KEIL SOFTWARE

C251 Compiler

Optimizing C Compiler and Library Reference for the MCS® 251 Architecture

User's Guide 09.97

Preface

This manual describes how to use the C251 Optimizing C Compiler to compile C programs for your target MCS® 251 environment. The C251 compiler package can be used on all MCS® 251 family processors and is executable under MS-DOS or Windows MS-DOS box. This manual assumes that you are familiar with the MS–DOS commands, know how to program 251 processors, and have a working knowledge of the C language.

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 the C251 compiler within an integrated development environment, examples in this manual are universal in that they apply to all programming environments.

NOTE
This manual describes the features of C251 Compiler Version 2 or higher. It does not refers also to C51 Compiler and C251 Compiler Version 1, but does not contain detailed information for C251 Version 1.

Manual Organization

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

"Chapter 1. Introduction," gives an overview about the operation and features of the C251 compiler.

"Chapter 2. Compiling with C251," explains how to compile a source file using the C251 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 251 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 C251 preprocessor and includes examples.

"Chapter 5. 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 C251 with other 251 and 8051 programming languages.

"Chapter 6. Error Messages," lists the fatal errors, syntax errors, and warnings that you may encounter while using the C251 compiler.

"Chapter 7. Library Reference," provides you with extensive C251 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** **L251.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: C251 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 C251 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. The C251 compiler is not a universal C compiler adapted for the 251 target. It is a ground-up implementation dedicated to generating extremely fast and compact code for the 251 microprocessor family. The C251 compiler 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 the C251 compiler 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 91 for more detailed information.

Books About the C Language

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

Overview of the C251 Compiler

The Keil C251 Compiler is a dedicated ANSI C compiler designed explicitly for the MCS® 251 microcontroller family. The C251 compiler is an extended ANSI C compiler which allows full access to all resources in a 251 microcontroller system. You may re-compile your existing C51 code with the C251 compiler.

Data Types

The C251 compiler supports the following data types.

Data Type	Bits	Bytes	Value Range
Bit	1		0 or 1
signed char	8	1	-128 to +127
Unsigned char	8	1	0 to 255
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
Double	64	8	±1.7E-308 to ±1.7E+308
Pointer		1, 2, 3, or 4	Object address
Data Types for SFR access:			
Sbit	1		0 or 1
Sfr	8	1	0 to 255
sfr16	16	2	0 to 65535

Memory Types

The C251 compiler provides full support for the 251 architecture and can access all system components. Each variable can be explicitly located anywhere in the 251 address space. The linear 16 Mbyte address space can be accessed with many addressing modes. In addition, all addressing modes of the 8051 are fully supported by the 251.

Memory Type	251 Address Space
near	64 Kbyte direct and indirect memory addressing.
far	16 Mbyte indirect memory addressing; object size < 64 Kbytes.
huge	16 Mbyte indirect memory addressing; any object size.
data	Direct memory addressing for on-chip RAM (128 bytes); fast 8-bit accesses.
bdata	Bit-addressable RAM; mixed bit and byte accesses (16 bytes).
ebdata, **ebit**	Extended bit-addressable RAM; mixed bit and byte accesses.
idata	Indirect memory addressing for on-chip RAM (256 bytes); access with **MOV @Ri**.
pdata	Paged **XDATA** memory (256 bytes); access with **MOVX @Ri**.
xdata	**XDATA** memory (64 Kbytes); access with **MOVX @DPTR**.
code	**CODE** memory (64 Kbytes); access with **MOVC**.

Memory Models

The memory model determines the default memory selector used for automatic variables and parameter passing areas. With the **HOLD** directive you can specify additional memory selectors for small objects, for example, the following command line:

```
C251 PROG.C HOLD (2, 4, 8)
```

directs the C251 compiler to locate global variables 2 bytes in size or smaller in **data** memory; variables 3 or 4 bytes in size in **near** memory; and variables 5 to 8 bytes in size in **xdata** memory.

The following table lists the memory areas used for each memory model.

Memory Model	Parameters & Automatic Variables	Default Global Variables	Default Constant Variables	Default Pointer Definition	Default Pointer Size
TINY	**data**	**data**	**near**	**near ***	2 bytes
XTINY	**near**	**near**	**near**	**near ***	2 bytes
SMALL	**data**	**data**	**code**	**far ***	4 bytes
XSMALL	**near**	**near**	**code**	**far ***	4 bytes
LARGE	**xdata**	**xdata**	**code**	**far ***	4 bytes

Program Size

The MCS® 251 microcontroller family allows program sizes up to 16 Mbytes. The generated 251 code can be optimized by using specific **JMP** and **CALL** instructions. The **ROM** directive lets you choose the combination of **JMP** and **CALL** instructions that is used.

ROM Directive	JMP Instruction	CALL Instruction Inside the Module	CALL Instruction extern Functions
SMALL	**AJMP**	**ACALL**	**ACALL**
MEDIUM	**AJMP**	**ACALL**	**LCALL**
COMPACT	**AJMP**	**LCALL**	**LCALL**
LARGE	**LJMP**	**LCALL**	**LCALL**
HUGE	**LJMP**	**LCALL / ECALL**	**ECALL**

Register Optimization

Depending on the program context, the C251 compiler allocates up to 24 CPU registers for register variables. Any registers modified during function execution are noted within each module. The linker/locator generates a global, project-wide register file which contains information about the registers altered by external functions. Consequently, the C251 compiler *knows* the registers used by each function in an application. With this information, the C251 compiler can optimize the overall CPU register allocation of those functions.

Registers R0-R7 and R11 are used for parameter passing. This technique yields very efficient code that compares favorably to assembly programming. Additional parameters are passed via fixed memory locations or the 251's hardware stack.

Reentrant Code

The 251 supports stack-based variable addressing. This permits the C251 compiler to support fast reentrant functions. The **#pragma functions (reentrant)** and **#pragma functions (static)** preprocessor directives control code generation. Non-reentrant code stores variables in directly addressable memory locations and yields the fastest program execution. Data overlaying considerably reduces the memory requirements in C applications of this type.

Registerbanks

C251 allows direct to control the usage of the four CPU register banks which can be used for example in connection with interrupt procedures. The code generated by C251 compiler including all library functions is independent from the registerbank currently selected. Almost all C251 run-time library functions are reentrant. This allows that the main program and one or more interrupt service routines can call simultaneously the same C251 run-time functions or C251 user functions which are translated with the **reentrant** attribute.

Interrupt Functions

The C251 compiler, and is language extensions, allow the programmer to exercise complet control over all aspects of interrupt and registerbank usage. Such support allows the system programmer to create efficient interrupt procedures. The user need only be concerned with the interrupt and necessary register bank switch-over operation, in a general and high level manner. The C251 compiler generates only the code necessary to effect the most expedient handling. If you are coding very simple interrupt routines it is possible to omit the **using** attribute for the register bank. In this case only the registers altered by the interrupt function are saved on the system stack, which gives very efficient code on small interrupt functions.

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 */
  }
}
```

C Run-Time Library

The run-time libraries provided with the C251 compiler contain over 100 routines, all of which are reentrant. Source code for I/O and memory allocation functions is also included.

Listing File Example

The C251 compiler produces a listing file that contains source code, directive information, an assembly listing, and a symbol table.

```
C251 COMPILER  V2.1,  SAMPLE           14/08/97  13:01:59  PAGE 1

DOS C251 COMPILER V2.11, COMPILATION OF MODULE SAMPLE
OBJECT MODULE PLACED IN SAMPLE.OBJ
COMPILER INVOKED BY: C:\C251\BIN\C251V2.EXE SAMPLE.C CODE

stmt  level    source
    1          #include <reg251s.h>  /* SFRs for the 251Sx CPU */
    2
    3          unsigned long outsqr (
    4            unsigned long num,
    5            unsigned int  power)   {
    6   1
    7   1        unsigned long result = 1;
    8   1
    9   1        do  {
   10   2          result *= num;
   11   2          power--;
   12   2        } while (power);
   13   1        return (result);
   14   1      }

ASSEMBLY LISTING OF GENERATED OBJECT CODE
;       FUNCTION outsqr (BEGIN)
000000 7D41           MOV      WR8,WR2
;---- Variable 'power' assigned to Register 'WR8' ----
000002 7F61           MOV      DR24,DR4
;---- Variable 'num' assigned to Register 'DR24' ----
                                             ; SOURCE LINE # 5
                                             ; SOURCE LINE # 7
000004 7E780001       MOV      DR28,#01H
;---- Variable 'result' assigned to Register 'DR28' ----
                                             ; SOURCE LINE # 9
               ?C0001:
                                             ; SOURCE LINE # 10
000008 7F17           MOV      DR4,DR28
00000A 7F06           MOV      DR0,DR24
00000C 120000      E  LCALL    ?C?LMUL
00000F 7F71           MOV      DR28,DR4
                                             ; SOURCE LINE # 11
000011 1B44           DEC      WR8,#01H
                                             ; SOURCE LINE # 12
000013 78F3           JNE      ?C0001
                                             ; SOURCE LINE # 13
000015 7F17           MOV      DR4,DR28
                                             ; SOURCE LINE # 14
000017 22             RET
;       FUNCTION outsqr (END)

Module Information          Static   Overlayable
------------------------------------------------
  code size            =        24     ------
  ecode size           =    ------     ------
  data size            =    ------     ------
  idata size           =    ------     ------
  pdata size           =    ------     ------
  xdata size           =    ------     ------
  xdata-const size     =    ------     ------
  edata size           =    ------     ------
  bit size             =    ------     ------
  ebit size            =    ------     ------
  bitaddressable size  =    ------     ------
  ebitaddressable size =    ------     ------
  far data size        =    ------     ------
  huge data size       =    ------     ------
  const size           =    ------     ------
  hconst size          =    ------     ------
End of Module Information.

C251 COMPILATION COMPLETE.  0 WARNING(S),  0 ERROR(S)
```

The C251 compiler produces a listing file with page numbers as well as time and date of the compilation. Remarks about the compiler invocation and object file output are displayed in this listing.

The listing includes a line number for each statement and a nesting level for each block enclosed within curly braces (‘{‘ and ‘}’).

Error messages and warning messages are included in the listing file.

The **CODE** compiler option includes an assembly code listing in the listing file. Source line numbers are embedded within the generated code.

A memory overview provides information about the 251 memory areas that are used.

The total number of errors and warnings is stated at the end of the listing file.

L251 Code Banking Linker/Locator

The L251 linker/locator is a code banking linker for 251-based tools. The L251 linker lets you combine object modules created Keil 251 tools with object modules created with Keil or Intel 8051 tools. The L251 linker/locator combines one or more object modules into a single executable 251 program. The linker also resolves external and public references, and assigns absolute addresses to relocatable programs segments.

The L251 linker/locator processes object modules created by the Keil C51 compiler, C251 compiler, A51 assembler, and A251 assembler and also processes object modules created by the Intel PL/M-51 compiler and ASM-51 assembler. The linker automatically selects the appropriate run-time library and links only the library modules that are required.

Normally, you specify the names of the object modules to combine. The default settings for the linker have been chosen to accommodate most applications without the need to specify additional directives. However, it is easy for you to specify custom settings for your application.

Address Management

The C251 compiler assigns each code and data segment to a specific memory class name. The class name refers to the different memory areas: for example the class **NCONST** contains constants which must be directed to ROM space in the memory region 0x000000-0x00FFFF; the class **EDATA** contains variable sections which must be directed to RAM space in the memory area 0x000000-0x00FFFF. Refer to the *L251 Linker/Locator User's Guide, Locating Segments* for more information about 251 memory classes.

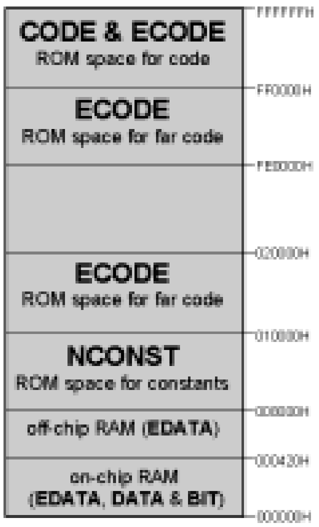
During the link/locate process it is simple to direct **all** segments with the same class name to the specific memory area. However, also the segment can be reference which allows exact placing of the specified segment to an absolute memory location.

Example

```
L251  SAMPLE.OBJ  CLASSES (NCONST (4000H-0FFFFH))
```

With this L251 invocation the area for near const (**NCONST**) is defined in the address space 4000h-0FFFFh. L251 locates all segments with the class name NCONST to this memory area which is usually an EPROM space.

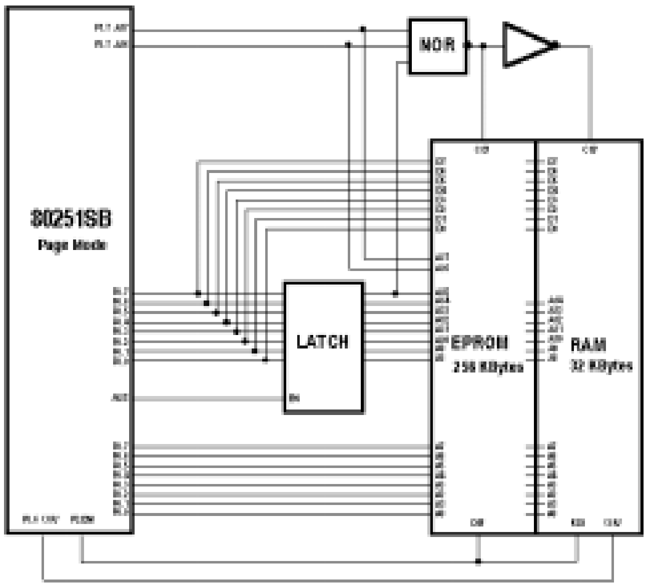
Sample 251 Hardware



The **TINY** and **XTINY** memory models of the C251 compiler place constants like implicit text strings into the memory class NCONST. This has the advantage that you can access all objects with 2-byte near pointers and that you can access constant objects with direct 16-bit addressing.

However, the NCONST memory class must be located in the address range 0x000000-0x00FFFF. This requires that you have ROM space somewhere in the lower 64KB area.

The memory map for this hardware design is shown on the right and a simple schematic is shown below.



The hardware design shown above provides 32KB RAM and a total of up to 224KB EPROM space. For this hardware design, the most efficient memory model is **TINY** or **XTINY**. You can use this memory model together with the C251 directive **ROM (HUGE)** which allows program sizes of up to 16 Mbytes.

The following CLASSES linker directive locates the memory classes appropriately for ths memory layout of this application.

```
L251  ....
  CLASSES (
     EDATA  (0-7FFFH),
     NCONST (8000H – 0FFFFH),
     ECODE  (8000H – 1FFFFH,
             0FE0000H – 0FFFFFFH))
```

Chapter 2. Compiling with C251

This chapter explains how to use the C251 compiler to compile C source files and discusses the control directives you may specify. These directives allow you to:

- Direct the C251 compiler 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	**\C251\BIN**	This environment variable specifies the path of the C251 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.
C251INC	**\C251\INC**	This environment variable specifies the location of the standard C251 include files. The compiler accepts a maximum of 5 path declarations. If more then one path declaration is required, the path names must be separated by semicolons: Example: **SET C251INC=C:\C251\INC;C:\CHIP_DIR** If you specify #include ”filename.h”; the C251 Compiler searches first the current directory and the directory of the source file. When this fails or when #include <filename.h> is used, the paths specified by the INCDIR directive are searched. When this still fails, the paths specified by the C251INC environment variable are used.
C251LIB	**\C251\LIB**	This environment variable specifies the location of the standard C251 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:\C251\BIN
SET TMP=D:\
SET C251INC=C:\C251\INC
SET C251LIB=C:\C251\LIB
```

Running C251

To invoke the C251 compiler, type **C251** 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 C251 compiler command line is:

C251 ***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 17 for a detailed list of the available directives.

The following command line example invokes the C251 compiler, specifies the source file **SAMPLE.C**, and uses the controls **DEBUG**, **CODE**, and **PREPRINT**.

```
C251 SAMPLE.C DEBUG CODE PREPRINT
```

The C251 compiler displays the following information upon successful invocation and compilation.

```
MS-DOS C251 COMPILER V2.1

C251 COMPILATION COMPLETE.  0 WARNING(S),  0 ERROR(S)
```

DOS ERRORLEVEL

After compilation, the number of errors and warnings detected is output to the screen. The C251 compiler 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.

C251 Output Files

The C251 compiler 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 L251 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 A251 assembler. See the **SRC** directive in the following sections for more information.

Control Directives

The C251 compiler 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

```
C251  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.

NOTES
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.

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

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 & Abbreviation	Class	Description
ASM, ENDASM	Source	Insert 251 assembly instructions into a **.SRC** file.
[NO]ASMEXPAND, [NO]AE	Source	Specify preprocessor expansion within __asm{} blocks and #pragma asm / endasm sections.
[NO]CASE †	Object	Specify case sensitivity of identifiers for the object file
BROWSE, BR †	Object	Enable generation of browser information.
CODE, CD †	Listing	Add an assembly listing to the listing file.
[NO]COND, [NO]CO	Listing	List/do not list source lines skipped from the preprocessor.
DEBUG, DB †	Object	Include debugging information in the object file.
DEFINE, DF	Source	Define preprocessor names in the C251 invocation line.
DISABLE	Object	Disable interrupts for a function.
EJECT, EJ	Listing	Insert a page break into the listing file.
FIXDRK, FD †	Object	Bypass the INC DRk,#const chip bug in C-step devices.
FLOAT64, F64 †	Object	Select double precision floating point and libraries.
FLOATFUZZY, FF	Object	Specify number of bits rounded during floating compare.
FUNCTIONS, FC	Object	Select default function attributes for functions.
HOLD, HL †	Object	Specify size limits for global variable locations.
INCDIR, ID †	Source	Specify additional path names for include files.
INTERVAL, IT †	Object	Specify the interrupt vector interval.
INTR2, I2 †	Object	Specify that C251 uses 8051 compatible interrupt frames.
[NO]INTVECTOR, [NO]IV †	Object	Specify base address for interrupt vectors or disable vectors.
LARGE, LA †	Object	Select **LARGE** memory model.

Directive & Abbreviation	Class	Description
LISTINCLUDE, LC	Listing	Display contents of include files in the listing file.
MAXARGS, MA †	Object	Specify size of variable argument lists.
MODBIN, MB †	Object	Select MCS® 251 binary mode.
NAME, NA †	Object	Override the default module name.
NOALIAS, NOAL †	Object	Disable alias checking for pointer access optimization.
NOAMAKE, NOAM †	Object	Disable information records for AutoMAKE.
NOCASE †	Object	Convert indentifiers to upper case for the object file.
NOEXTEND, NOEXT †	Source	Disable C251 extensions to ANSI C.
[NO]OBJECT, [NO]OJ †	Object	Specify a name for or suppress the object file.
OPTIMIZE, OT	Object	Specify the level of optimization performed by C251.
ORDER †	Object	Variables are allocated in the order in which they appear in the source file.
PACK	Object	Control the packing of structure types.
PAGELENGTH, PL †	Listing	Specify number of rows on the page.
PAGEWIDTH, PW †	Listing	Specify number of columns on the page.
PARM251, P251	Object	Use C251 parameter passing conventions (default).
PARM51, P51	Object	Use C51 parameter passing conventions.
PREPRINT, PP †	Listing	Produce a preprocessor listing file where all macros are expanded.
PREPRINTONLY, PPO †	Source	Create a preprocessor output file only, no translation and code generation takes place.
PRINT, NOPRINT †	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.
[NO]REGPARMS	Object	Control register parameter passing (default: enabled).
RENAMECODE, RC	Object	Rename a code segment.
ROM †	Object	Control generation of **JMP**/**CALL** instructions.
SAVE / RESTORE	Object	Save or restore current settings.
SMALL, SM †	Object	Select **SMALL** memory model.
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.
TINY, TY †	Object	Select **TINY** memory model (Default).
UNSIGNED_CHAR, UCH †	Object	Treat plain **char** as **unsigned char**.
USERCLASS, UCL	Object	Override a memory class name.
WARNING, WA	Listing	Change a Warning to an Error or disable a warning.
WARNINGLEVEL, WL †	Listing	Selects the level of Warning detection.
XSMALL, XSM †	Object	Select **XSMALL** memory model.
XTINY, XTY †	Object	Select **XTINY** memory model.

† 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.

Reference

The remainder of this chapter is devoted to describing each of the available C251 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.
µVision Control:	Gives the dialog box in the µVision Version 1 Windows environment which allows you to specify the directive. *µVision2* uses sligthly different dialog boxes. Therefore some specifications might not apply to *µVision2*.
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.

ASM / ENDASM

Abbreviation:	None.
Arguments:	None.
Default:	None.
µVision Control:	This directive cannot be specified on the command line.

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 inline assembly. However, it is output to the source file generated only when using the **SRC** directive. The source text is not assembled and the generation of the object file must be done with the A251 assembler.

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. #pragma asm/endasm is implemented for compatibility with previous compiler versions and converted to __asm {} blocks.*

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
```

ASMEXPAND / NOASMEXPAND

Abbreviation: **AE / NOAE**

Arguments: None.

Default: Expand preprocessor symbols in asm/endasm sections.

µVision Control: Options–C251 Compiler–Misc–Additional Options: Enter NOASMEXPAND

Description: The **NOASMEXPAND** directive instructs the C251 compiler to copy the text between #pragma asm & #pragma endasm without any preprocessor text expansion. The default setting of C251 is **ASMEXPAND** where all symbols used are expanded, even those symbols which are inside asm/endasm.

Example:

```
C251 SAMPLE.C SRC
```

Without **NOASMEXPAND**, the preprocessor expands the identifiers given before EQU. Since these identifiers are already defined at preprocessor level the expanded text would give **1 EQU 2.** This would result in errors when the file is assembled.

```
#pragma NOASMEXPAND
#define  abc  1
#define  xx0  2
#define  xx1  3

#pragma asm
  abc EQU  2    ; above <abc> not expanded
  xx0 EQU 10    ; above <xx0> not expanded
#pragma endasm

int a = abc;   /* abc replaced by 1 */
```

BROWSE

Abbreviation:	**BR**
Arguments:	None.
Default:	No browse information is created
µVision Control:	Browse information will be displayed µVision Version 2. In µVision Version 1 browse information is not available.
Description:	With **BROWSE**, the compiler creates browse information. The browse information covers identifiers (including preprocessor symbols), their memory space and type, and the definition and reference lists.

Example:

```
C251 SAMPLE.C BROWSE

#pragma browse
```

CODE

Abbreviation:	**CD**
Arguments:	None.
Default:	No assembly code listing is generated.
µVision Control:	Options–C251 Compiler–Listing–Include assembly code.
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:

```
C251 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           unsigned int  a, b, c;
   2
   3           void main (void)  {
   4   1         a = 14 * 15 + ((b / c) + 252);
   5   1       }

;       FUNCTION main (BEGIN)
                                   ; SOURCE LINE # 3
                                   ; SOURCE LINE # 4
000000 7E2500      R  MOV      WR4,c
000003 7E3500      R  MOV      WR6,b
000006 8D32           DIV      WR6,WR4
000008 2E3401CE       ADD      WR6,#01CEH
00000C 7A3500      R  MOV      a,WR6
                                   ; SOURCE LINE # 5
00000F 22             RET
;       FUNCTION main (END)
```

COND / NOCOND

Abbreviation: **CO, NOCO**

Arguments: None.

Default: **COND**

µVision Control: Options–C251 Compiler–Listing–Include coditional code.

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.
µVision Control:	Options–C251 Compiler–Object–Include debug information.

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 symbolic debug of your programs. It contains global and local variable definitions and their addresses, as well as function names and line numbers. Debug information contained in each object module is included in the L251 linker/locator output file and can be used by dScope or other debugging tools.

Example:

```
C251 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.
µVision Control:	Options–C251 Compiler–Misc–Symbols for DEFINE command.
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 may 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:

```
C251 SAMPLE.C DEFINE (check, NoExtRam)

C251 MYPROG.C DF (X1="1+5",iofunc="getkey ()")
```

DISABLE

Abbreviation: None.

Arguments: None.

Default: None.

µVision Control: This directive cannot be specified on the command line.

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
    4          uchar noInts (uchar p1, uchar p2)  {
    5   1        return (p1 * p2 + p2 * p1);
    6   1      }

;       FUNCTION noInts (BEGIN)
                                     ; SOURCE LINE # 4
000000 D3             SETB     C
000001 10AF01         JBC      EA,?C0002
000004 C3             CLR      C
               ?C0002:
000005 C0D0           PUSH     PSW
;---- Variable 'p2' assigned to Register 'R7' ----
000007 7C6B           MOV      R6,R11        ; A=R11
;---- Variable 'p1' assigned to Register 'R6' ----
                                     ; SOURCE LINE # 5
000009 7CB7           MOV      R11,R7        ; A=R11
00000B ACB6           MUL      R11,R6        ; A=R11
00000D 2CBB           ADD      R11,R11       ; A=R11
                                     ; SOURCE LINE # 6
00000F D0D0           POP      PSW
000011 92AF           MOV      EA,C
000013 22             RET
;       FUNCTION noInts (END)
```

EJECT

Abbreviation: **EJ**

Arguments: None.

Default: None.

µVision Control: This directive cannot be specified on the command line.

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
```

FIXDRK

Abbreviation:	**FD**
Arguments:	None.
Default:	**INC DRk,#const** instruction is used.
µVision Control:	Options–C251 Compiler–Misc–Additional Options: Enter FIXDRK
Description:	The **FIXDRK** directive instructs the compiler to replace the **INC DRk,#const** instruction by an **ADD DRk,#const** instruction. In the Intel 251SB C-step CPU, the **INC DRk,#const** does not work correctly. Check the stepping level or contact the silicon vendor to find out if you need to specify **FIXDRK**.

Example: The following example is a source and code listing of a function which adds a constant value to a variable of type long. This kind of statement is most likely to generate an **INC DRk,#const** instruction.

```
.
.
.
stmt  level    source

   1           long  l1;
   2
   3           void incL1 (void)  {
   4   1         l1 += 2;
   5   1       }

Code without FIXDRK directive:

;       FUNCTION incL1 (BEGIN)
                                      ; SOURCE LINE # 3
                                      ; SOURCE LINE # 4
000000 7E1D00      R  MOV    DR4,l1
000003 0B1D           INC    DR4,#02H
000005 7A1D00      R  MOV    l1,DR4
                                      ; SOURCE LINE # 5
000008 22             RET
;       FUNCTION incL1 (END)

Code with FIXDRK directive:

;       FUNCTION incL1 (BEGIN)
                                      ; SOURCE LINE # 3
                                      ; SOURCE LINE # 4
000000 7E1D00      R  MOV    DR4,l1
000003 2E180002       INC    DR4,#02H ; --> ADD DR4,#2
000007 7A1D00      R  MOV    l1,DR4
                                      ; SOURCE LINE # 5
00000A 22             RET
;       FUNCTION incL1 (END)
```

FLOAT64

Abbreviation:	**F64**
Arguments:	None.
Default:	Single precision floating point arithmetic (32 Bit).
µVision Control:	Options–C251 Compiler–Misc–Additional Options: Enter FLOAT64
Description:	With the **FLOAT64** directive double precision floating point arithmetic is enabled. If **FLOAT64** is not specified, C251 uses single precision floating point arithmetic even if the double keyword is used in the C source file.

Example:

```
C251 SAMPLE.C FLOAT64

#pragma float64
```

FLOATFUZZY

Abbreviation:	**FF**
Arguments:	A number between 0 and 7.
Default:	**FLOATFUZZY (3)**
µVision Control:	Options–C251 Compiler–Optimization–Bits to round for float compare.
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:

```
C251 SAMPLE.C FLOATFUZZY (2)

#pragma FF(0)
```

FUNCTIONS

Abbreviation: **FC**

Arguments: Memory type and reentrant/static attribute.

Default: Static code generation in the default memory type.

µVision Control: Options–C251 Compiler–Memory Model–Use static memory for non-register automatics.

Description: With the **FUNCTIONS** directive, the memory model and reentrant attribute can be changed. The syntax of this directive is:

FUNCTIONS (⟦***fmodel,***⟧ ⟦***attribute***⟧ **)**

The ***fmodel*** specifies the memory type used for non-register variables. The **FUNCTIONS (*fmodel*)** sub-control does not influence the pointer size. The following table describes the available options for ***fmodel***.

fmodel	Descrption
LARGE, LA	use **xdata** memory for non-register automatics.
PLM	use Intel **PL/M-51** parameter passing via **data** memory, register parameters are never used.
SMALL, SM	use **data** memory for non-register automatics
XSMALL, XSM	use **edata** memory for non-register automatics.

The ***attribute*** specifies static or reentrant code generation. The available options are listed below:

attribute	Descrption
static	use static memory for non-register automatic and parameter variables. The linker/locater overlays the static memory areas of functions when they do not call each other.
reentrant	use the 251 stack space for non-register automatic and parameter variables. *mem_model* has not influence on the variable placement and should be not specified together with reentrant.

Example: The following example is a source and code listing of various functions, each modified with **#pragma functions**. Note that the placement of local variables is affected. In case of reentrant functions the function name is mangled. The PLM function style provides an interface to programs written with Intel's PL/M-51. It implies parameter passing via data memory (static approach).

