



Cx51 Compiler

**Optimizing C Compiler and Library Reference
for Classic and Extended 8051 Microcontrollers**

User's Guide 09.2001

Information in this document is subject to change without notice and does not represent a commitment on the part of the Keil Software, Inc. The software described in this document is furnished under license agreement or nondisclosure agreement and may be used or copied only in accordance with the terms of the agreement. It is against the law to copy the software on any medium except as specifically allowed in the license or nondisclosure agreement. The purchaser may make one copy of the software for backup purposes. No part of this manual may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or through information storage and retrieval systems, for any purpose other than for the purchaser's personal use, without the express written permission of Keil Software, Inc.

© Copyright 1988-2001 Keil Elektronik GmbH. and Keil Software, Inc.
All rights reserved.

Keil C51™, Keil CX51™, and μ Vision2 are a trademarks of Keil Elektronik GmbH.

Microsoft® and Windows™ are trademarks or registered trademarks of Microsoft Corporation.

IBM®, PC®, and PS/2® are registered trademarks of International Business Machines Corporation.

Intel®, MCS® 51, MCS® 251, ASM-51®, and PL/M-51® are registered trademarks of Intel Corporation.

Every effort is made to ensure accuracy in this manual and to give appropriate credit to persons, companies, and trademarks referred to herein.

Preface

This manual describes how to use the **Cx51** Optimizing C Compilers to compile C programs for your target 8051 environment. The **Cx51** Compiler package may be used on all 8051 family processors and is executable under the Windows 32-Bit command line prompt. This manual assumes that you are familiar with the Windows operating system, know how to program 8051 processors, and have a working knowledge of the C programming language.

NOTE

This manual uses the term Windows to refer to the 32-bit Windows Versions Windows 95, Windows 98, Windows ME, Windows NT, Windows 2000 and Windows XP.

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 16.

Many of the examples and descriptions in this manual discuss invoking the compiler from the Windows command prompt. While this may not be applicable to you if you are running **Cx51** within an integrated development environment like **µVision2**, examples in this manual are universal and apply to all programming environments.

Manual Organization

This user's guide is divided into the following chapters and appendices:

“Chapter 1. Introduction,” describes the **Cx51** compiler.

“Chapter 2. Compiling with the Cx51,” explains how to compile a source file using the **Cx51** cross compiler. This chapter describes the command-line directives that control file processing, compiling, and output.

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

“Chapter 4. Preprocessor,” describes the components of the **Cx51** compiler preprocessor and includes examples.

“Chapter 5. 8051 Derivatives,” describes the 8051 family derivatives supported by the **Cx51** compiler. This chapter also includes tips that may help you improve your target program's performance.

“Chapter 6. Advanced Programming Techniques,” lists important information for the experienced developer. This chapter includes customization file descriptions, optimizer details, and segment naming conventions. This chapter also discusses how to interface programs created with the **Cx51** compiler with other 8051 programming languages.

“Chapter 7. Error Messages,” lists fatal errors, syntax errors, and warnings you may encounter while using the **Cx51** compiler.

“Chapter 8. Library Reference,” provides you with an extensive **Cx51** library reference. The library routines are listed by category and by include file. An alphabetical reference section, which includes example code for each of the library routines, concludes this 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 capitalized text is used for the names of executable programs, data files, source files, environment variables, and commands you enter at the Windows command prompt. This text usually represents commands that you must type in manually (capital letters are not required). Example: CLS DIR BL51.EXE
Language Elements	Elements of the C language are presented in bold type including keywords, operators and library functions. Example: if != long isdigit main >>
Courier	Text in this typeface represents information that displays on screen or prints out. This font is also used within the text when discussing or describing command line items.
<i>Variables</i>	Text in italics represents information that you must provide. For example, <i>projectfile</i> in a syntax string means that you are required to supply the actual project filename. 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 a fragment of the program is omitted. Example: <pre>void main (void) { . . . while (1);</pre>
[<i>Optional Items</i>]	Optional arguments in command-line and option fields are indicated by double brackets. Example: C51 TEST.C PRINT [<i>(filename)</i>]
{ <i>opt1</i> <i>opt2</i> }	Text contained within braces, separated by a vertical bar, represents a group of items from which one item in the list must be selected. <ul style="list-style-type: none"> ▪ Braces enclose all of the choices. ▪ Vertical bars separate the choices.
Keys	Text in the sans serif typeface represents keys on the keyboard. For example, "Press Enter to continue."

