegal octal digit** The specified character is not a valid octal digit.
252	**Misplaced primary control, line ignored** Primary controls must be specified at the start of the C module before any **#include** directives or declarations.

Number	Error Message and Description
253	**Internal error (ASMGEN\CLASS)** This error can occur under the following circumstances: ▪ An intrinsic function (for example, _**testbit**_) was activated incorrectly. This is the case when no prototype of the function exists and the number of actual parameters or their type is incorrect. For this reason, the appropriate declaration files must always be used (**INTRINS.H**, **STRING.H**). See Chapter 8 for more information on **intrinsic** functions. ▪ C51 recognized an internal consistency problem. Please contact technical support if this error occurs.
255	**Switch expression has illegal type** The expression in a switch statement has not a legal data type.
256	**Conflicting memory model** A function which contains the **alien** attribute may contain only the model specification **small**. The parameters of the function must lie in internal data memory. This applies to all external **alien** declarations and **alien** functions. For example: `alien plm_func (char c) large { ... }` generates error 256.
257	**Alien function cannot be reentrant** A function that contains the **alien** attribute cannot simultaneously contain the attribute **reentrant**. The parameters of the function cannot be passed via the virtual stack. This applies to all external **alien** declarations and **alien** functions.

258	**Mspace illegal on struct/union member**
	Mspace on parameter ignored
	A member of a structure or a parameter may not contain the specification of a memory type. The object to which the pointer refers may, however, contain a memory type. For example:
	`struct vp { char code c; int xdata i; };`
	generates error 258.
	`struct v1 { char c; int xdata *i; };`
	is the correct declaration for the **struct**.

259	**Pointer: different mspace**
	A spaced pointer has been assigned another spaced pointer with a different memory space. For example:
	`char xdata *p1;`
	`char idata *p2;`
	`p1 = p2;  /* different memory spaces */`

260	**Pointer truncation** A spaced pointer has been assigned some constant value which exceeds the range covered by the pointers memory space. For example: `char idata *p1 = 0x1234;  /* result is 0x34 */`

Number	Error Message and Description

261	**Bit(s) in reentrant ( )**
	A function with the attribute reentrant cannot have bit objects declared inside the function. For example:
	
	`|int func1 (int i1) reentrant {`
	`  bit b1, b2;  /* not allowed ! */`
	`  return (i1 - 1);`
	`}`

262	**'using/disable': can't return bit value**
	Functions declared with the **using** attribute and functions which rely on disabled interrupts (**#pragma disable**) cannot return a **bit** value to the caller. For example:
	
	`bit test (void) using 3`
	`{`
	`  bit b0;`
	`  return (b0);`
	`}`
	
	produces error 262.

263	**Save/restore: save-stack overflow/underflow** The maximum nesting depth **#pragma save** comprises eight levels. The pragmas **save** and **restore** work with a stack according to the LIFO (last in, first out) principal.
264	**Intrinsic '<intrinsic_name>': declaration/activation error** This error indicates that an **intrinsic** function was defined incorrectly (parameter number or ellipsis notation). This error should not occur if you are using the standard **.H** files. Make sure that you are using the **.H** files that were included with C51. Do not try to define your own prototypes for intrinsic library functions.
265	**Recursive call to non-reentrant function** Non reentrant functions cannot be called recursively since such calls would overwrite the parameters and local data of the function. If you need recursive calls, you should declare the function with the reentrant attribute.
267	**Funcdef requires ANSI-style prototype** A function was invoked with parameters but the declaration specifies an empty parameter list. The prototype should be completed with the parameter types in order to give the compiler the opportunity to pass parameters in registers and have the calling mechanism consistent over the application.
268	**Bad taskdef (taskid/priority/using)** The task declaration is incorrect.

271	**Misplaced 'asm/endasm' control**
	The asm and endasm statements may not be nested. Endasm requires that an asm block be opened by a previous asm statement. For example:
	
	`#pragma asm`
	`.`
	`.`
	`.`
	`assembler instruction(s)`
	`.`
	`.`
	`.`
	`#pragma endasm`

Number	Error Message and Description
272	**'asm' requires SRC control to be active** The use of asm and endasm in a source file requires that the file be compiled using the SRC directive. The compiler then generates an assembly source file which may then be assembled with A51.
273	**'asm/endasm' not allowed in include file** The use of asm and endasm is not permitted within include files. For debug reasons executable code should be avoided in include files anyway.
274	**Absolute specifier illegal** The absolute address specification is not allowed on bit objects, functions, and function locals. The address must conform to the memory space of the object. For example, the following declaration is invalid because the range of the indirectly addressable data space is 0x00 to 0xFF. `idata int _at_ 0x1000;`
278	**Constant too big** This error occurs when a floating-point argument lies outside of the valid range for 32-bit floating values. The numeric range of the 32-bit IEEE values is: ±1.175494E-38 to ±3.402823E+38.
279	**Multiple initialization** An attempt has been made to initialize some object more than once.
300	**Unterminated comment** This message occurs when a comment does not have a closing delimiter (*/).
301	**Identifier expected** The syntax of a preprocessor directive expects an identifier.
302	**Misused # operator** This message occurs if the stringize operator **'#'** is not followed by an identifier.
303	**Formal argument expected** This message occurs if the stringize operator **'#'** is not followed by an identifier representing a formal parameter name of the macro currently being defined.
304	**Bad macro parameter list** The macro parameter list does not represent a brace enclosed, comma separated list of identifiers.
305	**Unterminated string/char constant** A string or character constant is invalid. Typically, this error is encountered if the closing quote is missing.
306	**Unterminated macro call** The end of the input file was reached while the preprocessor was collecting and expanding actual parameters of a macro call.
307	**Macro 'name': parameter count mismatch** The number of actual parameters in a macro call does not match the number of parameters of the macro definition. This error indicates that too few parameters were specified.
308	**Invalid integer constant expression** The numerical expression of an if/elif directive contains a syntax error.
309	**Bad or missing file name** The filename argument in an include directive is invalid or missing.
310	**Conditionals too nested(20)** The source file contains too many nested directives for conditional compilation. The maximum nesting level allowed is 20.

Number	Error Message and Description
311 312	**Misplaced elif/else control** **Misplaced endif control** The directives elif, else, and endif are legal only within an if, ifdef, or ifndef directive.
313	**Can't remove predefined macro 'name'** An attempt was made to remove a predefined macro. Existing macros may be deleted using the **#undef** directive. Predefined macros cannot be removed. The compiler recognizes the following predefined macros: **_ _C51_ _** **_ _DATE_ _** **_ _FILE_ _** **_ _MODEL_ _** **_ _LINE_ _** **_ _STDC_ _** **_ _TIME_ _**
314	**Bad # directive syntax** In a preprocessor directive, the character **'#'** must be followed by either a newline character or the name of a preprocessor command (for example, if/define/ifdef, …).
315	**Unknown # directive 'name'** The name of the preprocessor directive is not known to the compiler.
316	**Unterminated conditionals** The number of endifs does not match the number of if or ifdefs after the end of the input file.
318	**Can't open file 'filename'** The given file could not be opened.
319	**'File' is not a disk file** The given file is not a disk file. Files other than disk files are not legal for compilation.
320	**User_error_text** This error number is reserved for errors introduced with the #error directive of the preprocessor. The #error directive causes the user error text to come up with error 320 which counts like some other error and prevents the compiler from generating code.
321	**Missing <character>** In the filename argument of an include directive, the closing character is missing. For example: #include <stdio.h
325	**Duplicate formal parameter 'name'** A formal parameter of a macro may be define only once.
326	**Macro body cannot start or end with '##'** The concat operator (**'##'**) cannot be the first or last token of a macro body.
327	**Macro 'macroname': more than 50 parameters** The number of parameters per macro is limited to 50.

Warnings

Warnings produce information about potential problems which may occur during the execution of the resulting program. Warnings do not hinder compilation of the source file.

Warnings produce a message in the list file. These warning messages are in the following format:

```
*** WARNING number IN LINE line OF file: warning message
```

where:

number	is the error number.
line	corresponds to the line number in the source file or include file.
file	is the name of the source or include file in which the error was detected.
warning message	is descriptive text that is dependent upon the type of warning encountered.

The following table lists warnings by number. The warning message displayed is listed along with a brief description and possible cause and correction.

Number	Warning Message and Description
173	**Missing return-expression** A function which returns a value of any type but int, must contain a return statement including an expression. Because of compatibility to older programs, no check is done on functions which return an int value.
182	**Pointer to different objects** A pointer was assigned the address of a different type.
185	**Different memory space** The memory space of an object declaration differs from the memory space of a prior declaration for the same object.
196	**Mspace probably invalid** This warning is caused by the assignment of an invalid constant value to a pointer. Valid pointer constants are **long** or **unsigned long**. The compiler uses 24 bits (3 bytes) for pointer objects. The low-order 16 bits represent the offset. The high-order 8 bits represent the memory space selector.
198	**Sizeof returns zero** The calculation of the size of an object yields zero. This value may be wrong if the object is external or if not all dimension sizes of an array are known.

Number	Warning Message and Description
206	**Missing function prototype** The called function is unknown because no prototype declaration exists. Calls to unknown functions are always at risk that the number of parameters does not correspond to the actual requirements. If this is the case, the function is called incorrectly. The compiler has no way to check for missing or excessive parameters and their types. Include prototypes of the functions used in your program. Prototypes must be specified before the functions are actually called. The definition of a function automatically produces a prototype.
209	**Too few actual parameters** Too few actual parameters were included in a function call.
219	**Long constant truncated to int** The value of a constant expression must be capable of being represented by an int type.
245	**Unknown pragma, line ignored** The #pragma statement is unknown, so the entire pragma line is ignored.

258	**Mspace illegal on struct/union member**
	Mspace on parameter ignored
	A member of a structure or a parameter may not contain the specification of a memory type. The object to which the pointer refers may, however, contain a memory type. For example:
	
	`struct vp { char code c; int xdata i; };`
	
	generates error 258.
	
	`struct v1 { char c; int xdata *i; };`
	
	is the correct declaration for the **struct**.

259	**Pointer: different mspace** This warning is generated when two pointers that do not refer to the same memory type of object are compared.
260	**Pointer truncation** This error or warning occurs when converting a pointer to a pointer with a smaller offset area. The conversion takes place, but the offset of the larger pointer is truncated to fit into the smaller pointer.
261	**Bit in reentrant function** A **reentrant** function cannot contain bits because **bit** scalars cannot be stored on the virtual stack.
265	**'name': recursive call to non-reentrant function** A direct recursion to a non-reentrant function was discovered. This can be intentional but should be functionally checked (through the generated code) for each individual case. Indirect recursions are discovered by the linker/locator.

Number	Warning Message and Description

271	**Misplaced 'asm/endasm' control**
	The asm and endasm statements may not be nested. Endasm requires that an asm block be opened by a previous asm statement. For example:
	
	`#pragma asm`
	`.`
	`.`
	`.`
	`assembler instruction(s)`
	`.`
	`.`
	`.`
	`#pragma endasm`

275	**Expression with possibly no effect**
	The compiler detected an expression which does not generate code. For example:
	
	`void test (void) {`
	`  int i1, i2, i3;`
	`  i1, i2, i3;  /* dead expression */`
	`  i1 << 3;  /* result is not used */`
	`}`

276	**Constant in condition expression**
	The compiler detected a conditional expression with a constant value. In most cases this is a typing mistake. For example:
	
	`void test (void) {`
	`  int i1, i2, i3;`
	`  if (i1 = 1)  i2 = 3;  /* const assigned with = */`
	`  while (i3 = 2);  /* const assigned with = */`
	`}`

277	**Different mspaces to pointer**
	A typedef declaration has a conflict of the memory spaces. For example:
	
	`typedef char xdata XCC;  /* mspace xdata */`
	`typedef XCC idata PICC;  /* mspace collision */`

280	**Unreferenced symbol/label** This message identifies a symbol or label which has been defined but not used.
307	**Macro 'name': parameter count mismatch** The number of actual parameters in a macro call does not match the number of parameters of the macro definition. This warning indicates that too many parameters were used. Excess parameters are ignored.
317	**Macro 'name': invalid redefinition** A predefined macro cannot be redefined or removed. The compiler recognizes the following predefined macros: **__C51__ __DATE__ __FILE__ __MODEL__** **__LINE__ __STDC__ __TIME__**
322	**Unknown identifier** The identifier in an #if directive line is undefined (evaluates to FALSE).
323	**Newline expected, extra characters found** A #directive line is correct but contains extra non commented characters. For example: `#include <stdio.h>  foo`

Number	Warning Message and Description
324	**Preprocessor token expected** A preprocessor token was expected but a newline character was found in input. For example: #line where the arguments to the #line directive are missing.

Chapter 8. Library Reference

The C51 run-time library provides you with more than 100 predefined functions and macros to use in your 8051 C programs. This library makes embedded software development easier by providing you with routines that perform common programming tasks such as string and buffer manipulation, data conversion, and floating-point math operations.

Typically, the routines in this library conform to the ANSI C Standard. However, some functions differ slightly in order to take advantage of the features found in the 8051 architecture. For example, the function **isdigit** returns a **bit** value as opposed to an **int**. Where possible, function return types and argument types are adjusted to use the smallest possible data type. In addition, unsigned data types are favored over signed types. These alterations to the standard library provide a maximum of performance while also reducing program size.

All routines in this library are implemented to be independent of and to function using any register bank.

Intrinsic Routines

The C51 compiler supports a number of intrinsic library functions. Non-intrinsic functions generate **ACALL** or **LCALL** instructions to perform the library routine. Intrinsic functions generate in-line code to perform the library routine. The generated in-line code is much faster and more efficient than a called routine would be. The following functions are available in intrinsic form:

crol	**_iror_**	**_nop_**
cror	**_lrol_**	**_testbit_**
irol	**_lror_**	

These routines are described in detail in the following sections.

Library Files

The C51 library includes six different compile-time libraries which are optimized for various functional requirements. These libraries support most of the ANSI C function calls.

Library File	Description
C51S.LIB	Small model library without floating-point arithmetic
C51FPS.LIB	Small model floating-point arithmetic library
C51C.LIB	Compact model library without floating-point arithmetic
C51FPC.LIB	Compact model floating-point arithmetic library
C51L.LIB	Large model library without floating-point arithmetic
C51FPL.LIB	Large model floating-point arithmetic library
80C751.LIB	Library for use with the Signetics 8xC751 and derivatives.

Several library modules are provided in source code form. These routines are used to perform low-level hardware-related I/O for the stream I/O functions. You can find the source for these routines in the **LIB** directory. You may modify these source files and substitute them for the library routines. By using these routines, you can quickly adapt the library to perform (using any hardware I/O device available in your target) stream I/O. Refer to “Stream Input and Output” on page 187 for more information.

Standard Types

The C51 standard library contains definitions for a number of standard types which may be used by the library routines. These standard types are declared in include files which you may access from your C programs.

jmp_buf

The **jmp_buf** type is defined in **SETJMP.H** and specifies the buffer used by the **setjmp** and **longjmp** routines to save and restore the program environment. The **jmp_buf** type is defined as follows:

```
#define _JBLEN   7
typedef char jmp_buf[_JBLEN];
```

va_list

The **va_list** array type is defined in **STDARG.H**. This type holds data required by the **va_arg** and **va_end** routines. The **va_list** type is defined as follows:

```
typedef char *va_list;
```

Absolute Memory Access Macros

The C51 standard library contains definitions for a number of macros that allow you to access explicit memory addresses. These macros are defined in **ABSACC.H**. Each of these macros is defined to be used like an array.

CBYTE

The **CBYTE** macro allows you to access individual bytes in the program memory of the 8051 and is defined as follows:

```
#define CBYTE ((unsigned char volatile code *)0)
```

You may use this macro in your programs as follows:

```
rval = CBYTE [0x0002];
```

to read the contents of the byte in program memory at address 0002h.

CWORD

The **CWORD** macro allows you to access individual words in the program memory of the 8051 and is defined as follows:

```
#define CWORD ((unsigned int volatile code *) 0)
```

You may use this macro in your programs as follows:

```
rval = CWORD [0x0002];
```

to read the contents of the word in program memory at address 0004h (2 × **sizeof** (**unsigned int**) = 4).

NOTE
The index used with this macro does not represent the memory address of the integer value. To obtain the memory address, you must multiply the index by the sizeof an integer (2 bytes).

DBYTE

The **DBYTE** macro allows you to access individual bytes in the internal data memory of the 8051 and is defined as follows:

```
#define DBYTE ((unsigned char volatile idata *) 0)
```

You may use this macro in your programs as follows:

```
rval = DBYTE [0x0002];
DBYTE [0x0002] = 5;
```

to read or write the contents of the byte in internal data memory at address 0002h.

DWORD

The **DWORD** macro allows you to access individual words in the internal data memory of the 8051 and is defined as follows:

```
#define DWORD ((unsigned int volatile idata *) 0)
```

You may use this macro in your programs as follows:

```
rval = DWORD [0x0002];
DWORD [0x0002] = 57;
```

to read or write the contents of the word in internal data memory at address 0004h (2 × **sizeof** (**unsigned int**) = 4).

NOTE
The index used with this macro does not represent the memory address of the integer value. To obtain the memory address, you must multiply the index by the sizeof an integer (2 bytes).

PBYTE

The **PBYTE** macro allows you to access individual bytes in one page of the external data memory of the 8051 and is defined as follows:

```
#define PBYTE ((unsigned char volatile pdata *) 0)
```

You may use this macro in your programs as follows:

```
rval = PBYTE [0x0002];
PBYTE [0x0002] = 38;
```

to read or write the contents of the byte in **pdata** memory at address 0002h.

PWORD

The **PWORD** macro allows you to access individual words in one page of the external data memory of the 8051 and is defined as follows:

```
#define PWORD ((unsigned int volatile pdata*) 0)
```

You may use this macro in your programs as follows:

```
rval = PWORD [0x0002];
PWORD [0x0002] = 57;
```

to read or write the contents of the word in **pdata** memory at address 0004h (2 × **sizeof** (**unsigned int**) = 4).

NOTE
The index used with this macro does not represent the memory address of the integer value. To obtain the memory address, you must multiply the index by the sizeof an integer (2 bytes).

XBYTE

The **XBYTE** macro allows you to access individual bytes in the external data memory of the 8051 and is defined as follows:

```
#define XBYTE ((unsigned char volatile xdata*) 0)
```

You may use this macro in your programs as follows:

```
rval = XBYTE [0x0002];
XBYTE [0x0002] = 57;
```

to read or write the contents of the byte in external data memory at address 0002h.

XWORD

The **XWORD** macro allows you to access individual words in the external data memory of the 8051 and is defined as follows:

```
#define XWORD ((unsigned int volatile xdata*) 0)
```

You may use this macro in your programs as follows:

```
rval = XWORD [2];
XWORD [2] = 57;
```

to read or write the contents of the word in external data memory at address 0004h (2 × **sizeof** (**unsigned int**) = 4).

NOTE
The index used with this macro does not represent the memory address of the integer value. To obtain the memory address, you must multiply the index by the sizeof an integer (2 bytes).

Routines by Category

This sections gives a brief overview of the major categories of routines available in the C51 standard library. Refer to "Reference" on page 195 for a complete description of routine syntax and usage.

NOTE
Many of the routines in the C51 standard library are reentrant, intrinsic, or both. These specifications are listed under attributes in the following tables. Unless otherwise noted, routines are non-reentrant and non-intrinsic.

Buffer Manipulation

Routine	Attributes	Description
memccpy		Copies data bytes from one buffer to another until a specified character or specified number of characters has been copied.
memchr	reentrant	Returns a pointer to the first occurrence of a specified character in a buffer.
memcmp	reentrant	Compares a given number of characters from two different buffers.
memcpy	reentrant	Copies a specified number of data bytes from one buffer to another.
memmove	reentrant	Copies a specified number of data bytes from one buffer to another.
memset	reentrant	Initializes a specified number of data bytes in a buffer to a specified character value.