```
stmt  level    source

   1           typedef unsigned int    uint;
   2           typedef unsigned char  uchar;
   3
   4           #pragma functions (PLM)
   5           uint PlmFunc (uchar u1, uint n1)  {
   6    1        return (n1 + u1 + 10);
   7    1      }
   8
   9           #pragma functions (reentrant)
  10           uint XsmFunc (uchar u1, uint n1)  {
  11    1        return (n1 + u1 + 10);
  12    1      }
  13
  14           #pragma functions (XSMALL, static)
  15           uint XsmStat (uchar u1, uint n1)  {
  16    1        uchar  arr[10];
  17    1        for ( n1 = 0 ; n1 < 10 ; ++n1 )  {
  18    2          arr[n1] = 0x10;
  19    2        }
  20    1        return (n1 + u1 + 10);
  21    1      }
  22
  23           #pragma functions (reentrant)
  24           uint XsmRent (uchar u1, uint n1)  {
  25    1        uchar  arr[10];
  26    1        for ( n1 = 0 ; n1 < 10 ; ++n1 )  {
  27    2          arr[n1] = 0x10;
  28    2        }
  29    1        return (n1 + u1 + 10);
  30    1      }

;       FUNCTION PLMFUNC (BEGIN)
                                      ; SOURCE LINE # 5
                                      ; SOURCE LINE # 6
000000 E500        R  MOV      A,u1          ; A=R11
000002 0A3B           MOVZ     WR6,R11       ; A=R11
000004 2E3500      R  ADD      WR6,n1
000007 2E34000A       ADD      WR6,#0AH
                                      ; SOURCE LINE # 7
00000B 22             RET
;       FUNCTION PLMFUNC (END)

;       FUNCTION XsmFunc?_ (BEGIN)
                                      ; SOURCE LINE # 10
;---- Variable 'n1' assigned to Register 'WR6' ----
;---- Variable 'u1' assigned to Register 'R11' ----
                                      ; SOURCE LINE # 11
000000 0A2B           MOVZ     WR4,R11       ; A=R11
```

```
000002 2D32           ADD      WR6,WR4
000004 2E34000A       ADD      WR6,#0AH
                                     ; SOURCE LINE # 12
000008 22             RET
;       FUNCTION XsmFunc?_ (END)

;       FUNCTION XsmStat (BEGIN)
                                     ; SOURCE LINE # 15
000000 7D23           MOV      WR4,WR6
;---- Variable 'n1' assigned to Register 'WR4' ----
000002 7C7B           MOV      R7,R11        ; A=R11
;---- Variable 'u1' assigned to Register 'R7' ----
                                     ; SOURCE LINE # 17
000004 6D22           XRL      WR4,WR4
               ?C0006:
                                     ; SOURCE LINE # 18
000006 7410           MOV      A,#010H       ; A=R11
000008 19B20000    R  MOV      @WR4+arr,R11  ; A=R11
                                     ; SOURCE LINE # 19
00000C 0B24           INC      WR4,#01H
00000E BE24000A       CMP      WR4,#0AH
000012 78F2           JNE      ?C0006
                                     ; SOURCE LINE # 20
000014 0A37           MOVZ     WR6,R7
000016 2E34000A       ADD      WR6,#0AH
00001A 2E34000A       ADD      WR6,#0AH
                                     ; SOURCE LINE # 21
00001E 22             RET
;       FUNCTION XsmStat (END)

;       FUNCTION XsmRent?_ (BEGIN)
                                     ; SOURCE LINE # 24
000000 7D23           MOV      WR4,WR6
;---- Variable 'n1' assigned to Register 'WR4' ----
000002 7C7B           MOV      R7,R11        ; A=R11
;---- Variable 'u1' assigned to Register 'R7' ----
000004 2EF8000A       ADD      DR60,#0AH
                                     ; SOURCE LINE # 26
000008 6D22           XRL      WR4,WR4
               ?C0012:
                                     ; SOURCE LINE # 27
00000A 7410           MOV      A,#010H       ; A=R11
00000C 7F0F           MOV      DR0,DR60
00000E 2E14FFF7       ADD      WR2,#0FFF7H
000012 2D12           ADD      WR2,WR4
000014 7A19B0         MOV      @WR2,R11      ; A=R11
                                     ; SOURCE LINE # 28
000017 0B24           INC      WR4,#01H
000019 BE24000A       CMP      WR4,#0AH
00001D 78EB           JNE      ?C0012
                                     ; SOURCE LINE # 29
00001F 0A37           MOVZ     WR6,R7
000021 2E34000A       ADD      WR6,#0AH
000025 2E34000A       ADD      WR6,#0AH
                                     ; SOURCE LINE # 30
000029 9EF8000A       SUB      DR60,#0AH
00002D 22             RET
;       FUNCTION XsmRent?_ (END)
```

HOLD

Abbreviation: **HL**

Arguments: Numeric size limits for global variables placed in **data**, **near**, and **xdata** memory. These numbers must be separated by commas and enclosed in parentheses. The **HOLD** directive has the following format:

HOLD (*data_limit* ⟦*, near_limit* ⟦*, xdata_limit*⟧⟧)

where:

data_limit	is the maximum size for global variables stored in **data** memory. This parameter specifies the object size limit for **data** memory. If ***data_limit*** is 2, only variables of type **char** and **int** and arrays with 2 bytes or less are placed into **data** memory. If ***data_limit*** is 0, **data** memory is not used for implicit variable declarations.
near_limit	is the maximum size for global variables stored in **near** memory. This parameter specifies the object size limit for the **near** memory. If ***near_limit*** is 10, variables that are less than 10 bytes long are stored in **near** memory. If ***near_limit*** is 0, **near** memory is not used for implicit variable declarations. Since ***near_limit*** is checked after ***data_limit***, it must be larger than ***data_limit***.
xdata_limit	is the maximum size for global variables stored in **xdata** memory. This parameter specifies the object size limit for **xdata** memory. If ***xdata_limit*** is 1000, variables that are 1000 bytes long or less are stored in **xdata** memory and variables that are larger are stored in **far** memory. If ***xdata_limit*** is 0, **xdata** memory is not used for implicit variable declarations. Since ***xdata_limit*** is

checked after ***data_limit*** and ***near_limit*** it must be larger than these parameters.

Default: **HOLD (0, 0, 0)**

µVision Control: Options–C251 Compiler–Memory Model–Data Allocation Limits

Description: Use the **HOLD** directive to specify size limits that the C251 compiler uses when locating global variables that are declared without explicit memory types.

Normally, the C251 compiler locates all variables in the default memory areas defined by the **SMALL**, **LARGE**, **TINY**, **XSMALL** and **XTINY** directives. You can use the **HOLD** directive to store global variables in different memory areas based on the size of the variable.

For example, **HOLD(2,4,6)** places variables that are 1 or 2 bytes long into **data** memory, 3 or 4 bytes long into **near** memory, and 5 or 6 bytes long into **xdata** memory. Variables that are larger than 6 bytes are stored in **far** memory. Refer to “251 Memory Areas” 251 Memory Types on page 92 for more information.

NOTE
*The **HOLD** directive does not influence the location of automatic variables and static variables declared within a function. When the **HOLD** directive is active, the default memory type depends on the size of the variable.*

When all three **HOLD** parameters (***data_limit***, ***near_limit***, ***xdata_limit***) are zero, the C251 compiler uses the default memory type defined by the memory model. Refer to “Memory Models” on page 96 for more information.

NOTE
*The **HOLD** directive affects **extern** definitions. In order for the C251 compiler to determine the location of external variables, you must compile all files in a project with the same **HOLD** settings. Furthermore, you should explicitly*

specify array dimensions for ***extern*** *arrays. Avoid array definitions with empty brackets, for example:*

```
extern int table [];       /* Array size unknown */
                           /* at compile time    */
```

See Also: **LARGE**, **SMALL, TINY, XTINY, XSMALL**

Examples:

```
C251 SAMPLE.C HOLD (2, 10, 0x2000)
```

In this example, variables with an object size less than or equal to 2 bytes are stored in **data** memory, variables with an object size less than or equal to 10 bytes are stored in **near** memory, and variables with an object size less than or equal to 0x2000 bytes are stored in **xdata** memory. Objects with larger sizes are stored in the memory model based default memory.

```
C251 SAMPLE.C HOLD (0, 20)
```

In this example, variables with an object size 20 bytes and smaller are stored in **near** data memory. All other variables are stored in **far** memory.

INCDIR

Abbreviation:	**ID**
Arguments:	Path specifications for include files enclosed in parentheses.
Default:	None.
µVision Control:	This directive cannot be set with µVision. You can use Options–Environment Pathspecs–INC File Directory.
Description:	The **INCDIR** directive specifies the location of C251 include files. The compiler accepts a maximum of 5 path declarations. If more then one path declaration is required, the path names must be separated by semicolons. If you specify #include "filename.h", the C251 Compiler searches first the current directory and then directory of the source file. When this fails or when #include <filename.h> is used, the paths specified by the **INCDIR** directive are searched. When this still fails, the paths specified by the **C251INC** environment variable are used.

Example:

```
C251 SAMPLE.C INCDIR(C:\C251\MYINC;C:\CHIP_DIR)
```

INTERVAL

Abbreviation: **IT**

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

Default: INTERVAL (8).

µVision Control: Options–C251 Compiler–Misc–Additional Options: Enter INTERVAL.

Description: The **INTERVAL** directive specifies an interval for interrupt vectors. The interval specification is required for derivatives that use an interval other than 8. Using this directive, the compiler will locate 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:

```
C251 SAMPLE.C INTERVAL (16)

#pragma interval (16)
```

INTR2

Abbreviation:	**I2**
Arguments:	None.
Default:	the C251 compiler assumes that an interrupt pushes 4 bytes onto the stack: a 24-bit return address and **PSW1**.
µVision Control:	Options–C251 Compiler–Object–Save PSW1 in interrupt code.
Description:	The **INTR2** directive informs the C251 compiler that the 251 CPU saves the low order 16 bits of the program counter but does not automatically save **PSW1** when entering an interrupt. When **INTR2** is used, the C251 compiler generates code that manually saves and restores the **PSW1** register in interrupt functions. You must use this directive if you want the 8051 compatible behaviour.

Example:

```
C251 SAMPLE.C INTR2

#pragma i2
```

INTVECTOR / NOINTVECTOR

Abbreviation:	**IV / NOIV**
Arguments:	An optional offset, in parentheses, for the interrupt vector table.
Default:	**INTVECTOR (0)**
µVision Control:	Options–C251 Compiler–Object–Interrupt Vectors

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 **AJMP**, **LJMP**, or **EJMP** 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 jump instruction. Vectors are located starting at absolute address:

(8 × *n*) + *offset* + 3,

where:

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

Example:

```
C251 SAMPLE.C INTVECTOR(0x8000)

#pragma iv(0x8000)

C251 SAMPLE.C NOINTVECTOR

#pragma noiv
```

LARGE

Abbreviation:	**LA**
Arguments:	None.
Default:	**TINY**
µVision Control:	Options–C251 Compiler–Memory Model–Memory Model.

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 xdata memory of the 251 system. Up to 64 Kbytes of xdata memory may be accessed. This, however, requires the long and therefore inefficient form of data access through the data pointer (**DPTR** or **DR56**). The default data pointer size is 4 bytes which allows full access to the entire 251 address space.

Regardless of memory model type, you may declare variables in any of the 251 memory areas. However, placing frequently used variables (such as loop counters and array indices) in **data** or **near** memory significantly improves system performance.

NOTE
*The stack required for function calls is always placed in near memory, refered as memory class **EDATA**.*

Instead of using the LARGE memory model you should try the XTINY or XSMALL memory model.

See Also: **HOLD, ROM, SMALL, TINY, XSMALL, XTINY**

Example:

```
C251 SAMPLE.C LARGE

#pragma large
```

LISTINCLUDE

Abbreviation: **LC**

Arguments: None.

Default: **NOLISTINCLUDE**

µVision Control: Options–C251 Compiler–Listing–List #include files

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:

```
C251 SAMPLE.C LISTINCLUDE

#pragma listinclude
```

MAXARGS

Abbreviation: None.

Arguments: Number of bytes the compiler reserves for variable-length argument lists when static code is generated.

Default:

MAXARGS(15)	for small and tiny memory model.
MAXARGS(40)	for large, xtiny and xsmall model.

µVision Control: Options–C251 Compiler–Misc–Additional Options: Enter MAXARGS

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 is relevant for non-reentrant functions only and has no impact on the maximum number of arguments that may be passed to reentrant functions.

Example:

```
C251 SAMPLE.C MAXARGS(20)
```

```
#pragma maxargs (10) /* allow 10 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 */
}
```

MODBIN

Abbreviation:	None.
Arguments:	None.
Default:	Generate code for the source mode of the 251.
µVision Control:	Options–C251 Compiler–Object–Code Generation
Description:	The **MODBIN** directive instructs the C251 compiler to generate instructions for the binary mode of the 251. When the **MODBIN** directive is specified, the C251 compiler generates binary mode code that uses the **ESC** prefix (0A5h). Binary mode code generated with the C251 compiler is not compatible with the 8051.
Example:	`C251 SAMPLE.C MODSRC`

NAME

Abbreviation: **NA**

Arguments: The module name for the object module

Default: **NAME** (**basename**)

µVision Control: This control cannot be entered with µVision.

Description: With the **NAME** directive, the name of the object module can be changed. The name of the object module is normally the source file name without path and file extension. The name may be up to 40 characters, a longer name is truncated at 40 characters. The purpose of the **NAME** directive is to create model independant libraries, when one module is contained several times, each time compiled with a different memory model or different function attributes like static or reentrant.

Example:

```
C251 SAMPLE.C NAME (S1MODULE)

#pragma NAME (S2MODULE)
```

NOALIAS

Abbreviation: **NOAL**

Arguments: None.

Default: All write operations via indirect addresses (pointers) are destructive. The C251 compiler does not re-use the contents of registers in case of variable accesses.

µVision Control: Options–C251 Compiler–Object–Alias checking on pointer accesses

Description: The **NOALIAS** control instructs the compiler to ignore pointer write operations during the optimization phase. If a register holds a variable it gets reused, even when a pointer could modify that variable.

Example:

```
C251 SAMPLE.C NOALIAS

#pragma NOAL
```

The following example demonstrates code generated using the **NOALIAS** control directive.

```
stmt lvl  source

   1      struct { int  i1; int  i2; } *p_struct;
   2      int val;
   3
   4      /* with NOALIAS, the C251 Compiler does
   5         not reload p_struct and val, even when
   6         the *p_val = 0 assignment destroys val */
   7
   8      void func1 (int *p_val)  {
   9   1    p_struct->i1 = val;
  10   1    *p_val = 0;
  11   1    p_struct->i2 = val;
  12   1  }
  13
  14      void func2 (void)  {
  15   1    func1 (&val);
  16   1  }
```

Code generated without NOALIAS

```
             ; FUNCTION func1 (BEGIN)
                    ; SOURCE LINE # 8
;--- 'p_val' assigned to Register 'WR6'
                    ; SOURCE LINE # 9
000000 7E1500    MOV    WR2,val
000003 7E2500    MOV    WR4,p_struct
000006 1B2810    MOV    @WR4,WR2
                    ; SOURCE LINE # 10
000009 6D22      XRL    WR4,WR4
00000B 1B3820    MOV    @WR6,WR4
                    ; SOURCE LINE # 11
00000E 7E3500    MOV    WR6,val
000011 7E2500    MOV    WR4,p_struct
000014 59320002  MOV    @WR4+2,WR6
                    ; SOURCE LINE # 12
000018 22        RET
;       FUNCTION func1 (END)

;       FUNCTION func2 (BEGIN)
                    ; SOURCE LINE # 14
                    ; SOURCE LINE # 15
000000 7E340000  MOV    WR6,#WORD0 val
000004 120000    LCALL  func1
                    ; SOURCE LINE # 16
000007 22        RET
;       FUNCTION func2 (END)
```

CODE SIZE = 33 Bytes

Code generated with NOALIAS

```
; FUNCTION func1 (BEGIN)
                    ; SOURCE LINE # 8
000000 7D13      MOV    WR2,WR6
;--- 'p_val' assigned to Register 'WR2'
                    ; SOURCE LINE # 9
000002 7E3500    MOV    WR6,val
000005 7E2500    MOV    WR4,p_struct
000008 1B2830    MOV    @WR4,WR6
                    ; SOURCE LINE # 10
00000B 6D00      XRL    WR0,WR0
00000D 1B1800    MOV    @WR2,WR0
                    ; SOURCE LINE # 11
000010 59320002  MOV    @WR4+2,WR6
                    ; SOURCE LINE # 12
000014 22             RET
;       FUNCTION func1 (END)

;       FUNCTION func2 (BEGIN)
                    ; SOURCE LINE # 14
                    ; SOURCE LINE # 15
000000 7E340000  MOV    WR6,#WORD0 val
000004 120000    LCALL  func1
                    ; SOURCE LINE # 16
000007 22        RET
;       FUNCTION func2 (END)
```

CODE SIZE = 29 Bytes

NOAMAKE

Abbreviation:	**NOAM**
Arguments:	None.
Default:	AutoMAKE information is generated.
µVision Control:	This control cannot be entered with µVision.
Description:	**NOAMAKE** disables the project information records produced by the C251 compiler for use with AutoMAKE. This option also disables the register optimization information. If **NOAMAKE** is used, changes to the include files are not recognized and the global register optimization cannot be performed for that source module.

Example:

```
C251 SAMPLE.C NOAMAKE

#pragma NOAM
```

NOCASE

Abbreviation:	**NOCA**
Arguments:	None.
Default:	Output all indentifiers as written in the C source file.
µVision Control:	Options–C251 Compiler–Misc–Addtional Options: Enter **NOCASE**.

Description: This directive controls the case sensitivity of imported or exported identifiers in the object file, the front end of the compiler is not affected by case/nocase.

If **NOCASE** is given, the compiler converts identifiers to upper case before using them for object file records. With **CASE**, the identifiers are used with no upper case translation.

Example:

```
C251 SAMPLE.C NOCASE

#pragma nocase
```

NOEXTEND

Abbreviation:	**NOEXT**
Arguments:	None.
Default:	All language extensions are enabled.
µVision Control:	Options–C251 Compiler–Misc–Disable non-ANSI extensions
Description:	The **NOEXTEND** directive instructs the compiler to process only ANSI C language constructs. The C251 language extensions are disabled. Reserved keywords such as **bit**, **reentrant**, and **using** are not recognized and generate compilation errors or warnings.

Example:

```
C251 SAMPLE.C NOEXTEND

#pragma NOEXT
```

OBJECT / NOOBJECT

Abbreviation:	**OJ** / **NOOJ**
Arguments:	An optional filename enclosed in parentheses.
Default:	**OBJECT (*basename*.OBJ)**
µVision Control:	Project–Edit Project–Object
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:

```
C251 SAMPLE.C OBJECT(sample1.obj)

#pragma oj(sample_1.obj)

C251 SAMPLE.C NOOBJECT

#pragma nooj
```

OPTIMIZE

Abbreviation: **OT**

Arguments: A decimal number between 0 and 7 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 (7, SPEED)**

µVision Control: Options–C251 Compiler–Optimization–Optimization Level / Emphasis

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 251 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 L251 linker/locator has the capability, through global data flow analysis, of selecting segments which can then be overlaid. **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
3	**Register Variables:** Automatic variables and function arguments are located in registers when possible. Reservation of data memory for these variables is omitted.
4	**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 and 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. **Case/Switch Optimizing:** Code involving switch and case statements is optimized as jump tables or jump strings.
5	**Life Variable Analysis:** Removes redundant and dead store operations to automatic variables.
6	**Constant Propagation:** The values of expressions are tracked and when possible constant values are inserted instead of variable accesses. **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.
7	**Common Tail Merging and Instruction Simplification:** The compiler analyzes the generated code function by function and tries to find common heads and tails. If the compiler detects common code sequences, it will replace one code sequence by a jump instruction to the other equivalent code sequence. This situation arises many times with switch/case statements, for example. While analyzing the code, the compiler also tries to replace instruction sequences with cheaper instructions.

OPTIMIZE level 7 includes all optimizations of levels 0 to 6.

Example:

```
C251 SAMPLE.C OPTIMIZE (size)
C251 SAMPLE.C OPTIMIZE (6, speed)
#pragma ot(4)
#pragma ot(size)
```

ORDER

Abbreviation:	**OD**
Arguments:	None.
Default:	The variables are not ordered.
µVision Control:	Options–C251 Compiler–Object–Keep variables in order.
Description:	The **ORDER** directive instructs the C251 compiler 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 C251 compiler runs a little slower when this directive is specified.

Example:

```
C251 SAMPLE.C ORDER

#pragma OR
```

PACK

Abbreviation:	None.
Arguments:	A decimal 1 or 2 in parentheses or empty parentheses.
Default:	**PACK(1)**
µVision Control:	Options–C251 Compiler–Misc–Additional Options: Enter PACK.

Description: The **PACK** directive specifies the structure packing. With **PACK(1)**, structure members are packed without alignment gaps, regardless of their size. With **PACK(2)**, structure members of type int, long, float or 2/4 byte pointer are aligned to even offsets. The compiler inserts a dummy placeholder to adjust the offset to an even value. The **PACK** directive may be used to make structure types compatible to other architectures, for example in dual port RAM application. Note that **PACK(2)** may increase the size of a structure, since one or more alignment gaps are inserted for padding.

The **PACK** directive can be given many times to change the packing of individual structures. Make sure that the structure packing is consistent over all modules of the application. To ensure this, place the PACK directives and structure declarations in a global .H file which is included in each C source file of the application.

Example:

```
C251 SAMPLE.C PACK(2)

stmt  level    source

   1           #pragma pack(1)
   2           struct s1  {
   3             char    n1;
   4             int     n2;
   5           };
   6
   7           #pragma pack(2)
   8           struct s2  {
   9             char    n1;
  10             int     n2;   /* preceeded by gap ! */
  11           };
  12
  13           #pragma pack()  /* default packing: 1 */
  14           struct s3  {
  15             char    n1;
  16             int     n2;
  17           };

Name                Class    Space Type   Offset   Size
-------------------------------------------------------
s3 . . . . . . . . .*tag*          struct 000003H  3
  n1 . . . . . . . .member         char   000000H  1
  n2 . . . . . . . .member         int    000001H  2
s2 . . . . . . . . .*tag*          struct 000004H  4
  n1 . . . . . . . .member         char   000000H  1
  <gap>. . . . . . .member         uchar  000001H  1
  n2 . . . . . . . .member         int    000002H  2
s1 . . . . . . . . .*tag*          struct 000003H  3
  n1 . . . . . . . .member         char   000000H  1
  n2 . . . . . . . .member         int    000001H  2
```

PAGELENGTH

Abbreviation:	**PL**
Arguments:	A decimal number up to 65535 enclosed in parentheses.
Default:	**PAGELENGTH (65)**
µVision Control:	Options–C251 Compiler–Listing–Page length
Description:	The **PAGELENGTH** directive specifies the number of lines printed per page in the listing file. The default is 65 lines per page (in US, otherwise 69 lines per page). This includes headers and empty lines.
See Also:	**PAGEWIDTH**

Example:

```
C251 SAMPLE.C PAGELENGTH (70)

#pragma pl(70)
```

PAGEWIDTH

Abbreviation:	**PW**
Arguments:	A decimal number in range 78 to 132 enclosed in parentheses.
Default:	**PAGEWIDTH (120)**
µVision Control:	Options–C251 Compiler–Listing–Page width
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 split into two or more lines.
See Also:	**PAGELENGTH**

Example:

```
C251 SAMPLE.C PAGEWIDTH(79)

#pragma pw(79)
```

PARM51 / PARM251

Abbreviation: **P51** / **P251**

Arguments: None

Default: **PARM251**

µVision Control: Options–C251 Compiler–Object–C51 argument passing

Description: The **PARM51** directive instructs the C251 compiler to use the parameter passing conventions of the C51 compiler. This directive is useful when you need to interface to existing 8051 code.

The **PARM251** directive directs the C251 compiler to use the default 251 parameter passing conventions.

Example:

```
#pragma PARM51   /* switch to C51 calling conventions */
extern char *function_51 (char);     /*  C51 function */

#pragma PARM251 /* switch to C251 calling conventions */
extern char *function_251 (int);     /* C251 function */

#pragma PARM51   /* switch to C51 calling conventions */

void f51 (char *p)  {    /* a C51 compatible function */
  *p = 0;
}

#pragma PARM251 /* switch to C251 calling conventions */

void main (void)  {
  char *p;

  p = function_51 (0);        /* call a C51 function  */
  p = function_251 (1);       /* call a C251 function */
}
```

PREPRINT

Abbreviation: **PP**

Arguments: An optional filename enclosed in parentheses.

Default: No preprocessor listing is generated.

µVision Control: Options–C251 Compiler–Misc–Additional Options: Enter PREPRINT.

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, the C251 compiler 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:

```
C251 SAMPLE.C PREPRINT

C251 SAMPLE.C PP (PREPRO.LSI)
```

PREPRINTONLY

Abbreviation: **PPO**

Arguments: An optional filename enclosed in parentheses.

Default: The nodule is preprocessed and compiled.

µVision Control: Options–C251 Compiler–Misc–Additional Options: Enter PREPRINTONLY.

Description: The **PREPRINTONLY** directive instructs the compiler to produce a preprocessor listing only, no parsing and code generation is performed. By default, the C251 compiler runs the parsing and code generation phase after preprocessing the input file.

NOTE
*The **PREPRINTONLY** 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:

```
C251 SAMPLE.C PREPRINTONLY

C251 SAMPLE.C PREPRINTONLY(PREPRO.C)
```

PRINT / NOPRINT

Abbreviation:	**PR** / **NOPR**
Arguments:	An optional filename enclosed in parentheses.
Default:	**PRINT (*basename*.LST)**
µVision Control:	Options–C251 Compiler–Listing–Generate .LST file. You may not change the listing file name with µVision.
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:

```
C251 SAMPLE.C PRINT(CON:)