Contents

Chapter 1. Introduction.....	15
Support for all 8051 Variants.....	15
Books About the C Language.....	16
Chapter 2. Compiling with the Cx51 Compiler	17
Environment Variables.....	17
Running Cx51 from the Command Prompt.....	18
ERRORLEVEL.....	19
Cx51 Output Files.....	19
Control Directives.....	20
Directive Categories.....	20
Reference.....	23
AREGS / NOAREGS.....	24
ASM / ENDASM.....	26
BROWSE.....	28
CODE.....	29
COMPACT.....	30
COND / NOCOND.....	31
DEBUG.....	33
DEFINE.....	34
DISABLE.....	35
EJECT.....	37
FLOATFUZZY.....	38
INCDIR.....	39
INTERVAL.....	40
INTPROMOTE / NOINTPROMOTE.....	41
INTVECTOR / NOINTVECTOR.....	44
LARGE.....	46
LISTINCLUDE.....	47
MAXARGS.....	48
MOD517 / NOMOD517.....	49
MODA2 / NOMODA2.....	51
MODAB2 / NOMODAB2.....	52
MODDA2 / NOMODDA2.....	53
MODDP2 / NOMODDP2.....	54
MODP2 / NOMODP2.....	55
NOAMAKE.....	56
NOEXTEND.....	57
OBJECT / NOOBJECT.....	58
OBJECTADVANCE.....	59
OBJECTEXTEND.....	60
ONEREBBANK.....	61
OMF2.....	62
OPTIMIZE.....	63

ORDER	65
PAGELength	66
PAGEWIDTH	67
PREPRINT	68
PRINT / NOPRINT	69
REGFILE	70
REGISTERBANK	71
REGPARMS / NOREGPARMS	72
RET_PSTK, RET_XSTK	74
ROM	76
SAVE / RESTORE	77
SMALL	78
SRC	79
STRING	80
SYMBOLS	81
USERCLASS	82
VARBANKING	84
WARNINGLEVEL	85
XCROM	86
Chapter 3. Language Extensions	89
Keywords	89
Memory Areas	90
Program Memory	90
Internal Data Memory	91
External Data Memory	92
Far Memory	93
Special Function Register Memory	93
Memory Models	94
Small Model	94
Compact Model	95
Large Model	95
Memory Types	95
Explicitly Declared Memory Types	96
Implicit Memory Types	97
Data Types	97
Bit Types	98
Bit-addressable Objects	99
Special Function Registers	101
sfr	101
sfr16	102
sbit	102
Absolute Variable Location	104
Pointers	106
Generic Pointers	106
Memory-specific Pointers	109
Pointer Conversions	111
Abstract Pointers	114

Function Declarations	118
Function Parameters and the Stack	119
Passing Parameters in Registers	120
Function Return Values.....	120
Specifying the Memory Model for a Function	121
Specifying the Register Bank for a Function.....	122
Register Bank Access.....	124
Interrupt Functions.....	125
Reentrant Functions	129
Alien Function (PL/M-51 Interface)	132
Real-time Function Tasks.....	133
Chapter 4. Preprocessor	135
Directives.....	135
Stringize Operator.....	136
Token-pasting operator	137
Predefined Macro Constants.....	138
Chapter 5. 8051 Derivatives	139
Analog Devices MicroConverter B2 Series.....	140
Atmel 89x8252 and Variants	141
Dallas 80C320, 420, 520, and 530.....	142
Dallas 80C390, 80C400, 5240, and Variants.....	143
Arithmetic Accelerator.....	144
Infineon C517, C509, 80C537, and Variants.....	145
Data Pointers.....	145
High-speed Arithmetic	146
Library Routines.....	146
Philips 8xC750, 8xC751, and 8xC752.....	147
Philips 80C51MX Architecture	148
Philips and Atmel WM Dual DPTR	148
Chapter 6. Advanced Programming Techniques.....	149
Customization Files	150
STARTUP.A51	151
INIT.A51	153
XBANKING.A51	154
Basic I/O Functions.....	156
Memory Allocation Functions.....	156
Optimizer	157
General Optimizations	157
8051-Specific Optimizations.....	158
Options for Code Generation	158
Segment Naming Conventions.....	159
Data Objects.....	160
Program Objects.....	161
Interfacing C Programs to Assembler	163
Function Parameters.....	163