The buffer manipulation routines are used to work on memory buffers on a character-by-character basis. A buffer is an array of characters like a string, however, a buffer is usually not terminated with a null character (**'\0'**). For this reason, these routines require a buffer length or count argument.

All of these routines are implemented as functions. Function prototypes are included in the **STRING.H** include file.

Character Conversion and Classification

Routine	Attributes	Description
isalnum	reentrant	Tests for an alphanumeric character.
isalpha	reentrant	Tests for an alphabetic character.
iscntrl	reentrant	Tests for a Control character.
isdigit	reentrant	Tests for a decimal digit.
isgraph	reentrant	Tests for a printable character with the exception of space.
islower	reentrant	Tests for a lowercase alphabetic character.
isprint	reentrant	Tests for a printable character.
ispunct	reentrant	Tests for a punctuation character.
isspace	reentrant	Tests for a whitespace character.
isupper	reentrant	Tests for an uppercase alphabetic character.
isxdigit	reentrant	Tests for a hexadecimal digit.
toascii	reentrant	Converts a character to an ASCII code.
toint	reentrant	Converts a hexadecimal digit to a decimal value.
tolower	reentrant	Tests a character and converts it to lowercase if it is uppercase.
_tolower	reentrant	Unconditionally converts a character to lowercase.
toupper	reentrant	Tests a character and converts it to uppercase if it is lowercase.
_toupper	reentrant	Unconditionally converts a character to uppercase.

The character conversion and classification routines allow you to test individual characters for a variety of attributes and convert characters to different formats.

The **_tolower**, **_toupper**, and **toascii** routines are implemented as macros. All other routines are implemented as functions. All macro definitions and function prototypes are found in the **CTYPE.H** include file.

Data Conversion

Routine	Attributes	Description
abs	reentrant	Generates the absolute value of an integer type.
atof / **atof517**		Converts a string to a float.
atoi		Converts a string to an int.
atol		Converts a string to a long.
cabs	reentrant	Generates the absolute value of a character type.
labs	reentrant	Generates the absolute value of a long type.

The data conversion routines convert strings of ASCII characters to numbers. All of these routines are implemented as functions and most are prototyped in the include file **STDLIB.H**. The **abs**, **cabs**, and **labs** functions are prototyped in the **MATH.H** include file. The **atof517** function is prototyped in the include file **80C517.H.**

Math

Routine	Attributes	Description
acos / **acos517**		Calculates the arc cosine of a specified number.
asin / **asin517**		Calculates the arc sine of a specified number.
atan / **atan517**		Calculates the arc tangent of a specified number.
atan2		Calculates the arc tangent of a fraction.
ceil		Finds the integer ceiling of a specified number.
cos / **cos517**		Calculates the cosine of a specified number.
cosh		Calculates the hyperbolic cosine of a specified number.
exp / **exp517**		Calculates the exponential function of a specified number.
fabs	reentrant	Finds the absolute value of a specified number.
floor		Finds the largest integer less than or equal to a specified number.
log / **log517**		Calculates the natural logarithm of a specified number.
log10 / **log10517**		Calculates the common logarithm of a specified number.
modf		Generates integer and fractional components of a specified number.
pow		Calculates a value raised to a power.
rand	reentrant	Generates a pseudo random number.
sin / **sin517**		Calculates the sine of a specified number.
sinh		Calculates the hyperbolic sine of a specified number.
srand		Initializes the pseudo random number generator.

Routine	Attributes	Description
sqrt / sqrt517		Calculates the square root of a specified number.
tan / tan517		Calculates the tangent of a specified number.
tanh		Calculates the hyperbolic tangent of a specified number.
chkfloat	intrinsic, reentrant	Checks the status of float numbers.
crol	intrinsic, reentrant	Rotates an unsigned char left a specified number of bits.
cror	intrinsic, reentrant	Rotates an unsigned char right a specified number of bits.
irol	intrinsic, reentrant	Rotates an unsigned int left a specified number of bits.
iror	intrinsic, reentrant	Rotates an unsigned int right a specified number of bits.
lrol	intrinsic, reentrant	Rotates an unsigned long left a specified number of bits.
lror	intrinsic, reentrant	Rotates an unsigned long right a specified number of bits.

The math routines perform common mathematical calculations. Most of these routines work with floating-point values and therefore include the floating-point libraries and support routines.

All of these routines are implemented as functions. Most are prototyped in the include file **MATH.H**. Functions which end in 517 (**acos517**, **asin517**, **atan517**, **cos517**, **exp517**, **log517**, **log10517**, **sin517**, **sqrt517**, and **tan517**) are prototyped in the **80C517.H** include file. The **rand** and **srand** functions are prototyped in the **STDLIB.H** include file.

The **_chkfloat_**, **_crol_**, **_cror_**, **_irol_**, **_iror_**, **_lrol_**, and **_lror_** functions are prototyped in the **INTRINS.H** include file.

Memory Allocation

Routine	Attributes	Description
calloc		Allocates storage for an array from the memory pool.
free		Frees a memory block that was allocated using calloc, malloc, or realloc.
init_mempool		Initializes the memory location and size of the memory pool.
malloc		Allocates a block from the memory pool.
realloc		Reallocates a block from the memory pool.

The memory allocation functions provide you with a means to specify, allocate, and free blocks of memory from a memory pool. All memory allocation functions are implemented as functions and are prototyped in the **STDLIB.H** include file.

Before using any of these functions to allocate memory, you must first specify, using the **init_mempool** routine, the location and size of a memory pool from which subsequent memory requests are satisfied.

The **calloc** and **malloc** routines allocate blocks of memory from the pool. The **calloc** routine allocates an array with a specified number of elements of a given size and initializes the array to 0. The **malloc** routine allocates a specified number of bytes.

The **realloc** routine changes the size of an allocated block, while the **free** routine returns a previously allocated memory block to the memory pool.

Stream Input and Output

Routine	Attributes	Description
getchar	reentrant	Reads and echoes a character using the _getkey and putchar routines.
_getkey		Reads a character using the 8051 serial interface.
gets		Reads and echoes a character string using the getchar routine.
printf / **printf517**		Writes formatted data using the putchar routine.
putchar		Writes a character using the 8051 serial interface.
puts	reentrant	Writes a character string and newline ('\n') character using the putchar routine.
scanf / **scanf517**		Reads formatted data using the getchar routine.
sprintf / **sprintf517**		Writes formatted data to a string.
sscanf / **sscanf517**		Reads formatted data from a string.
ungetchar		Places a character back into the getchar input buffer.
vprintf		Writes formatted data using the putchar function.
vsprintf		Writes formatted data to a string.

The stream input and output routines allow you to read and write data to and from the 8051 serial interface or a user-defined I/O interface. The default **_getkey** and **putchar** functions found in the C51 library read and write characters using the 8051 serial interface. You can find the source for these functions in the **LIB** directory. You may modify these source files and substitute them for the library routines. When this is done, other stream functions then perform input and output using the new **_getkey** and **putchar** routines.

If you want to use the existing **_getkey** and **putchar** functions, you must first initialize the 8051 serial port. If the serial port is not properly initialized, the default stream functions do not function. Initializing the serial port requires manipulating special function registers SFRs of the 8051. The include file **REG51.H** contains definitions for the required SFRs.

The following example code must be executed immediately after reset, before any stream functions are invoked.

```
.
.
.
#include <reg51.h>
.
.
.
SCON = 0x50;          /* Setup serial port control register */
                      /* Mode 1: 8-bit uart var. baud rate */
                      /* REN: enable receiver */

PCON &= 0x7F;         /* Clear SMOD bit in power ctrl reg */
                      /* This bit doubles the baud rate */

TMOD &= 0xCF          /* Setup timer/counter mode register */
                      /* Clear M1 and M0 for timer 1 */
TMOD |= 0x20;         /* Set M1 for 8-bit autoreload timer */

TH1 = 0xFD;           /* Set autoreload value for timer 1 */
                      /* 9600 baud with 11.0592 MHz xtal */

TR1 = 1;              /* Start timer 1 */

TI = 1;               /* Set TI to indicate ready to xmit */
.
.
.
```

The stream routines treat input and output as streams of individual characters. There are routines that process characters as well as functions that process strings. Choose the routines that best suit your requirements.

All of these routines are implemented as functions. Most are prototyped in the **STDIO.H** include file. The **printf517**, **scanf517**, **sprintf517**, and **sscanf517** functions are prototyped in the **80C517.H** include file.

String Manipulation

Routine	Attributes	Description
strcat		Concatenates two strings.
strchr	reentrant	Returns a pointer to the first occurrence of a specified character in a string.
strcmp	reentrant	Compares two strings.
strcpy	reentrant	Copies one string to another.
strcspn		Returns the index of the first character in a string that matches any character in a second string.
strlen	reentrant	Returns the length of a string.
strncat		Concatenates up to a specified number of characters from one string to another.
strncmp		Compares two strings up to a specified number of characters.
strncpy		Copies up to a specified number of characters from one string to another.
strpbrk		Returns a pointer to the first character in a string that matches any character in a second string.
strpos	reentrant	Returns the index of the first occurrence of a specified character in a string.
strrchr	reentrant	Returns a pointer to the last occurrence of a specified character in a string.
strrpbrk		Returns a pointer to the last character in a string that matches any character in a second string.
strrpos	reentrant	Returns the index of the last occurrence of a specified character in a string.
strspn		Returns the index of the first character in a string that does not match any character in a second string.

The string routines are implemented as functions and are prototyped in the **STRING.H** include file. They perform the following operations:

- Copying strings
- Appending one string to the end of another
- Comparing two strings
- Locating one or more characters from a specified set in a string

All string functions operate on null-terminated character strings. To work on non-terminated strings, use the buffer manipulation routines described earlier in this section.

Variable-length Argument Lists

Routine	Attributes	Description
va_arg	reentrant	Retrieves an argument from an argument list.
va_end	reentrant	Resets an argument pointer.
va_start	reentrant	Sets a pointer to the beginning of an argument list.

The variable-length argument list routines are implemented as macros and are defined in the **STDARG.H** include file. These routines provide you with a portable method of accessing arguments in a function that takes a variable number of arguments. These macros conform to the ANSI C Standard for variable-length argument lists.

Miscellaneous

Routine	Attributes	Description
setjmp	reentrant	Saves the current stack condition and program address.
longjmp	reentrant	Restores the stack condition and program address.
nop	intrinsic, reentrant	Inserts an 8051 NOP instruction.
testbit	intrinsic, reentrant	Tests the value of a bit and clears it to 0.

Routines found in the miscellaneous category do not fit easily into any other library routine category. The **setjmp** and **longjmp** routines are implemented as functions and are prototyped in the **STDJMP.H** include file.

The **_nop_** and **_testbit_** routines are used to direct the compiler to generate a **NOP** instruction and a **JBC** instruction respectively. These routines are prototyped in the **INTRINS.H** include file.

Include Files

The include files that are provided with the C51 standard library are found in the **INC** subdirectory. These files contain constant and macro definitions, type definitions, and function prototypes. The following sections describe the use and contents of each include file. Macros and functions included in the file are listed as well.

8051 Special Function Register Include Files

The C51 library provides you with a number of include files that define manifest constants for the special function registers found on many 8051 derivatives. These files are listed below:

REG151S.H
REG152.H
REG320.H
REG410.H
REG451.H
REG452.H
REG509.H
REG51.H
REG515.H
REG515A.H
REG515C.H
REG517.H
REG517A.H
REG51F.H
REG51G.H
REG51GB.H
REG52.H
REG552.H
REG592.H
REG781.H

80C517.H

The **80C517.H** include file contains routines that use the enhanced operational features of the 80C517 CPU and its derivatives. These routines are:

acos517
asin517
atan517
atof517
cos517
exp517
log10517
log517
printf517
scanf517
sin517
sprintf517
sqrt517
sscanf517
tan517

ABSACC.H

The **ABSACC.H** include file contains definitions for macros that allow you to directly access the different memory areas of the 8051.

CBYTE
CWORD
DBYTE
DWORD
PBYTE
PWORD
XBYTE
XWORD

ASSERT.H

The **ASSERT.H** include file defines the **assert** macro you can use to create test conditions in your programs.

CTYPE.H

The **CTYPE.H** include file contains definitions and prototypes for routines which classify ASCII characters and routines which perform character conversions. The following is a list of these routines:

isalnum
isalpha
iscntrl
isdigit
isgraph
islower
isprint
ispunct
isspace
isupper
isxdigit
toascii
toint
tolower
_tolower
toupper
_toupper

INTRINS.H

The **INTRINS.H** include file contains prototypes for routines that instruct the compiler to generate in-line intrinsic code.

chkfloat
crol
cror
irol
iror
lrol
lror
nop
testbit

MATH.H

The **MATH.H** include file contains prototypes and definitions for all routines that perform floating-point math calculations. Other math functions are also included in this file. All of the math function routines are listed below:

abs
acos
asin
atan
atan2
cabs
ceil
cos
cosh
exp
fabs
floor
fprestore
fpsave
labs
log
log10
modf
pow
sin
sinh
sqrt
tan
tanh

SETJMP.H

The **SETJMP.H** include file defines the **jmp_buf** type and prototypes the **setjmp** and **longjmp** routines which use it.

STDARG.H

The **STDARG.H** include file defines macros that allow you to access arguments in functions with variable-length argument lists. The macros include:

va_arg	**va_end**	**va_start**

In addition, the **va_list** type is defined in this file.

STDDEF.H

The **STDDEF.H** include file defines the **offsetof** macro you can use to determine the offset of members of a structure.

STDIO.H

The **STDIO.H** include file contains prototypes and definitions for stream I/O routines. They are:

getchar	**putchar**	**sscanf**
_getkey	**puts**	**ungetchar**
gets	**scanf**	**vprintf**
printf	**sprintf**	**vsprintf**

The **STDIO.H** include file also defines the **EOF** manifest constant.

STDLIB.H

The **STDLIB.H** include file contains prototypes and definitions for the type conversion and memory allocation routines listed below:

atof	**free**	**realloc**
atoi	**init_mempool**	**srand**
atol	**malloc**	
calloc	**rand**	

The **STDLIB.H** include file also defines the **NULL** manifest constant.

STRING.H

The **STRING.H** include file contains prototypes for the following string and buffer manipulation routines:

memccpy	**strchr**	**strncpy**
memchr	**strcmp**	**strpbrk**
memcmp	**strcpy**	**strpos**
memcpy	**strcspn**	**strrchr**
memmove	**strlen**	**strrpbrk**
memset	**strncat**	**strrpos**
strcat	**strncmp**	**strspn**

The **STRING.H** include file also defines the **NULL** manifest constant.

Reference

The following pages constitute the C51 standard library reference. The routines included in the standard library are described here in alphabetical order and each is divided into several sections:

Summary: Briefly describes the routine's effect, lists include file(s) containing its declaration and prototype, illustrates the syntax, and describes any arguments.

Description: Provides you with a detailed description of the routine and how it is used.

Return Value: Describes the value returned by the routine.

See Also: Names related routines.

Example: Gives a function or program fragment demonstrating proper use of the function.

abs

Summary:

#include <math.h>
int abs (
    int *val*); /* number to take absolute value of */

Description: The **abs** function determines the absolute value of the integer argument *val*.

Return Value: The **abs** function returns the absolute value of *val*.

See Also: **cabs, fabs, labs**

Example:

```
#include <math.h>
#include <stdio.h>                      /* for printf */

void tst_abs (void)  {
  int x;
  int y;

  x = -42;

  y = abs (x);

  printf ("ABS(%d) = %d\n", x, y);
}
```

acos / acos517

Summary:

```
#include <math.h>
float acos (
    float x);                    /* number to calculate arc
cosine of */
```

Description: The **acos** function calculates the arc cosine of the floating-point number *x*. The value of *x* must be between -1 and 1. The floating-point value returned by **acos** is a number in the 0 to π range.

The **acos517** function is identical to **acos**, but uses the arithmetic unit of the Siemens 80C517 to provide faster execution. For using this function include the header file 80C517.H. Do not use this routine with a CPU that does not support this feature.

Return Value: The **acos** function returns the arc cosine of *x*.

See Also: **asin, atan, atan2**

Example:

```
#include <math.h>
#include <stdio.h>                      /* for printf */

void tst_acos (void)  {
  float x;
  float y;

  for (x = -1.0; x <= 1.0; x += 0.1)  {
    y = acos (x);

    printf ("ACOS(%f) = %f\n", x, y);
  }
}
```

asin / asin517

Summary:

```
#include <math.h>
float asin (
    float x);                    /* number to calculate arc sine
of */
```

Description: The **asin** function calculates the arc sine of the floating-point number *x*. The value of *x* must be in the range -1 to 1. The floating-point value returned by **asin** is a number in the $-\pi/2$ to $\pi/2$ range.

The **asin517** function is identical to **asin**, but uses the arithmetic unit of the Siemens 80C517 to provide faster execution. For using this function include the header file 80C517.H. Do not use this routine with a CPU that does not support this feature.

Return Value: The **asin** function returns the arc sine of *x*.

See Also: **acos, atan, atan2**

Example:

```
#include <math.h>
#include <stdio.h>                      /* for printf */

void tst_asin (void)  {
  float x;
  float y;

  for (x = -1.0; x <= 1.0; x += 0.1)  {
    y = asin (x);

    printf ("ASIN(%f) = %f\n", x, y);
  }
}
```

assert

Summary:

#include <assert.h>
void assert (
expression);

Description: The **assert** macro tests *expression* and prints a diagnostic message using the **printf** library routine if it is false.

Return Value: None.

Example:

```
#include <assert.h>
#include <stdio.h>

void check_parms (
  char *string)
{
  assert (string != NULL);    /* check for NULL ptr */
  printf ("String %s is OK\n", string);
}
```

atan / atan517

Summary:

```
#include <math.h>
float atan (
    float x);                    /* number to calculate arc
tangent of */
```

Description: The **atan** function calculates the arc tangent of the floating-point number *x*. The floating-point value returned by **atan** is a number in the $-\pi/2$ to $\pi/2$ range.

The **atan517** function is identical to **atan**, but uses the arithmetic unit of the Siemens 80C517 to provide faster execution. For using this function include the header file 80C517.H. Do not use this routine with a CPU that does not support this feature.

Return Value: The **atan** function returns the arc tangent of *x*.

See Also: **acos, asin, atan2**

Example:

```
#include <math.h>
#include <stdio.h>                        /* for printf */

void tst_atan (void)  {
  float x;
  float y;

  for (x = -10.0; x <= 10.0; x += 0.1)  {
    y = atan (x);

    printf ("ATAN(%f) = %f\n", x, y);
  }
}
```

atan2

Summary:

```
#include <math.h>
float atan2 (
    float y,          /* denominator for arc tangent */
    float x);                 /* numerator for arc tangent */
```

Description: The **atan2** function calculates the arc tangent of the floating-point ratio y / x. This function uses the signs of both x and y to determine the quadrant of the return value. The floating-point value returned by **atan2** ia a number in the $-\pi$ to π range.

Return Value: The **atan2** function returns the arc tangent of y / x.

See Also: **acos, asin, atan**

Example:

```
#include <math.h>
#include <stdio.h>                          /* for printf */

void tst_atan2 ()  {
  float x;
  float y;
  float z;

  x = -1.0;

  for (y = -10.0; y < 10.0; y += 0.1)  {
    z = atan2 (y,x);

    printf ("ATAN2(%f/%f) = %f\n", y, x, z);
  }

  /* z approaches -pi as y goes from -10 to 0 */
  /* z approaches +pi as y goes from +10 to 0 */

}
```

atof / atof517

Summary:

```
#include <stdlib.h>
float atof (
    void *string);          /* string to convert */
```

Description: The **atof** function converts *string* into a floating-point value. The input *string* is a sequence of characters that can be interpreted as a floating-point number. This function stops processing characters from *string* at the first one it cannot recognize as part of the number.