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

C251 SAMPLE.C NOPRINT

#pragma nopr
```

REGFILE

Abbreviation: **RF**

Arguments: A file name enclosed in parentheses.

Default: None.

µVision Control: Options–Make–Misc–Global Register Optimization.

Description: With **REGFILE**, the C251 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 C251 compiler *knows* about the register utilization of external functions. This enables global program-wide register optimization.

Example:

```
C251 SAMPLE.C REGFILE(sample.reg)

#pragma REGFILE(sample.reg)
```

REGPARMS / NOREGPARMS

Abbreviation:	None.
Arguments:	None.
Default:	**REGPARMS**
µVision Control:	Options–C251 Compiler–Misc–Additional Options: Enter **NOREGPARMS**.

Description: The **REGPARMS** directive enables the compiler to generate code that passes function arguments in registers to the callee. This kind of parameter passing is similar to what you would use when writing assembly language code and is significantly faster than storing function arguments in static memory or on the stack. Parameters that do not fit within the available registers are passed using static memory locations or the stack, depending on the function attribute reentrant or non-reentrant.

The **NOREGPARMS** directive forces all function arguments to be passed in static memory locations or the via stack.

NOTE
*You may specify the **REGPARMS** and **NOREGPARMS** directive several times within a source program. This allows you to create program sections with register parameters and other sections using the old style of parameter passing.*

Example:

```
C251 SAMPLE.C NOREGPARMS

#pragma NOREGPARMS



stmt level source

   1       #pragma NOREGPARMS
   2       extern void test1 (char c1, int i1, int i2);
   3
   4       #pragma REGPARMS
   5       extern void test2 (char c1, int i1, int i2);
   6
   7       void main (void)  {
   8   1     test1 (1, 2, 3);    // passed via mem
   9   1     test2 (1, 2, 3);    // passed via regs
  10   1   }

;       FUNCTION main (BEGIN)
                                           ; SOURCE LINE # 7
                                           ; SOURCE LINE # 8
000000 750001      E  MOV      ?test1?BYTE,#01H
000003 7E340002       MOV      WR6,#02H
000007 7A3500      E  MOV      ?test1?BYTE+01H,WR6
00000A 7E340003       MOV      WR6,#03H
00000E 7A3500      E  MOV      ?test1?BYTE+03H,WR6
000011 120000      E  LCALL    test1
                                           ; SOURCE LINE # 9
000014 7401           MOV      A,#01H        ; A=R11
000016 7E340002       MOV      WR6,#02H
00001A 7E240003       MOV      WR4,#03H
00001E 120000      E  LCALL    test2
                                           ; SOURCE LINE # 10
000021 22             RET
;       FUNCTION main (END)
```

RENAMECODE

Abbreviation:	**RC**
Arguments:	The new name of a code segment.
Default:	**N**o code segment renaming takes place.
µVision Control:	This control cannot be entered with µVision.
Description:	With the **RENAMECODE** directive you can give a code segment a different name. **RENAMECODE** can be used only if **ROM(HUGE)** or **ROM(MEDIUM)** is specified, since all all functions of a module must reside in a single code segment. For all other **ROM** settings each function compiles into a separate code segment and the **RENAMECODE** directive is ignored.

Example:

```
C251 SAMPLE.C ROM(HUGE) RENAMECODE (?PR?COMMONSEG)

#pragma renamecode(?PR?MYSEG)
```

ROM

Abbreviation: None.

Arguments: **SMALL**, **COMPACT**, **LARGE**, **MEDIUM**, or **HUGE**.

Default: **ROM (LARGE)**

µVision Control: Options–C251 Compiler–Memory Model–Code size limits.

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.
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.
MEDIUM	External targets are coded as **LCALL**, targets within the current code module are coded as **AJMP** or **ACALL**. A single code segment is limited to 2 Kbytes of code.
HUGE	All function targets not explicitely modified with **near** are invoked with **ECALL** instructions. Intrasegment branches are codes with short jumps or **LJMP**'s, depending on the target distance.

NOTE
ROM(HUGE) *lets your program use the entire 16MB address space of the MCS® 251 microcontroller. In this memory model, a function declaration without a* ***near*** *modifier is a* ***far*** *function.* ***far*** *functions are invoked using the* ***ECALL*** *instruction (with a 24-bit address) and return using the* ***ERET*** *instruction. The compiler appends a question mark character to each* ***far*** *function in the generated object module to differentiate between* ***near*** *and*

***far** functions. For example, MyFunction() is converted to MyFunction? when the object module is created.*

Example:

```
C251 SAMPLE.C ROM(HUGE)

#pragma ROM (MEDIUM)
```

SAVE / RESTORE

Abbreviation: None.

Arguments: None.

Default: None.

µVision Control: This directive cannot be specified on the command line.

Description: The **SAVE** directive stores the current settings of the current **OPTIMIZE** level and emphasis, the function attributes specified by **FUNCTIONS** and the **HOLD** values. 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 functions (reentrant)

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

#pragma RESTORE
```

In the above example, the two external functions, `test1` and `test2` are defined to be reentrant. The settings at the time of the **SAVE** directive are restored by the **RESTORE** directive.

SMALL

Abbreviation: **SM**

Arguments: None.

Default: **TINY**

µVision Control: Options–C251 Compiler–Memory Model–Memory Model.

Description: This directive implements the **SMALL** memory model. The **SMALL** memory model places all function variables and local data segments in the data memory of the 251 system. This ensures the most efficient access to data objects. The default data pointers size is 4 bytes which allows full access to the entire 251 address space.

Regardless of memory model type, you may declare variables in any of the 251 memory areas. However, placing frequently used directives (such as loop counters and array indices) in data memory significantly improves system performance.

NOTE
*The stack required for function calls is always placed in near memory referred as the **EDATA** memory class.*

*If your total variable space is less than 64 Kbytes you should attempt to use the **TINY** memory model instead of **SMALL**. The **TINY** memory model increases execution speed and reduces code space requirements of your program.*

See Also: **HOLD, LARGE**, **ROM**, **TINY**, **XTINY**, **XSMALL**

Example:

```
C251 SAMPLE.C SMALL

#pragma small
```

SRC

Abbreviation: None.

Arguments: An optional filename in parentheses.

Default: None.

µVision Control: µVision

Description: Use the **SRC** directive to create an assembler source file instead of an object file. This source file may be assembled with the A251 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:

```
C251 SAMPLE.C SRC

C251 SAMPLE.C SRC(SAMPLE.ASM)
```

SYMBOLS

Abbreviation: **SB**

Arguments: None.

Default: No list of symbols is generated.

µVision Control: Options–C251 Compiler–Listing–Include symbols

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:

```
C251 SAMPLE.C SYMBOLS

#pragma SYMBOLS
```

TINY

Abbreviation: **TY**

Arguments: None.

Default: **TINY**

µVision Control: Options–C251 Compiler–Memory Model–Memory Model.

Description: This directive implements the **TINY** memory model. The **TINY** memory model places all function variables and local data segments in the **data** memory of the 251 system. This ensures the most efficient access to data objects. The default data pointer size is 2 bytes and refers to **near** (NDATA / NCONST) space.

Regardless of memory model type, you may declare variables in any of the 251 memory areas except **far** and **huge** spaces. However, placing frequently used directives (such as loop counters and array indices) in **data** or **near** memory significantly improves system performance.

NOTE
*The stack required for function calls is always placed in near memory, refered as memory class **EDATA**.*

*Always start by using the **TINY** 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 or by using the **HOLD** directive. If you data memory requirements grow you might use the **XTINY** memory model. Only for applications with huge data requirements the **SMALL**, **XSMALL** or **LARGE** memory model might be a good choice.*

See Also: **HOLD, LARGE**, **ROM**, **SMALL**, **XSMALL, XTINY**

Example:

```
C251 SAMPLE.C TINY

#pragma small
```

UNSIGNED_CHAR

Abbreviation: **UCH**

Arguments: None.

Default: Variables of type **char** are treated as **signed char**

µVision Control: Options–C251 Compiler–Misc–Additional Options: Enter UNSIGNED_CHAR.

Description: The **UNSIGNED_CHAR** directive instructs the C251 compiler to treat all variables declared with **char** as **unsigned char** variables.

Example:

```
C251 SAMPLE.C UNSIGNED_CHAR

#pragma UCH
```

The following example demonstrates code generated using the **UNSIGNED_CHAR** control directive.

```
Stmt lvl  source

   1                char  ch1;
   2      unsigned char uch2; /* always unsigned */
   3        signed char sch3; /* always signed   */
   4
   5                int I;
   6
   7      void test (void)  {
   8   1    if (ch1  > 0) i = ch1;
   9   1    if (uch2 > 0) i = uch2;
  10   1    if (sch3 > 0) i = sch3;
  11   1  }
```

Code generated with UNSIGNED_CHAR

```
;       FUNCTION test (BEGIN)
                ; SOURCE LINE # 6
                ; SOURCE LINE # 7
000000 E500     R  MOV   A,ch1
000002 BEB000      CMP   R11,#00H
000005 2807        JLE   ?C0001
000007 E500     R  MOV   A,ch1
000009 0A3B        MOVZ  WR6,R11
00000B 7A3500   R  MOV   i,WR6
                ?C0001:
                ; SOURCE LINE # 8
00000E E500     R  MOV   A,uch2
000010 BEB000      CMP   R11,#00H
000013 2807        JLE   ?C0002
000015 E500     R  MOV   A,uch2
000017 0A3B        MOVZ  WR6,R11
000019 7A3500   R  MOV   i,WR6
                ?C0002:
                ; SOURCE LINE # 9
00001C E500     R  MOV   A,sch3
00001E BEB000      CMP   R11,#00H
000021 0807        JSLE  ?C0003
000023 E500     R  MOV   A,sch3
000025 1A3B        MOVS  WR6,R11
000027 7A3500   R  MOV   i,WR6
                ?C0003:
                ; SOURCE LINE # 10
00002A 22        RET
;       FUNCTION test (END)
```

Code generated with UNSIGNED_CHAR

```
;       FUNCTION test (BEGIN)
                ; SOURCE LINE # 6
                ; SOURCE LINE # 7
000000 E500     R  MOV    A,ch1
000002 BEB000      CMP    R11,#00H
000005 0807        JSLE   ?C0001
000007 E500     R  MOV    A,ch1
000009 1A3B        MOVS   WR6,R11
00000B 7A3500   R  MOV    i,WR6
                ?C0001:
                ; SOURCE LINE # 8
00000E E500     R  MOV    A,uch2
000010 BEB000      CMP    R11,#00H
000013 2807        JLE    ?C0002
000015 E500     R  MOV    A,uch2
000017 0A3B        MOVZ   WR6,R11
000019 7A3500   R  MOV    i,WR6
                ?C0002:
                ; SOURCE LINE # 9
00001C E500     R  MOV    A,sch3
00001E BEB000      CMP    R11,#00H
000021 0807        JSLE   ?C0003
000023 E500     R  MOV    A,sch3
000025 1A3B        MOVS   WR6,R11
000027 7A3500   R  MOV    i,WR6
                ?C0003:
                ; SOURCE LINE # 10
00002A 22             RET
;       FUNCTION test (END)
```

USERCLASS

Abbreviation: **UCL**

Arguments: (***mspace* = *user_classname***)

mspace refers to the default memory space used for the variable or function allocation and is explained in the table below:

mspace	Description
UCODE	program code; refers to the class **CODE** or **ECODE** depending on the ROM directive.
DATA	**data** variables located in the **DATA** class.
IDATA	**idata** variables located in the **IDATA** class.
XDATA	**xdata** variables located in the **XDATA** class.
NEAR	**near** variables located in the **EDATA** class.
FAR	**far** variables located in the **HDATA** class.
HUGE	**huge** variables located in the **HDATA** class.

user_classname is the name for a memory class. You can supply any valid identifier for a class name. If you specify **default**, C251 selects the default user class again.

Default: Segments receive the default class name.

µVision Control: Options–C251 Compiler–

Description: The **USERCLASS** directive assigns a user defined class name to a compiler generated segment. By default, C251 uses the basic class name for segment definitoions. The user class name can be referred to at the L251 level to locate all segments with a class name, such as **EDATA_NVRAM** to a specific memory section.

Example:

```
C251 UCL.C SRC

#pragma userclass (near = app1)
int    near    x1 [10];
int    near    x2 [10];

#pragma userclass (near = default)
int    near    y1 [10];
int    near    y2 [10];

#pragma userclass (far = app2)
int    far     x3 [10];
int    far     x4 [10];

#pragma userclass (ucode = t2code)
int test1 (int i1, int i2)  {
  return (i1 + i2 * 10);
}
```

The example above creates the following assembly source:

```
                NAME UCL

  ?ED?APP1?UCL  SEGMENT  'EDATA_APP1'
  ?FA?APP2?UCL  SEGMENT  'HDATA_APP2'
?PR?T2CODE?UCL  SEGMENT  'ECODE_T2CODE'
       ?ED?UCL  SEGMENT  EDATA

     RSEG       ?ED?APP1?UCL
           x1:  DS  20
           x2:  DS  20

     RSEG       ?FA?APP2?UCL
           x3:  DS  20
           x4:  DS  20

     RSEG       ?ED?UCL
           y1:  DS  20
           y2:  DS  20

     RSEG       ?PR?T2CODE?UCL
        test1? PROC FAR
;---- Variable 'i2' assigned to Register 'WR4' ----
        MOV     WR2,WR6
;---- Variable 'i1' assigned to Register 'WR2' ----
; line 14:   return (i1 + i2 * 10);
        MOV     WR6,#0AH
        MUL     WR6,WR4
        ADD     WR6,WR2
; line 15: }
        ERET
        ENDP

        END
```

```
#pragma userclass (near=app1)
```

assignes the user class **EDATA_APP1** to variables with memory space **near** which are placed to the memory class **EDATA** by default. Therefore the arrays **x1** and **x2** are located in the memory class **EDATA_APP1**.

```
#pragma userclass (near=default)
```

resets the memory class for near variables to the compiler default.

WARNING

Abbreviation:	**WA**
Arguments:	A level and a numeric list of warning messages. The **WARNING** directive has the following format:

WARNING ***level*** **=** ***warning_num*** ⟦**,** ***warning_num*** ⟧

where:

level	is one of the following:	
	disable	disables the warning.
	once	lists the warning only once.
	error	treats the warning as an error.
	0	shifts the warning to level 0.
	1	shifts the warning to level 1.
	2	shifts the warning to level 2.
	3	shifts the warning to level 3.
warning_num	is the number of the warning message you want to modify.	

Default:	None.
µVision Control:	This directive cannot be specified on the command line.
Description:	The **WARNING** directive lets you change the level of specific compiler warnings. You may disable warnings, list them only once, or treat them as errors. The **WARNING** directive also allows you to change the warning level in which the message is listed. Refer to “Chapter 6. Error Messages” on page 165 for a complete list of warning messages.
See Also:	**WARNINGLEVEL**

Example:

```
#pragma warning error = 95
                     /* change warning 95 to an error */
#pragma warning once = 56, 57
        /* report warning 56 & 57 only the first time */
```

WARNINGLEVEL

Abbreviation: **WL**

Arguments: A number from 0 to 3.

Default: **WARNINGLEVEL (2)**

µVision Control: µVision

Description: The **WARNINGLEVEL** directive lets you suppress compiler warnings. Refer to “Chapter 6. Error Messages” on page 165 for a full list of 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:

```
C251 SAMPLE.C WL(1)

#pragma WARNINGLEVEL (0)
```

XSMALL

Abbreviation: **XSM**

Arguments: None.

Default: **TINY**

µVision Control: Options–C251 Compiler–Memory Model–Memory Model.

Description: This directive implements the **XSMALL** memory model. The **XSMALL** memory model places all function variables and local data segments in the **near** memory of the 251 system. This ensures efficient access to near data objects. The default data pointers is 4 bytes which allows full access to the entire 251 address space.

Regardless of memory model type, you may declare variables in any of the 251 memory areas. However, placing frequently used directives (such as loop counters and array indices) in **data** memory significantly improves system performance.

NOTE
*The stack required for function calls is always placed in near memory, refered as memory class **EDATA**.*

*If your total variable space is less than 64 Kbytes you should attempt to use the **XTINY** memory model instead of **XSMALL**.*

See Also: **HOLD**, **LARGE**, **ROM**, **SMALL**, **TINY, XTINY**

Example:

```
C251 SAMPLE.C XSMALL

#pragma xsmall
```

XTINY

Abbreviation:	**XTY**
Arguments:	None.
Default:	**TINY**
µVision Control:	Options–C251 Compiler–Memory Model–Memory Model.

Description: This directive implements the **XTINY** memory model. The **XTINY** memory model places all function variables and local data segments in the **near** memory of the 251 system. This ensures efficient access to near data objects. The default data pointer size is 2 bytes and refers to **near** (NDATA / NCONST) space.

Regardless of memory model type, you may declare variables in any of the 251 memory areas. However, placing frequently used directives (such as loop counters and array indices) in **data** memory significantly improves system performance.

NOTE
*The stack required for function calls is always placed in near memory, refered as memory class **EDATA**.*

*The **XTINY** memory model is typically the best choice for applications with medium data requirements.*

See Also: **HOLD**, **LARGE**, **ROM**, **SMALL**, **TINY**, **XSMALL**

Example:

```
C251 SAMPLE.C XTINY

#pragma xtiny
```

Chapter 3. Language Extensions

The C251 compiler provides a number of extensions for ANSI Standard C. Most of these provide direct support for elements of the MCS® 251 architecture. The C251 compiler includes extensions for:

- Memory Types and Areas on the 251
- 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 MCS® 251 architecture, the C251 compiler adds a number of new keywords to the scope of the C language. The following is a list of the keywords available:

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

You can disable these extensions using the **NOEXTEND** control directive. Refer to “NOEXTEND” on page 57 for more information.

251 Memory Areas

The MCS® 251 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 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 MCS® 251 Embedded Microcontroller* handbook or other 251 data books for more information about the MCS® 251 memory architecture.

Code Memory

The **code** memory area is typically used for program code and constants. The code memory resides in the address range FF:0000 to FF:FFFF. The **code** memory may reside within the 251 CPU, it may be external, or it may be both, depending on the 251 derivative and the hardware design. There may be up to 64 Kbytes of code memory.

Code memory can be accessed by using the **code** memory type specifier.

NOTE
*The **code** memory type is implemented for compatibility to the C51 compiler. Instead of code you should use **const near** or **const far**. This generates typically better code and should be used in new 251 applications instead of the memory type **code**.*

Data, BData, EBData and IData Memory

The **data** and **idata** memory areas reside within the 251 CPU and can be read and written. The **data** memory are the first 128 bytes of the on-chip memory. The **idata** memory are the first 256 bytes of the on-chip memory. There is also a 16 byte area starting at 20h that is bit-addressable with short 8051-compatible instructions.

Access to **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 128 bytes. The data memory can be broken down into three distinct data types: **data**, **bdata** and **ebdata**.

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

The **ebdata** memory specifier refers to the extended 251 bit-addressable memory in the internal data area (20H to 7FH). This memory type specifier allows you to declare data types that can also be accessed at the bit level. However compared to the bdata memory area, bit access to ebdata memory requires a longer opcode encoding.

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.

Near Memory

The **near** memory area is in the address range 00:0000 to 00:FFFF which is the first 64 Kbytes of the 251 memory. Part of the **near** memory resides within the 251 CPU and can be accessed very quickly using 16-bit addressing. The **near** memory area is represented at the assembly language level by the memory classes **EDATA** or **ECONST**. Variables stored in **near** memory are accessed using direct addressing.

Far and Huge Memory

The **far** memory area represents the full address range of the 251. The **far** memory area is represented at the assembly language level with the memory classes **HDATA** or **HCONST**.

Variables defined with memory type **far** are accessed using indirect addressing. The address calculation is 16 bits only and limits therefore the data object size of far objects to 64KB. However far variables can reside anywhere in the memory which allows a total size of up to 16 Mbytes.

In contrast to far, **huge** uses 32 bit operations for the address calculation and allows therefore unlimited object size. You can have a single array or structure with a size of several Mbytes.

Variables declared in **huge** memory may have any size up to 16 Mbytes - 1.

XData Memory

The **xdata** memory is a 64 Kbyte area which can be read and written. This memory area is implemented for compatibility with the 8051. Access to **xdata** is much slower than access to **data** or **near** memory. This is because external data memory is accessed indirectly through the data pointer (**DPTR** or **DR56**) register which must be loaded with a 16-bit address.

There may be up to 64 Kbytes of **xdata** 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 **xdata** memory to program and control the peripheral. This technique is referred to as memory-mapped I/O.

There are two different data types in C251 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.

NOTE
*The **pdata** memory type is implemented for compatibility to the C51 compiler and should not be used in new 251 applications.*

Special Function Register Memory

The 251 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 111 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 C251 command line using the **LARGE, SMALL, TINY, XSMALL** or **XTINY** directives. Refer to “Control Directives” on page 17 for more information about these directives.

The memory model does not influence the maximum program code you can generate. If you want to create a big program with a size of up to 16 Mbytes you must use the C251 “ROM” directive described on page 74. The memory model primarily specifies the default memory type used and depends therefore only on the variable space required by the application.

NOTE
*You should try to use the default **TINY** memory model. It generates the fastest, most efficient code. If the data requirements of your application grows you should try the **XTINY** memory model.*

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 97 for more information.

Tiny and XTiny Model

In the **Tiny** memory model, all variables, by default, reside in the internal **data** memory of the 251 system. (This is the same as if they were declared explicitly using the **data** memory type specifier.) The variable and pointer access is very efficient. The default pointers size is just 2 bytes and represents pointers to **near** memory. Constants are placed in **NCONST** memory class. Using the **HOLD** directive, you can direct variables above a specific object size to other memory areas. Typically, if the linker/locator is configured to overlay variables in the internal data memory, the **Tiny** model is the best model to use.

The **XTiny** memory model is identical to **Tiny**, except that all variables, by default, use the memory type **near,** known as **EDATA** memory class. For applications with medium data requirements the **XTiny** memory model is a good choice.

Small and XSmall Model

In the **Small** memory model, all variables, by default, reside in the internal **data** memory of the 251 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. Using the **HOLD** directive, you can direct variables above a specific object size to alternative memory areas. Data pointers are 4 byte pointers which allow the addressing of the entire 16 Mbyte address space. The **Small** memory model is a good choice if you need to variable access to the complete 16MB memory area.

The **XSmall** memory model is identical to **Small**, except that all variables, by default, use the memory type **near,** known as **EDATA** memory class. Typically, the **XSmall** model is the right choice for applications with huge requirements.

Large Model

In **Large** memory model, all variables, by default, use the memory type **xdata**, known as **XDATA** memory class. The data pointer (**DPTR** or **DR56**) is used for addressing. Memory access through this data pointer is inefficient. This type of data access mechanism generates more code than the **XTiny** or **XSmall** memory model. Therefore try to use the **XTiny** or **XSmall** memory model instead of **LARGE**. Using the **HOLD** directive, you can direct variables above a specific object size to alternative memory areas.

Memory Types

The C251 compiler explicitly supports the 251 architecture and provides access to all memory areas of the 251. Each variable may be explicitly assigned to a specific memory space.

Accessing the **data** or **near** memory is considerably faster than accessing the **xdata**, **far** or **huge** memory. For this reason, you should place frequently used variables into the **data** or **near** memory and place larger, less frequently used variables in **far** or **xdata** 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
data	Directly addressable internal data memory; fastest access to variables (128 bytes).
bdata	Bit-addressable internal data memory; allows mixed bit and byte access (16 bytes).
ebdata	Bit-addressable internal data memory, allows mixed bit and byte access. The **ebdata** address range is 20H to 7FH.
idata	Indirectly addressable internal data memory; accessed across the full internal address space (256 bytes).
near	64KB direct and indirectly addressable memory. Overlaps with the data and idata spaces; accessed by MOV @WRj or MOV dir16 instructions. If **near** is applied to a function, a LCALL or ACALL (16-bit call) is generated to this function.
far	the full 16MB address space; indirectly addressed by MOV @DRk instructions. The address calculation is performed with 16-bit operations which limites the size of a single object (array or structure) to 64KB. If **far** is applied to a function, a ECALL (24-bit call) is generated to this function.
huge	the full 16MB address space; indirectly addressed by MOV @DRk instructions. The address calculation is performed with 32-bit operations which supports objects with unlimited size.
xdata	External data memory (64 Kbytes); accessed by opcode MOVX @DPTR.
pdata	Paged (256 bytes) external data memory; accessed by opcode MOVX @Ri.
code	Code memory (64 Kbytes); accessed by opcode MOVC @A+DPTR.

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

Examples of Variable Declarations with Memory Type:

The following examples illustrate the use of the memory type within variable declarations:

```
char data var1;
char code text[] = "ENTER PARAMETER:";
unsigned long xdata array[100];
float near x, near y, far z;
unsigned char xdata vector[10][4][4];
unsigned char huge farray[100000];
unsigned char near narray[400];
const char near table[5] = { 1, 2, 3, 4, 5 };
char bdata flags;
```

Examples of Function Definitions with Memory Type:

The following examples illustrate the use of the memory type within a function definition:

```
/************************************************/
/* function with 'near' CALL and RET instruction */
/************************************************/
void near func1 (char *p)  {
  *p = 1;
}

/*******************************************************/
/* function with 'extended' ECALL and ERET instruction */
/*******************************************************/
void far func2 (char *p)  {
  *p = 1;
}
```

NOTE
*If **ROM (HUGE)** is specified, the **near** memory type can be applied to static functions to speed-up calls to this functions and to save stack space.*

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.

For global variables, the **HOLD** directive lets you specify size limits and alternative default memory spaces. Refer to HOLD on page 41 for more information.

The default memory type is determined by the **LARGE, SMALL, TINY, XSMALL** and **XTINY** compiler control directives. Refer to "Memory Models" on page 96 for more information.

The following table lists the default memory areas used for each memory model.

Memory Model	Parameters & Automatic Variables	Default Global Variables	Default Constant Variables	Default Pointer Definition	Default Pointer Size
TINY	**data**	**data**	**near**	**near ***	2 bytes
XTINY	**near**	**near**	**near**	**near ***	2 bytes
SMALL	**data**	**data**	**code**	**far ***	4 bytes
XSMALL	**near**	**near**	**code**	**far ***	4 bytes
LARGE	**xdata**	**xdata**	**code**	**far ***	4 bytes

Using a Memory Type in Pointer Declarations

The memory type can also be used in pointer declarations to define the pointer size. A pointer without explicit memory type is called generic pointer. Pointers with explicit memory type are called memory-specific pointers. A memory specific pointer allows for each variable access the generation of efficient code. In general, pointers should be declared according the following rules:

Variable Declaration	Pointer Size	Declaration of the Pointer
char c;	16/32 bit	char *ptr; (Pointer size depends from the memory model in use.)
int near nc;	16 bit	int near *np;
float data df;	8 bit	float data *dp; /* or */ float near *dp;
char idata index;	16 bit	char idata *ip; /* or */ char near *ip;
long xdata x;	16 bit	long xdata *xp;
const char code c = 'A';	16 bit	char code *cp; /* const can be ignored */
const char near n = 5;	16 bit	char near *np; /* const can be ignored */
unsigned long far l;	32 bit	long far *lp;
char huge hc;	32 bit	char huge *hc_ptr;
void near func1 (void);	16 bit	void (near *fp1) (void);
int far func2 (void);	32 bit	int (far *fp2) (void);

NOTES
Pointers to ***data*** *or* ***idata*** *objects may be declared using the* ***near*** *memory type since a* ***near*** *pointer can also access* ***data*** *or* ***idata*** *memory.*

In the ***TINY*** *and* ***XTINY*** *memory model the default pointer size is 16 bit and the default memory type for pointers is* ***near ****. *To access memory areas above 0x010000 you must specify the memory type* ***far **** *or* ***huge **** *in the pointer declaration.*

With a ***far*** *or* ***huge*** *pointer also* ***near***, ***data*** *or* ***idata*** *objects can be addressed. The difference is that* ***far*** *or* ***huge*** *pointer accesses are using the* ***@DRk*** *instructions, whereas a* ***near*** *pointer uses the more efficient* ***@WRj*** *instructions. A* ***far***, ***huge*** *or* ***xhuge*** *object cannot be addressed with a* ***near*** *pointer.*

Declaration Examples:

char near *px; declares a pointer which refers to an object of the type *char* in *near* memory type. The pointer itself is stored in the default memory type (depends from the memory model in use) and has a size of 2 bytes.

char near *data pdx;
this declaration corresponds to the prior declaration except that the pointer itself is placed explicitly into the on-chip data memory (*data*), independent of the memory model used.

struct time { char hour; char min; char sec; struct time near *pxtime; };
besides other elements, the structure shown contains a pointer *pxtime* to another structure which must located in the *near* memory type. The pointer *pxtime* has a size of 2 bytes.

struct time far *ptime;
this declaration defines a pointer which is stored in the default memory type and refers to a structure *time* which is located in the *far* memory type. The pointer *ptime* requires 4 bytes.

ptime->pxtime->hour = 12;
use of the prior two declarations; the pointer *pxtime* is indirectly loaded from the structure. It refers to the structure *time* which is located in the *near* memory type. The value 12 is assigned to the member *hour* in this structure.