Parameter Passing in Registers	164
Parameter Passing in Fixed Memory Locations	165
Function Return Values	165
Using the SRC Directive	166
Register Usage.....	168
Overlaying Segments.....	168
Example Routines.....	168
Small Model Example	169
Compact Model Example	171
Large Model Example	173
Interfacing C Programs to PL/M-51	175
Data Storage Formats	176
Bit Variables.....	176
Signed and Unsigned Characters, Pointers to data, idata, and pdata	177
Signed and Unsigned Integers, Enumerations, Pointers to xdata and code	177
Signed and Unsigned Long Integers	177
Generic and Far Pointers	178
Floating-point Numbers.....	179
Floating-point Errors	182
Accessing Absolute Memory Locations	184
Absolute Memory Access Macros	184
Linker Location Controls	185
The <code>_at_</code> Keyword	186
Debugging.....	187
Chapter 7. Error Messages	189
Fatal Errors	189
Actions	190
Errors.....	191
Syntax and Semantic Errors	193
Warnings.....	205
Chapter 8. Library Reference.....	209
Intrinsic Routines	209
Library Files.....	210
Standard Types.....	211
<code>jmp_buf</code>	211
<code>va_list</code>	211
Absolute Memory Access Macros.....	212
<code>CBYTE</code>	212
<code>CWORD</code>	212
<code>DBYTE</code>	213
<code>DWORD</code>	213
<code>FARRAY</code> , <code>FCARRAY</code>	214
<code>FVAR</code> , <code>FCVAR</code> ,.....	215
<code>PBYTE</code>	216
<code>PWORD</code>	216

XBYTE	217
XWORD	217
Routines by Category	218
Buffer Manipulation	218
Character Conversion and Classification	219
Data Conversion	220
Math Routines	221
Memory Allocation Routines	223
Stream Input and Output Routines	224
String Manipulation Routines	226
Variable-length Argument List Routines	227
Miscellaneous Routines	227
Include Files	228
8051 Special Function Register Include Files	228
80C517.H	228
ABSACC.H	229
ASSERT.H	229
CTYPE.H	229
INTRINS.H	229
MATH.H	230
SETJMP.H	230
STDARG.H	230
STDDEF.H	230
STDIO.H	231
STDLIB.H	231
STRING.H	231
Reference	232
abs	233
acos / acos517	234
asin / asin517	235
assert	236
atan / atan517	237
atan2	238
atof / atof517	239
atoi	240
atol	241
cabs	242
calloc	243
ceil	244
_chkfloat	245
cos / cos517	246
cosh	247
_crol	248
_cror	249
exp / exp517	250
fabs	251
floor	252

fmod	253
free	254
getchar	255
_getkey	256
gets	257
init_mempool.....	258
lrol	259
lror	260
isalnum	261
isalpha	262
iscentrl.....	263
isdigit.....	264
isgraph.....	265
islower.....	266
isprint.....	267
ispunct.....	268
isspace.....	269
isupper.....	270
isxdigit.....	271
labs	272
log / log517	273
log10 / log10517	274
longjmp	275
lrol	277
lror	278
malloc.....	279
memccpy	280
memchr.....	281
memcmp	282
memcpy	283
memmove.....	284
memset	285
modf	286
nop	287
offsetof	288
pow	289
printf / printf517	290
putchar.....	296
puts	297
rand.....	298
realloc.....	299
scanf	300
setjmp	304
sin / sin517	305
sinh	306
sprintf / sprintf517	307
sqrt / sqrt517.....	309