The **atof517** function is identical to atof, but uses the arithmetic unit of the Siemens 80C517 to provide faster execution. For using this function include the header file 80C517.H. Do not use this routine with a CPU that does not support this feature.

The **atof** function requires *string* to have the following format:

$$[\{+ \mid -\}]\ digits\ [.\ digits]\ [\{e \mid E\}\ [\{+ \mid -\}]\ digits]$$

where:

digits may be one or more decimal digits.

Return Value: The **atof** function returns the floating-point value that is produced by interpreting the characters in *string* as a number.

See Also: **atoi, atol**

Example:

```
#include <stdlib.h>
#include <stdio.h>                      /* for printf */

void tst_atof (void)  {
  float f;
  char s [] = "1.23";

  f = atof (s);
  printf ("ATOF(%s) = %f\n", s, f);
}
```

atoi

Summary:

```
#include <stdlib.h>
int atoi (
    void *string);          /* string to convert */
```

Description: The **atoi** function converts *string* into an integer value. The input *string* is a sequence of characters that can be interpreted as an integer. This function stops processing characters from *string* at the first one it cannot recognize as part of the number.

The atoi function requires *string* to have the following format:

$[\![whitespace]\!]\ [\![\{+ \mid -\}]\!]\ digits$

where:

digits may be one or more decimal digits.

Return Value: The **atoi** function returns the integer value that is produced by interpreting the characters in *string* as a number.

See Also: **atof, atol**

Example:

```
#include <stdlib.h>
#include <stdio.h>                      /* for printf */

void tst_atoi (void)  {
  int i;
  char s [] = "12345";

  i = atoi (s);
  printf ("ATOI(%s) = %d\n", s, i);
}
```

atol

Summary:

```
#include <stdlib.h>
long atol (
    void *string);          /* string to convert */
```

Description: The **atol** function converts *string* into a long integer value. The input *string* is a sequence of characters that can be interpreted as a long. This function stops processing characters from *string* at the first one it cannot recognize as part of the number.

The **atol** function requires *string* to have the following format:

$$[whitespace]\,[\{+\mid -\}]\,digits$$

where:

digits may be one or more decimal digits.

Return Value: The **atol** function returns the long integer value that is produced by interpreting the characters in *string* as a number.

See Also: **atof, atoi**

Example:

```
#include <stdlib.h>
#include <stdio.h>                      /* for printf */

void tst_atol (void)  {
  long l;
  char s [] = "8003488051";

  l = atol (s);
  printf ("ATOL(%s) = %ld\n", s, l);

}
```

cabs

Summary:

```
#include <math.h>
char cabs (
    char val);        /* character to take absolute value of */
```

Description: The **cabs** function determines the absolute value of the character argument *val*.

Return Value: The **cabs** function returns the absolute value of *val*.

See Also: **abs, fabs, labs**

Example:

```
#include <math.h>
#include <stdio.h>                          /* for printf */

void tst_cabs (void)  {
  char x;
  char y;

  x = -23;

  y = cabs (x);

  printf ("CABS(%bd) = %bd\n", x, y);
}
```

calloc

Summary:

```
#include <stdlib.h>
void *calloc (
    unsigned int num,       /* number of items */
    unsigned int len);      /* length of each item */
```

Description: The **calloc** function allocates memory for an array of *num* elements. Each element in the array occupies *len* bytes and is initialized to 0. The total number of bytes of memory allocated is *num* × *len*.

NOTE
Source code is provided for this routine in the ***LIB*** *directory. You can modify the source to customize this function for your hardware environment.* *Refer to "Chapter 6. Advanced Programming Techniques" on page 113 for more information.*

Return Value: The **calloc** function returns a pointer to the allocated memory or a null pointer if the memory allocation request cannot be satisfied.

See Also: **free, init_mempool, malloc, realloc**

Example:

```
#include <stdlib.h>
#include <stdio.h>                      /* for printf */

void tst_calloc (void)  {
  int xdata *p;           /* ptr to array of 100 ints */

  p = calloc (100, sizeof (int));

  if (p == NULL)
  printf ("Error allocating array\n");
  else
    printf ("Array address is %p\n", (void *) p);

}
```

ceil

Summary:

```
#include <math.h>
float ceil (
    float val);        /* number to calculate ceiling for */
```

Description: The **ceil** function calculates the smallest integer value that is greater than or equal to *val*.

Return Value: The **ceil** function returns a **float** that contains the smallest integer value that is not less than *val*.

See Also: **floor**

Example:

```
#include <math.h>
#include <stdio.h>                        /* for printf */

void tst_ceil (void)  {
  float x;
  float y;

  x = 45.998;
  y = ceil (x);

  printf ("CEIL(%f) = %f\n", x, y);

  /* output is "CEIL(45.998) = 46" */

}
```

chkfloat

Summary:

```
#include <intrins.h>
unsigned char _chkfloat_ (
    float val);        /* number for error checking */
```

Description: The **_chkfloat_** function checks the status of a floating-point number.

Return Value: The **_chkfloat_** function returns an **unsigned char** that contains the following status information:

Return Value	Meaning
0	Standard floating-point numbers
1	Floating-point value 0
2	**+INF** (positive overflow)
3	**-INF** (negative overflow)
4	**NaN** (Not a Number) error status

Example:

```
#include <intrins.h>
#include <stdio.h>                        /* for printf */

char _chkfloat_ (float);

float f1, f2, f3;

void tst_chkfloat (void)  {
  f1 = f2 * f3;

  switch (_chkfloat_ (f1))  {
    case 0:
      printf ("result is a number\n"); break;
    case 1:
      printf ("result is zero\n");     break;
    case 2:
      printf ("result is +INF\n");     break;
    case 3:
      printf ("result is -INF\n");     break;
    case 4:
      printf ("result is NaN\n");      break;
  }
}
```

cos / cos517

Summary:

```
#include <math.h>
float cos (
    float x);                    /* number to calculate cosine
for */
```

Description: The **cos** function calculates the cosine of the floating-point value *x*. The value of *x* must be between -65535 and 65535. Values outside this range result in an **NaN** error.

The **cos517** function is identical to **cos**, but uses the arithmetic unit of the Siemens 80C517 to provide faster execution. For using this function include the header file 80C517.H. Do not use this routine with a CPU that does not support this feature.

Return Value: The **cos** function returns the cosine for the value *x*.

See Also: **sin, tan**

Example:

```
#include <math.h>
#include <stdio.h>                        /* for printf */

void tst_cos (void)  {
  float x;
  float y;

  for (x = 0; x < (2 * 3.1415); x += 0.1)  {
    y = cos (x);

    printf ("COS(%f) = %f\n", x, y);
  }
}
```

cosh

Summary:

#include <math.h>
float cosh (
    float *x*); /* value for hyperbolic cos function */

Description: The **cosh** function calculates the hyperbolic cosine of the floating-point value *x*.

Return Value: The **cosh** function returns the hyperbolic cosine for the value *x*.

See Also: **sinh, tanh**

Example:

```
#include <math.h>
#include <stdio.h>                      /* for printf */

void tst_cosh (void)  {
  float x;
  float y;

  for (x = 0; x < (2 * 3.1415); x += 0.1)  {
    y = cosh (x);

    printf ("COSH(%f) = %f\n", x, y);
  }
}
```

crol

Summary:

```
#include <intrins.h>
unsigned char _crol_ (
    unsigned char c,        /* character to rotate left */
    unsigned char b);       /* bit positions to rotate */
```

Description: The **_crol_** routine rotates the bit pattern for the character *c* left *b* bits. This routine is implemented as an intrinsic function. The code required is included in-line rather than being called.

Return Value: The **_crol_** routine returns the rotated value of *c*.

See Also: **_cror_, _irol_, _iror_, _lrol_, _lror_**

Example:

```
#include <intrins.h>

void tst_crol (void)  {
  char a;
  char b;

  a = 0xA5;

  b = _crol_(a,3);                  /* b now is 0x2D */

}
```

cror

Summary:

```
#include <intrins.h>
unsigned char _cror_ (
    unsigned char c,        /* character to rotate right */
    unsigned char b);       /* bit positions to rotate */
```

Description: The **_cror_** routine rotates the bit pattern for the character *c* right *b* bits. This routine is implemented as an intrinsic function. The code required is included in-line rather than being called.

Return Value: The **_cror_** routine returns the rotated value of *c*.

See Also: **_crol_, _irol_, _iror_, _lrol_, _lror_**

Example:

```
#include <intrins.h>

void tst_cror (void)  {
  char a;
  char b;

  a = 0xA5;

  b = _crol_(a,1);                  /* b now is 0xD2 */

}
```

exp / exp517

Summary:

```
#include <math.h>
float exp (
    float x);                  /* power to use for e^x function
*/
```

Description: The **exp** function calculates the exponential function for the floating-point value *x*.

The **exp517** function is identical to **exp**, but uses the arithmetic unit of the Siemens 80C517 to provide faster execution. For using this function include the header file 80C517.H. Do not use this routine with a CPU that does not support this feature.

Return Value: The **exp** function returns the floating-point value e^x.

See Also: **log, log10**

Example:

```
#include <math.h>
#include <stdio.h>                        /* for printf */

void tst_exp (void)  {
  float x;
  float y;

  x = 4.605170186;

  y = exp (x);                              /* y = 100 */

  printf ("EXP(%f) = %f\n", x, y);

}
```

fabs

Summary:

#include <math.h>
float fabs (
    float *val*); /* number to calc absolute value for */

Description: The **fabs** function determines the absolute value of the floating-point number *val*.

Return Value: The **fabs** function returns the absolute value of *val*.

See Also: **abs, cabs, labs**

Example:

```
#include <math.h>
#include <stdio.h>                          /* for printf */

void tst_fabs (void)  {
  float x;
  float y;

  x = 10.2;
  y = fabs (x);
  printf ("FABS(%f) = %f\n", x, y);

  x = -3.6;
  y = fabs (x);
  printf ("FABS(%f) = %f\n", x, y);

}
```

floor

Summary:

```
#include <math.h>
float floor (
    float val);        /* value for floor function */
```

Description: The **floor** function calculates the largest integer value that is less than or equal to *val*.

Return Value: The **floor** function returns a **float** that contains the largest integer value that is not greater than *val*.

See Also: **ceil**

Example:

```
#include <math.h>
#include <stdio.h>                        /* for printf */

void tst_floor (void)  {
  float x;
  float y;

  x = 45.998;

  y = floor (x);

  printf ("FLOOR(%f) = %f\n", x, y);     /* prints 45 */
}
```

free

Summary:

```
#include <stdlib.h>
void free (
    void xdata *p);          /* block to free */
```

Description: The **free** function returns a memory block to the memory pool. The *p* argument points to a memory block allocated with the calloc, malloc, or realloc functions. Once it has been returned to the memory pool by the free function, the block is available for subsequent allocation.

If *p* is a null pointer, it is ignored.

NOTE
Source code for this routine is located in the **\C51\LIB** *directory. You may modify the source to customize this function for your particular hardware environment. Refer to "Chapter 6. Advanced Programming Techniques" on page 113 for more information.*

Return Value: None.

See Also: **calloc, init_mempool, malloc, realloc**

Example:

```
#include <stdlib.h>
#include <stdio.h>                      /* for printf */

void tst_free (void)  {
  void *mbuf;

  printf ("Allocating memory\n");
  mbuf = malloc (1000);

  if (mbuf == NULL)  {
    printf ("Unable to allocate memory\n");
  }
  else  {
      free (mbuf);
      printf ("Memory free\n");
  }

}
```

getchar

Summary:

```
#include <stdio.h>
char getchar (void);
```

Description: The **getchar** function reads a single character from the input stream using the **_getkey** function. The character read is then passed to the **putchar** function to be echoed.

NOTE
*This function is implementation-specific and is based on the operation of the **_getkey** and/or **putchar** functions. These functions, as provided in the standard library, read and write characters using the serial port of the 8051. Custom functions may use other I/O devices.*

Return Value: The **getchar** function returns the character read.

See Also: **_getkey, putchar, ungetchar**

Example:

```
#include <stdio.h>

void tst_getchar (void)  {
  char c;

  while ((c = getchar ()) != 0x1B)  {
    printf ("character = %c %bu %bx\n", c, c, c);
  }

}
```

_getkey

Summary:

```
#include <stdio.h>
    char _getkey (void);
```

Description: The **_getkey** function waits for a character to be received from the serial port.

NOTE
*This routine is implementation-specific, and its function may deviate from that described above. Source is included for the **_getkey** and **putchar** functions which may be modified to provide character level I/O for any hardware device. Refer to "Customization Files" on page 113 for more information.*

Return Value: The **_getkey** routine returns the received character.

See Also: **getchar, putchar, ungetchar**

Example:

```
#include <stdio.h>

void tst_getkey (void)  {
  char c;

  while ((c = _getkey ()) != 0x1B)  {
    printf ("key = %c %bu %bx\n", c, c, c);
  }

}
```

gets

Summary:

```
#include <stdio.h>
char *gets (
    char *string,    /* string to read */
    int len);                /* maximum characters to read
*/
```

Description: The **gets** function calls the **getchar** function to read a line of characters into *string*. The line consists of all characters up to and including the first newline character (**'\n'**). The newline character is replaced by a null character (**'\0'**) in *string*.

The *len* argument specifies the maximum number of characters that may be read. If *len* characters are read before a newline is encountered, the **gets** function terminates *string* with a null character and returns.

NOTE
*This function is implementation-specific and is based on the operation of the **_getkey** and/or **putchar** functions. These functions, as provided in the standard library, read and write characters using the serial port of the 8051. Custom functions may use other I/O devices.*

Return Value: The **gets** function returns *string*.

See Also: **printf, puts, scanf**

Example:

```
#include <stdio.h>

void tst_gets (void)  {
  xdata char buf [100];

  do  {
    gets (buf, sizeof (buf));
    printf ("Input string \"%s\"", buf);
  } while (buf [0] != '\0');
}
```

init_mempool

Summary:

```
#include <stdlib.h>
void init_mempool (
    void xdata *p,          /* start of memory pool */
    unsigned int size);     /* length of memory pool */
```

Description: The **init_mempool** function initializes the memory management routines and provides the starting address and size of the memory pool. The *p* argument points to a memory area in **xdata** which is managed using the **calloc**, **free**, **malloc**, and **realloc** library functions. The *size* argument specifies the number of bytes to use for the memory pool.

NOTE
*This function must be used to setup the memory pool before any other memory management functions (**calloc, free, malloc, realloc**) can be called. Call the **init_mempool** function only once at the beginning of your program.*

*Source code is provided for this routine in the **\LIB** directory. You can modify the source to customize this function for your hardware environment. Refer to "Chapter 6. Advanced Programming Techniques" on page 113 for more information.*

Return Value: None.

See Also: **calloc, free, malloc, realloc**

Example:

```
#include <stdlib.h>

void tst_init_mempool (void)  {
  xdata void *p;
  int i;

  init_mempool (&XBYTE [0x2000], 0x1000);
/* initialize memory pool at xdata 0x2000
   for 4096 bytes */

  p = malloc (100);
  for (i = 0; i < 100; i++) ((char *) p)[i] = i;
  free (p);

}
```

irol

Summary:

```
#include <intrins.h>
unsigned int _irol_ (
    unsigned int i,          /* integer to rotate left */
    unsigned char b);        /* bit positions to rotate */
```

Description: The **_irol_** routine rotates the bit pattern for the integer *i* left *b* bits. This routine is implemented as an intrinsic function. The code required is included in-line rather than being called.

Return Value: The **_irol_** routine returns the rotated value of *i*.

See Also: **_cror_, _crol_, _iror_, _lrol_, _lror_**

Example:

```
#include <intrins.h>

void tst_irol (void)  {
  int a;
  int b;

  a = 0xA5A5;

  b = _irol_(a,3);                /* b now is 0x2D2D */

}
```

iror

Summary:

```
#include <intrins.h>
unsigned int _iror_ (
    unsigned int i,          /* integer to rotate right */
    unsigned char b);        /* bit positions to rotate */
```

Description: The **_iror_** routine rotates the bit pattern for the integer *i* right *b* bits. This routine is implemented as an intrinsic function. The code required is included in-line rather than being called.

Return Value: The **_iror_** routine returns the rotated value of *i*.

See Also: **_cror_, _crol_, _irol_, _lrol_, _lror_**

Example:

```
#include <intrins.h>

void tst_iror (void)  {
  int a;
  int b;

  a = 0xA5A5;

  b = _irol_(a,1);                /* b now is 0xD2D2 */

}
```

isalnum

Summary:

```
#include <ctype.h>
bit isalnum (
    char c);                          /* character to test */
```

Description: The **isalnum** function tests *c* to determine if it is an alphanumeric character (**'A'-'Z'**, **'a'-'z'**, **'0'-'9'**).

Return Value: The **isalnum** function returns a value of 1 if *c* is an alphanumeric character or a value of 0 if it is not.

See Also: **isalpha, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit**

Example:

```
#include <ctype.h>
#include <stdio.h>                        /* for printf */

void tst_isalnum (void)  {
  unsigned char i;
  char *p;

  for (i = 0; i < 128; i++)  {
    p = (isalnum (i) ? "YES" : "NO");

    printf ("isalnum (%c) %s\n", i, p);
  }

}
```

isalpha

Summary:

```
#include <ctype.h>
bit isalpha (
    char c);                    /* character to test */
```

Description: The **isalpha** function tests *c* to determine if it is an alphabetic character (**'A'-'Z'** or **'a'-'z'**).

Return Value: The **isalpha** function returns a value of 1 if *c* is an alphabetic character and a value of 0 if it is not.

See Also: **isalnum, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit**

Example:

```
#include <ctype.h>
#include <stdio.h>                      /* for printf */

void tst_isalpha (void)  {
  unsigned char i;
  char *p;

  for (i = 0; i < 128; i++)  {
    p = (isalpha (i) ? "YES" : "NO");

    printf ("isalpha (%c) %s\n", i, p);
  }

}
```

iscntrl

Summary:

```
#include <ctype.h>
bit iscntrl (
    char c);                    /* character to test */
```

Description: The **iscntrl** function tests *c* to determine if it is a control character (0x00-0x1F or 0x7F).

Return Value: The **iscntrl** function returns a value of 1 if *c* is a control character and a value of 0 if it is not.

See Also: **isalnum, isalpha, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit**

Example:

```
#include <ctype.h>
#include <stdio.h>                      /* for printf */

void tst_iscntrl (void)  {
  unsigned char i;
  char *p;

  for (i = 0; i < 128; i++)  {
    p = (iscntrl (i) ? "YES" : "NO");

    printf ("iscntrl (%c) %s\n", i, p);
  }

}
```

isdigit

Summary:

```
#include <ctype.h>
bit isdigit (
    char c);                    /* character to test */
```

Description: The **isdigit** function tests *c* to determine if it is a decimal digit (**'0'-'9'**).

Return Value: The **isdigit** function returns a value of 1 if *c* is a decimal digit and a value of 0 if it is not.

See Also: **isalnum, isalpha, iscntrl, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit**

Example:

```
#include <ctype.h>
#include <stdio.h>                      /* for printf */

void tst_isdigit (void)  {
  unsigned char i;
  char *p;

  for (i = 0; i < 128; i++)  {
    p = (isdigit (i) ? "YES" : "NO");

    printf ("isdigit (%c) %s\n", i, p);
  }

}
```

isgraph

Summary:

```
#include <ctype.h>
bit isgraph (
    char c);                    /* character to test */
```

Description: The **isgraph** function tests *c* to determine if it is a printable character (not including space). The character values tested for are 0x21-0x7E.