Pointer Conversions

The C251 compiler can convert the memory types of pointers. Pointer conversions can be forced by explicit program code using type casts or can be coerced by the compiler. A generic pointer is identical to a **near *** pointer definition for the **TINY** or **XTINY** memory model. For all other memory models a generic pointer is the same as a **far *** pointer definition.

The C251 compiler converts 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 C251 run-time library functions such as **printf**, **sprintf**, and **gets** which use generic pointers as arguments. For example:

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

extern int myfunc (void near *p, int data *dq);

int data *dp;
const char near *fmt = "value = %d | %04XH\n";

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

In the call to **printf**, the argument `fmt` which represents a 2-byte **near** pointer is automatically converted or coerced into generic pointer. This is done because the prototype for **printf** requires a generic pointer as the first argument. Depending on the memory model the size of a generic pointer is 2-bytes or 4-bytes.

NOTE
*In the memory models **TINY** or **XTINY** it is not possible to access memory locations above 0x010000 with the standard C251 run-time library functions. Therefore the format string to the printf must be declared as **const char near**. If you declare the format string as **char code** (like it is done in the C51 programs) the printf only works in the **SMALL**, **XSMALL** or **LARGE** memory model.*

Pointer conversion are mostly done by extending or truncating the high part of the pointer, since the address space is viewed as a linear entity. The following table details the process involved in converting pointers.

Conversion Type	Description
far * to **code** *	The low-order word (2 bytes) of the **far** * is used as offset. The high-order word is discarded.
far * to **data *** **far** * to **idata** *	The low-order byte of the **far** * is used as offset.
far * to **near** *	The low-order word (2 bytes) of the **far** * is used as offset.
far * to **xdata** *	The low-order word (2 bytes) of the **far** * is used as offset.
far * to **pdata** *	The low-order byte of the **far** * offset is used as offset.
code * to **far** *	The high-order word of the **far** * is set to the **code** base (default 0xFF). The 2-byte offset of the **code** * is used.
idata * to **far** * **data** * to **far** *	The high-order word of the **far** * is set to 0x00 for **idata** / **data**. The 1-byte offset of the **idata *** / **data *** is converted to an **unsigned int** and used as low-order word.
near * to **far** *	The high-order word of the **far** * is set to 0x00 for **near**. The 2-byte offset of the **near** * is used as low-order word.
xdata * to **far** *	The high-order word of the **far** * is set to **xdata** base (default 1). The 2-byte offset of the **xdata** * is used as low-order word.
pdata * to **far** *	The high-order word of the **far** * is set to the **xdata** base (default 1). The high-order byte of the offset is set to the **pdata** page. The 1-byte offset of the **pdata *** is used as the low-order byte of the offset.
near * to **data *** **near** * to **idata** *	Only the low-order byte of the near * pointer is used as offset.
data * to **near *** **idata** * to **near** *	The 1-byte offset of the **idata *** / **data *** is converted to an **unsigned int** and used as offset for the **near** * pointer.

All pointer conversions not listed in this table are illegal
and generate a WARNING message at compile time.

The following example listing illustrates a few pointer conversions.

```
stmt level  source
   1        int *p1;           /* generic ptr (3 bytes) */
   2        int xdata *p2;     /* xdata ptr (2 bytes) */
   3        int near *p3;      /* near 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;           /* near* to generic* */
   9    1    p1 = p4;           /* code*  to generic* */
  10    1
  11    1    p4 = p1;           /* generic* to code* */
  12    1    p3 = p1;           /* generic* to near* */
  13    1    p2 = p1;           /* generic* to xdata* */
  14    1
  15    1    p2 = p3;           /* near* to xdata* (WARNING) */
*** WARNING 259 IN LINE 15 OF P.C: pointer: different mspace
  16    1    p3 = p4;           /* code*  to near* (WARNING) */
*** WARNING 259 IN LINE 16 OF P.C: pointer: different mspace
  17    1    }
```

Data Types

The C251 compiler provides you with a number of basic data types to use in your C programs. The C251 compiler offers you the standard C data types and also supports several data types that are unique to the 8051 and 251 platforms. The following table lists the available 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
double	64	8	±1.7E-308 to ±1.7E+308
idata *, data *, pdata *	8	1	0 to 0xFF
near *, xdata *, code *	16	2	0 to 0xFFFF
far *, huge *	32	4	0 to 0xFFFFFF
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 C251. These data types are described in detail in the following sections.

Bit Types

The C251 compiler 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 */
}
```

C251 supports two bit areas. By default, all **bit** variables are stored in a bit segment located in the 8051-compatible bit memory area of the 251. Because this area is only 16 bytes long, a maximum of 128 **bit** variables may be declared within any one scope.

NOTE
Accesses to the 251's extended bits generate more code than accesses to the standard 8051-compatible bit area. Therefore, you should use the standard 8051-compatibitble bit area as much as possible to maximize performance.

The 251 supports an extended bit-addressable area that is located in the data memory address range 0x20-0x7F. This extended bit area supports a maximum of 768 **bit** variables. To locate a bit in the extended bit area, specify **bit ebdata** as shown in the example below.

NOTE
It is not possible to locate bit function parameters in the extended 251 bit space.

Example for extended 251 bit declarations:

```
bit ebdata state1;          /* bit in 251 extended space         */

bit function (bit flag)  {  /* parameters are only in 8051 space */
  bit ebdata  local_bit;    /* local bit in 251 space            */

  local_bit = flag;

  return (local_bit);       /* return 251 extended bit           */
}
```

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 C251 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 can be addressed as bytes or as bits. The C251 compiler places variables declared with the memory type **bdata** or **ebdata** into the bit-addressable data memory of the 251. The memory type **bdata** refers to the 8051-compatible bit space; **ebdata** specifies the extended 251 bit space. Both the **bdata** and **ebdata** memory type are handled like the **data** memory type except that variables reside in the bit-addressable portion of the on-chip 251 data memory. Note that the total size of this area of memory may not exceed 16 bytes in case of **bdata** or 96 bytes in case of **ebdata**.

You may declare global **bdata/ebdata** variables as shown below:

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

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

int ebdata ebase;       /* Ebit-addressable int */

char ebdata eary [4];   /* Ebit-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] */

sbit ebBit0  = ebase ^ 0;     /* bit 0 of ebase   */
sbit ebBit15 = ebase ^ 15;    /* bit 15 of ebase  */

sbit ebAry07 = eary[0] ^ 7;   /* bit 7 of eary[0] */
sbit ebAry37 = eary[3] ^ 7;   /* bit 7 of eary[3] */
```

The above example represents declarations, not assignments to the bits of the **`ibase/ebase`** and **`bary/eary`** 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] */

extern bit ebdata eBit15;   /* external ebdata based bit */
extern bit ebdata ebAry37;  /* external ebdata based bit */
```

Declarations involving the **sbit** type require that the base object be declared with the memory type **bdata** or **ebdata**. Note that the declaration of external bits which are **ebdata** based require the memory type **ebdata**. Without he explicit memory, the default space **bdata** is assumed.

If you want to declare special function register based bits, the memory space must be omitted. Refer to "Special Function Registers" on page 111 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 */
```

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;
};

struct bad  {
  char     m1;
  union lft u;
} bdata tcp;                        /* ebdata would be ok too. */

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

NOTES

*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.*

*Variables declared with the **bdata** and **ebdata** type must be global; it is not possible to define bit-addressable objects inside a function.*

Special Function Registers

The 251 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 251/930 Embedded Controllers User's Manuals* or other 251 data books.

Within the 251 family, the number and type of SFRs vary. Note that no SFR names are predefined by the C251 compiler. However, declarations for SFRs are provided in include files.

The C251 compiler provides you with a number of include files for various 251 derivatives. Each file contains declarations for the SFRs available on that derivative. See "251 Special Function Register Include Files" on page 205 for more information about include files.

The C251 compiler 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 special function registers consists of two SFRs with consecutive addresses to specify 16-bit values. For example, the 80C251SB uses addresses 0xCC and 0xCD for the low and high bytes of timer/counter 2. The C251 compiler 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 251 applications, it is often necessary to access individual bits within an SFR. The C251 compiler makes this possible with the **sbit** data type. The **sbit** data type allows you to access any SFR register. The C251 compiler determines automatically, if the SFR bit can be accessed with short 8051-compatible bit instructions or uses the extended 251 instruction set.

Example:

```
sbit EA = 0xA8 ^ 7;
```

This declaration defines **`EA`** to be the SFR bit at address **`0xAF`**. On the 251, this is the *enable all* bit in the interrupt enable register.

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 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 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 8051-bit address for the **sbit** and is implemented for compatibility with the C51 compiler. For example:

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

sbit EA = 0xAF;
```

NOTES

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** and **ebdata** memory type specifier. Refer to “Bit-addressable Objects” on page 108 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:

type ⟦*memory_type*⟧ *variable_name* **_at_** *const_expr* ;

where:

type	is the variable type.
memory_type	is the memory type for the variable. If missing from the declaration, the default memory type is used.
variable_name	is the variable name.
const_expr	is an expression which must evaluate to an absolute constant value. The result of the expression 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. For the **xdata** and **code** memory types this expression is interpreted as an offset from the base address of the memory class. The C251 compiler generates the **OFFS** relocation type for **xdata** and **code** variables (Refer to the *A251 Assembler User's Guide* for more information about relocation types). For all other memory types the **_at_** expression represents the absolute memory location in the physical 251 memory. The C251 compiler checks for invalid address specifications.

With C251 Version 2 or higher, any type of variable or even a function may be located on an absolute address. The following example demonstrates how to locate several different variable types using the **_at_** keyword.

```
#define base 0x1000   /* base address for declaration */
int near var0 _at_ (base + 10);
int near var1 _at_ (base + 10 + sizeof (var0));

/* locate string at address 0xFFFF80 */
const char far string[12] _at_0xFFFF80 = "TEST PROGRAM";

/* locate text at OFFS 0x7E00, the default gives 0xFF7E00 */
char code text[20] _at_ 0x7E00 = "Hello World\n";

struct led  {
  char  digit0;
  char  digit1;
};

/* display is accessed at XDATA:0x4000, the default gives 0x014000 */
struct led xdata display _at_ 0x4000;

/* locate main at OFFS 0x8000, the default gives 0xFF8000 */
void main (void)  _at_ 0x8000  {
  display.digit0 = 'A';
  display.digit1 = '1';
}
```

With C251 you may also locate functions to absolute memory addresses. When you are using the C251 directive **ROM(HUGE)** the **_at_** address evaluates to a 251 physical memory address. Otherwise, C251 gives the function code segment the relocation type OFFS, which represents the offset address to the base memory address of the memory class code.

Function Declarations

The C251 compiler provides you with a number of extensions for standard C function declarations. These extensions allow you to:

- Choose the register bank used
- Specify a function as an interrupt procedure
- Specify reentrancy
- Select the memory model
- 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 function declarations.

⟦*return_type*⟧ *funcname* (⟦*args*⟧) ⟦{**small | 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 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.

Specifying the Register Bank for a Function

The lowest 32 bytes of all members of the 251 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 fast interrupt functions. 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 program example shows you 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          extern bit alarm;
    2          int alarm_count;
    3
    4          void falarm (void) using 3  {
    5   1        alarm_count += 4;
    6   1        alarm = 0;
    7   1      }


ASSEMBLY LISTING OF GENERATED OBJECT CODE

;       FUNCTION falarm (BEGIN)
                                             ; SOURCE LINE # 4
000000 C0D0           PUSH     PSW
```

```
000002 75D118          MOV      PSW1,#018H
                                             ; SOURCE LINE # 5
000005 7E3500      R   MOV      WR6,alarm_count
000008 0B36            INC      WR6,#04H
00000A 7A3500      R   MOV      alarm_count,WR6
                                             ; SOURCE LINE # 6
00000D C200        E   CLR      alarm
                                             ; SOURCE LINE # 7
00000F D0D0            POP      PSW
000011 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 **000Fh** 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.

Interrupt Functions

The 251 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 the 80C251SB are listed in the following table:

Interrupt Number	Interrupt Description	Address
0	EXTERNAL 0	**FF:0003h**
1	TIMER/COUNTER 0	**FF:000Bh**
2	EXTERNAL 1	**FF:0013h**
3	TIMER/COUNTER 1	**FF:001Bh**
4	SERIAL PORT	**FF:0023h**
5	TIMER/COUNTER 2	**FF:002Bh**
6	PROG. COUNTER ARRAY (PCA)	**FF:0033h**

As 251 chip vendors create new parts, more interrupts will be added. The Keil C251 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	**FF:0003h**
1	**FF:000Bh**
2	**FF:0013h**
3	**FF:001Bh**
4	**FF:0023h**
5	**FF:002Bh**
6	**FF:0033h**
7	**FF:003Bh**
8	**FF:0043h**
9	**FF:004Bh**
10	**FF:0053h**
11	**FF:005Bh**
12	**FF:0063h**
13	**FF:006Bh**
14	**FF:0073h**
15	**FF:007Bh**
16	**FF:0083h**
17	**FF:008Bh**
18	**FF:0093h**
19	**FF:009Bh**
20	**FF:00A3h**
21	**FF:00ABh**
22	**FF:00B3h**
23	**FF:00BBh**
24	**FF:00C3h**
25	**FF:00CBh**
26	**FF:00D3h**
27	**FF:00DBh**
28	**FF:00E3h**
29	**FF:00EBh**
30	**FF:00F3h**
31	**FF:00FBh**

The C251 compiler provides you with a method of calling a C function when an interrupt occurs. This support allows you to 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 data 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 **PSW** and **PSW1** 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 **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           void falarm (void) interrupt 1 using 3  {
   5   1         alarm_count *= 10;
   6   1         alarm = 1;
   7   1       }


ASSEMBLY LISTING OF GENERATED OBJECT CODE

;       FUNCTION falarm (BEGIN)
                                                  ; SOURCE LINE # 4
000000 75D118         MOV      PSW1,#018H
                                                  ; SOURCE LINE # 5
000003 7E24000A       MOV      WR4,#0AH
000007 7E3500      R  MOV      WR6,alarm_count
00000A AD32           MUL      WR6,WR4
00000C 7A3500      R  MOV      alarm_count,WR6
                                                  ; SOURCE LINE # 6
00000F D200        E  SETB     alarm
                                                  ; SOURCE LINE # 7
000011 32             RETI
;       FUNCTION falarm (END)
```

In the example above, note that the **`PSW1`** register is already saved by the 251 hardware interrupt**.** Note the **`RETI`** instruction generated to exit the interrupt.

The following rules apply to interrupt functions.

- Parameter passing is not possible. The compiler emits an error message when the interrupt procedure contains any parameter declarations.
- The return of a value is not possible. The compiler emits an error message if any attempt is made to define a return value. The return type **void** should be applied to all interrupt procedure declarations. The implicit **int** return value, however, is ignored by the compiler.
- The compiler recognizes direct invocations of interrupt functions and summarily rejects them. An interrupt procedure is exited with the **RETI** instruction. **RETI** restores the **PSW1** value and affects the interrupt system of the 251. Because no interrupt request on 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 using the C251 directive **NOINTVECTOR**. In this case, you must provide interrupt vectors from separate assembly modules. Refer to "INTVECTOR / NOINTVECTOR" on page 47 for more information about the interrupt vector table.
- The C251 compiler allows **interrupt** numbers within the 0 to 31 range. Refer to your 251 derivative document to determine which interrupts are available.
- Since most of the C251 library functions are fully reentrant there is no restriction for calling any of these functions. C251 uses in reentrant code the stack for automatic variables. Therefore you can call any C251 function which is compiled with the reentrant attribute or which uses just registers during the function execution.
- C251 never generates absolute addresses for register accesses. Each C251 function is independent from the actual registerbank in use. The **NOAREGS** directive known from the C51 Compiler is no longer required.

NOTE
*It is sometimes more efficient to remove the **using** attribute and let the C251 compiler **PUSH** the used registers. This also saves the data memory required for the additional registerbank. You can compare the code generated with or without using attribute with the C251 **CODE** directive.*

Reentrant Functions

A reentrant function may 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. By default, the C251 compiler generates static code, which means that functions cannot normally be called recursively or in a fashion which causes reentrancy. The advantage of static code is that function arguments and local variables are stored in fixed memory locations which allows the 251 CPU fast variable access. 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);
}

#pragma FUNCTIONS (reentrant)         /* C251 generates reentrant CODE */
int func (unsigned char index)  {
  return (table[index);
}

#pragma FUNCTIONS (static)            /* select static CODE again       */
```

Reentrant functions may be called recursively and may 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 or the **FUNCTIONS** directive) functions as reentrant. Reentrant code uses the 251 hardware stack area for non-register parameters and automatic variables. Refer to “Hardware Stack” on page 142 for a detailed explanation about the stack layout in reentrant code. For reentrant functions a few restrictions exists due to the 251 hardware architecture.

- **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 code cannot use the **alien** attribute since PL/M-51 argument passing is static only.
- Reentrant functions may have other attributes like **using** and **interrupt**.

NOTE
Variable access using the 251 hardware stack is slower than accessing static memory. For this reason, use reentrant functions only when required.

Specifying the Memory Model for a Function

C251 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 96 for more information.

You may, however, specify which memory model to use for a single function by including the **small** 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 on-chip 251 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.

NOTES
*C251 Version 2 or higher provides the **#pragma functions** directive. With this directive, you can specify the model or reentrant attribute for one or more functions individually without having to modify the function declaration. For more information, refer to the FUNCTIONS directive on page 38.*

Alien Function (Intel PL/M-51 Interface)

The C251 compiler 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** specifier. As alternative you can also use **#pragma FUNCTONS (PLM)** to specify PL/M-51 parameter passing. For example:

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

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:

```
char c_func (char a,  int b) alien {
  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 C251 compiler internally provides support for the **RTX251 Full** and **RTX251 Tiny** real-time multitasking operating systems. The **_task_** keyword allows you to define a function as a real-time task.

For example:

```
void func (void) _task_ num
```

where:

num	is a task ID number from 0 to 255.

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

Chapter 4. Preprocessor

The preprocessor built into the C251 compiler handles directives found in the source file. The C251 compiler 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.
if	Evaluates an expression for conditional compilation.
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.
include	Reads source text from an external file. The notation sequence determines the search sequence of the included files. The C251 compiler searches for include files specified with less-than/greater-than symbols (‘<‘ ’>‘) in the include file directory. C251 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 C251 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

The C251 compiler 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
_ _**C251**_ _	Version number of the C251 compiler (for example, 210 for version 2.10).
_ _**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, 2 for LARGE, 3 for TINY, 4 for XTINY and 5 for XSMALL).
_ _**TIME**_ _	Time when the compilation was started.
_ _**STDC**_ _	Defined to 1 to indicate full conformance with the ANSI C Standard.
_ _**FLOAT64**_ _	Defined to 0 to indicate that 32-bit floating-point numbers are used. Defined to 1 if the **FLOAT64** directive is given either by invocation or #pragma. **FLOAT64** changes the floating point math to double precision (64-bit).
_ _**MODSRC**_ _	Defined to 0 if the binary mode of 251 CPU is specified either by invocation or #pragma. Defined to 1 if the source mode of the 251 CPU is specified.

Chapter 5. Advanced Programming Techniques

This chapter describes advanced programming information for the experienced software engineer. Knowledge of most of these topics is not necessary to successfully create an embedded 251 target program using the C251 compiler. However, the following sections provide insight into how many non-standard procedures can be accomplished (for example, interfacing to C51 or 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
- Conventions the C251 compiler uses to name code and data segments
- How to interface C251 functions to assembly, C51 and PL/M-51 routines
- Data storage formats for the different C251 data types
- Different optimizing features of the C251 optimizing compiler

Customization Files

The C251 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 (**START251.A51**), code that is used to initialize static variables (**INITEDAT.A51** and **INITHDAT.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 **START251.A51** and **PUTCHAR.C.**

```
L251 MYMODUL1.OBJ, MYMODUL2.OBJ, START251.OBJ, PUTCHAR.OBJ
```

START251.A51

The **START251.A51** file contains the startup code for a C251 target program. This source file is located in the **\C251\LIB** directory. Include a copy of this file in each 251 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 **near** memory
- Clears **xdata** memory
- Clears **far** memory
- Initializes the 251 hardware stack pointer
- Transfers control to the main C function

The **START251.A51** file contains some definitions at the beginning which are used for the hardware configuration of the 251 CPU and for the setup of the C runtime system.

The functions of the individual EQU statements are described below.

Name	Description
CONFIGB	251 Configuration Bytes Definition for off-chip (external) config bytes. Set this definition to 1 if you want to generate the 251 configuration byte information at address 0xFFFFF8 and 0xFFFFF9.
WSA	Wait State for PSEN#/RD#/WR# signal except region 01:xxxx.
XALE	Extend ALE pulse.
RDRG	RD# and PSEN# memory signal selection (RD1 and RD0 Bits).
PAGM	Page mode selection.
INTR	Interrupt mode selection.
EDF	Extended data float (EDF) timing feature.
WSB	Wait State for PSEN#/RD#/WR# signal for region 01:xxxx.
EMAP	EPROM/ROM Mapping.
EDATALEN	Indicates the number of bytes of **edata** that are to be initialized to 0. The default is 420h because most 251 derivatives contain at least 1056 bytes of on-chip **edata** memory.
XDATASTART	Specifies the **xdata** address to start initializing to 10000H.
XDATALEN	Indicates the number of bytes of **xdata** to be initialized to 0. The default is 0.
HDATASTART	Specifies the **hdata** address to start initializing to 10000H.
HDATALEN	Indicates the number of bytes of **hdata** to be initialized to 0. The default is 0.
STACKSIZE	Specifies the number of bytes use as hardware stack area. The **STACKSIZE** value must be adjusted according the actual requirements of the application. The default is 100H bytes for stack size.
IBPSTACK	Not required in C251 Version 2 or higher, only for compatibility to C51 & C251 Version 1.x. 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.
IBPSTACKTOP	Specifies the top start address of the small model reentrant stack area. The default is 0xFF in idata memory. The C51 or C251 Version 1 compiler 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	Not required in C251 Version 2 or higher, only for compatibility to C51 & C251 Version 1.x. 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. The C51 or C251 Version 1 compiler 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.

INITEDAT.A51, INITHDAT.A51

These files contain the routines for data initialization of variables that where explicitly initialized in the C source file. **INITEDAT.A51** performs the initialization for variables located in the data, idata and near memory space; **INITHDAT.A51** initializes the xdata, far and huge memory space.

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 use an 80C251SB, the macro could be defined as follows:

```
WATCHDOG    MACRO
            MOV    WDTRST,#01EH
            MOV    WDTRST,#0E1H
            ENDM
```

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 C251 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 C251 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 C251 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 C251 program runs as quickly as possible.

The C251 compiler provides six different levels of optimizing. Each increasing level includes the optimizations of the levels below it. Refer to "OPTIMIZE" on page 59 for more information.

General Optimizations

Optimization	Description
Constant Folding	Multiple constant values in an expression or address calculation are combined into a single constant.
Jump Optimizing	Jumps are extended to the final target address. Jumps to jumps are eliminated. Conditional jumps are examined to determine if they can be streamlined or eliminated by the inversion of the test logic.
Dead Code Elimination	Code which cannot be reached (dead code) is removed from the program. This includes code which is not executed due to constant conditional expressions or unreferenced labels.
Register Variables	Automatic variables and function arguments are located in registers when possible. No data memory is reserved for these variables.
Parameter Passing Via Registers	A maximum of three function arguments may 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.
Life Variable Analysis	Removes redundant store operations to automatic variables.
Constant Propagation	The values of expressions are tracked and when possible constant values are inserted instead of variable accesses.

251-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 L251 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)	Some C constructs (like shift) are coded as loops to reduce code size.
NOALIAS	Disable the alias checking for pointer accesses. C251 ignores pointer write optimizations during the optimization phase. If a register holds a variable it gets reused, even when a pointer could modify that variable.

Segment Naming Conventions

Objects generated by the C251 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 absolute and has a memory class and a segment name.

Some segment names generated by C251 include the ***module*** name and optionally the ***function*** name. The ***module*** name is the name of the C source file in which the object is declared and excludes the drive letter, path specification, and file extension. The ***function*** name refers to the function which defines the objects. The ***function*** name may be slightly modified as described under "Function Symbols" on page 140. All segment names are converted to uppercase letters and are therefore not case sensitive. For example, the segment ?PR?ERROR_CHECK?SAMPLE refers to the function **error_check** in the file **SAMPLE.C**.

The C251 compiler assigns all variables and functions to a segment. The only exception are reentrant variables which are stored on the hardware stack of the 251 CPU. For each memory type used in the C source a separate segment is generated. The segment name contains a ***prefix*** followed by the module name. The ***prefix*** depends on the memory type and the C **const** modifier.

The C251 compiler generates four different formats for segment names. The format depends on the information contained in the segment:

- Static segments contain data and const variables which are defined at file level of the C module or inside a C function as static or const. Segments of these type have the format ***?prefix?module***. Also program code segments use this format, if you apply the C251 directive **ROM (HUGE)** or **ROM (MEDIUM)**.
- Overlayable segments are used in non-reentrant code and hold data variables which are declared inside a C function. These segments are typically overlaid during link time and use the format ***?prefix?function?module.*** This is also the default format for program code segments.
- Absolute segments are used for variable or function declarations with the ***_at_*** keyword and have the format ***?prefix?AT_address?module***. For xdata and code segments the format is ***?prefix?OF_address?module***, since these segments represent offsets to the memory class base address. ***address*** refers to the absolute address specified in the C source file.

- Code for interrupt vectors is store in segments with the format ***?PR?IV?n*** whereby ***n*** represents the interrupt number of the function.

The following table shows the segment name ***prefix*** together with the memory type and their associated memory class names. The ***prefix*** corresponds to the memory class used for the segment and is enclosed in question marks (?). The memory class names are used during the link/locate process to simplify address assignments and ensure that objects from different modules are placed in a valid address range. Refer to the *L251 Linker/Locater User's Guide* for more information about locating segments.