Return Value: The **isgraph** function returns a value of 1 if *c* is a printable character and a value of 0 if it is not.

See Also: **isalnum, isalpha, iscntrl, isdigit, islower, isprint, ispunct, isspace, isupper, isxdigit**

Example:

```
#include <ctype.h>
#include <stdio.h>                      /* for printf */

void tst_isgraph (void)  {
  unsigned char i;
  char *p;

  for (i = 0; i < 128; i++)  {
    p = (isgraph (i) ? "YES" : "NO");

    printf ("isgraph (%c) %s\n", i, p);
  }

}
```

islower

Summary:

```
#include <ctype.h>
bit islower (
    char c);                    /* character to test */
```

Description: The **islower** function tests *c* to determine if it is a lowercase alphabetic character (**'a'-'z'**).

Return Value: The **islower** function returns a value of 1 if *c* is a lowercase letter and a value of 0 if it is not.

See Also: **isalnum, isalpha, iscntrl, isdigit, isgraph, isprint, ispunct, isspace, isupper, isxdigit**

Example:

```
#include <ctype.h>
#include <stdio.h>                      /* for printf */

void tst_islower (void)  {
  unsigned char i;
  char *p;

  for (i = 0; i < 128; i++)  {
    p = (islower (i) ? "YES" : "NO");

    printf ("islower (%c) %s\n", i, p);
  }

}
```

isprint

Summary:

```
#include <ctype.h>
bit isprint (
    char c);                          /* character to test */
```

Description: The **isprint** function tests *c* to determine if it is a printable character (0x20-0x7E).

Return Value: The **isprint** function returns a value of 1 if *c* is a printable character and a value of 0 if it is not.

See Also: **isalnum, isalpha, iscntrl, isdigit, isgraph, islower, ispunct, isspace, isupper, isxdigit**

Example:

```
#include <ctype.h>
#include <stdio.h>                        /* for printf */

void tst_isprint (void)  {
  unsigned char i;
  char *p;

  for (i = 0; i < 128; i++)  {
    p = (isprint (i) ? "YES" : "NO");

    printf ("isprint (%c) %s\n", i, p);
  }

}
```

ispunct

Summary:

```
#include <ctype.h>
bit ispunct (
    char c);                    /* character to test */
```

Description: The **ispunct** function tests *c* to determine if it is a punctuation character. The following symbols are punctuation characters:

```
!    "    #    $    %    &    '    (
)    *    +    ,    -    .    /    :
;    <    =    >    ?    @    [    \
]    ^    _    `    {    |    }    ~
```

Return Value: The **ispunct** function returns a value of 1 if *c* is a punctuation character and a value of 0 if it is not.

See Also: **isalnum, isalpha, iscntrl, isdigit, isgraph, islower, isprint, isspace, isupper, isxdigit**

Example:

```
#include <ctype.h>
#include <stdio.h>                      /* for printf */

void tst_ispunct (void)  {
  unsigned char i;
  char *p;

  for (i = 0; i < 128; i++)  {
    p = (ispunct (i) ? "YES" : "NO");

    printf ("ispunct (%c) %s\n", i, p);
  }

}
```

isspace

Summary:

```
#include <ctype.h>
bit isspace (
    char c);                          /* character to test */
```

Description: The **isspace** function tests *c* to determine if it is a whitespace character (0x09-0x0D or 0x20).

Return Value: The **isspace** function returns a value of 1 if *c* is a whitespace character and a value of 0 if it is not.

See Also: **isalnum, isalpha, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isupper, isxdigit**

Example:

```
#include <ctype.h>
#include <stdio.h>                        /* for printf */

void tst_isspace (void)  {
  unsigned char i;
  char *p;

  for (i = 0; i < 128; i++)  {
    p = (isspace (i) ? "YES" : "NO");

    printf ("isspace (%c) %s\n", i, p);
  }

}
```

isupper

Summary:

```
#include <ctype.h>
bit isupper (
    char c);                    /* character to test */
```

Description: The **isupper** function tests *c* to determine if it is an uppercase alphabetic character (**'A'-'Z'**).

Return Value: The **isupper** function returns a value of 1 if *c* is an uppercase character and a value of 0 if it is not.

See Also: **isalnum, isalpha, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isxdigit**

Example:

```
#include <ctype.h>
#include <stdio.h>                      /* for printf */

void tst_isupper (void)  {
  unsigned char i;
  char *p;

  for (i = 0; i < 128; i++)  {
    p = (isupper (i) ? "YES" : "NO");

    printf ("isupper (%c) %s\n", i, p);
  }

}
```

isxdigit

Summary:

```
#include <ctype.h>
bit isxdigit (
    char c);                    /* character to test */
```

Description: The **isxdigit** function tests *c* to determine if it is a hexadecimal digit (**'A'-'Z'**, **'a'-'z'**, **'0'-'9'**).

Return Value: The **isxdigit** function returns a value of 1 if *c* is a hexadecimal digit and a value of 0 if it is not.

See Also: **isalnum, isalpha, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper**

Example:

```
#include <ctype.h>
#include <stdio.h>                      /* for printf */

void tst_isxdigit (void)  {
  unsigned char i;
  char *p;

  for (i = 0; i < 128; i++)  {
    p = (isxdigit (i) ? "YES" : "NO");

    printf ("isxdigit (%c) %s\n", i, p);
  }

}
```

labs

Summary:

```
#include <math.h>
long labs (
    long val);          /* value to calc. abs. value for */
```

Description: The **labs** function determines the absolute value of the long integer *val*.

Return Value: The **labs** function returns the absolute value of *val*.

See Also: **abs, cabs, fabs**

Example:

```
#include <math.h>
#include <stdio.h>                          /* for printf */

void tst_labs (void)  {
  long x;
  long y;

  x = -12345L;

  y = labs (x);

  printf ("LABS(%ld) = %ld\n", x, y);

}
```

log / log517

Summary:

```
#include <math.h>
float log (
    float val);       /* value to take natural logarithm of */
```

Description: The **log** function calculates the natural logarithm for the floating-point number *val*. The natural logarithm uses the base *e* or 2.718282…

The **log517** function is identical to **log**, but uses the arithmetic unit of the Siemens 80C517 to provide faster execution. When using this function, include the header file 80C517.H. Do not use this routine with a CPU that does not support this feature.

Return Value: The **log** function returns the floating-point natural logarithm of *val*.

See Also: **exp, log10**

Example:

```
#include <math.h>
#include <stdio.h>                        /* for printf */

void tst_log (void)  {
  float x;
  float y;

  x = 2.71838;
  x *= x;

  y = log (x);                                 /* y = 2 */

  printf ("LOG(%f) = %f\n", x, y);

}
```

log10 / log10517

Summary:

```
#include <math.h>
float log10 (
    float val);        /* value to take common logarithm of */
```

Description: The **log10** function calculates the common logarithm for the floating-point number *val*. The common logarithm uses base 10.

The **log10517** function is identical to log10, but uses the arithmetic unit of the Siemens 80C517 to provide faster execution. When using this function, include the header file 80C517.H. Do not use this routine with a CPU that does not support this feature.

Return Value: The **log10** function returns the floating-point common logarithm of *val*.

See Also: **exp, log**

Example:

```
#include <math.h>
#include <stdio.h>                        /* for printf */

void tst_log10 (void)  {
  float x;
  float y;

  x = 1000;

  y = log10 (x);                               /* y = 3 */

  printf ("LOG10(%f) = %f\n", x, y);

}
```

longjmp

Summary:

```
#include <setjmp.h>
void longjmp (
    jmp_buf env,    /* environment to restore */
    int retval);    /* return value */
```

Description: The **longjmp** function restores the state which was previously stored in *env* by the **setjmp** function. The *retval* argument specifies the value to return from the **setjmp** function call.

The **longjmp** and **setjmp** functions can be used to execute a non-local goto and are usually utilized to pass control to an error recovery routine.

Local variables and function arguments are restored only if declared with the **volatile** attribute.

Return Value: None.

See Also: **setjmp**

Example:

```
#include <setjmp.h>
#include <stdio.h>                          /* for printf */

jmp_buf env;     /* jump environment (must be global) */
bit error_flag;


void trigger (void)  {
  .
  .
  .
  /* put processing code here */
  .
  .
  .
  if (error_flag != 0)  {
    longjmp (env, 1);           /* return 1 to setjmp */
  }
  .
  .
  .
}


void recover (void)  {
            /* put recovery code here */
}


void tst_longjmp (void)  {
  .
  .
  .
  if (setjmp (env) != 0)  {     /* setjmp returns a 0 */
    printf ("LONGJMP called\n");
    recover ();
  }
  else  {
    printf ("SETJMP called\n");

    error_flag = 1;                 /* force an error */

    trigger ();
  }
}
```

lrol

Summary:

```
#include <intrins.h>
unsigned long _lrol_ (
    unsigned long l,        /* 32-bit integer to rotate left */
    unsigned char b);       /* bit positions to rotate */
```

Description: The **_lrol_** routine rotates the bit pattern for the long integer *l* left *b* bits. This routine is implemented as an intrinsic function. The code required is included in-line rather than being called.

Return Value: The **_lrol_** routine returns the rotated value of *l*.

See Also: **_cror_, _crol_, _irol_, _iror_, _lror_**

Example:

```
#include <intrins.h>

void tst_lrol (void)  {
  long a;
  long b;

  a = 0xA5A5A5A5;

  b = _lrol_(a,3);             /* b now is 0x2D2D2D2D */

}
```

lror

Summary:

```
#include <intrins.h>
unsigned long _lror_ (
    unsigned long l,        /* 32-bit int to rotate right */
    unsigned char b);       /* bit positions to rotate */
```

Description: The **_lror_** routine rotates the bit pattern for the long integer *l* right *b* bits. This routine is implemented as an intrinsic function. The code required is included in-line rather than being called.

Return Value: The **_lror_** routine returns the rotated value of *l*.

See Also: **_cror_, _crol_, _irol_, _iror_, _lrol_**

Example:

```
#include <intrins.h>

void tst_lror (void)  {
  long a;
  long b;

  a = 0xA5A5A5A5;

  b = _lrol_(a,1);             /* b now is 0xD2D2D2D2 */

}
```

malloc

Summary:

```
#include <stdlib.h>
void *malloc (
    unsigned int size);       /* block size to allocate */
```

Description: The **malloc** function allocates a memory block from the memory pool of *size* bytes in length.

> ***NOTE***
> *Source code is provided for this routine in the* ***LIB*** *directory. You may modify the source to customize this function for your hardware environment. Refer to "Chapter 6. Advanced Programming Techniques" on page 113 for more information.*

Return Value: The **malloc** function returns a pointer to the allocated block or a null pointer if there was not enough memory to satisfy the allocation request.

See Also: **calloc, free, init_mempool, realloc**

Example:

```
#include <stdlib.h>
#include <stdio.h>                      /* for printf */

void tst_malloc (void)  {
  unsigned char xdata *p;

  p = malloc (1000);           /* allocate 1000 bytes */

  if (p == NULL)
    printf ("Not enough memory space\n");
    else
      printf ("Memory allocated\n");

}
```

memccpy

Summary:

```
#include <string.h>
void *memccpy (
    void *dest,     /* destination buffer */
    void *src,      /* source buffer */
    char c,         /* character which ends copy */
    int len);               /* maximum bytes to copy */
```

Description: The **memccpy** function copies 0 or more characters from *src* to *dest*. Characters are copied until the character *c* is copied or until *len* bytes have been copied, whichever comes first.

Return Value: The **memccpy** function returns a pointer to the byte in *dest* that follows the last character copied or a null pointer if the last character copied was *c*.

See Also: **memchr, memcmp, memcpy, memmove, memset**

Example:

```
#include <string.h>
#include <stdio.h>                      /* for printf */

void tst_memccpy (void)  {
  static char src1 [100] = "Copy this string
                            to dst1";
  static char dst1 [100];

void *c;

  c = memccpy (dst1, src1, 'g', sizeof (dst1));

  if (c == NULL)
    printf ("'g' was not found in the src
             buffer\n");
    else
      printf ("characters copied up to 'g'\n");

}
```

memchr

Summary:

```
#include <string.h>
void *memchr (
    void *buf,      /* buffer to search */
    char c,         /* byte to find */
    int len);               /* maximum buffer length */
```

Description: The **memchr** function scans *buf* for the character *c* in the first *len* bytes of the buffer.

Return Value: The **memchr** function returns a pointer to the character *c* in *buf* or a null pointer if the character was not found.

See Also: **memccpy, memcmp, memcpy, memmove, memset**

Example:

```
#include <string.h>
#include <stdio.h>                      /* for printf */

void tst_memchr (void)  {
  static char src1 [100] =
  "Search this string from the start";

  void *c;

  c = memchr (src1, 'g', sizeof (src1));

  if (c == NULL)
    printf ("'g' was not found in the buffer\n");
  else
    printf ("found 'g' in the buffer\n");

}
```

memcmp

Summary:

```
#include <string.h>
char memcmp (
    void *buf1,      /* first buffer */
    void *buf2,      /* second buffer */
    int len);                /* maximum bytes to compare
*/
```

Description: The **memcmp** function compares two buffers *buf1* and *buf2* for *len* bytes and returns a value indicating their relationship as follows:

Value	Meaning
< 0	*buf1* less than *buf2*
= 0	*but1* equal to *buf2*
> 0	*buf1* greater than *buf2*

Return Value: The **memcmp** function returns a positive, negative, or zero value indicating the relationship of *buf1* and *buf2*.

See Also: **memccpy, memchr, memcpy, memmove, memset**

Example:

```
#include <string.h>
#include <stdio.h>                        /* for printf */

void tst_memcmp (void)  {
  static char hexchars  [] = "0123456789ABCDEF";
  static char hexchars2 [] = "0123456789abcdef";

  char i;

  i = memcmp (hexchars, hexchars2, 16);

  if (i < 0)
    printf ("hexchars < hexchars2\n");

  else if (i > 0)
    printf ("hexchars > hexchars2\n");

  else
    printf ("hexchars == hexchars2\n");

}
```

memcpy

Summary:

```
#include <string.h>
void *memcpy (
    void *dest,      /* destination buffer */
    void *src,       /* source buffer */
    int len);                /* maximum bytes to copy */
```

Description: The **memcpy** function copies *len* bytes from *src* to *dest*. If these memory buffers overlap, the **memcpy** function cannot guarantee that bytes in *src* are copied to *dest* before being overwritten. If these buffers do overlap, use the **memmove** function.

Return Value: The **memcpy** function returns *dest*.

See Also: **memccpy, memchr, memcmp, memmove, memset**

Example:

```
#include <string.h>
#include <stdio.h>                          /* for printf */

void tst_memcpy (void)  {
  static char src1 [100] =
  "Copy this string to dst1";

  static char dst1 [100];

  char *p;

  p = memcpy (dst1, src1, sizeof (dst1));

  printf ("dst = \"%s\"\n", p);

}
```

memmove

Summary:

```
#include <string.h>
void *memmove (
    void *dest,      /* destination buffer */
    void *src,       /* source buffer */
    int len);                /* maximum bytes to move */
```

Description: The **memmove** function copies *len* bytes from *src* to *dest*. If these memory buffers overlap, the **memmove** function ensures that bytes in *src* are copied to *dest* before being overwritten.

Return Value: The **memmove** function returns *dest*.

See Also: **memccpy, memchr, memcmp, memcpy, memset**

Example:

```
#include <string.h>
#include <stdio.h>                      /* for printf */

void tst_memmove (void)  {
  static char buf [] = "This is line 1  "
                       "This is line 2  "
                       "This is line 3  ";

  printf ("buf before = %s\n", buf);

  memmove (&buf [0], &buf [16], 32);

  printf ("buf after  = %s\n", buf);

}
```

memset

Summary:

```
#include <string.h>
void *memset  (
    void *buf,        /* buffer to initialize */
    char c,           /* byte value to set */
    int len);                 /* buffer length */
```

Description: The **memset** function sets the first *len* bytes in *buf* to *c*.

Return Value: The **memset** function returns *dest*.

See Also: **memccpy, memchr, memcmp, memcpy, memmove**

Example:

```
#include <string.h>
#include <stdio.h>                        /* for printf */

void tst_memset (void)  {
  char buf [10];

  memset (buf, '\0', sizeof (buf));
                  /* fill buffer with null characters */

}
```

modf

Summary:

```
#include <math.h>
float modf (
    float val,      /* value to calculate modulo for */
    float *ip);     /* integer portion of modulo */
```

Description: The **modf** function splits the floating-point number *val* into integer and fractional components. The fractional part of *val* is returned as a signed floating-point number. The integer part is stored as a floating-point number at *ip*.

Return Value: The **modf** function returns the signed fractional part of *val*.

Example:

```
#include <math.h>
#include <stdio.h>                      /* for printf */

void tst_modf (void)  {
  float x;
  float int_part, frc_part;

  x = 123.456;

  frc_part = modf (x, &int_part);

  printf ("%f = %f + %f\n", x, int_part,frc_part);

}
```

nop

Summary: **#include <intrins.h>**
void _nop_ (void);

Description: The **_nop_** routine inserts an 8051 NOP instruction into the program. This routine can be used to pause for 1 CPU cycle. This routine is implemented as an intrinsic function. The code required is included in-line rather than being called.

Return Value: None.

Example:

```
#include <intrins.h>
#include <stdio.h>                          /* for printf */

void tst_nop (void)  {

  P1 = 0xFF;

  _nop_ ();                        /* delay for hardware */
  _nop_ ();
  _nop_ ();

  P1 = 0x00;

}
```

offsetof

Summary:

#include <stddef.h>
int offsetof (
structure, /* structure to use */
member); /* member to get offset for */

Description:

The **offsetof** macro calculates the offset of the *member* structure element from the beginning of the structure. The *structure* argument must specify the name of a structure. The *member* argument must specify the name of a member of the structure.

Return Value:

The **offsetof** macro returns the offset, in bytes, of the *member* element from the beginning of **struct** *structure*.

Example:

```
#include <stddef.h>

struct index_st
  {
  unsigned char type;
  unsigned long num;
  unsigned ing len;
  };

typedef struct index_st index_t;

void main (void)
{
int x, y;

x = offsetof (struct index_st, len);  /* x = 5 */
y = offsetof (index_t, num);          /* x = 1 */
}
```

pow

Summary:

```
#include <math.h>
float pow (
    float x,          /* value to use for base */
    float y);                 /* value to use for exponent */
```

Description: The **pow** function calculates x raised to the yth power.

Return Value: The **pow** function returns the value x^y. If $x \neq 0$ and $y = 0$, **pow** returns a value of 1. If $x = 0$ and $y \leq 0$, **pow** returns **NaN**. If $x < 0$ and y is not an integer, **pow** returns **NaN**.

See Also: **sqrt**

Example:

```
#include <math.h>
#include <stdio.h>                        /* for printf */

void tst_pow (void)  {
  float base;
  float power;
  float y;

  base  = 2.0;
  power = 8.0;

  y = pow (base, power);                    /* y = 256 */

  printf ("%f ^ %f = %f\n", base, power, y);

}
```

printf / printf517

Summary:

```
#include <stdio.h>
int printf (
    const char *fmtstr          /* format string */
    [, arguments]...);          /* additional arguments */
```

Description: The **printf** function formats a series of strings and numeric values and builds a string to write to the output stream using the **putchar** function. The *fmtstr* argument is a format string and may be composed of characters, escape sequences, and format specifications.

Ordinary characters and escape sequences are copied to the stream in the order in which they are interpreted. Format specifications always begin with a percent sign (**'%'**) and require additional *arguments* to be included in the function call.

The format string is read from left to right. The first format specification encountered references the first argument after *fmtstr* and converts and outputs it using the format specification. The second format specification accesses the second argument after *fmtstr*, and so on. If there are more arguments than format specifications, the extra arguments are ignored. Results are unpredictable if there are not enough arguments for the format specifications.

Format specifications have the following format:

% [*flags*] [*width*] [*. precision*] [{**b** | **B** | **l** | **L**}] *type*

Each field in the format specification can be a single character or a number which specifies a particular format option.

The *type* field is a single character that specifies whether the argument is interpreted as a character, string, number, or pointer, as shown in the following table.

Character	Argument Type	Output Format
d	**int**	Signed decimal number
u	**unsigned int**	Unsigned decimal number
o	**unsigned int**	Unsigned octal number
x	**unsigned int**	Unsigned hexadecimal number using "0123456789abcdef"
X	**unsigned int**	Unsigned hexadecimal number using "0123456789ABCEDF"
f	**float**	Floating-point number using the format [-]*dddd.dddd*
e	**float**	Floating-point number using the format [-]*d.dddd*e[-]*dd*
E	**float**	Floating-point number using the format [-]*d.dddd*E[-]*dd*
g	**float**	Floating-point number using either e or f format, whichever is more compact for the specified value and precision
G	**float**	Identical to the g format except that (where applicable) E precedes the exponent instead of e
c	**char**	Single character
s	**generic ***	String with a terminating null character
p	**generic ***	Pointer using the format *t*:*aaaa* where *t* is the memory type the pointer references (c: code, i: data/idata, x: xdata, p: pdata) and *aaaa* is the hexadecimal address

The optional characters **b** or **B** and **l** or **L** may immediately precede the type character to respectively specify **char** or **long** versions of the integer types **d**, **i**, **u**, **o**, **x**, and **X**.

The *flags* field is a single character used to justify the output and to print +/- signs and blanks, decimal points, and octal and hexadecimal prefixes, as shown in the following table.

Flag	Meaning
-	Left justify the output in the specified field width.
+	Prefix the output value with a + or - sign if the output is a signed type.
***blank* (' ')**	Prefix the output value with a blank if it is a signed positive value. Otherwise, no blank is prefixed.
#	Prefixes a non-zero output value with 0, 0x, or 0X when used with o, x, and X field types, respectively. When used with the e, E, f, g, and G field types, the # flag forces the output value to include a decimal point. The # flag is ignored in all other cases.
*****	Ignore format specifier.

The *width* field is a non-negative number that specifies the minimum number of characters printed. If the number of characters in the output value is less than *width*, blanks are added on the left or right (when the - flag is specified) to pad to the minimum width. If *width* is prefixed with a **'0'**, zeros are padded instead of blanks. The *width* field never truncates a field. If the length of the output value exceeds the specified width, all characters are output.

The *width* field may be an asterisk (**'*'**), in which case an **int** argument from the argument list provides the width value. Specifying a **'b'** in front of the asterisk specifies that the argument used is an **unsigned char**.

The *precision* field is a non-negative number that specifies the number of characters to print, the number of significant digits, or the number of decimal places. The *precision* field can cause truncation or rounding of the output value in the case of a floating-point number as specified in the following table.

Type	Meaning of Precision Field
d, u, o, x, X	The *precision* field is where you specify the minimum number of digits that are included in the output value. Digits are not truncated if the number of digits in the argument exceeds that defined in the *precision* field. If the number of digits in the argument is less than the *precision* field, the output value is padded on the left with zeros.
f	The *precision* field is where you specify the number of digits to the right of the decimal point. The last digit is rounded.
e, E	The *precision* field is where you specify the number of digits to the right of the decimal point. The last digit is rounded.
g, G	The *precision* field is where you specify the maximum number of significant digits in the output value.
c, p	The *precision* field has no effect on these field types.
s	The *precision* field is where you specify the maximum number of characters in the output value. Excess characters are not output.

The *precision* field may be an asterisk (**'*'**), in which case an **int** argument from the argument list provides the value for the precision. Specifying a **'b'** in front of the asterisk specifies that the argument used is an **unsigned char**.

The **printf517** function is identical to **printf**, but uses the arithmetic unit of the Siemens 80C517 to provide faster execution. When using this function, include the header file 80C517.H. Do not use this routine with a CPU that does not support this feature.

NOTE
This function is implementation-specific and is based on the operation of the ***putchar*** *function. This function, as provided in the standard library, writes characters using the serial port of the 8051. Custom functions may use other I/O devices.*

You must ensure that the argument type matches that of the format specification. You can use type casts to ensure that the proper type is passed to ***printf****.*

The total number of bytes that may be passed to ***printf*** *is limited due to the memory restrictions imposed by the 8051. A maximum of 15 bytes may be passed in small model or compact model. A maximum of 40 bytes may be passed in large model.*

Return Value: The **printf** function returns the number of characters actually written to the output stream.

See Also: **gets, puts, scanf, sprintf, sscanf, vprintf, vsprintf**

Example:

```
#include <stdio.h>

void tst_printf (void)  {
  char a;
  int  b;
  long c;
  unsigned char x;
  unsigned int  y;
  unsigned long z;
  float f,g;
  char buf [] = "Test String";
  char *p = buf;

  a = 1;
  b = 12365;
  c = 0x7FFFFFFF;
  x = 'A';
  y = 54321;
  z = 0x4A6F6E00;
  f = 10.0;
  g = 22.95;

  printf ("char %bd  int %d  long %ld\n",a,b,c);
  printf ("Uchar %bu Uint %u Ulong %lu\n",x,y,z);
  printf ("xchar %bx xint %x xlong %lx\n",x,y,z);
  printf ("String %s is at address %p\n",buf,p);
  printf ("%f != %g\n", f, g);
  printf ("%*f != %*g\n", 8, f, 8, g);
}
```

putchar

Summary:

```
#include <stdio.h>
char putchar (
    char c);                    /* character to output */
```

Description: The **putchar** function transmits the character *c* using the 8051 serial port.

NOTE
*This routine is implementation-specific and its function may deviate from that described above. Source is included for the **_getkey** and **putchar** functions which may be modified to provide character level I/O for any hardware device. Refer to "Customization Files" on page 113 for more information.*

Return Value: The **putchar** routine returns the character output, *c*.

See Also: **getchar, _getkey, ungetchar**

Example:

```
#include <stdio.h>

void tst_putchar (void)  {
  unsigned char i;

  for (i = 0x20; i < 0x7F; i++)
    putchar (i);
}
```

puts

Summary:

```
#include <stdio.h>
int puts (
    const char *string);      /* string to output */
```

Description: The **puts** function writes *string* followed by a newline character (**'\n'**) to the output stream using the putchar function.

> ***NOTE***
> *This function is implementation-specific and is based on the operation of the **putchar** function. This function, as provided in the standard library, writes characters using the serial port of the 8051. Custom functions may use other I/O devices.*

Return Value: The **puts** function returns **EOF** if an error occurred and a value of 0 if no errors were encountered.

See Also: **gets, printf, scanf**

Example:

```
#include <stdio.h>

void tst_puts (void)  {

  puts ("Line #1");
  puts ("Line #2");
  puts ("Line #3");

}
```

rand

Summary: **#include <stdlib.h>**
int rand (void);

Description: The **rand** function generates a pseudo-random number between 0 and 32767.

Return Value: The **rand** function returns a pseudo-random number.

See Also: **srand**

Example:

```
#include <stdlib.h>
#include <stdio.h>                      /* for printf */

void tst_rand (void)  {
  int i;
  int r;

  for (i = 0; i < 10; i++)  {
    printf ("I = %d, RAND = %d\n", i, rand ());
  }

}
```

realloc

Summary:

```
#include <stdlib.h>
void *realloc (
    void xdata *p,          /* previously allocated block */
    unsigned int size);     /* new size for block */
```

Description: The **realloc** function changes the size of a previously allocated memory block. The *p* argument points to the allocated block and the *size* argument specifies the new size for the block. The contents of the existing block are copied to the new block. Any additional area in the new block, due to a larger block size, is not initialized.

NOTE
Source code is provided for this routine in the **\C51\LIB** *directory. You can modify the source to customize this function for your hardware environment. Refer to* *"Chapter 6. Advanced Programming Techniques" on page 113 for more information.*

Return Value: The **realloc** function returns a pointer to the new block. If there is not enough memory in the memory pool to satisfy the memory request, a null pointer is returned and the original memory block is not affected.

See Also: **calloc, free, init_mempool, malloc**

Example:

```
#include <stdlib.h>
#include <stdio.h>                          /* for printf */

void tst_realloc (void)  {
  void xdata *p;
  void xdata *new_p;

  p = malloc (100);
  if (p != NULL)  {
    new_p = realloc (p, 200);

    if (new_p != NULL)  p = new_p;
    else  printf ("Reallocation failed\n");
  }
}
```

scanf

Summary:

```
#include <stdio.h>
int scanf (
    const char *fmtstr          /* format string */
    [, argument]...);           /* additional arguments */
```

Description: The **scanf** function reads data using the **getchar** routine. Data input are stored in the locations specified by *argument* according to the format string *fmtstr*. Each *argument* must be a pointer to a variable that corresponds to the type defined in *fmtstr* which controls the interpretation of the input data. The *fmtstr* argument is composed of one or more whitespace characters, non-whitespace characters, and format specifications as defined below.

The **scanf517** function is identical to **scanf**, but uses the arithmetic unit of the Siemens 80C517 to provide faster execution. When using this function include the header file 80C517.H. Do not use this routine with a CPU that does not support this feature.

- Whitespace characters, blank (**' '**), tab (**'\t'**), or newline (**'\n'**), causes **scanf** to skip whitespace characters in the input stream. A single whitespace character in the format string matches 0 or more whitespace characters in the input stream.
- Non-whitespace characters, with the exception of the percent sign (**'%'**), cause **scanf** to read but not store a matching character from the input stream. The **scanf** function terminates if the next character in the input stream does not match the specified non-whitespace character.
- Format specifications begin with a percent sign (**'%'**) and cause **scanf** to read and convert characters from the input stream to the specified type values. The converted value is stored to an *argument* in the parameter list. Characters following a percent sign that are not recognized as a format specification are treated as an ordinary character. For example, **%%** matches a single percent sign in the input stream.

The format string is read from left to right. Characters that are not part of the format specifications must match characters in the input stream. These characters are read from the input stream but are discarded and not stored. If a character in the input stream conflicts with the format string, **scanf** terminates. Any conflicting characters remain in the input stream.

The first format specification encountered in the format string references the first argument after *fmtstr* and converts input characters and stores the value using the format specification. The second format specification accesses the second argument after *fmtstr*, and so on. If there are more arguments than format specifications, the extra arguments are ignored. Results are unpredictable if there are not enough arguments for the format specifications.

Values in the input stream are called input fields and are delimited by whitespace characters. When converting input fields, **scanf** ends a conversion for an argument when a whitespace character is encountered. Additionally, any unrecognized character for the current format specification ends a field conversion.

Format specifications have the following format:

% [[*****]] [[*width*]] [[{**b** | **h** | **l**}]] *type*

Each field in the format specification can be a single character or a number which specifies a particular format option.

The *type* field is where a single character specifies whether input characters are interpreted as a character, string, or number. This field can be any one of the characters in the following table.

Character	Argument Type	Input Format
d	**int ***	Signed decimal number
i	**int ***	Signed decimal, hexadecimal, or octal integer
u	**unsigned int ***	Unsigned decimal number

Character	Argument Type	Input Format
o	**unsigned int ***	Unsigned octal number
x	**unsigned int ***	Unsigned hex number
e	**float ***	Floating-point number
f	**float ***	Floating-point number
g	**float ***	Floating-point number
c	**char ***	A single character
s	**char ***	A string of characters terminated by whitespace

An asterisk (*****) as the first character of a format specification causes the input field to be scanned but not stored. The asterisk suppresses assignment of the format specification.

The *width* field is a non-negative number that specifies the maximum number of characters read from the input stream. No more than *width* characters are read from the input stream and converted for the corresponding *argument*. However, fewer than *width* characters may be read if a whitespace character or an unrecognized character is encountered first.

The optional characters **b**, **h**, and **l** may immediately precede the type character to respectively specify **char**, **short**, or **long** versions of the integer types **d**, **i**, **u**, **o**, and **x**.

NOTE
*This function is implementation-specific and is based on the operation of the **_getkey** and/or **putchar** functions. These functions, as provided in the standard library, read and write characters using the serial port of the 8051. Custom functions may use other I/O devices.*

*The total number of bytes that may be passed to **scanf** is limited due to the memory restrictions imposed by the 8051. A maximum of 15 bytes may be passed in small model or compact model. A maximum of 40 bytes may be passed in large model.*

Return Value: The **scanf** function returns the number of input fields that were successfully converted. An **EOF** is returned if an error is encountered.

See Also: **gets, printf, puts, sprintf, sscanf, vprintf, vsprintf**

Example:

```
#include <stdio.h>

void tst_scanf (void)  {
  char a;
  int  b;
  long c;

  unsigned char x;
  unsigned int  y;
  unsigned long z;

  float f,g;

  char d, buf [10];

  int argsread;

  printf ("Enter a signed byte, int, and long\n");
  argsread = scanf ("%bd %d %ld", &a, &b, &c);
  printf ("%d arguments read\n", argsread);

  printf ("Enter an unsigned byte, int, and long\n");
  argsread = scanf ("%bu %u %lu", &x, &y, &z);
  printf ("%d arguments read\n", argsread);

  printf ("Enter a character and a string\n");
  argsread = scanf ("%c %9s", &d, buf);
  printf ("%d arguments read\n", argsread);

  printf ("Enter two floating-point numbers\n");
  argsread = scanf ("%f %f", &f, &g);
  printf ("%d arguments read\n", argsread);

}
```

setjmp

Summary:

```
#include <setjmp.h>
int setjmp (
    jmp_buf env);          /* current environment */
```

Description: The **setjmp** function saves the current state of the CPU in *env*. The state can be restored by a subsequent call to the **longjmp** function. When used together, the **setjmp** and **longjmp** functions provide you with a way to execute a non-local goto.

A call to the **setjmp** function saves the current instruction address as well as other CPU registers. A subsequent call to the **longjmp** function restores the instruction pointer and registers, and execution resumes at the point just after the **setjmp** call.

Local variables and function arguments are restored only if declared with the **volatile** attribute.

Return Value: The **setjmp** function returns a value of 0 when the current state of the CPU has been copied to *env*. A non-zero value indicates that the **longjmp** function was executed to return to the **setjmp** function call. In such a case, the return value is the value passed to the **longjmp** function.

See Also: **longjmp**

Example: See **longjmp**

sin / sin517

Summary:

```
#include <math.h>
float sin (
    float x);                    /* value to calculate
sine for */
```

Description: The **sin** function calculates the sine of the floating-point value *x*. The value of *x* must be in the -65535 to +65535 range or an **NaN** error value is generated.

The **sin517** function is identical to **sin**, but uses the arithmetic unit of the Siemens 80C517 to provide faster execution. When using this function include the header file 80C517.H. Do not use this routine with a CPU that does not support this feature.

Return Value: The **sin** function returns the sine of *x*.

See Also: **cos, tan**

Example:

```
#include <math.h>
#include <stdio.h>                        /* for printf */

void tst_sin (void)  {
  float x;
  float y;

  for (x = 0; x < (2 * 3.1415); x += 0.1)  {
    y = sin (x);

    printf ("SIN(%f) = %f\n", x, y);
  }

}
```

sinh

Summary:

```
#include <math.h>
float sinh (
    float val);        /* value to calc hyperbolic sine for */
```

Description: The **sinh** function calculates the hyperbolic sine of the floating-point value *x*. The value of *x* must be in the -65535 to +65535 range or an **NaN** error value is generated.

Return Value: The **sinh** function returns the hyperbolic sine of *x*.

See Also: **cosh, tanh**

Example:

```
#include <math.h>
#include <stdio.h>                      /* for printf */

void tst_sinh (void)  {
  float x;
  float y;

  for (x = 0; x < (2 * 3.1415); x += 0.1)  {
    y = sinh (x);
    printf ("SINH(%f) = %f\n", x, y);
  }

}
```

sprintf / sprintf517

Summary:

```
#include <stdio.h>
int sprintf (
    char *buffer,           /* storage buffer */
    const char *fmtstr      /* format string */
    [, argument]...);       /* additional arguments */
```

Description: The **sprintf** function formats a series of strings and numeric values and stores the resulting string in *buffer*. The *fmtstr* argument is a format string and has the same requirements as specified for the **printf** function. Refer to "printf / printf517" on page 252 for a description of the format string and additional arguments.

The **sprintf517** function is identical to **sprintf**, but uses the arithmetic unit of the Siemens 80C517 to provide faster execution. When using this function include the header file 80C517.H. Do not use this routine with a CPU that does not support this feature.

NOTE
The total number of bytes that may be passed to ***sprintf*** *is limited due to the memory restrictions imposed by the 8051. A maximum of 15 bytes may be passed in small model or compact model. A maximum of 40 bytes may be passed in large model.*

Return Value: The **sprintf** function returns the number of characters actually written to *buffer*.

See Also: **gets, printf, puts, scanf, sscanf, vprintf, vsprintf**

Example:

```
#include <stdio.h>

void tst_sprintf (void)  {
  char buf [100];
  int n;

  int a,b;
  float pi;

  a = 123;
  b = 456;
  pi = 3.14159;

  n =  sprintf (buf,   "%f\n", 1.1);
  n += sprintf (buf+n, "%d\n", a);
  n += sprintf (buf+n, "%d %s %g", b, "---", pi);
  printf (buf);
}
```

sqrt / sqrt517

Summary:

```
#include <math.h>
float sqrt (
    float x);                  /* value to calculate square root
of */
```

Description: The **sqrt** function calculates the square root of *x*.

The **sqrt517** function is identical to **sqrt**, but uses the arithmetic unit of the Siemens 80C517 to provide faster execution. When using this function include the header file 80C517.H. Do not use this routine with a CPU that does not support this feature.

Return Value: The **sqrt** function returns the positive square root of *x*.

See Also: **exp, log, pow**

Example:

```
#include <math.h>
#include <stdio.h>                          /* for printf */

void tst_sqrt (void)  {
  float x;
  float y;

  x = 25.0;

  y = sqrt (x);                                  /* y = 5 */

  printf ("SQRT(%f) = %f\n", x, y);

}
```

srand

Summary:

```
#include <stdlib.h>
void srand (
    int seed);          /* random number generator seed */
```

Description: The **srand** function sets the starting value *seed* used by the pseudo-random number generator in the **rand** function. The random number generator produces the same sequence of pseudo-random numbers for any given value of *seed*.

Return Value: None.

See Also: **rand**

Example:

```
#include <stdlib.h>
#include <stdio.h>                      /* for printf */

void tst_srand (void)  {
  int i;
  int r;

  srand (56);

  for (i = 0; i < 10; i++)  {
    printf ("I = %d, RAND = %d\n", i, rand ());
  }

}
```

sscanf / sscanf517

Summary:

```
#include <stdio.h>
int sscanf (
    char *buffer,          /* scanf input buffer */
    const char *fmtstr     /* format string */
    [, argument]...);      /* additional arguments */
```

Description: The **sscanf** function reads data from the string *buffer*. Data input are stored in the locations specified by *argument* according to the format string *fmtstr*. Each *argument* must be a pointer to a variable that corresponds to the type defined in *fmtstr* which controls the interpretation of the input data. The *fmtstr* argument is composed of one or more whitespace characters, non-whitespace characters, and format specifications, as defined in the **scanf** function description. Refer to "scanf" on page 262 for a complete description of the formation string and additional arguments.

The **sscanf517** function is identical to **sscanf**, but uses the arithmetic unit of the Siemens 80C517 to provide faster execution. When using this function include the header file 80C517.H. Do not use this routine with a CPU that does not support this feature.