Segment Prefix	Memory Type	Description
?PR?	**---**	Executable program code in **CODE** or **ECODE** class.
?CO?	**code**	Constants in the **CODE** memory class.
?ED?	**near**	**EDATA** memory class.
?NC?	**const near**	**NCONST** memory class.
?FD?	**far**	Far variables in **HDATA** class (within 64 Kbyte block).
?FC?	**const far**	Far constants in **HCONST** class (within 64 Kbyte block).
?HD?	**huge**	Huge variables in **HDATA** class (no block limitation).
?HC?	**const huge**	Huge constants in **HCONST** class (no block limitation).
?XD?	**xdata**	**XDATA** memory class.
?DT?	**data**	**DATA** memory class.
?ID?	**idata**	**IDATA** memory class.
?BI?	**bit**	8051 compatible bits in **BIT** memory class.
?EB?	**bit ebdata**	Extended 251 bits in **EBIT** memory class.
?BA?	**bdata**	8051 bit-addressable variables in **DATA** memory class.
?EA?	**ebdata**	Extended 251 bit-addressable variables in **DATA** class.
?PD?	**pdata**	Page xdata variables in **XDATA** memory class.

Initialization Segments

Segments which contain initialization data or code have a fixed predefined names. The information in this segments is use in the startup code which is executed prior the main function.

Segment Name	Memory Class	Description
?C_INITEDATA	**NCONST**	contains initialization values for data, idata and near variables.
?C_INITHDATA	**NCONST**	contains initialization values for xdata, far and huge variables.
?C_C51STARTUP **?C_C51STARTUP1 ..** **C_C51STARTUP3**	**CODE**	Startup code for the C main function. Most of the startup code is contained in the files **START251.A51**, **INITEDAT.A51** and **INITHDAT.A51**. However, the startup code contains also initialization code for bits and small data objects which are stored in a C251 generated segment. The L251 Linker/Locater locates all startup segments in ascending order without gaps.

Function Symbols

By default, C251 uses the unmodified function name as symbol reference in the object files. The function names for code which is reentrant, uses C51 parameter passing, or ends with the ERET instruction are modified slightly to avoid run-time errors. The following table lists symbol name generated for the function declaration:

```
void TestFunction (char c) ...
```

Symbol Name	Description
TestFunction	For standard code generation the function names are not modified.
TestFunction?	For code which ends with returns with a ERET instruction a question mark (?) is appended at the symbol name.
TestFunction?_	For reentrant code a (?) and (_) is appended at the symbol name.
TestFunction??	For reentrant code which returns with a ERET instruction two question mars (??) are appended at the symbol name.
TESTFUNCTION	For functions with PL/M-51 or C51 parameter passing the function name is converted in uppercase.
_TESTFUNCTION	For functions with C51 parameter passing, the function name is converted in uppercase and prefixed with ”_” when the function receives parameters in registers.

All symbol names are converted in uppercase letters when the C251 directive **NOCASE** is applied. Refer to “NOCASE” on page 56 for more information.

Parameter Passing Symbols

Function arguments were historically passed using fixed memory locations. This is still true for routines written in PL/M-51. However, the C251 compiler can pass up to 9 function arguments in registers. Other arguments are passed using the traditional fixed memory areas. The parameter areas must be publicly known to any calling module. So, they are publicly defined using the following symbol names:

```
?function?BYTE
?function?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. Depending on the memory model, the **?*function*?BYTE** symbol refers to a DATA, EDATA or XDATA memory location. Refer to "Example Routines" on page 149 for examples of the function argument symbols.

Hardware Stack

The design of the 251 architecture is biased towards allowing structured languages like C. The instruction set of the CPU therefore supports stack-relative variables with the MOV @DR60 instruction. The stack direction of the 251 hardware stack is from the bottom upwards. The hardware stack pointer is initialized in the file START251.A51 which is executed before the C *main* function is called. The START251.A51 reserves also the space for the hardware stack.

The 251 hardware stack holds parameters and automatic variables for reentrant code. The stack layout during the execution of a reentrant function is show in the following example. For information about reentrant code refer to “Reentrant Functions” on page 123.

C Function

```
pragma functions (reentrant)

void func (long a, long b, int c)  {
  char  x1, x2;
  int   y;
  float z;

  :
  :
}
```

User Stack Layout after Execution of the Epilog of *func*:

`int c`	`@DR60-19`
`Ret Addr.`	`@DR60-17`
`long b`	`@DR60-15`
`long a`	`@DR60-11`
`float z`	`@DR60-7`
`int y`	`@DR60-3`
`char x2`	`@DR60-1`
`char x1`	`@DR60`

NOTES

The above stack layout is generated if C251 works without optimization (C251 control directive OPTIMIZE (0) is in affect). In regular C programs the C251 optimizer allocates parts of the stack variables to CPU registers.

The shown code is called with standard LCALL instructions. If you translate with the ROM (HUGE) directive the space required for the return address is 3 bytes.

*The parameters **a** and **b** are passed in registers and therefore located on the stack after the return address. The parameter **c** cannot be passed anymore in registers and is located before the return address on stack.*

Calculating Stack Size

The C251 compiler cannot calculate exact stack sizes of your application, since the compiler does not detect the actual program flow and does not know about the interrupt functions in your system. The 251 hardware stack is also used for storing reentrant variables, return addresses and PUSH/POP instructions. The size of the hardware stack area is defined in the file START251.A51. The default value of the system stack is 100 bytes which is typically enough for most non-reentrant applications. If you use heavily reentrant code in your application, you should increase the value for the hardware stack to 500 bytes. If you want to reduce the stack size, we suggest the following procedure for calculating the user stack size of your system:

Stop the program execution at the *main* entry and fill the memory area allocated to the hardware stack with a constant value (i.e. 0xA5). The address range of the user stack is listed in the linker map file (*.MAP) under the headline MEMORY MAP OF MODULE. The segment **?STACK** lists the address range for the hardware stack.

Execute all functions of your application and check how many constants are overwritten. We recommend that you have at least enough space to execute the most complex interrupt function of your system. At good value for the spare space on the user stack is 20 bytes.

If your application behaves strange and if you think that the stack size could be a problem, you can also increase the user stack size and try again.

Interfacing C Programs to Assembler

You can easily interface C251 routines to routines written in 8051 assembler. The A251 assembler is a 251 macro assembler that emits object modules in OMF-251 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

C functions may pass parameters in registers and/or fixed memory locations. The **REGPARMS** and **NOREGPRAMS** directives enable and disable parameter passing using registers. Parameters are passed in fixed memory locations if parameter passing in registers is disabled, or if too many parameters are passed to fit in registers. Typically, you will use register parameters, so the enclosed examples only describe this way of parameter passing. Refer to “Using the SRC Directive” on page 148 for an example.

Passing Parameters in Registers

The C251 Compiler uses the registers R11 and R0-R7 for parameter passing. A maximum of 9 parameters may be passed in registers. All other parameters are passed using fixed memory locations or the 251 hardware stack (depending on the function reentrant / static attribute). The following table lists what registers are used for passing parameters.

Allocation Order	char, 1-byte ptr	int, 2-byte ptr	long, float 4-byte ptr	double
1	R11	WR6 (MSB in R6, LSB in R7)	DR4	DR0, DR4
2	R6	WR4	DR0	
3	R5	WR2		
4	R4	WR0		
5	R3			
6	R2			
7	R1			
8	R0			

For parameter passing to C51 functions 3-byte pointers are used; 3-byte pointers are passed in the registers R1/R2/R3. Refer to "PARM51 / PARM251" on page 66 for more information.

C251 allocates registers in the order R11, R7, R6, R5, R4, R3, R2, R1, and R0. If a register is already in use for a parameter variable, C251 continues down the list until a free register is found.

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 WR6.
func2 (int b, int c, int far *d)	The first argument, ***b***, is passed in register WR6. The second argument, ***c***, is passed in register WR4. The third argument, ***d***, is passed in registers DR0.
func3 (long e, long f long g)	The first argument, ***e***, is passed in register DR4. The second argument, ***f***, is passed in register DR0. The third argument, ***g***, cannot be located in registers since those available for a type of long are already used by other arguments. This parameter is passed using fixed memory locations or the 251 hardware stack.
func4 (char h, float i)	The first argument, ***h***, passed in registers R11. The second parameter, ***i***, is passed in register DR0, since the first character argument also blocks the register R7 from being used for other variables.

Parameter Passing in Fixed Memory Locations

Parameters passed to assembly routines in fixed memory locations use symbols named ***?function?BYTE*** and ***?function?BIT*** to hold the parameter values passed to the ***function***. Bit parameters are copied into the memory referred by ***?function?BIT*** prior to calling the function. All other parameters are copied into the memory starting at ***?function_name?BYTE***. Parameters are stored in the order in which they are specified in the function declaration.

The fixed memory locations used for parameter passing may be in the memory class **DATA**, **EDATA** or **XDATA** depending upon the memory model used.

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	Single bit returned in the carry flag
char / unsigned char, 1-byte pointer	R11	Single byte returned in R11
int / unsigned int, 2-byte pointer	WR6	MSB in R6, LSB in R7
long / unsigned long	DR4	MSB in R4, LSB in R7
float	DR4	32-Bit IEEE format: 'exponent' and 'sign' in WR4
double	DR0, DR4	64-Bit IEEE format: 'exponent' and 'sign' in WR0
4-byte pointer	DR0	MSB in R0, LSB in R3

For parameter passing to C51 functions; 3-byte pointers are returned in the registers R1/R2/R3. Refer to "PARM51 / PARM251" on page 66 for more information.

Using the SRC Directive

The **SRC** directive instructs the C251 compiler to generate an assembly routine instead of an object module. You may use this directive to generate the shell of an assembly source file or to help you determine the parameter passing conventions the compiler expects. The following C source file:

```
#pragma SRC   /* the SRC directive generates assembler output */
#pragma XTINY /* specify the memory model you want to use     */

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

generates the following assembly output file.

```
$NOMACRO
$MODSRC
$MODEL(5)
$CASE
;
; 'ASM1.SRC' generated from 'ASM1.C'
; COMPILER INVOKED BY:
;        C:\C251\BIN\C251V2.EXE ASM1.C
;
                NAME ASM1
        ?PR?ASMFUNC1?ASM1  SEGMENT  CODE

;--- special function registers and bits:
             ACC  DATA  0E0H
             PSW  DATA  0D0H
             DPL  DATA  082H
             DPH  DATA  083H
               B  DATA  0F0H
              SP  DATA  081H
            PSW1  DATA  0D1H
               Z  BIT   0D1H.1
              EA  BIT   0A8H.7
;--- end of special function registers and bits.

        PUBLIC        asmfunc1
; line 1: #pragma SRC   /* the SRC directive generates assembler output */
; line 2: #pragma XTINY /* specify the memory model you want to use     */
; line 3:
; line 4:
; line 5: unsigned int asmfunc1 (unsigned int arg)  {
        RSEG       ?PR?ASMFUNC1?ASM1
     USING  0
        asmfunc1 PROC
;---- Variable 'arg' assigned to Register 'WR6' ----
; line 6:   return (1 + arg);
        INC      WR6,#01H
; line 7: }
        RET
        ENDP
        END
```

5

Register Usage in Assembler Subroutines

In an assembler subroutine, the following registers can be used without prior saving of their contents: R0 to R11, WR16 to WR30, PSW, PSW1, and DPTR. DR60 is the *STACK POINTER*; at the end of the assembler subroutine the value of DR60 must be the same as at the entry of the routine. If the subroutine uses the registers R12 to R15, the contents of the registers must be saved and restored at return to the calling C function.

When calling a C function from assembler, it must be assumed that the register contents of R0 to R11, WR16 to WR30, PSW, PSW1, and DPTR are destroyed during the processing of the C function. The PSW flag UD is not used in C251 code can be freely used.

Overlaying Segments

If the overlay process is executed during program linking and locating, it is important that each subroutine are declared with the PROC / ENDP assembler directive or 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 C251 compiler segment naming conventions.
- Each assembler function with local variables must be assigned its own data segment. This data segment must be declared with the relocation type OVERLAYABLE. The overlayable data segment may be accessed by other functions only for passing parameters.

Example Routines

The following program examples show how to pass parameters to and from assembly routines. The equivalent C source for the assembler program is printed as assembler comment.

TINY Model

```
LOC    OBJ          LINE     SOURCE
                       1     $CASE ; make CASE sensitive for C interface
                       2
                       3                     NAME A251_MODULE
                       4
                       5     ; /***********************************/
                       6     ; /* Sample Assembler to C Interface */
                       7     ; /***********************************/
                       8     ;
                       9     ; extern char func1 (int a1, char a2, bit b);
                      10     ;
                      11     ; extern bit flag;
                      12     ; extern char val;
                      13     ;
                      14     ; int func2 (int *ip, char *cp)  {
                      15     ;   char c;
                      16     ;
                      17     ;   c = func1 (*ip, *cp, flag);
                      18     ;   val += c;
                      19     ;   return (val);
                      20     ; }
                      21
                      22
                      23     EXTRN  BIT       (flag)
                      24     EXTRN  DATA:BYTE (val)
                      25     EXTRN  CODE:NEAR (func1)
                      26     EXTRN  BIT       (?func1?BIT)
                      27
------                28     ?PR?FUNC2?ASM1  SEGMENT  CODE
                      29
                      30     PUBLIC func2  ; entry point for func2
                      31
------                32             RSEG       ?PR?FUNC2?ASM1
                      33     func2 PROC NEAR
                      34     ; Variable 'cp' is passed in Register 'WR4'
                      35     ; Variable 'ip' is passed in Register 'WR6'
                      36
                      37     ; func1 is called with parameters
                      38     ; copy flag to parameter area
                      39     ; all other parameters are passed in registers
000000 A200         E  40            MOV      C,flag
000002 9200         E  41            MOV      ?func1?BIT,C
000004 0B3830          42            MOV      WR6,@WR6
000007 7E29B0          43            MOV      R11,@WR4
00000A 120000       E  44            LCALL    func1
                       45     ; func1 returns a character in R11 (same as A)
00000D 2500         E  46            ADD      A,val
00000F F500         E  47            MOV      val,A
                       48     ; func2 returns a integer in WR6
000011 1A3B            49            MOVS     WR6,R11
000013 22              50            RET
                       51            ENDP
                       52
                       53            END
```

XTINY Model

```
LOC    OBJ          LINE     SOURCE

                       1     $CASE ; make CASE sensitive for C interface
                       2
                       3                     NAME A251_MODULE
                       4
                       5     ; /***********************************/
                       6     ; /* Sample Assembler to C Interface */
                       7     ; /***********************************/
                       8     ;
                       9     ; extern char func1 (int a1, char a2, bit b);
                      10     ;
                      11     ; extern bit flag;
                      12     ; extern char val;
                      13     ;
                      14     ; int func2 (int *ip, char *cp)  {
                      15     ;   char c;
                      16     ;
                      17     ;   c = func1 (*ip, *cp, flag);
                      18     ;   val += c;
                      19     ;   return (val);
                      20     ; }
                      21
                      22
                      23     EXTRN  BIT        (flag)
                      24     EXTRN  EDATA:BYTE (val)
                      25     EXTRN  CODE :NEAR (func1)
                      26     EXTRN  BIT        (?func1?BIT)
                      27
------                28     ?PR?FUNC2?ASM1  SEGMENT  CODE
                      29
                      30     PUBLIC func2  ; entry point for func2
                      31
------                32             RSEG       ?PR?FUNC2?ASM1
                      33     func2 PROC NEAR
                      34     ; Variable 'cp' is passed in Register 'WR4'
                      35     ; Variable 'ip' is passed in Register 'WR6'
                      36
                      37     ; func1 is called with parameters
                      38     ; copy flag to parameter area
                      39     ; all other parameters are passed in registers
000000 A200       E   40             MOV      C,flag
000002 9200       E   41             MOV      ?func1?BIT,C
000004 0B3830         42             MOV      WR6,@WR6
000007 7E29B0         43             MOV      R11,@WR4
00000A 120000     E   44             LCALL    func1
                      45     ; func1 returns a character in R11
00000D 2EB30000   E   46             ADD      R11,val
000011 7AB30000   E   47             MOV      val,R11
                      48     ; func2 returns a integer in WR6
000015 1A3B           49             MOVS     WR6,R11
000017 22             50             RET
                      51             ENDP
                      52
                      53             END
```

SMALL Model

```
LOC    OBJ         LINE     SOURCE

                      1     $CASE ; make CASE sensitive for C interface
                      2
                      3                     NAME A251_MODULE
                      4
                      5     ; /**********************************/
                      6     ; /* Sample Assembler to C Interface */
                      7     ; /**********************************/
                      8     ;
                      9     ; extern char func1 (int a1, char a2, bit b);
                     10     ;
                     11     ; extern bit flag;
                     12     ; extern char val;
                     13     ;
                     14     ; int func2 (int *ip, char *cp)  {
                     15     ;   char c;
                     16     ;
                     17     ;   c = func1 (*ip, *cp, flag);
                     18     ;   val += c;
                     19     ;   return (val);
                     20     ; }
                     21
                     22
                     23     EXTRN  BIT        (flag)
                     24     EXTRN  DATA :BYTE (val)
                     25     EXTRN  CODE :NEAR (func1)
                     26     EXTRN  BIT        (?func1?BIT)
                     27
------               28     ?PR?FUNC2?ASM1  SEGMENT  CODE
                     29
                     30     PUBLIC func2  ; entry point for func2
                     31
------               32             RSEG       ?PR?FUNC2?ASM1
                     33     func2 PROC NEAR
                     34     ; Variable 'cp' is passed in Register 'DR4'
                     35     ; Variable 'ip' is passed in Register 'DR0'
                     36
                     37     ; func1 is called with parameters
                     38     ; copy flag to parameter area
                     39     ; all other parameters are passed in registers
000000 A200       E  40             MOV      C,flag
000002 9200       E  41             MOV      ?func1?BIT,C
000004 7E1BB0        42             MOV      R11,@DR4
000007 0B0A30        43             MOV      WR6,@DR0
00000A 120000     E  44             LCALL    func1
                     45     ; func1 returns a character in R11 (same as A)
00000D 2500       E  46             ADD      A,val
00000F F500       E  47             MOV      val,A
                     48     ; func2 returns a integer in WR6
000011 1A3B          49             MOVS     WR6,R11
000013 22            50             RET
                     51             ENDP
                     52
                     53             END
```

XSMALL Model

```
LOC    OBJ          LINE     SOURCE

                       1     $CASE ; make CASE sensitive for C interface
                       2
                       3                     NAME A251_MODULE
                       4
                       5     ; /**********************************/
                       6     ; /* Sample Assembler to C Interface */
                       7     ; /**********************************/
                       8     ;
                       9     ; extern char func1 (int a1, char a2, bit b);
                      10     ;
                      11     ; extern bit flag;
                      12     ; extern char val;
                      13     ;
                      14     ; int func2 (int *ip, char *cp)  {
                      15     ;   char c;
                      16     ;
                      17     ;   c = func1 (*ip, *cp, flag);
                      18     ;   val += c;
                      19     ;   return (val);
                      20     ; }
                      21
                      22
                      23     EXTRN  BIT        (flag)
                      24     EXTRN  EDATA:BYTE (val)
                      25     EXTRN  CODE :NEAR (func1)
                      26     EXTRN  BIT        (?func1?BIT)
                      27
------                28     ?PR?FUNC2?ASM1  SEGMENT  CODE
                      29
                      30     PUBLIC func2  ; entry point for func2
                      31
------                32             RSEG       ?PR?FUNC2?ASM1
                      33     func2 PROC NEAR
                      34     ; Variable 'cp' is passed in Register 'DR4'
                      35     ; Variable 'ip' is passed in Register 'DR0'
                      36
                      37     ; func1 is called with parameters
                      38     ; copy flag to parameter area
                      39     ; all other parameters are passed in registers
000000 A200         E  40            MOV      C,flag
000002 9200         E  41            MOV      ?func1?BIT,C
000004 7E1BB0          42            MOV      R11,@DR4
000007 0B0A30          43            MOV      WR6,@DR0
00000A 120000       E  44            LCALL    func1
                       45     ; func1 returns a character in R11
00000D 2EB30000     E  46            ADD      R11,val
000011 7AB30000     E  47            MOV      val,R11
                       48     ; func2 returns a integer in WR6
000015 1A3B            49            MOVS     WR6,R11
000017 22              50            RET
                       51            ENDP
                       52
                       53            END
```

LARGE Model

```
LOC    OBJ         LINE     SOURCE

                      1     $CASE ; make CASE sensitive for C interface
                      2
                      3                     NAME A251_MODULE
                      4
                      5     ; /***********************************/
                      6     ; /* Sample Assembler to C Interface */
                      7     ; /***********************************/
                      8     ;
                      9     ; extern char func1 (int a1, char a2, bit b);
                     10     ;
                     11     ; extern bit flag;
                     12     ; extern char val;
                     13     ;
                     14     ; int func2 (int *ip, char *cp)  {
                     15     ;   char c;
                     16     ;
                     17     ;   c = func1 (*ip, *cp, flag);
                     18     ;   val += c;
                     19     ;   return (val);
                     20     ; }
                     21
                     22
                     23     EXTRN  BIT        (flag)
                     24     EXTRN  XDATA:BYTE (val)
                     25     EXTRN  CODE :NEAR (func1)
                     26     EXTRN  BIT        (?func1?BIT)
                     27
------               28     ?PR?FUNC2?ASM1  SEGMENT  CODE
                     29
                     30     PUBLIC func2  ; entry point for func2
                     31
------               32             RSEG       ?PR?FUNC2?ASM1
                     33     func2 PROC NEAR
                     34     ; Variable 'cp' is passed in Register 'DR4'
                     35     ; Variable 'ip' is passed in Register 'DR0'
                     36
                     37     ; func1 is called with parameters
                     38     ; copy flag to parameter area
                     39     ; all other parameters are passed in registers
000000 A200       E  40             MOV      C,flag
000002 9200       E  41             MOV      ?func1?BIT,C
000004 7E1BB0        42             MOV      R11,@DR4
000007 0B0A30        43             MOV      WR6,@DR0
00000A 120000     E  44             LCALL    func1
                     45     ; func1 returns a character in R11
00000D 7C7B          46             MOV      R7,R11
00000F 900000     E  47             MOV      DPTR,#WORD0 val
000012 E0            48             MOVX     A,@DPTR    ; A=R11
000013 2CB7          49             ADD      R11,R7
000015 F0            50             MOVX     @DPTR,A    ; A=R11
                     51     ; func2 returns a integer in WR6
000016 1A3B          52             MOVS     WR6,R11
000018 22            53             RET
                     54             ENDP
                     55
                     56             END
```

Data Storage Formats

This section describes the storage formats of the data types available in C251. The C251 compiler 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
double	64	8	±1.7E-308 to ±1.7E+308
data *, idata *, pdata *	8	1	0x00 to 0xFF
code*, xdata *, near *	16	2	0x0000 to 0xFFFF
far *, huge *	32	4	Linear 16 Mbyte address 0x000000 to 0xFFFFFF

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 251 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 251 CPU. The L251 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, Enum, Pointers to near, xdata, and code

Scalars of type **int**, scalars of type **short**, **enum** types, and memory-specific pointers that reference **near**, **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 Pointers to far and huge

Scalars of type **long** and memory-specific pointers that reference **far** or **huge** 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

Floating-Point Numbers

Scalars of type **float** are stored using 4 bytes (32 bits). Numbers of the type **float** have a precision of 24 bits, which represents about 7 decimal digits. 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.

Numbers of the type **float** are stored in 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 24-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 would appear 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 this example, we will use the value for -12.5 shown above.

Note that the floating-point storage representation is not an intuitive format. To convert this value to a floating-point number, the bits should be separated as specified in the floating-point number storage format table shown above.

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

From the illustration above, 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
```

Binary digits that are to the left of the decimal point represent the power of two corresponding to their position. For example, **`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 to the right of the decimal point also represent the power of two corresponding to their position. However, because these digits are to the right of the decimal point, the powers are negative.

For example, **`.100`**… represents $(1 \times 2^{-1)} + (0 \times 2^{-2}) + (0 \times 2^{-3}) + \ldots$ which equals **`.5`**.

Adding these values together, we get **`12.5`**. Because the sign bit was set, we must include a negative sign. So, the hexadecimal value **`0xC1480000`** is **`-12.5`**.

Float Errors

Since the 251 does not contain an interrupt vector to trap floating-point errors, your software must appropriately respond to these error conditions.

In addition to the normal floating-point values, a **float** 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	0xFFFFFFFF	Not a number
+INF	0x7F80000	Positive infinity (positive overflow)
-INF	0xFF80000	Negative infinity (negative overflow)

NOTE
*The C251 library function **_chkfloat_** allows you to quickly check floating-point status. Refer to "_chkfloat_ / _chkdouble_" on page 224 for more information.*

Double Numbers

Scalars of type **double** are stored using 8 bytes (64 bits). Numbers of the type **double** have a precision of 53 bits, which represents about 13 decimal digits. 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 11-bit value in the range 0-2047. It is stored relative to 1023. The actual value of the exponent is calculated by subtracting 1023 from the stored value (0-2047). The value of the exponent can be anywhere from +1024 to -1023.

The mantissa is a 53-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.

Numbers of the type **double** are stored in memory on word boundaries using the following format:

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

Address	+4	+5	+6	+7
Contents	MMMM MMMM	MMMM 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 1023.
M	is the 53-bit normal mantissa. The highest bit is always 1 and, therefore, is not stored

Double Errors

Since the 251 does not contain an interrupt vector to trap floating-point errors, your software must appropriately respond to these error conditions.

In addition to the normal floating-point values, a **double** 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	0xFFFFFFFF FFFFFFFF	Not a number
+INF	0x7FF0000 00000000	Positive infinity (positive overflow)
-INF	0xFFF0000 00000000	Negative infinity (negative overflow)

NOTE
*The C251 library function **_chkdouble_** allows you to quickly check floating-point status on numbers of the type **double**. Refer to "_chkfloat_ / _chkdouble_" on page 224 for more information.*

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. Nonetheless, there are three ways to reference explicit memory location. You may use the:

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

Each of these three methods is described below.

The _at_ Keyword

The best method for accessing absolute memory locations is to use the **_at_** keyword when you declare variables in your C source files. The **_at_** keyword also reserves the memory required for the object. In cases where you do not want to reserve memory, you may use the absolute memory access macros described below. For a detailed description of the **_at_** keyword refer to “Absolute Variable Location” on page 114.

Absolute Memory Access Macros

Absolute memory access macros are provided as part of the C251 library. Use the following macros to directly access the memory areas of the 251. Refer to “Absolute Memory Access Macros” on page 192 for definitions of these macros.

CARRAY	**DVAR**	**NBYTE**
CBYTE	**HARRAY**	**NVAR**
CVAR	**HBYTE**	**XARRAY**
DARRAY	**HVAR**	**XBYTE**
DBYTE	**NARRAY**	**XVAR**

NOTE
*The absolute memory access macros do not reserve memory at link/locate time. Therefore, it is possible to access any memory location in the 251 memory. The linker has no way to check if the memory location is reserved in a proper way. If you want to reserve memory for the variable access, you should use the **_at_** keyword described above.*

Linker Location Controls

The third 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 L251 linker/locator to specify an absolute memory address. You may use this method together with the "ORDER" directive described on page 61 to ensure that all variables inside a module are located in ascending order.

In the following example, assume that we have a structure called **`alarm_ctrl`** that we want to reside at address 02000h in **EDATA**. 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;
 };

struct alarm_st near alarm_ctrl;
.
.
.
```

The C251 compiler generates an object file for **ALMCTRL.C** and includes a segment for variables in the **near** memory type. Because it is the only variable declared in this module, **`alarm_ctrl`** is the only variable in that segment. The name of the segment is ?ED?ALMCTRL.

The L251 linker/locator allows you to specify the base address of any segment by using the location control directive SEGMENTS. The L251 **SEGMENTS** directive must be used as follows:

```
L251 … almctrl.obj SEGMENTS(?ED?ALMCTRL(02000h)) …
```

This instructs the linker to locate the segment named ?ED?ALMCTRL at address 02000h in the linear 16MB address space.

Debugging

The C251 compiler uses the Intel Object Format (OMF-251) for object files and generates complete symbol information. All compatible emulators may be used for program debugging. The **DEBUG** control directive embeds debugging information in the object file.

Chapter 6. Error Messages

This chapter lists Fatal Error, Syntax Error, and Warning messages that you may encounter as you develop an application. 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:

```
C251 FATAL-ERROR -
      ACTION:    <current action>
      LINE:      <line in which the error is detected>
      ERROR:     <corresponding error message>
C251 TERMINATED.
```

```
C251 FATAL-ERROR -
      ACTION:    <current action>
      FILE:      <file in which the error is detected>
      ERROR:     <corresponding error message>
C251 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 FILE/WORKFILE`
The compiler could not create the list file, object file, or temporary file. This error may occur if the disk is full or write-protected, or if the file already exists and is read only. For temporary files, check the current setting of the environment variable **TMP**. Make sure that **TMP** refers to an existing drive and/or directory.

`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 parsing and evaluating arguments on the command line or while evaluating parameters in a **#pragma** statement.

`PARSING SOURCE-FILE`
The source file violates a compiler limit, for example the statement nesting exceeds the maximum of 32 levels.

`PREPROCESSING INPUT FILE`
The compiler could not complete the preprocessing phase.

`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.

‘=’ EXPECTED
The specified WARNING control or pragma is syntactically incorrect.

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.

INVALID FUNCTION MODEL SPECIFIER
The function model selection is invalid. For example, functions using the PLM model can’t be reentrant.

INVALID SUB-CONTROL
The argument given to the **ROM** control must be one of: **SMALL, MEDIUM, COMPACT, LARGE,** or **HUGE**.

LINE TOO LONG
The maximum length of the invocation line must not exceed 500 characters. The same limit applies also if the compiler controls are passed via command file (@cmdfile).

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 C251 compiler uses a DOS extender to utilize all extended memory available.

MORE THAN 400 ERRORS IN SOURCE-FILE
During the compilation more than 400 errors were detected. This causes the termination of the compiler.

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

NUMERIC VALUE OUT OF RANGE
The numerical argument of a control parameter is out of range. For instance, the **OPTIMIZE** control allows the numbers 0 through 8. A value of 9 would generate this error message.

PARSE STACK OVERFLOW/ STATEMENST ARE TOO NESTED
The parse stack has overflowed. This can occur if the source program contains extremely complex expressions or if blocks are nested more than 32 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 a conflicting command-line parameters was 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 CLASSNAME

The first parameter given to the **USERCLASS** control must be the name of a predefined class name. The predefined classes are: **UCODE, DATA, IDATA, PDATA, XDATA, NEAR, FAR,** and **HUGE**.

UNKNOWN CONTROL

The selected control parameter is unrecognized by the compiler.

Errors and Warnings

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.

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.

Errors and warnings produce a message in the list file. These messages are in the following format:

```
*** WARNING number IN LINE line OF file: warning message
*** 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.
message	is descriptive text and is dependent upon the type of error or warning encountered.

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

For warning messages, the default warning level is listed in brackets. Refer to the "WARNING" directive on page 86 for more information on how to change the warning level.

Number	Message and Description
E1	**misplaced primary control, line ignored** Primary controls must be specified at the start of the C module before any #include directives or declarations.

Number	Message and Description
W2 [1]	**unknown #pragma/control, line ignored**
	The specified control is undefined. The compiler ignores the control along with the rest of the pragma line.
E3	**'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.
E4	**save/restore: save-stack overflow/underflow**
	The maximum nesting depth of #pragma save comprises eight levels. The pragma SAVE/RESTORE work with a stack according to the LIFO (last in, first out) principal.
E5	**'asm' requires SRC control to be active**
	The use of asm and endasm in a source file requires that the file is compiled using the SRC directive. The compiler then generates an assembly source file which is to be assembled using A251.
E6	**invalid argument given to #pragma**
	A numeric argument is out of range, for example the threshold value specified in the hold directive.
E7	**unused.**
E8	**Code size exceeds 64K**
	The maximum code size per module is limited to 64K bytes.
W9 [1]	**function pointer: 'huge' changed to 'far'**
	A function pointer with a **huge** specifier such as **int (huge *xpi) (void)** is changed to **int (far *xpi) (void)** since **huge** apply to data pointers, not to function pointers. Function pointers should use **near** or **far** if explicit override of model based pointer sizes is required.
W10 [2]	**conversion: 'type1' <operator> 'type2'**
	This message warns about pointer conversions and type conversions in assignment expressions such as **unsigned short** = pointer. In case of pointer conversions, it means that a pointer value is converted to a non pointer value.
E11	**internal error: <message>**
	C251 recognized an internal consistency problem. Please contact technical support if this error occurs.
W12 [1]	**unused.**
W13 [1]	**unsafe conversion: 'type' truncated to short pointer**
	A type such as long or unsigned long has been converted to a short pointer. Such an operation truncates the long value to a int value. A conversion of the short pointer back to a long value may yield a different result.
W14 [1]	**'operator': unsafe conversion - boolean to pointer**
	The result of a boolean expression such as ==, >=, ||, && has been converted to a pointer. Since the result could be 1, the resulting pointer may be invalid because the resulting memory selector part of the pointer may be invalid.
E15	**floating point error**
	This error occurs when a floating point argument lies outside of the valid range for 32-bit or 64-bit floating values.

Number	Message and Description
E16	**unprintable character 0xnn skipped** An illegal character was found in the source file. (Note that characters inside a comment are not checked.)
E17	**unclosed string** A string is not terminated with a quote (").
E18	**string too long** A string may not contain more than 4096 characters.
E19	**'\c': invalid character constant** A character constant has an invalid format. The notation '\c' is valid only when c is any printable ASCII character.
E20	**'\esc': value exceeds 255** A character constant has a value greater than 0xFF.
E21	**'c': illegal octal digit** The specified character is not a valid octal digit.
E22	**constant too big** The value of the specified constant cannot be represented specified character is not a valid octal digit.
E23	**bad syntax in floating point number** The fractional part of the specified floating point number should start with a digit.
E24	**hex constants must have at least one hex digit** A hex constant with no hex digits after the 0x prefix was detected.
E25	**syntax error near 'token'** The `token` as seen by the compiler is wrong. This usually results from missing semicolons or misspelled keywords.
E26	**expression too complex, simplify** The specified expression is too complex. Break it into two or more simpler expressions.
E27	**illegal type specifier** The type specifier is illegal - for example: signed struct x or unsigned float.
E28	**more than one storage class** A declaration may not have more than one storage class.
E29	**formal/parameter has illegal storage class** Formal parameters of a function type must not have a storage class other than **register**.
E30	**same type qualifier used more than once** A type qualifier (const or volatile) is specified more than once in one type declaration.
E31	**declarator too complex (20)** The declaration of an object may contain a maximum of 20 type modifiers such as nested array, pointer or function types.

Number	Message and Description
E32	**type-stack underflow** The type declaration stack has an underflow. This error is usually a side effect of error 31.
E33	**‘identifier’: invalid storage class** The storage class for the specified identifier is invalid. An example for this error would be storage class **static** or **extern** specified for a member of a structure or union.
W34 [2]	**missing declaration specifiers** A declaration of an identifier with no type and storage class information causes this type of warning.
W35 [2]	**‘identifier’: uses old style declarator** The function definition uses the old style declarator format where the return type is omitted. The return type defaults to type **int**. This may cause runtime problems if far or huge pointers are returned because of pointer truncation.
E36 [1]	**missing declarator** A type has been declared but no identifier, for example: unsigned long; /* just a type but no identifier */
E37 [2]	**nonstandard extension - unnamed parameter** The compiler has found an unnamed parameter or just a typedef name. This is a violation of the ANSI specification. C251 allows this for compatibility to compilers from other vendors.
W38 [2]	**empty translation unit** The specified source file is empty, it contains no external or file level declarations.
E39	**‘identifier’: typedef cannot be used for function definition** A function definition cannot be initiated by a typedef’d function type.
W40 [1]	**‘type’ converted to ‘<mspace>’ pointer** A non pointer value has been converted to a pointer value. If for example an **int** is converted to a far or huge pointer, then the memory space selector part of the pointer will be zero which is probably not what you want.
E41	**duplicate function-parameter** A formal parameter name exists more than once within the list of parameters of the function. The formal parameter names are required to be unique.

E42	**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 */`
	`{`
	`  /* … */`
	`}`

Number	Message and Description
E43 [1]	**mspace on parameter ignored** A function parameter cannot have an explicit memory space. The compiler derives the memory space for parameters based on the memory model used for that function. An exception to this rule is an array or function type. In this case, a parameters is converted to a pointer anyway. If a memory space is given, then a spaced pointer is created.
E44	**illegal use of mspace specifier** An mspace specifier cannot be used for the members of a structure or union type. It may be used to direct a structure or union (or any other type) to a specified memory space.
E45	**memory attribute ignored** unused in C251.
W46 [1]	**storage class 'class' ignored** The specified storage class has been ignored by the compiler. An example would be **register struct x1;** defined within a function. Since compound types are never placed in registers, the compiler ignores the storage class **register**.
W47 [1]	**'identifier': unreferenced <item>** An unreferenced label, local variable or parameter has been detected.
E48	**'identifier': struct/union type redefinition** A structure or union type has been redefined.
E49	**'tag-name': not a union tag** The tag name specified in the union type does not specify a union type tag.
E50	**'tag-name': not a struct tag** The tag name specified in the struct type does not specify a struct type tag.
E51	**'tag-name': not an enum tag** The tag name specified in the enum type does not specify an enumeration type tag.
W52 [2]	**use of undefined type 'tag-name'** An undefined struct or union type has been used. Depending on context, this message may be issued as an error or a warning.
E53	**redefinition of 'identifier'** The type of an identifier has been redefined, it does not conform to the type specified in an earlier declaration.
E54	**'identifier': duplicate label** Definitions of labels within a function must use unique names.
E55	**'identifier': undefined label** The specified label name referred to by a goto statement is undefined within the current function.
E56	**'{', scope stack overflow (31)** The maximum of 31 nested blocks has been exceeded. Additional levels of nested blocks are ignored.
W57 [2]	**parameter <num>: different from declaration** The parameter with number <num> has been defined with a type which is not compatible with an earlier function prototype.

Number	Message and Description
E58	**‘identifier’: requires ANSI style prototype.** A call to a **non-reentrant** functions with the **NOREGPARMS** attribute requires a prototype of that function. For example, when passing the constant value 1, C251 cannot anticipate the required type of the constant which could be bit, char/unsigned char or another type. The ANSI integer promotion rule is still not a solution because of the special cases of bit and character types.
E59	**‘=’: address of rvalue taken** The address of a structure member returned by a function cannot be taken. **Example:** p1 = &structFunc (1, 2)**.**nCount;
E60	**‘identifier’: duplicate member** The specified identifier is already a member of some struct or union type.
E61	**duplicate parameter** The specified identifier is already defined in the list of function parameters.
E62	**actual parameters must fit into registers** The actual parameters in an indirect function call must fit into registers, assumed that the function is non-reentrant. This limitation does not apply to reentrant functions.
E63	**‘bit’ illegal in reentrant function** A reentrant function cannot have parameter or local variables of type bit. Bit variables are never located on the stack, they are located in bit-addressable memory.
E64	**‘identifier’: static function declared but not defined** A static function was declared but not defined. Since a static function must be defined somewhere in the source module, this error signals a function which was not defined.
E65	**illegal use of ‘near/far/huge’** The language extension keywords near, far and huge are not allowed in with structure members or function parameters.
E66	**invalid dereference** An invalid dereference has been encountered. This error may be issued in case of previous errors where the type of an object may be unknown or invalid.
E67	**‘identifier’: undefined identifier** The identifier used in an expression is undefined.
E68	**left side of ‘->’ requires struct/union pointer** The left side of a member selection expression must be a pointer to a struct or union type.
E69	**left side of ‘.’ requires struct/union** The left side of a member selection expression must be a struct or union type.
E70	**‘identifier’: undefined member** The right side of a member selection expression must be the name of a member of the left sides struct or union type.
E71	**subscript requires array or pointer type** The left side of an array access expression **array[x]** does not represent an array or pointer type. In case of a pointer type, the pointer must not be of type **void *p**.

Number	Message and Description
W72 [1]	**sizeof returns zero** The calculation of an object size yields zero. This may happen for example if **sizeof (*vp)** is specified where vp represents a void pointer.
E73	**sizeof illegal operand taken** This error occurs if sizeof is applied to a function type. Such operations are illegal.
E74	**type follows 'void'** Prototype declarations of functions may contain a void parameter list (for example, int func (void)). This notation may not contain further type definitions after the void keyword.
E75	**'void' follows other types** Type **void** cannot follow other types in a function parameter list.

E76	**formal parameter ignored**
	
	This error is caused by a nested function prototype where one or more parameter are specified in an identifier list as shown:
	
	`int test (void)  {`
	`  int  i;`
	`  extern int Gm(i1);  /* illegal prototype */`
	
	`  i = func (1, 2);`
	`  return (i + 10)`
	`}`

E77	**'identifier': invalid or conflicting memory space**
	
	An invalid or conflicting memory space was detected. The reason may be a function with a memory space other than **near** or **far** or a conflict, where two memory spaces are present - one in the declaration and another one introduced by a spaced typedef type as shown below:
	
	`int idata f(void)  {  /* idata invalid for f() */`
	`  return (1);         /* should be near or far */`
	`}`
	
	`typedef char near NC; /* NC = char in near memory */`
	`NC far cc1;           /* far - near conflict */`

E78	**'plm-function' cannot be reentrant** The parameter passing convention for plm functions uses static data memory.
E79	**'*' 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.
E80	**'*' illegal indirection** The * operator may not be applied on non-pointer arguments.
E81	**call not to a function** The term of a function call does not evaluate to a function type or pointer to function type.
E82	**'+': two pointers** The sum of two pointers is not a legal C operation.

Number	Message and Description
E83	**‘operator’: incompatible types** An expression with operands of incompatible types has been detected. An example for such an operation is ***p = s1;** where p1 is of type **int *** and s1 is a structure.
E84	**illegal type conversion** An expression causes an invalid type conversion such as float to pointer or bit to pointer or if the target of the conversion is a function type.
E85	**‘void’ illegal with all types** A type conversion cannot have **void** as the target type. An exception to this rule is (void) test (1,2) where the return value of the function test is discarded without a type conversion actually performed.
E86	**‘operator’: void illegal with all types** The expression with the specified operator has at least one void type operand which is clearly invalid.
E87	**illegal type conversion on ‘struct/union’ type** A struct or union type cannot be converted to any type.
E88	**‘operator’: invalid left/right operand type** The specified expression operator has an invalid typed left or right operand.

W89 [2]	**‘operator’: different levels of indirection**
	A pointer initialization uses a pointer with a different level as an initializer. Different level means pointer vs. pointer to pointer.
	`int   i;`
	`int **ppi = &i;   /* ptr-ptr-int = ptr-int */`

E90	**‘operator’: different struct union types** Structure assignment must use the same struct or union type on both sides of the specified operator. The same applies when a structure is passed to a function by value.
W91 [1]	**‘operator’: pointer to different objects** The operands of the specified operator are pointers to different objects. For example: pi = pl where pi is int * and pl is long * will give warning 91.
E92	**‘operator’: requires l-value** The left hand side of an assignment must evaluate to an address which represents an object which can be modified.
E93	**‘operator’: bit or bit-field not allowed** An expression tried to take the address of a bit or bit field, this is not legal in C.
E94	**‘identifier’: declare with void parameter list** A function declared with a void parameter list must not be activated with parameters.
E95 [1]	**‘identifier’: too many actual parameters** A function call expression with too many actual parameters has been detected. In case of a non-reentrant function, Error 95 is displayed instead of Warning 95.
W96 [1]	**‘identifier’: too few actual parameters** A function call expression with too few actual parameters has been detected.

Number	Message and Description
E97	**parameter <number>: incompatible types** An actual parameter uses a type not compatible with an earlier function prototype or function definition, for example a structure is passed when another typed value is expected.
W98 [1]	**parameter <number>: pointer to different objects** An actual parameter uses a pointer to an object which is different from an earlier function prototype or function definition. An example would be passing a pointer to int where a pointer to long is expected.
E99	**'operator': l-value specifies a 'const' object** Objects declared with the **const** type qualifier cannot be modified.
E100	**'operator': left/right operand must be l-value** The specified operand must evaluate to an address which represents an object which can be modified.
W101 [1]	**l-value specifies a 'const/code' object** An l-value with memory type **code** may not represent the address of an object which can be modified.
W102 [2]	**'operator': different const/volatile qualifiers** The pointer operand(s) use different qualifiers. For example: p1 = p2 where p1 is **char volatile *** and p2 is **char const ***.
E103	**illegal function definition (missing ';' ?)** An illegal function definition was detected. This is usually caused by forgotten semicolons which may change the meaning of declarations.
E104	**different function types** A conditional expression (**e1 ? e2 : e3**) specifies different function types for expressions e2 and e3.

W105 [1]	**unnamed type definition in parentheses**
	A declaration of an unnamed type within a parameter list has been detected. Unnamed types are those declared without a tag name, such as an enumeration tag as shown below:
	`void test ( enum { a1, a2, a3 } x )  {`
	`    /* … */`
	`}`

E106	**'&': on constant** It is not allowed to take the address of a constant.
W107 [1]	**address of automatic: dangling reference** A function returns the address of an automatic object. This is a dangling reference because the value accessed later via pointer may be undefined.
E108	**unused.**
E109	**illegal constant expression** An invalid constant expression has been found. Constant expressions must not have float or double type when used for specification of a **using** or **interrupt** number, for example.
E110	**non-constant case/dim expression** A constant expression is required for case values and dimension values for arrays. Such an expression must not have components other than constants.

Number	Message and Description
W111 [2]	**div by zero** A division by zero was detected while evaluating a constant expression.
W112 [2]	**mod by zero** A modulus by zero was detected while evaluating a constant expression.

W113 [3]	**nonstandard extension: cast on lvalue was used**
	Casting the left hand side of an assignment is a nonstandard operation. The compiler allows this for historical reasons, as long as the size of the left hand value is not changed:
	`long    l1;`
	`void main (void)  {`
	`  (unsigned long) l1 += 3;   /* this gives the warning */`
	`}`

W114 [3]	**'operator': integral constant overflow** A constant overflow was detected while evaluating a constant expression.
W115 [1]	**'operator': applied to unsigned type, result still unsigned** The unary plus or minus operator has been applied to an unsigned type. The result of such an expression is still unsigned.
E116	**negative subscript** The constant value specifying a dimension size of an array must not be negative.
E117 [1]	**constant expression: value out of range** The constant value is out of range. This may happen for example when a bit field size uses a value greater than 16 or an address constant is not within the 16MB address range. Depending on context, E117 will show as an error or a warning.
W118 [1]	**'identifier': enumeration value out of range** This warning is issued on an enumeration declaration where an enumerator has a different signed value.

E119	**'identifier': bit-field type too small for number of bits**
	A bit field with a width greater than the width (in bits) of the base scalar was specified, for example:
	`struct fld  {`
	`   unsigned char f1 : 10;     /* 10 is illegal, maximum is 8 bits */`
	`};`

E120	**'identifier': named bit field cannot have 0 width** A named bit field must not have a width of zero bits.
E121	**'identifier': char/int type required for bit fields** The compiler supports bit fields based on signed and unsigned char, int and short int types.
E122	**'identifier': unknown size** The size of the specified object is unknown. This is for example the case, if an array declared within a function does not specify a dimension size.
E123	**'identifier': allocation exceeds 64K** An object declared near or far must not have a size greater than 64K. If you require a bigger object, you should use the huge specifier.

Number	Message and Description
W124	**‘identifier’: illegal pointer: bit ***
	A pointer cannot refer to a bit scalar since the CPU architecture does not support this kind of addressing.
E125	**‘identifier’: array of invalid type**
	An array of void or function types is not valid in C.
E126	**‘identifier’: function returning invalid type**
	A function cannot return a function type, change the declaration to return a pointer to a function.
E127	**‘identifier’: member has invalid type**
	A member of a structure or union cannot have type bit, function or void.
E128	**‘identifier’: self relative struct/union**
	A structure or union cannot have an instance of itself as a member.
E129	**‘identifier’: uses undefined struct/union ‘tagname’**
	The type tag name designates an undefined structure or union type.
E130	**‘break/continue’: missing enclosing loop**
	A break or continue statement must be placed within a for, while or do statement.
E131	**‘case/default’: missing enclosing switch**
	A case or default statement requires an enclosing switch statement.
E132	**case value ‘<value>’ already used**
	The case values in a switch statement must be unique.
E133	**more than one default**
	A switch statement cannot have more than one default statement.
E134	**unused.**
W135 [1]	**‘function name’: no return value**
	The specified function does not return a value at all. This may cause runtime problems since random values will be used by the caller.
E136	**return value on ‘void’ function**
	A function declared with a void return value cannot return a value.
W137 [1]	**constant in condition expression**
	This message warns about a possibly misspelled operator such as **if (i = 5)** where **if (i == 5)** was intended.
W138 [1]	**expression with possibly no effect**
	A dead expression such as **i << 4** was detected. Since the result of such an expression is not used anywhere, the compiler will remove the expression. This message is usually the result of a wrong operator, for example << where <<= was intended.
E139	**expression has illegal type**
	The type of a condition expression must not have struct, union or function type.

Number	Message and Description
W140 [1]	**'identifier' undefined; assuming extern int identifier()** A function call to a function with no currently existing prototype has been found. The compiler assumes that the function is external and returns an int value. The parameters if any, will be promoted using the default promotion rules.
E141	**'identifier': too many initializers** An initialization statement provides more initializers than required, for example char array[5] = { 1,2,3,4,5,6, };
E142	**illegal initialization** An attempt was made to initialize a symbol which represents a typedef name or a function parameter.
E143	**'identifier': initializer is not a constant** Initializer expressions for file level objects must be constant expressions. Such an expression can use constants and addresses of objects.
E144	**'identifier': initialization needs curly braces** Initialization of an array or a structure needs curly braces around the initializers. An exception to this rule is the initialization of an automatic structure or union, where another structure or union without curly braces can be specified for the initializer.

W145 [3]	**non-lvalue struct converted to pointer**
	A member of a structure or union directly returned by some function as been accessed. Since the returned structure or union is stored in a temporary location, the compiler converts the struct.member access type to a ptr->member access type.
	`struct abc  { int m; int z; };`
	`extern struct abc sfunc (void);`
	`int  x;`
	
	`void main (void)  {`
	`  if (sfunc().z > 3) x = 1;   /* sfunc().z: non-lvalue struct */`
	`}`

W146 [2]	**'identifier': slightly different type** The type of the initializer expression does not exactly match the type of the object being initialized.
E147	**unused.**
E148	**initializer uses an auto-address** A function local static object must not be initialized with a stack based address in a reentrant function.
W149 [2]	**potential pointer truncation** An expression causes a pointer to be converted to a pointer with smaller size. Due to truncation, the result may be an invalid pointer.
W150 [2]	**'mspace1' pointer converted to 'mspace2' pointer** A pointer with memory space 1 (near, far, huge, …) has been converted to a pointer with a different memory space. Depending on the memory spaces, a pointer may get truncated.
W151 [2]	**pointer truncation 'mspace1' to 'mspace2'** A pointer has been truncated to a pointer with smaller size.

Number	Message and Description
E152	**bad type conversion: ‘type1’ to ‘type2’**
	It is not possible to convert a float or double type constant to a pointer.
W153 [3]	**‘operator’: different spaced pointer (mspace1,mspace2)**
	A comparison of pointers with different memory spaces was attempted. In general, it makes no sense to compare pointers with different memory spaces since they could refer to different segments or pages.
E154	**cannot allocate an array of constant size 0**
	An array must not have dimension a zero size.
E155	**invalid base address**
	The base address of a SFR or sbit declaration does not specify the address of a special function register
E156	**invalid sbit declaration**
	A sbit declaration must use the address of a sfr or a **bdata** object and a bit position in range 0-7. If a bdata based sbit is declared, the bit position range depends on the base scalar. If the base scalar is a long value, then the bit position range is 0-31, for example.
E157	**unused.**
E158	**unused.**
E159	**invalid or out of range ‘_at_’ address**
	The absolute address given in an _at_ declaration violates the address boundaries for the given memory space. For example. **‘int idata x1 _at_ 0x100’** is invalid since an idata address must be in range 0x00 to 0xFF.
E160	**‘operator’: invalid memory space**
	The source or target address of a structure or union type copy specifies an invalid memory space, for example a bit address.
W161 [2]	**‘mspace’ pointer truncated to ‘<type>’**
	A pointer type has been truncated to a non-pointer type.
E162	**unused.**
E163	**unused.**
E164	**unused.**
E165	**unused.**
E166	**unused.**
E167	**intrinsic ‘name’: declaration/activation error**
	An intrinsic function has an invalid declaration or the activation of an intrinsic function uses invalid parameters. You should include the INTRINS.H file in your source file to provide consistent declaration of the intrinsic functions. You may also consult INTRINS.H for details on parameter requirements for the individual intrinsic functions.
E168	**‘item’: value not in range <n1>…<n2>**
	A numeric argument was out of range, for example the value given to **using** was outside the valid range 0 to 3.
E169	**illegal task definition (taskid/priority)**
	The task definition uses out of range numeric parameters. The taskid must be in range 0-0xFF the range for priority is 0-0x7F.

Number	Message and Description
E170	**constant value not in range num1 ... num2** The constant value is not within the specified range
E171	**interrupt/using: can’t receive or return values** A function declared with the using registerbank attribute or an interrupt function cannot receive parameters or return a value to the caller.
E172	**‘segment-name’: segment too big (act=actvalue, max=maxvalue)** Some segment violates the maximum size given by maxvalue. Example: the total size of all edata or near objects within one source module must not exceed 64K.
W173 [2]	**‘operator’: signed/unsigned type mismatch** The arguments for a comparison operator have different signs.
W174 [1]	**‘identifier’: unreferenced ‘static’ function** An unreferenced static function has been detected. Since static functions cannot be activated from outside the module, you may remove the function.
E175	**‘identifier’: more than <maxval> local bits** A single function may not have more than 128 (bit) or 768 (ebit) local bit declarations.
E176	**‘identifier’: automatics require more than 16384 bytes** The maximum size for automatic objects per functions is 16K.
W177 [3]	**unreachable code** The optimizer has detected one or more paths which cannot be reached from the regular control flow within a function. This message usually results from condition expressions with constant assignments
E178	**unused.**
W179 [1]	**‘function name’: out of memory while optimizing function** The compiler could not allocate enough memory for optimization of the specified function. The generated code is still correct, although it may be less efficient because one or more optimizer stages could not be executed.

W180 [1]	**‘function name’: not every path returns a value**
	Control flow analysis yields one or more exit paths which do not provide a return value. The example shows a situation where the warning would be issued:
	
	`int testfunc (int i, int k)  {`
	`   if (i < k) return (i + k);      /* true path returns a value */`
	`}                                 /* false path returns a random value ! */`

E181	**‘identifier’: unknown size** A static file level object with undefined size has been detected. This is not allowed since there is no BSS segment where such definitions are collected.
W182 [1]	**‘identifier’: unknown size, assumed to be ‘extern’** A non static file level object with an undefined size has been detected. The object is assumed to be defined in another source module.

Number	Message and Description

W183 [3]	**dead assignment eliminated**
	The optimizer has removed a dead assignment, that is, an assignment to a local variable whose value is not used subsequently. Note that such assignments may result from the different optimizer stages:
	`int func (void)  {`
	`  int    i, j, k;`
	
	`  i = 5;`
	`  j = i + 10;                  /* j = 15 */`
	`  k = 10;`
	`  return ( i + j * k);        /* result is 155 */`
	`}`
	The optimizer tracks constant assignments and finally evaluates the constant expression to calculate the return value. In this case, the assignments to the variables are dead and will be removed because their values are never used.