NOTE
*The total number of bytes that may be passed to **sscanf** is limited due to the memory restrictions imposed by the 8051. A maximum of 15 bytes may be passed in small model or compact model. A maximum of 40 bytes may be passed in large model.*

Return Value: The **sscanf** function returns the number of input fields that were successfully converted. An **EOF** is returned if an error is encountered.

See Also: **gets, printf, puts, scanf, sprintf, vprintf, vsprintf**

Example:

```
#include <stdio.h>

void tst_sscanf (void)  {
  char a;
  int  b;
  long c;

  unsigned char x;
  unsigned int  y;
  unsigned long z;

  float f,g;

  char d, buf [10];

  int argsread;

  printf ("Reading a signed byte, int,and long\n");
  argsread = sscanf ("1 -234 567890",
                     "%bd %d %ld", &a, &b, &c);
  printf ("%d arguments read\n", argsread);

  printf ("Reading an unsigned byte, int, and long\n");
  argsread = sscanf ("2 44 98765432",
                      "%bu %u %lu", &x, &y, &z);
  printf ("%d arguments read\n", argsread);

  printf ("Reading a character and a string\n");
  argsread = sscanf ("a abcdefg", "%c %9s", &d, buf);
  printf ("%d arguments read\n", argsread);

  printf ("Reading two floating-point numbers\n");
  argsread = sscanf ("12.5 25.0", "%f %f", &f, &g);
  printf ("%d arguments read\n", argsread);

}
```

strcat

Summary:

```
#include <string.h>
char *strcat (
    char *dest,      /* destination string */
    char *src);      /* source string */
```

Description: The **strcat** function concatenates or appends *src* to *dest* and terminates *dest* with a null character.

Return Value: The **strcat** function returns *dest*.

See Also: **strcpy, strlen, strncat, strncpy**

Example:

```
#include <string.h>
#include <stdio.h>                        /* for printf */

void tst_strcat (void)  {
  char buf [21];
  char s [] = "Test String";

  strcpy (buf, s);
  strcat (buf, " #2");

  printf ("new string is %s\n", buf);

}
```

strchr

Summary:

#include <string.h>
char *strchr (
    **const char ***string*, /* string to search */
    char *c*); /* character to find */

Description: The **strchr** function searches *string* for the first occurrence of *c*. The null character terminating *string* is included in the search.

Return Value: The **strchr** function returns a pointer to the character *c* found in *string* or a null pointer if no matching character was found.

See Also: **strcspn, strpbrk, strpos, strrchr, strrpbrk, strrpos, strspn**

Example:

```
#include <string.h>
#include <stdio.h>                      /* for printf */

void tst_strchr (void)  {
  char *s;
  char buf [] = "This is a test";

  s = strchr (buf, 't');

  if (s != NULL)
    printf ("found a 't' at %s\n", s);
}
```

strcmp

Summary:

```
#include <string.h>
char strcmp (
    char *string1,          /* first string */
    char *string2);         /* second string */
```

Description: The **strcmp** function compares the contents of *string1* and *string2* and returns a value indicating their relationship.

Return Value: The **strcmp** function returns the following values to indicate the relationship of *string1* to *string2*:

Value	Meaning
< 0	*string1* less than *string2*
= 0	*string1* equal to *string2*
> 0	*string1* greater than *string2*

See Also: **memcmp, strncmp**

Example:

```
#include <string.h>
#include <stdio.h>                          /* for printf */

void tst_strcmp (void)  {
  char buf1 [] = "Bill Smith";
  char buf2 [] = "Bill Smithy";
  char i;

  i = strcmp (buf1, buf2);

  if (i < 0)
    printf ("buf1 < buf2\n");

  else if (i > 0)
    printf ("buf1 > buf2\n");

  else
    printf ("buf1 == buf2\n");
}
```

strcpy

Summary:

```
#include <string.h>
char *strcpy (
    char *dest,      /* destination string */
    char *src);      /* source string */
```

Description: The **strcpy** function copies *src* to *dest* and appends a null character to the end of *dest*.

Return Value: The **strcpy** function returns *dest*.

See Also: **strcat, strlen, strncat, strncpy**

Example:

```
#include <string.h>
#include <stdio.h>                        /* for printf */

void tst_strcpy (void)  {
  char buf [21];
  char s [] = "Test String";

  strcpy (buf, s);
  strcat (buf, " #2");

  printf ("new string is %s\n", buf);
}
```

strcspn

Summary:

```
#include <string.h>
int strcspn (
    char *src,      /* source string */
    char *set);     /* characters to find */
```

Description: The **strcspn** function searches the *src* string for any of the characters in the *set* string.

Return Value: The **strcspn** function returns the index of the first character located in *src* that matches a character in *set*. If the first character in *src* matches a character in *set*, a value of 0 is returned. If there are no matching characters in *src*, the length of the string is returned.

See Also: **strchr, strpbrk, strpos, strrchr, strrpbrk, strrpos, strspn**

Example:

```
#include <string.h>
#include <stdio.h>                          /* for printf */

void tst_strcspn (void)  {
  char buf [] = "13254.7980";
  int i;

  i = strcspn (buf, ".,");

  if (buf [i] != '\0')
    printf ("%c was found in %s\n", (char)
    buf [i], buf);
}
```

strlen

Summary:

```
#include <string.h>
int strlen (
    char *src);       /* source string */
```

Description: The **strlen** function calculates the length, in bytes, of *src*. This calculation does not include the null terminating character.

Return Value: The **strlen** function returns the length of *src*.

See Also: **strcat, strcpy, strncat, strncpy**

Example:

```
#include <string.h>
#include <stdio.h>                        /* for printf */

void tst_strlen (void)  {
  char buf [] = "Find the length of this string";
  int len;

  len = strlen (buf);

  printf ("string length is %d\n", len);

}
```

strncat

Summary:

```
#include <string.h>
char *strncat (
    char *dest,      /* destination string */
    char *src,       /* source string */
    int len);                /* max. chars to concatenate */
```

Description: The **strncat** function appends at most *len* characters from *src* to *dest* and terminates *dest* with a null character. If *src* is shorter than *len* characters, *src* is copied up to and including the null terminating character.

Return Value: The **strncat** function returns *dest*.

See Also: **strcat, strcpy, strlen, strncpy**

Example:

```
#include <string.h>
#include <stdio.h>                          /* for printf */

void tst_strncat (void)  {
  char buf [21];

  strcpy  (buf, "test #");
  strncat (buf, "three", sizeof (buf) - strlen
           (buf));
}
```

strncmp

Summary:

```
#include <string.h>
char strncmp (
    char *string1,          /* first string */
    char *string2,          /* second string */
    int len);               /* max characters to
compare */
```

Description: The **strncmp** function compares the first *len* bytes of *string1* and *string2* and returns a value indicating their relationship.

Return Value: The **strncmp** function returns the following values to indicate the relationship of the first *len* bytes of *string1* to *string2*:

Value	Meaning
< 0	*string1* less than *string2*
= 0	*string1* equal to *string2*
> 0	*string1* greater than *string2*

See Also: **memcmp, strcmp**

Example:

```
#include <string.h>
#include <stdio.h>                          /* for printf */

void tst_strncmp (void)  {
  char str1 [] = "Wrodanahan     T.J.";
  char str2 [] = "Wrodanaugh     J.W.";

  char i;

  i = strncmp (str1, str2, 15);

  if (i < 0)       printf ("str1 < str2\n");
  else if (i > 0)  printf ("str1 > str2\n");
  else             printf ("str1 == str2\n");
}
```

strncpy

Summary:

```
#include <string.h>
char *strncpy (
    char *dest,          /* destination string */
    char *src,           /* source string */
    int len);                    /* max characters to
copy */
```

Description: The **strncpy** function copies at most *len* characters from *src* to *dest*. If *src* contains fewer characters than *len*, *dest* is padded out with null characters to *len* characters.

Return Value: The **strncpy** function returns *dest*.

See Also: **strcat, strcpy, strlen, strncat**

Example:

```
#include <string.h>
#include <stdio.h>                          /* for printf */

void tst_strncpy ( char *s)  {
  char buf [21];

  strncpy (buf, s, sizeof (buf));
  buf [sizeof (buf)] = '\0';
}
```

strpbrk

Summary:

```
#include <string.h>
char *strpbrk (
    char *string,     /* string to search */
    char *set);       /* characters to find */
```

Description: The **strpbrk** function searches *string* for the first occurrence of any character from *set*. The null terminator is not included in the search.

Return Value: The **strpbrk** function returns a pointer to the matching character in *string*. If *string* contains no characters from *set*, a null pointer is returned.

See Also: **strchr, strcspn, strpos, strrchr, strrpbrk, strrpos, strspn**

Example:

```
#include <string.h>
#include <stdio.h>                      /* for printf */

void tst_strpbrk (void)  {
  char vowels [] ="AEIOUaeiou";
  char text [] = "Seven years ago...";

  char *p;

  p = strpbrk (text, vowels);

  if (p == NULL)
    printf ("No vowels found in %s\n", text);

  else
    printf ("Found a vowel at %s\n", p);

}
```

strpos

Summary:

```
#include <string.h>
int strpos (
    const char *string,      /* string to search */
    char c);                 /* character to find */
```

Description: The **strpos** function searches *string* for the first occurrence of *c*. The null character terminating *string* is included in the search.

Return Value: The **strpos** function returns the index of the character matching *c* in *string* or a value of -1 if no matching character was found. The index of the first character in *string* is 0.

See Also: **strchr, strcspn, strpbrk, strrchr, strrpbrk, strrpos, strspn**

Example:

```
#include <string.h>
#include <stdio.h>                      /* for printf */

void tst_strpos (void)  {
  char text [] = "Search this string for
  blanks";

  int i;

  i = strpos (text, ' ');

  if (i == -1)
    printf ("No spaces found in %s\n", text);

  else
    printf ("Found a space at offset %d\n", i);

}
```

strrchr

Summary:

```
#include <string.h>
char *strrchr (
    const char *string,      /* string to search */
    char c);                 /* character to find */
```

Description: The **strrchr** function searches *string* for the last occurrence of *c*. The null character terminating *string* is included in the search.

Return Value: The **strrchr** function returns a pointer to the last character *c* found in *string* or a null pointer if no matching character was found.

See Also: **strchr, strcspn, strpbrk, strpos, strrpbrk, strrpos, strspn**

Example:

```
#include <string.h>
#include <stdio.h>                      /* for printf */

void tst_strrchr (void)  {
  char *s;
  char buf [] = "This is a test";

  s = strrchr (buf, 't');

  if (s != NULL)
    printf ("found the last 't' at %s\n", s);

}
```

strrpbrk

Summary:

```
#include <string.h>
char *strrpbrk (
    char *string,    /* string to search */
    char *set);      /* characters to find */
```

Description: The **strrpbrk** function searches *string* for the last occurrence of any character from *set*. The null terminator is not included in the search.

Return Value: The **strrpbrk** function returns a pointer to the last matching character in *string*. If *string* contains no characters from *set*, a null pointer is returned.

See Also: **strchr, strcspn, strpbrk, strpos, strrchr, strrpos, strspn**

Example:

```
#include <string.h>
#include <stdio.h>                        /* for printf */

void tst_strrpbrk (void)  {
  char vowels [] ="AEIOUaeiou";
  char text [] = "American National Standards
                  Institute";

  char *p;

  p = strpbrk (text, vowels);

  if (p == NULL)
    printf ("No vowels found in %s\n", text);

  else
    printf ("Last vowel is at %s\n", p);

}
```

strrpos

Summary:

```
#include <string.h>
int strrpos (
    const char *string,      /* string to search */
    char c);                 /* character to find */
```

Description: The **strrpos** function searches *string* for the last occurrence of *c*. The null character terminating *string* is included in the search.

Return Value: The **strrpos** function returns the index of the last character matching *c* in *string* or a value of -1 if no matching character was found. The index of the first character in *string* is 0.

See Also: **strchr, strcspn, strpbrk, strpos, strrchr, strrpbrk, strspn**

Example:

```
#include <string.h>
#include <stdio.h>                      /* for printf */

void tst_strrpos (  char *s)  {

  int i;

  i = strpos (s, ' ');

  if (i == -1)
    printf ("No spaces found in %s\n", s);

  else
    printf ("Last space in %s is at offset %d\n",
             s, i);
}
```

strspn

Summary:

```
#include <string.h>
int strspn (
    char *string,    /* string to search */
    char *set);      /* characters to allow */
```

Description: The **strspn** function searches the *src* string for characters not found in the *set* string.

Return Value: The **strspn** function returns the index of the first character located in *src* that does not match a character in *set*. If the first character in *src* does not match a character in *set*, a value of 0 is returned. If all characters in *src* are found in set, the length of *src* is returned.

See Also: **strchr, strcspn, strpbrk, strpos, strrchr, strrpbrk, strrpos**

Example:

```
#include <string.h>
#include <stdio.h>                      /* for printf */

void tst_strspn ( char *digit_str)  {
  char octd [] = "01234567";
  int i;

  i = strspn (digit_str, octd);

  if (digit_str [i] != '\0')
    printf ("%c is not an octal digit\n",
             digit_str [i]);

}
```

tan / tan517

Summary:

```
#include <math.h>
float tan (
    float x);                    /* value to calculate tangent of
*/
```

Description: The **tan** function calculates the tangent of the floating-point value *x*. The value of *x* must be in the -65535 to +65535 range or an **NaN** error value is generated.

The **tan517** function is identical to **tan**, but uses the arithmetic unit of the Siemens 80C517 to provide faster execution. When using this function, include the header file 80C517.H. Do not use this routine with a CPU that does not support this feature.

Return Value: The **tan** function returns the tangent of *x*.

See Also: **cos, sin**

Example:

```
#include <math.h>
#include <stdio.h>                          /* for printf */

void tst_tan (void)  {
  float x, y, pi;

  pi = 3.14159;

  for (x = -(pi/4); x < (pi/4); x += 0.1)  {
    y = tan (x);
    printf ("TAN(%f) = %f\n", x, y);
  }
}
```

tanh

Summary:

```
#include <math.h>
float tanh (
    float x);                    /* value to calc hyperbolic
tangent for */
```

Description: The **tanh** function calculates the hyperbolic tangent for the floating-point value *x*.

Return Value: The **tanh** function returns the hyperbolic tangent of *x*.

See Also: **cosh, sinh**

Example:

```
#include <math.h>
#include <stdio.h>                        /* for printf */

void tst_tanh (void)  {
  float x;
  float y;
  float pi;

  pi = 3.14159;

  for (x = -(pi/4); x < (pi/4); x += 0.1)  {
    y = tanh (x);
    printf ("TANH(%f) = %f\n", x, y);
  }

}
```

testbit

Summary:

#include <intrins.h>
bit _testbit_ (
    bit *b*); * bit to test and clear */

Description: The **_testbit_** routine produces a JBC instruction in the generated program code to simultaneously test the bit *b* and clear it to 0. This routine can be used only on directly addressable bit variables and is invalid on any type of expression. This routine is implemented as an intrinsic function. The code required is included in-line rather than being called.

Return Value: The **_testbit_** routine returns the value of *b*.

Example:

```
#include <intrins.h>
#include <stdio.h>                      /* for printf */

void tst_testbit (void){
  bit test_flag;

  if (_testbit_ (test_flag))
    printf ("Bit was set\n");

  else
    printf ("Bit was clear\n");
}
```

toascii

Summary:

```
#include <ctype.h>
char toascii (
    char c);                    /* character to convert */
```

Description: The **toascii** macro converts *c* to a 7-bit ASCII character. This macro clears all but the lower 7 bits of *c*.

Return Value: The **toascii** macro returns the 7-bit ASCII character for *c*.

See Also: **toint**

Example:

```
#include <ctype.h>
#include <stdio.h>                          /* for printf */

void tst_toascii ( char c)  {
  char k;

  k = toascii (c);

  printf ("%c is an ASCII character\n", k);
}
```

toint

Summary:

```
#include <ctype.h>
char toint (
    char c);                    /* digit to convert */
```

Description: The **toint** function interprets *c* as a hexadecimal value. ASCII characters **'0'** through **'9'** generate values of 0 to 9. ASCII characters **'A'** through **'F'** and **'a'** through **'f'** generate values of 10 to 15.

Return Value: The **toint** function returns the value of the ASCII hexadecimal character *c*.

See Also: **toascii**

Example:

```
#include <ctype.h>
#include <stdio.h>                      /* for printf */

void tst_toint (void)  {
  unsigned long l;
  char k;

  for (l = 0; isdigit (k = getchar ());
       l *= 10)  {

     l += toint (k);
  }
}
```

tolower

Summary:

```
#include <ctype.h>
char tolower (
    char c);                    /* character to convert */
```

Description: The **tolower** function converts *c* to a lowercase character. If *c* is not an alphabetic letter, the **tolower** function has no effect.

Return Value: The **tolower** function returns the lowercase equivalent of *c*.

See Also: **_tolower, toupper, _toupper**

Example:

```
#include <ctype.h>
#include <stdio.h>                      /* for printf */

void tst_tolower (void)  {
  unsigned char i;

  for (i = 0x20; i < 0x7F; i++)  {
    printf ("tolower(%c) = %c\n", i, tolower(i));
  }
}
```

_tolower

Summary:

```
#include <ctype.h>
char _tolower (
    char c);                    /* character to convert */
```

Description: The **_tolower** macro is a version of **tolower** that can be used when *c* is known to be an uppercase character.

Return Value: The **_tolower** macro returns a lowercase character.

See Also: **tolower, toupper, _toupper**

Example:

```
#include <ctype.h>
#include <stdio.h>                          /* for printf */

void tst__tolower (  char k)  {

  if (isupper (k))   k = _tolower (k);
}
```

toupper

Summary:

```
#include <ctype.h>
char toupper (
    char c);                    /* character to convert */
```

Description: The **toupper** function converts *c* to an uppercase character. If *c* is not an alphabetic letter, the **toupper** function has no effect.

Return Value: The **toupper** function returns the uppercase equivalent of *c*.

See Also: **tolower, _tolower, _toupper**

Example:

```
#include <ctype.h>
#include <stdio.h>                      /* for printf */

void tst_toupper (void)  {
  unsigned char i;

  for (i = 0x20; i < 0x7F; i++)  {
    printf ("toupper(%c) = %c\n", i, toupper(i));
  }
}
```

_toupper

Summary:

```
#include <ctype.h>
char _toupper (
    char c);                    /* character to convert */
```

Description: The **_toupper** macro is a version of **toupper** that can be used when *c* is known to be a lowercase character.

Return Value: The **_toupper** macro returns an uppercase character.

See Also: **tolower, _tolower, toupper**

Example:

```
#include <ctype.h>
#include <stdio.h>                          /* for printf */

void tst__toupper (  char k)  {
  if (islower (k))  k = _toupper (k);
}
```

ungetchar

Summary:

```
#include <stdio.h>
char ungetchar (
    char c);                    /* character to unget */
```

Description: The **ungetchar** function stores the character *c* back into the input stream. Subsequent calls to **getchar** and other stream input functions return *c*. Only one character may be passed to **unget** between calls to **getchar**.

Return Value: The **ungetchar** function returns the character *c* if successful. If **ungetchar** is called more than once between function calls that read from the input stream, **EOF** is returned indicating an error condition.

See Also: **_getkey, putchar, ungetchar**

Example:

```
#include <stdio.h>

void tst_ungetchar (void)  {
  char k;

  while (isdigit (k = getchar ()))  {
          /* stay in the loop as long as k is a digit */
  }
  ungetchar (k);
}
```

va_arg

Summary:

#include <stdarg.h>
type **va_arg (**
argptr, /* optional argument list */
type); /* type of next argument */

Description:

The **va_arg** macro is used to extract subsequent arguments from a variable-length argument list referenced by *argptr*. The *type* argument specifies the data type of the argument to extract. This macro may be called only once for each argument and must be called in the order of the parameters in the argument list.