W184 [3]	**value of 'identifier' possibly undefined** Data flow analysis shows an access to a local variable which may be undefined. This is the case if no definition of the variable reaches its use.
E185	**unused.**
E186 [1]	**meaningless use of an expression** The compiler has comparison with a constant which never yields true. For example, comparing the value of an unsigned char with a constant value greater than 0xFF is meaningless because it false regardless of the unsigned char value.
E187	**'identifier': size of actuals exceeds 'maxargs'** The total size of the parameters passed to a non-reentrant ellipsis function must not exceed 'maxargs'. This value is either 15 bytes in SMALL and TINY models and 40 bytes in all other memory models. If you need to enlarge the maximum size for your application, then you should increase the value with the **MAXARGS** directive. Note that all modules of your project should use exactly the same maxargs value.
E188 [2]	**'identifier': truncation of constant value** A constant value has been truncated by an explicit type conversion (cast).
E189 [2]	**pointer constant truncated to <n> bits** A constant pointer value has been truncated. The number of bits depends on the referring memory space of the pointer. A data * pointer constant is truncated to 8 bits.
E190 [1]	**'function()': recursive call to non-reentrant function** A recursive call to a non-reentrant function has been detected. Since the local variables are not located on the stack, they will be overwritten.
E191 [1]	**'function()': potential runtime problem with K&R parameters** Passing parameters to a non-reentrant function where a K&R function prototype (empty parameter list) may cause runtime problems (see also E58). K&R prototypes should be replaced by ANSI prototypes which specify the full type of each parameter. This avoids ambiguities with bit and char types.

Number	Message and Description
E192	**‘identifier’: different ‘near/far’ attribute** An attempt has been made to override a near or far attribute introduced via typedef such as shown below: typedef void near nFunc (void); typedef nFunc far *p1; /* gives error 192 */

E193 [1]	**memory space causes ‘identifier’ to be located in memory**
	A memory space is applied to a automatic variable in a reentrant function. This causes that the variable is located in static memory in the specified memory class. The function is therefore no longer reentrant, since the variable is not located on the stack.
	void test (void) reentrant {
	  int xdata i1; /* gives warning 193 */
	  :
	  :

E194 [1]	**obsolete declaration: use <type> space identifier** An old-style C51 declaration is used in the source program. You must change this to the new order as outlined in the following example: xdata int i1; /* gives warning 194 */ int xdata i1; /* this is the correct declaration */
E300	**unterminated comment** This message occurs when a comment does not have a closing delimiter (*/).
E301	**identifier expected** The syntax of a preprocessor directive expects an identifier.
E302	**misused # operator** This message occurs if the stringize operator (‘#’) is not followed by an identifier.
E303	**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.
E304	**bad macro parameter list** The macro parameter list does not represent a brace enclosed, comma separated list of identifiers.
E305	**unterminated string/char constant** A string or character constant is invalid. Typically, this error is encountered if the closing quote is missing.
E306	**unterminated macro call** The end of the input file was reached while the preprocessor was collecting and expanding actual parameters of a macro call.
E307 [1]	**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. In case of too many parameters, message 307 is shown as a warning.

Number	Message and Description
E308	**invalid integer constant expression**
	The numerical expression of an if/elif directive contains a syntax error.
E309	**bad or missing file name**
	The filename argument in an include directive is invalid or missing.
E310	**conditionals too nested (20)**
	The source file contains too many nested directives for conditional compilation. The maximum nesting level allowed is 20.
E311	**misplaced elif/else control**
E312	**misplaced endif control**
	The directives elif, else, and endif are legal only within an if, ifdef, or ifndef directive.
E313	**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:
	`__C251__` **`__FLOAT64__`** **`__DATE__`**
	`__FILE__` **`__MODEL__`** **`__LINE__`**
	`__STDC__` **`__TIME__`** **`__MODSRC__`**
E314	**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, …).
E315	**unknown # directive ‘name’**
	The name of the preprocessor directive is not known to the compiler.
E316	**unterminated conditionals**
	The number of endifs does not match the number of if or ifdefs after the end of the input file.
W317 [1]	**redefinition of macro ‘macro name’**
	The macro with the specified name is already defined with different parameter and or macro body.
E318	**can’t open file ‘filename’**
	The specified file could not be opened.
E319	**‘file’: not a disk file**
	The specified file is not a disk file. Files other than disk files are not acceptable for compilation.
E320	**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.
E321	**missing <character>**
	In the filename argument of an include directive, the closing character is missing. For example: #include <stdio.h>

Number	Message and Description
W322 [1]	**unknown identifier**
	The integer constant expression uses an undefined identifier.
W323 [1]	**newline expected, extra character found**
	A preprocessor directive is terminated by one or more extra characters, which are not interpreted. The directive should be terminated by a newline character.
W324 [1]	**preprocessor token expected**
	A preprocessor token was expected but a newline character was found.
E325	**duplicate formal parameter 'name'**
	The names of the formal parameters of a macro must be unique.
E326	**macro body cannot start or end with '##'**
	The concatenation operator (##) cannot be the first or last token of a macro body.
E327	**macro 'macroname': more than 50 parameters**
	The number of parameters per macro is limited to 50.
E328	**misplaced 'asm/endasm'**
	The preprocessor found nested asm directives. Asm and endasm must not be nested.

Chapter 7. Library Reference

The C251 run-time library provides you with more than 100 predefined functions and macros to use in your 251 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 251 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. The floating pointer error handling is performed via the functions **_chkfloat_** and **_chkdouble_** rather then using a global variable **errno** for the error status. This method fits the needs of embedded systems, where the error status is checked separately in the main and interrupt function or in each RTX251 task. 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 C251 compiler supports a number of intrinsic library functions. Non-intrinsic functions generate some **CALL** instructions to perform the library routine. Intrinsic functions generate inline code to perform the library routine. The generated inline code is much faster and more efficient than a called routine would be. The following functions are available in intrinsic form:

chkdouble	**_irol_**	**_nop_**
chkfloat	**_iror_**	**_testbit_**
crol	**_lrol_**	
cror	**_lror_**	

These routines are described in detail in the following sections.

Library Files

The C251 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
C2BT.LIB	Tiny model library without floating-point arithmetic (Binary Mode)
C2BFPT.LIB	Tiny model floating-point arithmetic library (Binary Mode)
C2BDPT.LIB	Tiny model double precision arithmetic library (Binary Mode)
C2BXT.LIB	XTiny model library without floating-point arithmetic (Binary Mode)
C2BFPXT.LIB	XTiny model floating-point arithmetic library (Binary Mode)
C2BDPXT.LIB	XTiny model double precision arithmetic library (Binary Mode)
C2BS.LIB	Small model library without floating-point arithmetic (Binary Mode)
C2BFPS.LIB	Small model floating-point arithmetic library (Binary Mode)
C2BDPS.LIB	Small model double precision arithmetic library (Binary Mode)
C2BXS.LIB	XSmall model library without floating-point arithmetic (Binary Mode)
C2BFPXS.LIB	XSmall model floating-point arithmetic library (Binary Mode)
C2BDPXS.LIB	XSmall model double precision arithmetic library (Binary Mode)
C2BL.LIB	Large model library without floating-point arithmetic (Binary Mode)
C2BFPL.LIB	Large model floating-point arithmetic library (Binary Mode)
C2BDPL.LIB	Large model double precision arithmetic library (Binary Mode)
C2ST.LIB	Tiny model library without floating-point arithmetic (Source Mode)
C2SFPT.LIB	Tiny model floating-point arithmetic library (Source Mode)
C2SDPT.LIB	Tiny model double precision arithmetic library (Source Mode)
C2SXT.LIB	XTiny model library without floating-point arithmetic (Source Mode)
C2SFPXT.LIB	XTiny model floating-point arithmetic library (Source Mode)
C2SDPXT.LIB	XTiny model double precision arithmetic library (Source Mode)
C2SS.LIB	Small model library without floating-point arithmetic (Source Mode)
C2SFPS.LIB	Small model floating-point arithmetic library (Source Mode)
C2SDPS.LIB	Small model double precision arithmetic library (Source Mode)
C2SXS.LIB	XSmall model library without floating-point arithmetic (Source Mode)
C2SFPXS.LIB	XSmall model floating-point arithmetic library (Source Mode)
C2SDPXS.LIB	XSmall model double precision arithmetic library (Source Mode)
C2SL.LIB	Large model library without floating-point arithmetic (Source Mode)
C2SFPL.LIB	Large model floating-point arithmetic library (Source Mode)
C2SDPL.LIB	Large model double precision arithmetic library (Source Mode)

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 **\C251\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 201 for more information.

Standard Types

The C251 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   9
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 C251 standard library contains definitions for macros that allow you to access explicit memory addresses. These macros are defined in **ABSACC.H** and allow you to access the different memory spaces of the 251 hardware. The absolute memory access macros do not reserve memory at link/locate time. If you want to reserve memory for an object, you should use the **_at_** keyword described under “Absolute Variable Location” on page 114. The function and usage of the absolute memory access macros are explained below.

CARRAY, DARRAY, HARRAY, NARRAY, XARRAY

These macros can be used to access an array of type object at the fixed absolute address base. The **ARRAY** macros scales the index by the size of type object and adds the result to base. The final address is then used to access the memory. The difference of the various macros is that various 251 memory classes are accessed. For example generates the **DARRAY** macro the most efficient code while the **HARRAY** macro allows to access the complete 16 Mbytes memory without any address limitations. The ARRAY macros are defined as follows:

```
#define CARRAY(object, base) ((object volatile code *) (base)) /* CODE  */
#define DARRAY(object, base) ((object volatile data *) (base)) /* IDATA */
#define HARRAY(object, base) ((object volatile huge *) (base)) /* HDATA */
#define NARRAY(object, base) ((object volatile near *) (base)) /* EDATA */
#define XARRAY(object, base) ((object volatile xdata *) (base))/* XDATA */
```

You may use these macros in your programs as shown in the example below:

```
#include <absacc.h>       /* include absolute memory access macros     */
   :
   :
int i;
long l;

l = NARRAY (long, 0x8000)[i];            /* long array at EDATA 0x8000 */
NARRAY (long, 0x8000)[i] = 0x12345678;    /* assign a value             */

#define DualPortRam HARRAY (int, 0x18000) /* int array at HDATA 0x18000 */
DualPortRam[l] = 0x1234;                  /* assign a value             */
```

CBYTE, DBYTE, HBYTE, NBYTE, XBYTE

These macros allows you to access individual bytes in the various 251 memory classes are defined as follows. You can access with the various macros only the memory class which is used in the macro definition. The **DARRAY** macro generates the most efficient code while the **HARRAY** macro enables access the complete 16 Mbytes address space without any limitations.

```
#define CBYTE ((unsigned char volatile code  *)0)  /* CODE memory       */
#define DBYTE ((unsigned char volatile data  *)0)  /* DATA/IDATA memory */
#define HBYTE ((unsigned char volatile huge  *)0)  /* HDATA memory      */
#define NBYTE ((unsigned char volatile near  *)0)  /* EDATA memory      */
#define XBYTE ((unsigned char volatile xdata *)0)  /* XDATA memory      */
```

You may use these macros in your programs as follows:

```
#include <absacc.h>       /* include absolute memory access macros      */
   :
   :
rval = CBYTE [0x0002];    /* read contents of code memory at 0002H      */
DBYTE[0x50] = 5;          /* write 0x05 to data memory at 000050H       */

#define IO NBYTE[0x8000]  /* define IO as address 0x8000 in near memory */
if (IO) IO = 0;           /* test address 008000H and assign a value    */

unsigned long address;
HBYTE[address] = 0;       /* set memory at location address to 0        */
```

CVAR, DVAR, HVAR, NVAR, XVAR

These macros can be used to access variables of any type at fixed absolute memory locations. The **VAR** macros accesses the variable type specified by *object* at the memory location *addr*. The difference of the various macros is that various 251 memory classes are accessed. For example generates the **DARRAY** macro the most efficient code while the **HARRAY** macro allows to access the complete 16 Mbytes memory without any address limitations. The ARRAY macros are defined as follows:

```
#define CVAR(object, addr) (*((object volatile code *)(addr))) /* CODE  */
#define DVAR(object, addr) (*((object volatile code *)(addr))) /* IDATA */
#define HVAR(object, addr) (*((object volatile code *)(addr))) /* CODE  */
#define NVAR(object, addr) (*((object volatile code *)(addr))) /* EDATA */
#define XVAR(object, addr) (*((object volatile code *)(addr))) /* XDATA */
```

You may use these macros in your programs as shown in the example below:

```
#include <absacc.h>       /* include absolute memory access macros      */
   :
   :
int i;
long l;

l = NVAR (long, 0xFFE0);        /* read long value at EDATA 0FFE0H      */
NVAR (int, 0x1000) = i;         /* write the value of i to EDATA 1000H  */

#define IO  XVAR (char, 0x1000) /* IO port at XDATA 1000H               */
if (IO == 0) IO = 1;            /* test port and write 1 to IO address  */
```

Routines by Category

This sections gives a brief overview of the major categories of routines available in the C251 standard library. Refer to “Reference” on page 210 for a complete description of routine syntax and usage.

NOTE
Many of the routines in the C251 standard library are reentrant, intrinsic, or both. Some routines like sprintf and sscanf are available in both forms reentrant and non-reentrant. 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
fmemcpy	reentrant	Copies a specified number of data bytes from one **far** memory buffer to another.
fmemset	reentrant	Initializes a specified number of data bytes in **far** memory.
memccpy	reentrant	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.
xmemcpy	reentrant	Copies a specified number of data bytes from one **xdata** memory buffer to another.
xmemset	reentrant	Initializes a specified number of data bytes in **xdata** memory.

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.
toupper	reentrant	Tests a character and converts it to uppercase if it is lowercase.
_tolower	reentrant	Unconditionally converts a character to 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	reentrant	Converts a string to a float.
atoi	reentrant	Converts a string to an int.
atol	reentrant	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**. **abs**, **cabs**, and **labs** are prototyped in the **MATH.H**.

Math

Routine	Attributes	Description
acos	reentrant	Calculates the arc cosine of a specified number.
asin	reentrant	Calculates the arc sine of a specified number.
atan	reentrant	Calculates the arc tangent of a specified number.
atan2	reentrant	Calculates the arc tangent of a fraction.
ceil	reentrant	Finds the integer ceiling of a specified number.
chkdouble	intrinsic, reentrant	Checks the status of double precision numbers.
chkfloat	intrinsic, reentrant	Checks the status of float numbers.
cos	reentrant	Calculates the cosine of a specified number.
cosh	reentrant	Calculates the hyperbolic cosine of a specified number.
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.
exp	reentrant	Calculates the exponential function of a specified number.
fabs	reentrant	Finds the absolute value of a specified number.
floor	reentrant	Finds the largest integer less than or equal to a specified number.
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.
log	reentrant	Calculates the natural logarithm of a specified number.
log10	reentrant	Calculates the common logarithm of a specified number.
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.
modf	reentrant	Generates integer and fractional components of a specified number.
pow	reentrant	Calculates a value raised to a power.
rand	reentrant	Generates a pseudo random number.
sin	reentrant	Calculates the sine of a specified number.
sinh	reentrant	Calculates the hyperbolic sine of a specified number.
sqrt	reentrant	Calculates the square root of a specified number.
srand	reentrant	Initializes the pseudo random number generator.
tan	reentrant	Calculates the tangent of a specified number.
tanh	reentrant	Calculates the hyperbolic tangent of a specified number.

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**. The **rand** and **srand** functions are prototyped in the **STDLIB.H** include file.

The _**chkfloat**_, _**chkdouble**_, **_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.
fcalloc		Allocates storage for an array from the **far** memory pool.
ffree		Frees a memory block that was allocated using fcalloc, fmalloc, or frealloc.
finit_mempool		Initializes the memory location and size of the **far** memory pool.
fmalloc		Allocates a block from the **far** memory pool.
frealloc		Reallocates a block from the **far** 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.
xcalloc		Allocates storage for an array from the **xdata** memory pool.
xfree		Frees a memory block that was allocated using xcalloc, xmalloc, or xrealloc.
xinit_mempool		Initializes the memory location and size of the **xdata** memory pool.
xmalloc		Allocates a block from the **xdata** memory pool.
xrealloc		Reallocates a block from the **xdata** 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		Reads and echoes a character using the _getkey and putchar routines.
_getkey		Reads a character using the 251 serial interface.
gets		Reads and echoes a character string using the getchar routine.
printf	refer to page 271	Writes formatted data using the putchar routine.
putchar	reentrant	Writes a character using the 251 serial interface.
puts	reentrant	Writes a character string and newline (‘\n’) character using the putchar routine.
scanf	refer to page 282	Reads formatted data using the getchar routine.
sprintf	refer to page 290	Writes formatted data to a string.
sscanf	refer to page 294	Reads formatted data from a string.
ungetchar		Places a character back into the getchar input buffer.
vprintf	refer to page 325	Writes formatted data using the putchar function.
vsprintf	refer to page 327	Writes formatted data to a string.

The stream input and output routines allow you to read and write data to and from the 251 serial interface or a user-defined I/O interface. The default **_getkey** and **putchar** functions found in the C251 library read and write characters using the 251 serial interface. You can find the source for these functions in the **\C251\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 251 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 251. The include file **REG251SB.H** contains definitions for the required SFRs.

The following example code must be executed immediately after reset, before any stream functions are invoked.

```
.
.
.
#include <reg251s.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 auto reload timer */

TH1 = 0xFD;           /* Set auto reload 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. All functions are prototyped in the **STDIO.H** include file.

String Manipulation

Routine	Attributes	Description
strcat	reentrant	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	reentrant	Concatenates up to a specified number of characters from one string to another.
strncmp	reentrant	Compares two strings up to a specified number of characters.
strncpy	reentrant	Copies up to a specified number of characters from one string to another.
strpbrk	reentrant	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	reentrant	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	reentrant	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.

The string routines provide you with the following operations to perform on character strings:

- 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
assert		Evaluate test expression and print diagnostic message.
setjmp	reentrant	Saves the current stack condition and program address.
longjmp	reentrant	Restores the stack condition and program address.
nop	intrinsic, reentrant	Inserts an 251 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.

The **assert** diagnostic macro is defined in the **ASSERT.H** include file.

Include Files

The include files that are provided with the C251 standard library are found in the **\C251\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.

251 Special Function Register Include Files

The C251 library provides you with a include files that define manifest constants for the special function registers found on the 251 derivatives. These files are listed below:

REG251S.H	**REG251A1.H**
REG930.H	**REG251G1.H**

ABSACC.H

The **ABSACC.H** include file contains definitions for macros that allow you to directly access the different memory areas of the 251.

CARRAY	**DVAR**	**NBYTE**
CBYTE	**HARRAY**	**NVAR**
CVAR	**HBYTE**	**XARRAY**
DARRAY	**HVAR**	**XBYTE**
DBYTE	**NARRAY**	**XVAR**

ASSERT.H

The **ASSERT.H** include file contains definitions for assert facility which prints a diagnostic message on the standard output stream and halts the program. The only function available in **ASSERT.H** is:

assert

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	**isprint**	**toint**
isalpha	**ispunct**	**_tolower**
iscntrl	**isspace**	**tolower**
isdigit	**isupper**	**_toupper**
isgraph	**isxdigit**	**toupper**
islower	**toascii**	

FLOAT.H

The **FLOAT.H** include file contains manifest constants that define range limits and other parameters for floating-point types. The following manifest constants are defined in this include file.

DBL_DIG	**FLT_DIG**	**LDBL_DIG**
DBL_EPSILON	**FLT_EPSILON**	**LDBL_EPSILON**
DBL_MANT_DIG	**FLT_MANT_DIG**	**LDBL_MANT_DIG**
DBL_MAX	**FLT_MAX**	**LDBL_MAX**
DBL_MAX_10_EXP	**FLT_MAX_10_EXP**	**LDBL_MAX_10_EXP**
DBL_MAX_EXP	**FLT_MAX_EXP**	**LDBL_MAX_EXP**
DBL_MIN	**FLT_MIN**	**LDBL_MIN**
DBL_MIN_10_EXP	**FLT_MIN_10_EXP**	**LDBL_MIN_10_EXP**
DBL_MIN_EXP	**FLT_MIN_EXP**	**LDBL_MIN_EXP**
DBL_RADIX	**FLT_RADIX**	**LDBL_RADIX**
DBL_ROUNDS	**FLT_ROUNDS**	**LDBL_ROUNDS**

INTRINS.H

The **INTRINS.H** include file contains prototypes for routines that instruct the compiler to generate inline intrinsic code.

chkdouble	**_irol_**	**_nop_**
chkfloat	**_iror_**	**_testbit_**
crol	**_lrol_**	
cror	**_lror_**	

LIMITS.H

The **LIMITS.H** include file contains manifest constants that define the value limits for the different variable types. The following manifest constants are defined in this include file.

CHAR_BIT	**LONG_MAX**	**SHRT_MIN**
CHAR_MAX	**LONG_MIN**	**UCHAR_MAX**
CHAR_MIN	**SCHAR_MAX**	**UINT_MAX**
INT_MAX	**SCHAR_MIN**	**ULONG_MAX**
INT_MIN	**SHRT_MAX**	**USHRT_MAX**

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	**cosh**	**pow**
acos	**exp**	**sin**
asin	**fabs**	**sinh**
atan	**floor**	**sqrt**
atan2	**labs**	**tan**
cabs	**log**	**tanh**
ceil	**log10**	
cos	**modf**	

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 following types, manifest constants, and macros.

NULL	**ptrdiff_t**
offsetof()	**size_t**

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	**fmalloc**	**srand**
atoi	**frealloc**	**xcalloc**
atol	**free**	**xfree**
calloc	**init_mempool**	**xinit_mempool**
fcalloc	**malloc**	**xmalloc**
ffree	**rand**	**xrealloc**
finit_mempool	**realloc**	

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 C251 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

Summary:

#include <math.h>
double acos (
    double *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.

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

Summary:

#include <math.h>
double asin (
  double *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 -π/2 to π/2 range.

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>
#define assert(expr) \          /* assert test argument */
if (expr) { ; } \
else  {\
  printf("Assert failed: " #expr " (file %s line %d)\n",
              __FILE__, (int) __LINE__ );\
  while (1);\
}
```

Description: The macro **assert** takes as argument any scalar type. If the value of *expr* is 0 or false, **assert** will print a diagnostic message via the printf function and halt the program execution with an endless loop.

If the macro **NDEBUG** is defined before the header file **ASSERT.H** is read, the assert facility is disabled, be expanding assert to be an empty statement. In this case no diagnostic message is printed and the *expr* is not evaluated.

Return Value: None.

See Also: **printf**

Example:

```
#include <assert.h>

void tst_assert (int x)  {

  /* x should be between 1 and 10 */
  assert (x > 0 && x < 10);
}
```

atan

Summary:

#include <math.h>
double atan (
  double *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 -π/2 to +π/2 range.

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>
double atan2 (
  double *y*, /* denominator for arc tangent */
  double *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** is a number in the -π 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 (void)  {
  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

Summary:

#include <stdlib.h>
double 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 **atof** function requires *string* to have the following format:

⟦{+ | -}⟧ *digits* ⟦. *digits*⟧ ⟦{e | E} ⟦{+ | -}⟧ *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*⟧ ⟦{+ | -}⟧ *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);
}
```

7

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*⟧ ⟦{+ | -}⟧ *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(%d) = %d\n", x, y);
}
```

calloc, fcalloc, xcalloc

Summary:

```
#include <stdlib.h>
void *calloc (
    unsigned int num,       /* number of items */
    unsigned int len);      /* length of each item */

void far *fcalloc (
    unsigned int num,       /* number of items */
    unsigned int len);      /* length of each item */

void xdata *xcalloc (
    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*.

The **fcalloc** and **xcalloc** functions work identically to **calloc**. **fcalloc** uses a *void far ** for the allocation of up to 16 Mbyte memory (which is also the default for **calloc** in the SMALL, XSMALL and LARGE memory models). **xcalloc** uses a *void xdata ** for memory allocation in the **XDATA** memory space.

NOTE
Source code is provided for this routine in the ***\C251\LIB*** *directory. You may modify the source to customize this function for your hardware environment. Refer to "Chapter 5. Advanced Programming Techniques" on page 131 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 */

char pool[0x2000];       /* space for memory pool */

void tst_calloc (void)  {
  int *p;                 /* ptr to array of 100 ints */

  init_mempool (pool, sizeof (pool));

  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>
double ceil (
    double *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 floating point number 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 / _chkdouble_

Summary:

#include <intrins.h>
unsigned char _chkfloat_ (
    float *val*); /* number for error checking */

unsigned char _chkdouble_ (
    double *val*); /* number for error checking */

Description: The **_chkfloat_** and **_chkdouble_** function checks the status of a floating point number.

Return Value: The **_chkfloat_** and **_chkdouble_** 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

Summary:

#include <math.h>
double cos (
    double *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.

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>
double cosh (
      double 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 inline 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 inline 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

Summary: **#include <math.h>**
double exp (
    double *x*);   /* power to use for e^x function */

Description: The **exp** function calculates the exponential function for the floating point value *x*.

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>
double fabs (
    double *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>
double floor (
    double *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 floating point number 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, ffree, xfree

Summary:

```
#include <stdlib.h>
void free (
    void *p);                 /* block to free */

void ffree (
    void far *p);             /* block to free */

void xfree (
    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.

The **ffree** and **xfree** functions work identically to **free**. **ffree** uses a *void far ** for the allocation of up to 16MB memory (which is also the default for **free** in the SMALL, XSMALL and LARGE memory models). **xfree** uses a *void xdata ** for memory allocation in the **XDATA** memory space.

NOTE
Source code for this routine is located in the ***\C251\LIB*** *directory. You may modify the source to customize this function for your particular hardware environment. Refer to "Chapter 5. Advanced Programming Techniques" on page 131 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 251. 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 %u %x\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 131 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 %u %x\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 251. 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');
}
```

7

init_mempool, finit_mempool, xinit_mempool

Summary:

```
#include <stdlib.h>
void init_mempool (
      void *p,                  /* start of memory pool */
      unsigned int size);       /* length of memory pool */

void finit_mempool (
      void far *p,              /* start of memory pool */
      unsigned int size);       /* length of memory pool */

void xinit_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.

The **finit_mempool** and **xinit_mempool** functions work identically to **init_mempool**. **finit_mempool** uses a *void far ** for the allocation of up to 16MB memory (which is also the default for **init_mempool** in the SMALL, XSMALL and LARGE memory models). **xinit_mempool** uses a *void xdata ** for memory allocation in the **XDATA** memory space.