The first call to **va_arg** returns the first argument after the *prevparm* argument specified in the **va_start** macro. Subsequent calls to **va_arg** return the remaining arguments in succession.

Return Value:

The **va_arg** macro returns the value for the specified argument type.

See Also:

va_end, va_start

Example:

```
#include <stdarg.h>
#include <stdio.h>                          /* for printf */

int varfunc (char *buf, int id, ...)  {
  va_list tag;

  va_start (tag, id);


  if (id == 0)  {
    int arg1;
    char *arg2;
    long arg3;

    arg1 = va_arg (tag, int);
    arg2 = va_arg (tag, char *);
    arg3 = va_arg (tag, long);
  }
  else  {
    char *arg1;
    char *arg2;
    long arg3;

    arg1 = va_arg (tag, char *);
    arg2 = va_arg (tag, char *);
    arg3 = va_arg (tag, long);
  }
}


void caller (void)  {
  char tmp_buffer [10];

  varfunc (tmp_buffer, 0, 27, "Test Code", 100L);
  varfunc (tmp_buffer, 1, "Test", "Code", 348L);
}
```

va_end

Summary:

```
#include <stdarg.h>
void va_end (
    argptr);          /* optional argument list */
```

Description: The **va_end** macro is used to terminate use of the variable-length argument list pointer *argptr* that was initialized using the **va_start** macro.

Return Value: None.

See Also: **va_arg, va_start**

Example: See **va_arg**.

va_start

Summary:

```
#include <stdarg.h>
void va_start (
    argptr,          /* optional argument list */
    prevparm);       /* arg preceding optional args */
```

Description: The **va_start** macro, when used in a function with a variable-length argument list, initializes *argptr* for subsequent use by the **va_arg** and **va_end** macros. The *prevparm* argument must be the name of the function argument immediately preceding the optional arguments specified by an ellipses (…). This function must be called to initialize a variable-length argument list pointer before any access using the **va_arg** macro is made.

Return Value: None.

See Also: **va_arg, va_end**

Example: See **va_arg**.

vprintf

Summary:

```
#include <stdio.h>
void vprintf (
    const char * fmtstr,      /* pointer to format string */
    char * argptr);           /* pointer to argument list */
```

Description: The **vprintf** function formats a series of strings and numeric values and builds a string to write to the output stream using the **putchar** function. The function is similar to the counterpart printf, but it accepts a pointer to a list of arguments instead of an argument list.

The *fmtstr* argument is a pointer to a format string and has the same form and function as the *fmtstr* argument for the **printf** function. Refer to "printf / printf517" on page 252 for a description of the format string. The *argptr* argument points to a list of arguments that are converted and output according to the corresponding format specifications in the format.

NOTE
*This function is implementation-specific and is based on the operation of the **putchar** function. This function, as provided in the standard library, writes characters using the serial port of the 8051. Custom functions may use other I/O devices.*

Return Value: The **vprintf** function returns the number of characters actually written to the output stream.

See Also: **gets, puts, printf, scanf, sprintf, sscanf, vsprintf**

Example:

```
#include <stdio.h>
#include <stdarg.h>

void error (char *fmt, ...)  {
  va_list arg_ptr;

  va_start (arg_ptr, fmt);           /* format string */
  vprintf (fmt, arg_ptr);
  va_end (arg_ptr);
}

void tst_vprintf (void)  {
  int i;
  i = 1000;
                     /* call error with one parameter */
  error ("Error: '%d' number too large\n", i);
              /* call error with just a format string */
  error ("Syntax Error\n");
}
```

vsprintf

Summary:

```
#include <stdio.h>
void vsprintf (
    char *buffer,          /* pointer to storage buffer */
    const char * fmtstr,   /* pointer to format string */
    char * argptr);        /* pointer to argument list */
```

Description: The **vsprintf** function formats a series of strings and numeric values and stores the string in *buffer*. The function is similar to the counterpart sprintf, but it accepts a pointer to a list of arguments instead of an argument list.

The *fmtstr* argument is a pointer to a format string and has the same form and function as the *fmtstr* argument for the **printf** function. Refer to “printf / printf517” on page 252 for a description of the format string. The *argptr* argument points to a list of arguments that are converted and output according the corresponding format specifications in the format.

Return Value: The **vsprintf** function returns the number of characters actually written to the output stream.

See Also: **gets, puts, printf, scanf, sprintf, sscanf, vprintf**

Example:

```
#include <stdio.h>
#include <stdarg.h>

xdata char etxt[30];                    /* text buffer */


void error (char *fmt, ...)  {
  va_list arg_ptr;

  va_start (arg_ptr, fmt);           /* format string */
  vsprintf (etxt, fmt, arg_ptr);
  va_end (arg_ptr);
}

void tst_vprintf (void)  {
  int i;
  i = 1000;

                      /* call error with one parameter */
  error ("Error: '%d' number too large\n", i);

              /* call error with just a format string */
  error ("Syntax Error\n");
}
```

Appendix A. Differences from ANSI C

The C51 compiler differs in only a few aspects from the ANSI C Standard. These differences can be grouped into compiler-related differences and library-related differences.

Compiler-related Differences

- **Wide Characters**
  Wide 16-bit characters are not supported by C51. ANSI provides wide characters for future support of an international character set.
- **Recursive Function Calls**
  Recursive function calls are not supported by default. Functions that are recursive must be declared using the **reentrant** function attribute. Reentrant functions can be called recursively because the local data and parameters are stored in a reentrant stack. In comparison, functions which are not declared using the **reentrant** attribute use static memory segments for the local data of the function. A recursive call to these functions overwrites the local data of the prior function call instance.

Library-related Differences

The ANSI C Standard Library includes a vast number of routines, most of which are included in C51. Many, however, are not applicable to an embedded application and are excluded from the C51 library.

The following ANSI Standard library routines are included in the C51 library:

abs	**cosh**	**isgraph**
acos	**exp**	**islower**
asin	**fabs**	**isprint**
atan	**floor**	**ispunct**
atan2	**free**	**isspace**
atof	**getchar**	**isupper**
atoi	**gets**	**isxdigit**
atol	**isalnum**	**labs**
calloc	**isalpha**	**log**
ceil	**iscntrl**	**log10**
cos	**isdigit**	**longjmp**

A

malloc
memchr
memcmp
memcpy
memmove
memset
modf
pow
printf
putchar
puts
rand
realloc
scanf
setjmp
sin
sinh
sprintf
sqrt
srand
sscanf
strcat
strchr
strcmp
strcpy
strcspn
strlen
strncat
strncmp
strncpy
strpbrk
strrchr
strspn
tan
tanh
tolower
toupper
va_arg
va_end
va_start
vprintf
vsprintf

The following ANSI Standard library routines are not included in the C51 library:

abort
asctime
atexit
bsearch
clearerr
clock
ctime
difftime
div
exit
fclose
feof
ferror
fflush
fgetc
fgetpos
fgets
fmod
fopen
fprintf
fputc
fputs
fread
freopen
frexp
fscanf
fseek
fsetpos
ftell
fwrite
getc
getenv
gmtime
ldexp
ldiv
localeconv
localtime
mblen
mbstowcs
mbtowc
mktime
perror
putc
qsort
raise
remove
rename
rewind
setbuf
setlocale
setvbuf
signal
strcoll
strerror
strftime
strstr
strtod
strtok
strtol
strtoul
strxfrm
system
time
tmpfile
tmpnam
ungetc
vfprintf
wcstombs
wctomb

The following routines are not found in the ANSI Standard Library but are included in the C51 library.

acos517
asin517
atan517
atof517
cabs
chkfloat
cos517
crol
cror
exp517
_getkey
init_mempool
irol
iror
log10517
log517
lrol
lror
memccpy
nop
printf517
scanf517
sin517
sprintf517
sqrt517
sscanf517
strpos
strrpbrk
strrpos
tan517
testbit
toascii
toint
_tolower
_toupper
ungetchar

A

Appendix B. Version Differences

This following appendix lists an overview of major product enhancements and differences between Version 5 and previous versions. The current version of the C51 compiler contains *all* enhancements listed below:

B

Version 4 Differences

- **Byte Order of Floating-point Numbers**
  Floating-point numbers are now stored in the big endian order. Previous releases of the C51 compiler stored floating-point numbers in little endian format. Refer to "Floating-point Numbers" on page 147 for more information.
- **_chkfloat_ Library Function**
  The intrinsic function **_chkfloat_** allows for fast testing of floating-point numbers for error (**NaN**), ±INF, zero and normal numbers. Refer to "_chkfloat_" on page 208 for more information.
- **FLOATFUZZY Directive**
  C51 now supports the **FLOATFUZZY** directive. This directive controls the number of bits ignored during the execution of a floating-point compare. Refer to "FLOATFUZZY" on page 23 for more information.
- **Floating-point Arithmetic is Fully Reentrant**
  Intrinsic floating-point arithmetic operations (add, subtract, multiply, divide, and compare) are now fully reentrant. The C library routines **fpsave** and **fprestore** are no longer needed. Several library routines are also reentrant. Refer to "Routines by Category" on page 182 for more information.
- **Long and Floating-point Operations no Longer use an Arithmetic Stack**
  The long and floating-point arithmetic is more efficient; the code generated is now totally register-based and does not use a simulated arithmetic stack. This also reduces the memory needs of the generated code.
- **Memory Types**
  The memory types have been changed to achieve better performance in the run-time library and to reflect the memory map of the MCS® 251 architecture.

- **Memory Type Bytes for Generic Pointers**
  The memory type bytes used in generic pointers have changed. The following table contains the memory type byte values and their associated memory type.

Memory Type	idata	data	bdata	xdata	pdata	code
C51 V5 Value	0x00	0x00	0x00	0x01	0xFE	0xFF
C51 V4 Value	0x01	0x04	0x04	0x02	0x03	0x05

- **WARNINGLEVEL Directive**
  C51 now supports the **WARNINGLEVEL** directive which lets you specify the strength of the warning detection for the C51 compiler. The C51 compiler now also checks for unused local variables, labels, and expressions. Refer to “WARNINGLEVEL” on page 55 for more information.

Version 3.4 Differences

- **_at_Keyword**
  C51 supports variable location using the **_at_** keyword. This new keyword allows you to specify the address of a variable in a declaration. Refer to "The _at_ Keyword" on page 152 for more information.

- **NOAMAKE Directive**
  C51 now supports the **NOAMAKE** directive. This directive causes C51 to generate object modules without project information and register optimization records. This is necessary only if you want to use object files with older versions of C51 tools. Refer to "NOAMAKE" on page 35 for more information.

- **OH51 Hex File Converter**
  The OHS51 Object-Hex-Symbol Converter provided with prior versions of C51 has been replaced with OH51.

- **Optimizer Level 6**
  C51 now supports optimizer level 6 which provides loop rotation. The resulting code is more efficient and executes faster. Refer to "OPTIMIZE" on page 39 for more information.

- **ORDER Directive**
  When you specify the **ORDER** directive, C51 locates variables in memory in the order in which they are declared in your source file. Refer to "ORDER" on page 42 for more information.

- **REGFILE Directive**
  C51 now supports the **REGFILE** directive which lets you specify the name of the register definition file generated by the linker. This file contains information that is used to optimize the use of registers between functions in different modules. Refer to "REGFILE" on page 47 for more information.

- **vprintf and vsprint Library Functions**
  The **vprintf** and **vsprintf** library functions have been added. Refer to "vprintf" on page 304 and "vsprintf" on page 306 for more information.

Version 3.2 Differences

- **ANSI Standard Automatic Integer Promotion**
  The latest version of the ANSI C Standard requires that calculations use **int** values if **char** or **unsigned char** values might overflow during the calculation. This new requirement is based on the premise that **int** and **char** operations are similar on 16-bit CPUs. C51 supports this feature as the default and provides you with two new control directives, **INTPROMOTE** and **NOINTPROMOTE**, to enable or disable integer promotion.

  There is a big difference between 8-bit and 16-bit operations on the 8-bit 8051 in terms of code size and execution speed. For this reason, you might want to disable integer promotion by using the **NOINTPROMOTE** control directive.

  However, if you wish to retain maximum compatibility with other C compilers and platforms, leave integer promotions enabled.

- **Assembly Source Generation with In-Line Assembly**
  You may use the new control directives **ASM** and **ENDASM** to include source text to output to **.SRC** files generated using the **SRC** command directive.

- **New Control Directives**
  The control directives **ASM**, **ENDASM**, **INTERVAL**, **INTPROMOTE**, **INTVECTOR**, **MAXARGS**, and **NOINTPROMOTE** have been added or enhanced.

- **Offset and Interval Can Now Be Specified for Interrupt Vectors**
  You may now specify the offset and interval for the interrupt vector table. These features provide support for the SIECO-51 derivatives and allow you to specify a different location for the interrupt vector in situations where the interrupt table is not located at address 0000h.

- **Parameter Passing to Indirectly Called Functions**
  Function parameters may now be passed to indirectly called functions if all of the parameters can be passed in CPU registers. These functions do not have to be declared with the reentrant attribute.

- **Source Code Provided For Memory Allocation Functions**
  C source code for the memory allocation routines is now provided with the C51 compiler. You may now more easily adapt these functions to the hardware architecture of your embedded system.

- **Trigraphs**
  C51 now supports trigraph sequences.
- **Variable-length Argument Lists for All Functions**
  Variable-length argument lists are now supported for all function types. Functions with a variable length argument list do not have to be declared using the **reentrant** attribute. The new command line directive **MAXARGS** determines the size of the parameter passing area.

Version 3.0 Differences

- **New Control Directive Added for Assembly Source File Output**
  The **SRC** control directive has been added to direct the compiler to generate an assembly language source file instead of an object file.
- **New Library Functions**
  The library functions **calloc**, **free**, **init_mempool**, **malloc**, and **realloc** have been added.

Version 2 Differences

- **Absolute Register Addressing**
  C51 now generates code that performs absolute register addressing. This improves execution speed. The control directives **AREGS** and **NOAREGS,** respectively, enable or disable this feature.

- **Bit-addressable Memory Type**
  Variable types of **char** and **int** can now be declared to reside in the bit-addressable internal memory area by using the **bdata** memory specifier.

- **Intrinsic Functions**
  Intrinsic functions have been added to the library to support some of the special instructions built in to the 8051.

- **Mixed Memory Models**
  Calls to and from functions of different memory models are now supported.

- **New Optimizer Levels**
  Two new levels of optimization have been added to the C51 compiler. These new levels support register variables, local common subexpression elimination, loop optimizations, and global common subexpression elimination, to name a few.

- **New Predefined Macros**
  The macros **_ _C51_ _** and **_ _MODEL_ _** are now defined by the preprocessor at compile time.

- **Reentrant and Recursive Functions**
  Individual functions may now be defined as being reentrant or recursive by using the **reentrant** function attribute.

- **Registers Used for Parameter Passing**
  C51 now passes up to 3 function arguments using registers. The **REGPARMS** and **NOREGPARMS** directives enable or disable this feature.

- **Support for Memory-specific Pointers**
  Pointers may now be defined to reference data in a particular memory area.

- **Support for PL/M-51 Functions**
  The **alien** keyword has been added to support PL/M-51 compatible functions and function calls.

- **Volatile Type Specifier**
  The **volatile** variable attribute may be used to enforce variable access and to prevent optimizations involving that variable.

Using C51 Version 5 with Previous Versions

You may wish to use the C51 Version 5 with older versions of the 8051 development tools such as BL51, OHS51, or debugging tools and emulators. The new compiler adds object file records for register optimization which makes the object format incompatible with the old tools. However, you can direct the compiler and linker to generate object modules that are compatible with the old tools.

1. Invoke C51 with the control **NOAMAKE** and do not use **REGFILE**.

   or

2. Invoke L51 or BL51 with the control **NOAMAKE**.

If you are using old debugging tools, you may have problems displaying floating-point numbers and pointers. Make sure that you have current versions of the debugging software.

B

Appendix C. Writing Optimum Code

This section lists a number of ways you can improve the efficiency (i.e., smaller code and faster execution) of the 8051 code generated by the C51 compiler. The following is by no means a complete list of things to try. These suggestions in most cases, however, improve the speed and code size of your program.

Memory Model

The most significant impact on code size and execution speed is memory model. Compiling in small model always generates the smallest, fastest code possible. The **SMALL** control directive instructs the C51 compiler to use the small memory model. In small model, all variables, unless declared otherwise, reside in the internal memory of the 8051. Memory access to internal data memory is fast (typically performed in 1 or 2 clock cycles), and the generated code is much smaller than that generated with the compact or large models. For example, the following loop:

```
for (i = 0; i < 100; i++)  {
  do_nothing ();
}
```

is compiled both in small model and in large model to demonstrate the difference in generated code. The following is the small model translation:

```
stmt level   source

   1         #pragma small
   2
   3         void do_nothing (void);
   4
   5
   6         void func (void)
   7         {
   8   1     unsigned char i;
   9   1
  10   1     for (i = 0; i < 100; i++)
  11   1       {
  12   2       do_nothing ();
  13   2       }
  14   1     }
       ; FUNCTION func (BEGIN)
                                  ; SOURCE LINE # 10
0000 E4                 CLR   A
0001 F500    R          MOV   i,A
0003            ?C0001:
0003 E500    R          MOV   A,i
0005 C3                 CLR   C
0006 9464               SUBB  A,#064H
```

```
0008 5007              JNC   ?C0004
                               ; SOURCE LINE # 12
000A 120000  E         LCALL do_nothing
                               ; SOURCE LINE # 13
000D 0500    R         INC   i
000F 80F2              SJMP  ?C0001
                               ; SOURCE LINE # 14
0011           ?C0004:
0011 22                RET
       ; FUNCTION func (END)
```

In small model, the variable **i** is maintained in internal data memory. The instructions to access **i**, **MOV A,i** and **INC i**, require only two bytes each of code space. In addition, each of these instructions executes in only one clock cycle. The total size for the main function when compiled in small model is 11h or 17 bytes.

The following is the same code compiled using the large model:

```
       ; FUNCTION func (BEGIN)
                               ; SOURCE LINE # 10
0000 E4                CLR   A
0001 900000  R         MOV   DPTR,#i
0004 F0                MOVX  @DPTR,A
0005           ?C0001:
0005 900000  R         MOV   DPTR,#i
0008 E0                MOVX  A,@DPTR
0009 C3                CLR   C
000A 9464              SUBB  A,#064H
000C 500B              JNC   ?C0004
                               ; SOURCE LINE # 12
000E 120000  E         LCALL do_nothing
                               ; SOURCE LINE # 13
0011 900000  R         MOV   DPTR,#i
0014 E0                MOVX  A,@DPTR
0015 04                INC   A
0016 F0                MOVX  @DPTR,A
0017 80EC              SJMP  ?C0001
                               ; SOURCE LINE # 14
0019           ?C0004:
0019 22                RET
       ; FUNCTION func (END)
```

In large model, the variable **i** is maintained in external data memory. To access **i**, the compiler must first load the data pointer and then perform an external memory access (see offset 0001h through 0004h in the above listing). These two instructions alone take 4 clock cycles. The code to increment **i** is found from offset 0011h to offset 0016h. This operation consumes 6 bytes of code space and takes 7 clock cycles to execute. The total size for the main function when compiled in small model is 19h or 25 bytes.

Variable Location

Frequently accessed data objects should be located in the internal data memory of the 8051. Accessing the internal data memory is much more efficient than accessing the external data memory. The internal data memory is shared among register banks, the bit data area, the stack, and other user defined variables with the memory type **data**.

Because of the limited amount of internal data memory (128 to 256 bytes), all your program variables may not fit into this memory area. In this case, you must locate some variables in other memory areas. There are two ways to do this.

One way is to change the memory model and let the compiler do all the work. This is the simplest method, but it is also the most costly in terms of the amount of generated code and system performance. Refer to "Memory Model" on page 321 for more information.

Another way to locate variables in other memory areas is to manually select the variables that can be moved into external data memory and declare them using the **xdata** memory specifier. Usually, string buffers and other large arrays can be declared with the **xdata** memory type without a significant degradation in performance or increase in code size.

Variable Size

Members of the 8051 family are all 8-bit CPUs. Operations that use 8-bit types (like **char** and **unsigned char**) are much more efficient than operations that use **int** or **long** types. For this reason, always use the smallest data type possible.

The C51 compiler directly supports all byte operations. Byte types are not promoted to integers unless required. See the **INTPROMOTE** directive for more information.

An example can be illustrated by examining of multiplication operations. The multiplication of two **char** objects is done inline with the 8051 instruction `MUL AB`. To accomplish the same operation with **int** or **long** types would require a call to a compiler library function.

Unsigned Types

The 8051 family of processors does not specifically support operations with signed numbers. The compiler must generate additional code to deal with sign extensions. Far less code is produced if unsigned objects are used wherever possible.

Local Variables

When possible, use local variables for loops and other temporary calculations. As part of the optimization process, the compiler attempts to maintain local variables in registers. Register access is the fastest type of memory access. The best effect is normally achieved with **unsigned char** and **unsigned int** variable types.

Other Sources

The quality of the compiler generated code is more often than not directly influenced by the algorithms implemented in the program. Sometimes, you can improve the performance or reduce the code size simply by using a different algorithm. For example, a heap sort algorithm always outperforms a bubble sort algorithm.

For more information on how to write efficient programs, refer to the following books:

The Elements of Programming Style, Second Edition
Kernighan & Plauger
McGraw-Hill
ISBN 0-07-034207-5

Writing Efficient Programs
Jon Louis Bentley
Prentice-Hall Software Series
ISBN 0-13-970244-X

Efficient C
Plum & Brodie
Plum Hall, Inc.
ISBN 0-911537-05-8

Appendix D. Compiler Limits

The C51 compiler embodies some known limitations that can be arranged into two distinct categories:

- Limitations of the compiler implementation
- Limitations of the Intel Object Module Format (OMF-51)

For the most part, there are no limits placed on the compiler with respect to components of the C language; for example, you may specify an unlimited number of symbols or number of **case** statements in a **switch** block. If there is enough address space, several thousand symbols could be defined. However, at this time, C51 is bound by a historical limit of 256 global symbols.

Limitations of the C51 Compiler Implementation

- A maximum of 19 levels of indirection (access modifiers) to any standard data type are supported. This includes array descriptors, indirection operators, and function descriptors.
- Number of functions in a module (see OMF-51 Limitation values).
- Names can be up to 255 characters long. However, only the first 32 are significant. The C language provides for case sensitivity in regard to function and variable names. However, for compatibility reasons, all names in the object file appear in capital letters. It is therefore irrelevant if an external object name within the source program is written in capital or small letters.
- The maximum number of **case** statements in a **switch** block is not fixed. Limits are imposed only by the available memory size and the maximum size of individual functions.
- The maximum number of nested function calls in an invocation parameter list is 10.
- The maximum number of nested include files is 9. This value is independent of list files, preprocessor files, or whether or not an object file is to be generated.
- The maximum depth of directives for conditional compilation is 20. This is a preprocessor limitation.

- Instruction blocks ({…}) may be nested up to 15 levels deep.
- Macros may be nested up to 8 levels deep.
- A maximum of 32 parameters may be passed in a macro or function call.
- The maximum length of a line or a macro definition is 2000 characters. Even after a macro expansion, the result may not exceed 2000 characters.

Limitations of the Intel Object Module Format

- There may be a maximum of 255 segments. The number of functions that may exist in a module is difficult to calculate. Each function definition in a source program module receives a separate code segment. If local variables exist within the function, a separate data segment is also created. If **bit** variables exist within the function, a separate **bit** segment is created too. For these reasons, the number of functions that may exist within a module depends upon the number of variables in the functions.
- There may be a maximum of 256 external symbols. All module names which have the memory class extern, are contained in this external symbol class. The compiler produces external names for external functions, that are dependent upon whether or not bit or data parameters are contained in the function call. Thus, the reference name to the external data and bit segments is produced in a manner analogous to the global functions.

Appendix E. Byte Ordering

Most microprocessors have a memory architecture that is composed of 8-bit address locations known as bytes. Many data items (addresses, numbers, and strings) are too long to be stored using a single byte and must be stored in a series of consecutive bytes.

When using data that are stored in multiple bytes, byte ordering becomes an issue. Unfortunately, there is not just one standard for the order in which bytes in multi-byte data are stored. There are two popular methods of byte ordering currently in widespread use.

The first method is called little endian and is often referred to as Intel order. In little endian, the least significant, or low-order byte is stored first. For example, a 16-bit integer value of 0x1234 (4660 decimal) would be stored using the little endian method in two consecutive bytes as follows:

Address	+0	+1
Contents	0x34	0x12

A 32-bit integer value of 0x57415244 (1463898692 decimal) would be stored using the little endian method as follows:

Address	+0	+1	+2	+3
Contents	0x44	0x52	0x41	0x57

A second method of accessing multi-byte data is called big endian and is often referred to as Motorola order. In big endian, the most significant, or high-order byte is stored first, and the least significant, or low-order byte is stored last. For example, a 16-bit integer value of 0x1234 would be stored using the big endian method in two consecutive bytes as follows:

Address	+0	+1
Contents	0x12	0x34

A 32-bit integer value of 0x004A4F4E would be stored using the big endian method as follows:

Address	+0	+1	+2	+3
Contents	0x00	0x4A	0x4F	0x4E

The 8051 is an 8-bit machine and has no instructions for directly manipulating data objects that are larger than 8 bits. Multi-byte data are stored according to the following rules.

- The 8051 **LCALL** instruction stores the address of the next instruction on the stack. The address is pushed onto the stack low-order byte first. The address is, therefore, stored in memory in little endian format.
- All other 16-bit and 32-bit values are stored, contrary to other Intel processors, in big endian format, with the high-order byte stored first. For example, the **LJMP** and **LCALL** instructions expect 16-bit addresses that are in big endian format.
- Floating-point numbers are stored according to the IEEE-754 format and are stored in big endian format with the high-order byte stored first.

If your 8051 embedded application performs data communications with other microprocessors, it may be necessary to know the byte ordering method used by the other CPU. This is certainly true when transmitting raw binary data.

Appendix F. Hints, Tips, and Techniques

This section lists a number of illustrations and tips which commonly require further explanation. Items in this section are listed in no particular order and are merely intended to be referenced if you experience similar problems.

Recursive Code Reference Error

The following program example:

```
#pragma code symbols debug oe

void func1(unsigned char *msg ) { ; }

void func2( void ) {
  unsigned char uc;
  func1("xxxxxxxxxxxxxx");
}

code void (*func_array[])() = { func2 };

void main( void ) {
  (*func_array[0])();
}
```

when compiled and linked using the following command lines:

```
C51 EXAMPLE1.C

BL51 EXAMPLE1.OBJ IX
```

fails and display the following error message.

```
*** WARNING 13:  RECURSIVE CALL TO SEGMENT
    SEGMENT:  ?CO?EXAMPLE1
    CALLER:   ?PR?FUNC2?EXAMPLE1
```

In this program example, **`func2`** defines a constant string (**`"xxx…xxx"`**) which is directed into the constant code segment ?CO?EXAMPLE1. The definition **`code void (*func_array[])() = { func2 };`** yields a reference between segment **?CO?EXAMPLE1** (where the code table is located) and the executable code segment **?PR?FUNC2?EXAMPLE1**. Because **`func2`** also refers to segment **?CO?EXAMPLE1**, BL51 assumes that there is a recursive call.

To avoid this problem, link using the following command line:

```
BL51 EXAMPLE1.OBJ IX OVERLAY &
(?CO?EXAMPLE1 ~ FUNC2, MAIN ! FUNC2)
```

?CO?EXAMPLE1 ~ FUNC2 deletes the implied call reference between **func2** and the code constant segment in the example. Then, **MAIN ! FUNC2** adds an additional call to the reference listing between **MAIN** and **FUNC2** instead. Refer to the *8051 Utilities User's Guide* for more information.

In summary, automatic overlay analysis cannot be successfully accomplished when references are made via pointers to functions. References of this type must be manually implemented, as in the example above.

Problems Using the printf Routines

The **printf** functions are implemented using a variable-length argument list. Arguments specified after the format string are passed using their inherent data type. This can cause problems when the format specification expects a data object of a different type than was passed. For example, the following code:

```
printf ("%c %d %u %bu", 'A', 1, 2, 3);
```

does *not* print the string "A 1 2 3". This is because the C51 compiler passes the arguments **1**, **2**, and **3** all as 8-bit byte types. The format specifiers **%d** and **%u** both expect 16-bit int types.

To avoid this type of problem, you must explicitly define the data type to pass to the **printf** function. To do this, you must type cast the above values. For example:

```
printf ("%c %d %u %bu", 'A',(int) 1, (unsigned int) 2, (char) 3);
```

If you are uncertain of the size of the argument that is passed, you may cast the value to the desired size.

Uncalled Functions

It is common practice during the development process to write but not call additional functions. While the compiler permits this without error, the Linker/Locator does not treat this code casually, because of the support for data overlaying, and emits a warning message.

Interrupt functions are never called, they are invoked by the hardware. An uncalled routine is treated as a potential interrupt routine by the linker. This means that the function is assigned non-overlayable data space for its local variables. This quickly exhausts all available data memory (depending upon the memory model used).

If you unexpectedly run out of memory, be sure to check for linker warnings relating to uncalled or unused routines. You can use the linker's **IXREF** control directive to include a cross reference list in the linker map (**.M51**) file.

Trouble with the bdata Memory Type

Some users have reported difficulties in using the **bdata** memory type. Using **bdata** is similar to using the **sfr** modifier. The most common error is encountered when referencing a **bdata** variable defined in another module. For example:

```
extern bdata char xyz_flag;

sbit xyz_bit1 = xyz_flag^1;
```

In order to generate the appropriate instructions, the compiler must have the absolute value of the reference to be generated. In the above example, this cannot be done, as this address of **`xyz_flag`** cannot be known until after the linking phase has been completed. Follow the rules below to avoid this problem.

1. A **bdata** variable (defined and used in the same way as an **sfr**) must be defined in global space; not within the scope of a procedure.

2. A **bdata bit** variable (defined and used in the same way as an **sbit**) must also be defined in global space, and cannot be located within the scope of a procedure.

3. The definition of the **bdata** variable and the creation of its **sbit** access component name must be accomplished where the compiler has a "view" of both the variable and the component.

For example, declare the bdata variable and the bit component in the same source module:

```
bdata char xyz_flag;
sbit xyz_bit1 = xyz_flag^1;
```

Then, declare the bit component external:

```
extern bit xyz_bit1;
```

As with any other declared and named C variable that reserves space, simply define your **bdata** variable and its component **sbits** in a module. Then, use the **extern bit** specifier to reference it as the need arises.

Using Monitor-51

If you want to test a C program with Monitor-51 and if the Monitor-51 is installed at code address 0, consider the following rules (the specification refers to a target system where the available code memory for user programs starts at address 8000H):

- All C modules which contain interrupt functions must be translated with the control directive INTVECTOR (0x8000).
- In the file STARTUP.A51 (directory: LIB) the statement CSEG AT 0 must be replaced with CSEG AT 8000H. The this file must be assembled and added to the linker/locator invocation according the specifications in the file header.

Function Pointers

Function pointers are one of the most difficult aspects of C to understand and to properly utilize. Most problems involving function pointers are caused by improper declaration of the function pointer, improper assignment, and improper dereferencing.

The following brief example demonstrates how to declare a function pointer (f), how to assign function addresses to it, and how to call the functions through the pointer. The **printf** routine is used for example purposes when running DS51 to simulate program execution.

```
#pragma code symbols debug oe

#include <reg51.h>             /* special function register declarations */
#include <stdio.h>           /* prototype declarations for I/O functions */

void func1(int d) {                                        /* function #1 */
  printf("In FUNC1(%d)\n", d);
}

void func2(int i) {                                        /* function #2 */
  printf("In FUNC2(%d)\n", i);
}

void main(void) {
  void (*f)(int i);                /* Declaration of a function pointer */
                                    /* that takes one integer arguments */
                                                 /* and returns nothing */

  SCON  = 0x50;                  /* SCON: mode 1, 8-bit UART, enable rcvr */
  TMOD |= 0x20;                    /* TMOD: timer 1, mode 2, 8-bit reload */
  TH1   = 0xf3;                       /* TH1:  reload value for 2400 baud */
  TR1   = 1;                                        /* TR1:  timer 1 run */
  TI    = 1;                   /* TI:   set TI to send first char of UART */

  while( 1 ) {
    f = (void *)func1;                         /* f points to function #1 */
    f(1);
    f = (void *)func2;                         /* f points to function #2 */
    f(2);
  }
}
```

NOTE

Because of the limited stack space of the 8051, the linker overlays function variables and arguments in memory. When you use a function pointer, the linker cannot correctly create a call tree for your program. For this reason, you may have to correct the call tree for the data overlaying. Use the ***OVERLAY*** *directive with the linker to do this. Refer to the 8051 Utilities User's Guide for more information.*

Glossary

A51
The command used to assemble programs using the 8051 Macro Assembler.

aggregate types
Arrays, structures, and unions.

ANSI
American National Standards Institute. The organization responsible for defining the C language standard.

argument
The value that is passed to a macro or function.

arithmetic types
Data types that are integral, floating-point, or enumerations.

array
A set of elements all of the same data type.

ASCII
American Standard Code for Information Interchange. This is a set of 256 codes used by computers to represent digits, characters, punctuation, and other special symbols.

basename
The part of the file name that excludes the drive letter, directory name, and file extension. For example, the basename for the file **`C:\C51\SAMPLE\SIO.C`** is **`SIO`**.

batch file
A text file that contains MS-DOS commands and programs that can be invoked from the command line.

BL51
The command used to link object files and libraries using the 8051 Code-Banking Linker/Locator.

block
A sequence of C statements, including definitions and declarations, enclosed within braces ({ }).

C51
The command used to compile programs using the 8051 Optimizing C Cross Compiler.

constant expression
Any expression that evaluates to a constant non-variable value. Constants may include character, integer, enumeration, and floating-point constant values.

declaration
A C construct that associates the attributes of a variable, type, or function with a name.

definition
A C construct that specifies the name, formal parameters, body, and return type of a function or that initializes and allocates storage for a variable.

directive
An instruction to the C preprocessor or a control switch to the C51 compiler.

disk cache
A software program usually installed as a TSR or device driver that buffers disk I/O operations in memory in an attempt to improve system performance by satisfying disk reads from the memory buffer.

DS51
The command used to load and execute the 8051 Simulator/Debugger.

environment table
The memory area used by MS-DOS to store environment variables and their values.

environment variable
A variable stored in the environment table. These variables provide MS-DOS programs with information such as where to find include files and library files.

escape sequence
A backslash (**'\'**) character followed by a single letter or a combination of digits that specifies a particular character value in strings and character constants.

expression
A combination of any number of operators and operands that produces a constant value.

formal parameters
The variables that receive the value of arguments passed to a function.

function
A combination of declarations and statements that can be called by name that perform an operation and/or return a value.

function body
A block that contains the declarations and statements that make up a function.

function call
An expression that invokes and possibly passes arguments to a function.

function declaration
A declaration that provides the name and return type of a function that is explicitly defined elsewhere in the program.

function definition
A definition that provides the name, formal parameters, return type, declarations, and statements that define what a function does.

function prototype
A function declaration that includes the list of formal parameters in parentheses following the function name.

in-circuit emulator (ICE)
A hardware device that aids in debugging embedded software by providing hardware-level single-stepping, tracing, and break-pointing. Some ICEs provide a trace buffer that stores the most recent CPU events.

include file
A text file that is incorporated into a source file using the **#include** preprocessor directive.

keyword
A reserved word with a predefined meaning for the compiler.

L51
The command used to link object files and libraries using the 8051 Linker/Locator.

LIB51
The command used to manipulate library files using the 8051 Library Manager.

library
A file that stores a number of possibly related object modules. The linker can extract modules from the library to use in building a target object file.

LSB
Least significant bit or byte.

macro
An identifier that represents a series of keystrokes that is defined using the **#define** preprocessor directive.

manifest constant
A macro that is defined to have a constant value.

MCS-51
The general name applied to the Intel family of 8051 compatible microprocessors.

memory manager
Any of the programs that utilize the extended memory of the 80386 and 80486 CPUs to reduce system overhead and provide convenient means of accessing the different types of memory on IBM AT/286/386 based machines or 100% compatibles.

memory model
Any of the models that specifies which memory areas are used for function arguments and local variables.

monitor51
An 8051 program that can be loaded into your target CPU to aid in debugging and rapid product development through rapid software downloading.

MSB
Most significant bit or byte.

newline character
The character used to mark the end of a line in a text file or the escape sequence (**'\n'**) used to represent the newline character.

null character
The ASCII character with the value 0 represented as the escape sequence (**'\0'**).

null pointer
A pointer that references nothing and has an offset of 0000h. A null pointer has the integer value 0.

object
An area of memory that can be examined. Usually used when referring to the memory area associated with a variable or function.

object file
A file, created by the compiler, that contains the program segment information and relocatable machine code.

OH51
The command used to convert absolute object files into Intel HEX file format using the Object File Converter.

operand
A variable or constant that is used in an expression.

operator
A symbol (e.g., +, -, *, /) that specifies how to manipulate the operands of an expression.

parameter
The value that is passed to a macro or function.

PL/M-51
A high-level programming language that provides a blocked structure, a facility for data structures, type checking, and a standard language for use on most Intel hardware architectures.

pointers
A variable that contains the address of another variable, function, or memory area.

pragma
A statement that passes an instruction to the compiler at compile time.

preprocessor
The compiler's first pass text processor that manipulates the contents of a C file. The preprocessor defines and expands macros, reads include files, and passes control directives to the compiler.

RAM disk
A memory area used by a device drive or TSR that emulates a disk drive, but provides much faster access.

relocatable
Able to be moved or relocated. Not containing absolute or fixed addresses.

RTX51 Full
An 8051 Real-TIME Executive that provides a multitasking operating system kernel and library of routines for its use.

RTX51 Tiny
A limited version of RTX51.

scalar types
In C, integer, enumerated, floating-point, and pointer types.

scope
The sections or a program where an item (function or variable) can be referenced by name. The scope of an item may be limited to file, function, or block.

source file
A text file containing C program code.

stack
An area of memory, indirectly accessed by a stack pointer, that shrinks and expands dynamically as items are pushed onto the stack and popped off of the stack. Items in the stack are removed on a LIFO (last-in first-out) basis.

static
A storage class that, when used with a variable declaration in a function, causes variables to retain their value after exiting the block or function in which they are declared.

stream functions
Routines in the library that read and write characters using the input and output streams.

string
An array of characters that is terminated with a null character (**'\0'**).

string literal
A string of characters enclosed within double quotes (**" "**).

structure
A set of elements of possibly different types grouped together under one name.

structure member
One element of a structure.

token
A fundamental symbol that represents a name or entity in a programming language.

TS51
The command used to load and execute the 8051 Target Debugger.

two's complement
A binary notation that is used to represent both positive and negative numbers. Negative values are created by complementing all bits of a positive value and adding 1.

type
A description of the range of values associated with a variable. For example, an **int** type can have any value within its specified range (-32768 to 32767).

type cast
An operation in which an operand of one type is converted to another type by specifying the desired type enclosed within parentheses immediately preceding the operand.

whitespace character
Characters that are used as delimiters in C programs such as space, tab, and newline.

wild card
One of the MS-DOS characters (? or *) that can be used in place of characters in a filename.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X