NOTES
*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.*

*The **fcalloc**, **ffree**, **fmalloc**, **frealloc** and **finit_mempool** functions are implemented for the memory management of a 16MB memory pool. These functions control an indepentant memory pool which is not the same as the memory pool controlled by **init_mempool**.*

*The **xcalloc**, **xfree**, **xmalloc**, **xrealloc** and **xinit_mempool** functions are implemented for the memory management of a 64KB memory pool in the xdata memory space. These functions control an indepentant memory pool which is not the same as the memory pool controlled by **init_mempool**.*

*Source code is provided for this routine in the **\C251\LIB** directory. You can modify the source to customize this function for your hardware environment. Refer to "Chapter 5. Advanced Programming Techniques" on page 131 for more information.*

Return Value: None

See Also: **calloc**, **free**, **malloc**, **realloc**

Example:

```
#include <stdlib.h>

char pool[0x2000];           /* space for memory pool */

void tst_init_mempool (void)  {
  void *p;
  int i;

  init_mempool (pool, sizeof (pool));

  p = malloc (100);
  for (i = 0; i < 100; i++) ((char *) p)[i] = i;
  free (p);
}
```

irol

Summary:

```
#include <intrins.h>
unsigned char _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 inline 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 char _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 inline 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'-'F', 'a'-'f', '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

Summary:

#include <math.h>
double log (
    double *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…

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

Summary:

#include <math.h>
double log10 (
    double *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.

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 char _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 inline 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 char _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 inline 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, fmalloc, xmalloc

Summary:

```
#include <stdlib.h>
void *malloc (
     unsigned int size);       /* block size to allocate */

void far *fmalloc (
     unsigned int size);       /* block size to allocate */

void xdata *xmalloc (
     unsigned int size);       /* block size to allocate */
```

Description: The **malloc** function allocates a memory block from the memory pool of *size* bytes in length.

The **fmalloc** and **xmalloc** functions work identically to **malloc**. **fmalloc** uses a *void far ** for the allocation of up to 16MB memory (which is also the default for **malloc** in the SMALL, XSMALL and LARGE memory models). **xmalloc** uses a *void xdata ** for memory allocation in the **XDATA** memory space.

NOTE
Source code is provided for this routine in the ***\C251\LIB*** *directory. You may modify the source to customize this function for your hardware environment. Refer to "Chapter 5. Advanced Programming Techniques" on page 131 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 */

char pool[0x2000];           /* space for memory pool */

void tst_malloc (void)  {

  init_mempool (pool, sizeof (pool));
  unsigned char *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, fmemcpy, xmemcpy

Summary:

```
#include <string.h>
void *memcpy (
      void *dest,        /* destination buffer */
      void *src,         /* source buffer */
      int len);          /* maximum bytes to copy */

void far *fmemcpy (
      void far *dest,    /* destination buffer */
      void far *src,     /* source buffer */
      int len);          /* maximum bytes to copy */

void xdata *xmemcpy (
      void xdata *dest,          /* destination buffer */
      void xdata *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.

The **fmemcpy** and **xmemcpy** functions work identical to memset. **fmemcpy** uses a *void far ** to access the complete 16MB memory; this is also the default for **memcpy** in the SMALL, XSMALL and LARGE memory model. **xmemcpy** uses a *void xdata ** to access the **XDATA** memory space.

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, fmemset, xmemset

Summary:

#include <string.h>
void *memset (
  **void ****buf*, /* buffer to initialize */
  char *c*, /* byte value to set */
  int *len*); /* buffer length */

void far *memset (
  **void far ****buf*, /* buffer to initialize */
  char *c*, /* byte value to set */
  int *len*); /* buffer length */

void xdata *xmemset (
  **void xdata ****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*.

The **fmemset** and **xmemset** functions work identical to memset. **fmemset** uses a *void far ** to access the complete 16MB memory; this is also the default for **memset** in the SMALL, XSMALL and LARGE memory model. **xmemset** uses a *void xdata ** to access the **XDATA** memory space.

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>
double modf (
  double *val*, /* value to calculate modulo for */
  double **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 251 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 inline 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;

}
```

pow

Summary:

```
#include <math.h>
double pow (
      double x,          /* value to use for base */
      double y);         /* value to use for exponent */
```

Description: The pow function calculated 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

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 where a single character specifies that the argument is interpreted as a character, a string, a number, or a 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, double**	Floating-point number using the format [-]*dddd.dddd*
e	**float, double**	Floating-point number using the format [-]*d.dddd*e[-]*ddd*
E	**float, double**	Floating-point number using the format [-]*d.dddd*E[-]*ddd*
g	**float, double**	Floating-point number using either e or f format, whichever is more compact for the specified value and precision
G	**float, double**	Identical to the g format except that (where applicable) E precedes the exponent instead of e
c	**char**	Single character
s	**pointer**	String with a terminating null character
p	**pointer**	Pointer as linear address value. The pointer must be a default pointer of the memory model in use.

The *flags* field is where a single character is 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.
*****	Supply width or precision from an integer argument.

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 *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.

The *width*, or *precision* or both may be indicated by an asterisk (*). In this case, an **int** (or if prefixed with ***b*** a **char**) argument supplies the field width or precision.

Type	Meaning of Precision Field
d, u, o, x, X	The *precision* field is where you specify the minimum number of digits 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 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 **b** or **B** prefix must be used in combination with an explict case operation. In contrast to C51 or C251 Version 1, all char arguments are casted be default to int variables to maintain full compatiblity to the ANSI standard. However, if you want to optimize the parameter space, you can use the the **b** or **B** prefix together with cast operations as shown below.

```
unsigned char uc;
char c;

/* char is passed as int; total space: 4 bytes      */
printf ("%x %i", uc, c);

/* explicit casts optimize required parameter space */
printf ("%bx %bi", (unsigned char) uc, (char) c);
```

NOTES
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 251. Custom functions may use other I/O devices.*

The ***printf*** *function exists in two versions: non-reentrant and reentrant. If you specify* ***#pragma functions (reentrant)*** *before you include the* ***stdio.h*** *header file you select the reentrant form. The default or #pragma functions (static) prior to the stdio.h header file selects the non-reentrant version of* ***printf****.*

The total number of bytes that may be passed to the non-reentrant ***printf*** *function is limited due to the memory restrictions imposed by the 251. A maximum of 15 bytes may be passed in tiny or small memory model. A maximum of 40 bytes may be passed in all other memory models. No limitation exists when the reentrant* ***printf*** *function is used.*

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 %d  int %d  long %ld\n",a,b,c);
  printf ("Uchar %u Uint %u Ulong %lu\n",x,y,z);
  printf ("xchar %x xint %x xlong %lx\n",x,y,z);
  printf ("String %s is at address %p\n",buf,p);
  printf ("%f != %g\n", f, g);
}
```

putchar

Summary:

#include <stdio.h>
char putchar (
    char *c*);   /* character to output */

Description: The **putchar** function transmits the character *c* using the 251 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 131 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 251. 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, frealloc, xrealloc

Summary:

```
#include <stdlib.h>
void *realloc (
     void *p,                   /* previously allocated block */
     unsigned int size);        /* new size for block */

void far *frealloc (
     void far *p,               /* previously allocated block */
     unsigned int size);        /* new size for block */

void xdata *xrealloc (
     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.

The **frealloc** and **xrealloc** functions work identically to **realloc**. **frealloc** uses a *void far* * for the allocation of up to 16MB memory (which this is also the default for **realloc** in the SMALL, XSMALL and LARGE memory models). **xrealloc** uses a *void xdata* * for memory allocation in the **XDATA** memory space.

NOTE
Source code is provided for this routine in the ***\C251\LIB*** *directory. You can modify the source to customize this function for your hardware environment. Refer to "Chapter 5. Advanced Programming Techniques" on page 131 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 */

char pool[0x2000];           /* space for memory pool */

void tst_realloc (void)  {
  void *p;
  void *new_p;

  init_mempool (pool, sizeof (pool));

  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.

- 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
o	**unsigned int ***	Unsigned octal number
x	**unsigned int ***	Unsigned hex number
e	**float *, double ***	Floating-point number
f	**float *, double ***	Floating-point number
g	**float *, double ***	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 251. Custom functions may use other I/O devices.*

The ***scanf*** *function exists in two versions: non-reentrant and reentrant. If you specify* ***#pragma functions (reentrant)*** *before you include the* ***stdio.h*** *header file you select the reentrant form. The default or #pragma functions (static) prior to the stdio.h header file selects the non-reentrant version of* ***scanf****.*

The total number of bytes that may be passed to the non-reentrant ***scanf*** *function is limited due to the memory restrictions imposed by the 251. A maximum of 15 bytes may be passed in tiny or small memory model. A maximum of 40 bytes may be passed in all other memory models. No limitation exists when the reentrant* ***scanf*** *function is used.*

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

Summary:

#include <math.h>
double sin (
    double *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.

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>**
double sinh (
  double *val*); /* value to calc hyperbolic sine for */

Description: The **sinh** function calculates the hyperbolic sine of the floating point value *x*.

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

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” on page 271 for a description of the format string and additional arguments.

NOTES
*The **sprintf** function exists in two versions: non-reentrant and reentrant. If you specify **#pragma functions (reentrant)** before you include the **stdio.h** header file you select the reentrant form. The default or #pragma functions (static) prior to the stdio.h header file selects the non-reentrant version of **sprintf**.*

*The total number of bytes that may be passed to the non-reentrant **sprintf** function is limited due to the memory restrictions imposed by the 251. A maximum of 15 bytes may be passed in tiny or small memory model. A maximum of 40 bytes may be passed in all other memory models. No limitation exists when the reentrant **sprintf** function is used.*

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

Summary:

#include <math.h>
double sqrt (
    double *x*);   /* value to calculate square root of */

Description: The **sqrt** function calculates the square root of *x*.

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

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 282 for a complete description of the formation string and additional arguments.

NOTES

*The **sscanf** function exists in two versions: non-reentrant and reentrant. If you specify **#pragma functions (reentrant)** before you include the **stdio.h** header file you select the reentrant form. The default or #pragma functions (static) prior to the stdio.h header file selects the non-reentrant version of **sscanf**.*

*The total number of bytes that may be passed to the non-reentrant **sscanf** function is limited due to the memory restrictions imposed by the 251. A maximum of 15 bytes may be passed in tiny or small memory model. A maximum of 40 bytes may be passed in all other memory models. No limitation exists when the reentrant **sscanf** function is used.*

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_.

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 Stds. Inst.";

  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

Summary:

#include <math.h>
double tan (
    double *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.

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>
double tanh (
    double *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 inline 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** 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);
}
```

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));
  }
}
```

_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);
}
```

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));
  }
}
```

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” on page 271 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.

NOTES
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 251. Custom functions may use other I/O devices.*

The ***vprintf*** *function exists in two versions: non-reentrant and reentrant. If you specify* ***#pragma functions (reentrant)*** *before you include the* ***stdio.h*** *header file you select the reentrant form. The default or #pragma functions (static) prior to the stdio.h header file selects the non-reentrant version of* ***vprintf****.*

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” on page 271 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.

NOTE
*The **vsprintf** function exists in two versions: non-reentrant and reentrant. If you specify **#pragma functions (reentrant)** before you include the **stdio.h** header file you select the reentrant form. The default or #pragma functions (static) prior to the stdio.h header file selects the non-reentrant version of **vsprintf**.*

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 text[30];                   /* text buffer */


void error (char *fmt, ...)  {
  va_list arg_ptr;

  va_start (arg_ptr, fmt);           /* format string */
  vsprintf (text, 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 C251 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 the C251 compiler. 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 on the hardware 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 C251. Many, however, are not applicable to an embedded application and are excluded from the C251 library.

The following ANSI Standard library routines are included in the C251 library:

abs
acos
asin
assert
atan
atan2
atof
atoi
atol
calloc
ceil
cos
cosh
exp
fabs
floor
free
getchar
gets
isalnum
isalpha
iscntrl
isdigit
isgraph
islower
isprint
ispunct
isspace
isupper
isxdigit
labs
log
log10
longjmp
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 C251 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
offsetof
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 C251 library.

cabs	**init_mempool**	**toint**
chkdouble	**_irol_**	**_tolower**
chkfloat	**_iror_**	**_toupper**
crol	**_lrol_**	**ungetchar**
cror	**_lror_**	**xcalloc**
fcalloc	**memccpy**	**xfree**
ffree	**_nop_**	**xinit_mempool**
finit_mempool	**strpos**	**xmalloc**
fmalloc	**strrpbrk**	**xmemcpy**
fmemcpy	**strrpos**	**xmemset**
fmemset	**_testbit_**	
_getkey	**toascii**	

Appendix B. C251 Differences

This following appendix lists an overview of major product enhancements and differences between the actual C251 version and previous versions. Also included are differences between C251 Compiler and C51 Version 5.

C251 Version 1 Differences

- **C251 optimizer improved**
  C251 now supports optimizer level 7 and the optimizer has been improved. The new optimizations are:
  - Global Constant Propagation.
  - Enhanced Peephole Optimization.
  - Loop rotation. The resulting code is more efficient and executes faster.
  - Common tail merging and instruction simplification
- **Global common subexpressions**
  The global common subexpression optimization has been improved.
- **Enhanced code generation for 251 Source Mode**
  The code generated for the 251 source mode has been improved. C251 now uses all registers of the 251 architecture. Also the default is now MODSRC which generates source mode code. C251 version 1 has generated binary mode code by default.
- **TINY, XTINY and XSMALL memory models**
  These new memory models support efficient code generation for the 251 CPU. The XTINY and XSMALL memory model can be used instead of the LARGE memory model to generate the most efficient code for the 251 architecture. The inefficient COMPACT memory model is no longer supported.
- **Reentrant run-time library functions**
  Most of the C251 run-time library functions are now reentrant. This includes also all complex float and double functions and the sprintf and sscanf functions.

- **New memory type huge for unlimited object size**
  The memory type huge uses 32-bit address arithmetic and allows unlimited object sizes.
- **Double precision floating point arithmetic**
  C251 supports now the **FLOAT64** directive which implements double precision floating point arithmetic.
- **FUNCTIONS directive**
  The new C251 FUNCTIONS directive allows to specify **REENTRANT** and **STATIC** code and supports PLM51 and the memory model override. With this new control it is no longer required to change the function prototypes.
- **const far, const near and const xdata support**
  C251 now allows to locate constants with all memory types.
- **UNSIGNED_CHAR directive**
  The **UNSIGNED_CHAR** directive specifies ***unsigned char*** as default type for plain char variables.
- **INCDIR directive**
  The **INCDIR** directive allows the specification of optional pathnames for include files on the C166 invocation line.
- **NOALIAS directive**
  The **NOALIAS** relaxes alias checking for pointer access optimization.
- **USERCLASS and RENAMECODE directive**
  allows to assign user defined memory class names to variable declarations and to rename the segment name for user code.
- **ASMEXPAND and PREPRINTONLY preprocessor directives**
  allow to run only the preprocessor and control preprocessor macros in inline assembler code.
- **WARNING directive**
  This new C251 directive allows you to change the level of compiler warnings.
- **BROWSER directive**
  Generates a source browser database for µVision Version 2.

- **_at_ function attribute improved**
  The **_at_** attribute may be used to locate initialized objects and functions at specific addresses.
- **STARTUP code enhanced**
  The **START251.A51** files which contain the startup code have been rewritten for initialization of 251 **CONFIG** registers. The initialization code has been improved. It is required to use the new startup code of version 2.1 with your application.
- **PACK directive controls alignment in structure definitions**
  With PACK(2) you specify WORD alignment of structure members.
- **FIXDRK directive for bypassing C-step chip bug**
  specifing FIXDRK disables the generation of INC DRk instructions. Also the complete C251 run-time library has been generated with INC DRk.

C51 Version 5 Differences

- **Register Bank Dependent Code is No Longer Necessary**
  The 251 instruction set promotes efficient code which is independent of the selected registerbank. The directives **AREGS**, **NOAREGS**, and **REGISTERBANK** are not required to support the 251 architecture.

- **Support for Source Mode and Binary Mode**
  The C251 compiler supports the **MODSRC** and **MODBIN** directives for selecting the source and binary mode of the MCS® 251 architecture. Both modes generate different code which is optimized for the operating mode of the 251 CPU. The binary mode allows you to interface with existing 8051 code written with the C51 compiler, the A51 assembler, the Intel PL/M-51 compiler, or the Intel ASM51 assembler. Refer to “MODBIN” on page 51 for more information.

- **Support for Near and Far Memory Types**
  The memory types **near** and **far** let you access the new memory areas of the MCS® 251 architecture. For absolute access to **near** memory the macros **NBYTE** and **NWORD** are available.

- **HOLD Directive**
  The C251 compiler lets you override the memory model defaults with the **HOLD** directive. This lets you use the **SMALL** model for your application even when the memory requirements exceed the limits of this memory model.

- **PARM51 and PARM251 Directives**
  The C251 compiler optimizes parameter passing for the MCS® 251 architecture. This differs from the parameter passing performed by the C51 compiler. The **PARM51** directive instructs the C251 compiler to pass parameters like the C51 compiler. The **PARM251** directive instructs the C251 compiler to optimize parameter passing for the MCS® 251. Refer to “PARM51 / PARM251” on page 66 for more information.

- **INTR2 Directive**
  By default, the C251 compiler assumes that the 251 pushes a 3-byte stack pointer and **PSW1** onto the stack when entering an interrupt. Use the **INTR2** directive to instruct the C251 compiler to assume that the 251 pushes only the 2-byte stack pointer (like the 8051). Refer to “INTR2” on page 46 for more information.

- **OBJECTEXTEND No Longer Required**
  The C251 compiler always stores type information in the object file. This allows the linker to perform complete type checking of all declarations in an application. The **OBJECTEXTEND** directive is no longer required.

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 251 code generated by the C251 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 **TINY** directive instructs the C251 compiler to use the **TINY** memory model. In **TINY** model, variables, unless declared otherwise, reside in the data memory of the 251. Pointers are **near *** by default. If your application has huge data requirements the XTINY memory model should be preferred over the XSMALL memory model. The following example shows the influence of the memory models:

```
char *p;

void test (unsigned int i)  {
  p[i] = 0;
}
```

is compiled TINY, XSMALL and LARGE model to demonstrate the difference in generated code.

The following is the TINY model translation:

```
stmt  level    source

   1           #pragma tiny
   2
   3           char *p;
   4
   5           void test (unsigned int i)  {
   6   1         p[i] = 0;
   7   1       }

;       FUNCTION test (BEGIN)
                                             ; SOURCE LINE # 5
;---- Variable 'i' assigned to Register 'WR6' ----
                                             ; SOURCE LINE # 6
000000 E4             CLR      A             ; A=R11
000001 2E3500      R  ADD      WR6,p
000004 7A39B0         MOV      @WR6,R11      ; A=R11
                                             ; SOURCE LINE # 7
000007 22             RET
;       FUNCTION test (END)

Total Code Size = 8 Bytes
```

In TINY model, the variable **p** is located in the data memory. Also the default pointer size is 2 bytes. In addition, each of these instructions executes very fast.

The following is the same code compiled using the **XSMALL** model:

```
stmt  level    source

   1           #pragma xsmall
   2
   3           char *p;
   4
   5           void test (unsigned int i)  {
   6   1         p[i] = 0;
   7   1       }

;       FUNCTION test (BEGIN)
                                             ; SOURCE LINE # 5
000000 7D13           MOV      WR2,WR6
;---- Variable 'i' assigned to Register 'WR2' ----
                                             ; SOURCE LINE # 6
000002 E4             CLR      A             ; A=R11
000003 7E1F0000    R  MOV      DR4,p
000007 2D31           ADD      WR6,WR2
000009 7A1BB0         MOV      @DR4,R11      ; A=R11
                                             ; SOURCE LINE # 7
00000C 22             RET
;       FUNCTION test (END)

Total Code Size = 13 Bytes
```

In XSMALL model, the variable **p** is located in the near memory. Also the default pointer size is 4 bytes. The generated code requires more CPU registers and is bigger compared to the TINY model.

The following is the same code compiled using the **XSMALL** model:

```
stmt  level    source

   1           #pragma large
   2
   3           char *p;
   4
   5           void test (unsigned int i)  {
   6   1         p[i] = 0;
   7   1       }

;       FUNCTION test (BEGIN)
                                             ; SOURCE LINE # 5
000000 7D13           MOV      WR2,WR6
;---- Variable 'i' assigned to Register 'WR2' ----
                                             ; SOURCE LINE # 6
000002 E4             CLR      A             ; A=R11
000003 900000      R  MOV      DPTR,#WORD0 p
000006 693E0002       MOV      WR6,@DR56+2   ; WORD0(DR56)=DPTR
00000A 0BEA20         MOV      WR4,@DR56     ; WORD0(DR56)=DPTR
00000D 2D31           ADD      WR6,WR2
00000F 7A1BB0         MOV      @DR4,R11      ; A=R11
                                             ; SOURCE LINE # 7
000012 22             RET
;       FUNCTION test (END)

Total Code Size = 19 Bytes
```

In LARGE model, the variable **p** is located in the xdata memory which generates inefficient memory accesses. For big applications you should really use the XSMALL model instead of LARGE.

Variable Location

Frequently accessed data objects should be located in the internal data memory of the 251. 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, and other user defined variables with the memory type **data**.

Because of the limited amount of data memory (128 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 three 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 339 for more information.

Another way is to use the **HOLD** directive. **HOLD** allows you to specify object size limits for variables. Objects above a specific size, like arrays, longs or floats may be placed to **near**, **xdata** or **far** memory.

The third way to locate variables in other memory areas is to manually select the variables that can be moved into **near**, **far**, or **xdata** memory and declare them using the **near**, **far**, or **xdata** memory specifier. Usually, string buffers and other large arrays can be declared with the **near** memory type without a significant degradation in performance or increase in code size.

Variable Size

Members of the 251 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.

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.

Use 251 Source Mode

The source mode of the 251 CPU generates better code for complex applications. You should select the 251 binary mode only, when your code is primarily using the data type unsigned char and most memory accesses are to the data memory.

Other Sources of Information

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 C251 compiler embodies some known limitations. 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.

Limitations of the C251 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.
- Names can be up to 255 characters long. However, only the first 40 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 32 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 8180 characters. Even after a macro expansion, the result may not exceed 8180 characters.

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 251 is an 8-bit machine that is based on the 8051 which has no instructions for directly manipulating data objects that are larger than 8 bits. Multi-byte data objects are stored according to the following rules.

- The **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 251 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.

Problems with Program Execution

There are a couple of common user problems which arise due to the complexity of the 251 hardware. Check the following if your program hangs or behaves strangely.

- If your program runs on the dScope Simulator but not on the 251 hardware you may have problems with the initialization of the **CONFIG** registers in the START251 file. Check carefully that all parameters of the START251 file match your target hardware. In most cases, the hardware is not correctly configured. Run a simple program on your target hardware before you try a complex application. It is a good choice to start with is the HELLO program example provided in **\C251\EXAMPLES\HELLO**.
- Another common problem is incorrectly setting the memory classes in L251 for your RAM and EPROM. Check the areas listed under MEMORY MAP in the linker/locator MAP file (*.MAP) and verify that they match the memory allocation of your target hardware. Segments with the memory class name ***x*CONST** or ***x*CODE** must be located in a EPROM device and segments of the class ***x*DATA** must be located in a RAM area.

Problems with the Stack

Another reason for your program hanging-up could be that your application is running out of stack space. Since embedded applications usually use interrupt functions, it is not possible to calculate the stack requirements at compile-time. If you think that your application has problems with the available stack space you have the following options:

- Execute your program with a debugger or emulator. Stop after the memory initialization at the beginning of the **main** function. Fill the memory space of the system stack and the user stack with a constant value (like 0xA5) and execute all functions of the application. Use the debugger to determine if there is enough stack space to execute your most complex interrupt procedure. The location of the stack area is listed in the MEMORY MAP of the linker/locator listing file (*.MAP) under the segment ?STACK.
- If you cannot work with a debugger in your target system, increase the stack space and check to see if the problems go away.

Problems with Interrupts

To enable an interrupt source you need to set global interrupt enable all (EA) bit in the SFR register IE0. In addition, each individual interrupt source has it is own interrupt enable bit which must be set to 1.

Several interrupt routines with the same interrupt priority can share a common registerbank. However you cannot state using 0, since this registerbank is typically used in the main function.

If you need to disable interrupts for a C statement or a C function you can use the following methods:

- To disable interrupts for a complete C function you can use the *#pragma disable* compiler directive. Refer to “DISABLE” on page 31 for more information.
- You can reset the EA bit in the IE0 SFR register.

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 on-chip RAM of the 251, 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 251 Linker/Locater User's Guide for more information.*

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 L251 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 (**.MAP**) 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.

Problems when Porting C51 Code

When porting existing C51 code there are a number of minor differences between C51 and C251 that you must consider. Enclosed is a list of typical problems you may encounter:

- The code generated by C251 is independent of the current registerbank selected and the C51 directives **NOAREGS** and **REGISTERBANK** are no longer necessary. If these directives are used, C251 generates a WARNING or ERROR message.
- The memory type specifier must be given after the variable type as shown in the following example:

```
int xdata value;    /* correct in both C51 and C251                */
xdata int value;    /* illegal: old C51 form not supported by C251 */
```

- The **pdata** memory type usually gives problems in the PAGE mode of the 251 CPU and should not be used for new applications.
- The **printf** library function and the variable argument list makes default promotion to int even for char and unsigned char variables. The following printf statement will generate incorrect results in C251.

```
unsigned char c1;

     /* works in C51, but gives wrong output in C251:        */
printf ("%bx %bi", c1, 1);

     /* C251 defaults to int and does not need the b prefix: */
printf ("%x %i", c1, 1);

     /* the following two forms work in both C51 and C251:   */
printf ("%bx %bi", (unsigned char) c1, (char) 1);  /* optimal code */
printf ("%x %i", (unsigned int) c1, (int) 1);      /* passes int   */
```

Problems with the Intel RISM Monitor

Some users have reported problems with the tool setup on target boards which use the Intel RISM-251 Monitor program. To help you quickly get started using the RISM monitor, we have provided a sample application written for the RISM monitor in the directory **\C251\EXAMPLES\RISM**. If you still have problems after trying this example, make sure to check the following:

- Since RISM uses memory locations in the on-chip RAM of the 251 chip, you need to change the START251.A51 code so that it does not initialize the internal memory of the chip. Specifically, the startup code may no longer clear the EDATA memory to 0 (since this causes RISM251 to stop working). Therefore, change the setting of EDATA_LEN to 0 in START251.A51 as shown below. START251.A51 must then be included in your application or project file.

```
EDATALEN   EQU   0H      ; the length of EDATA memory in bytes.
```

- You must instruct the L251 linker to avoid placing program data in the memory space used by RISM. You must specify the RESERVE directive as follows:

```
L251 ... RESERVE (29H – 3DH)
```

- RISM expects your target program to begin at address 0x004000. You must instruct the linker to locate the program code at this address. Additionally, there are a few other data ranges that must be specified. For example,

```
L251 ... CLASSES (EDATA  (0-3FFFH),
                  CODE   ($4000H, 4000H – 0FFFFH)
                  HCONST (4000H – 0FFFFH))
```

NOTE
If your code still does not run, you may have located a segment with the ***xCONST*** *or* ***ECODE*** *memory classes to a non-existing memory area. Check the memory map in the linker MAP file to make sure your segments are located in the proper physical address ranges.*

Glossary

A251
The command used to assemble programs using the 251 Macro Assembler.

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:\C251\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.

block
A sequence of C statements, including definitions and declarations, enclosed within braces ({ }).

C251
The command used to compile programs using the 251 Optimizing C Cross Compiler.

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 C251 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.

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 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.

function
A combination of declarations and statements that can be called by name that perform an operation and/or return a value.

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.

L251
The command used to link object files and libraries using the 251 linker/locator.

LIB251
The command used to manipulate library files using the 251 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® 251
The general name applied to the Intel family of 251 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.

Monitor-251
An 251 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 file
A file, created by the compiler, that contains the program segment information and relocatable machine code.

object
An area of memory that can be examined. Usually used when referring to the memory area associated with a variable or function.

OH251
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.

RTX251 Full
An 251 Real-Time Executive that provides a multitasking operating system kernel and library of routines for its use.

RTX251 Tiny
A limited version of RTX251.

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 literal
A string of characters enclosed within double quotes (**" "**).

string
An array of characters that is terminated with a null character (**'\0'**).

structure member
One element of a structure.

structure
A set of elements of possibly different types grouped together under one name.

token
A fundamental symbol that represents a name or entity in a programming language.

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 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.

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).

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