srand.....	310
sscanf / sscanf517.....	311
strcat.....	313
strchr.....	314
strcmp.....	315
strcpy.....	316
strcspn.....	317
strlen.....	318
strncat.....	319
strncmp.....	320
strncpy.....	321
strpbrk.....	322
strpos.....	323
strrchr.....	324
strrbrk.....	325
strrpos.....	326
strspn.....	327
strstr.....	328
strtod / strtod517.....	329
strtol.....	331
strtoul.....	333
tan / tan517.....	335
tanh.....	336
_testbit.....	337
toascii.....	338
toint.....	339
tolower.....	340
_toupper.....	341
toupper.....	342
_toupper.....	343
ungetchar.....	344
va_arg.....	345
va_end.....	347
va_start.....	348
vprintf.....	349
vsprintf.....	351
Appendix A. Differences from ANSI C.....	353
Compiler-related Differences.....	353
Library-related Differences.....	353
Appendix B. Version Differences.....	357
Version 6.0 Differences.....	357
Version 5 Differences.....	358
Version 4 Differences.....	359
Version 3.4 Differences.....	361
Version 3.2 Differences.....	362
Version 3.0 Differences.....	363

Version 2 Differences	364
Appendix C. Writing Optimum Code	367
Memory Model	367
Variable Location.....	369
Variable Size.....	369
Unsigned Types.....	370
Local Variables	370
Other Sources.....	370
Appendix D. Compiler Limits.....	371
Appendix E. Byte Ordering.....	373
Appendix F. Hints, Tips, and Techniques.....	375
Recursive Code Reference Error.....	375
Problems Using the printf Routines	376
Uncalled Functions.....	377
Using Monitor-51.....	377
Trouble with the bdata Memory Type.....	378
Function Pointers	379
Glossary.....	383
Index.....	391

Chapter 1. Introduction

1

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 **Cx51** Optimizing C Compiler is a complete implementation of the American National Standards Institute (ANSI) standard for the C language. **Cx51** is not a universal C compiler adapted for the 8051 target. It is a ground-up implementation dedicated to generating extremely fast and compact code for the 8051 microprocessor. **Cx51** provides you with 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 **Cx51** 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 89 for more detailed information.

Support for all 8051 Variants

The 8051 Family is one of the fastest growing Microcontroller Architectures. More than 400 device variants from various silicon vendors are today available. New extended 8051 Devices, like the Philips 80C51MX architecture are dedicated for large application with several Mbytes code and data space.

For optimum support of these different 8051 variants, Keil provides the several development tools that are listed in the table below. A new output file format (OMF2) allows direct support of up to 16MB code and data space. The CX51 compiler is a variant of the C51 compiler that is designed for the new Philips 80C51MX architecture.

Development Tools	Support Microcontrollers, Description
C51 Compiler A51 Macro Assembler BL51 Linker/Locater	Development Tools for classic 8051 . Includes support for 32 x 64KB code banks.
C51 Compiler (with OMF2 Output) AX51 Macro Assembler LX51 Linker/Locater	Development Tools for classic 8051 and extended 8051 variants (like the Dallas 390). Includes support for code banking and up to 16MB code and xdata memory.
CX51 Compiler AX51 Macro Assembler LX51 Extended Linker/Locater	Development Tools for the Philips 80C51MX . Supports up to 16MB code and xdata memory.

The **Cx51** compiler is available in different packages. The table above refers to the entire line of the 8051 development tools.

NOTE

*The term **Cx51** is used to refer to both compiler variants: the **C51** compiler and the **CX51** compiler.*

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

C and the 8051: Programming and Multitasking

Schultz
P T R Prentice-Hall, Inc.
ISBN 0-13-753815-4

Chapter 2. Compiling with the Cx51 Compiler

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

- Direct the **Cx51** compiler to generate a listing file
- Control the amount of information included in the object file
- Specify optimization level and memory models

NOTE

*Typically you will use the **Cx51** compiler within the **µVision2 IDE**. For more information on using the **µVision2 IDE**, refer to the User's Guide "**Getting Started with µVision2 and C51**".*

Environment Variables

If you run the Cx51 compiler within the **µVision2 IDE**, you need no additional settings on your computer. If you want to run the **Cx51** compiler and utilities from the command prompt, you must manually create the following environment variables.

Variable	Path	Environment Variable specifies ...
PATH	\C51\BIN	path of the C51 and CX51 executable programs.
TMP		path to use for temporary files generated by the compiler. If the specified path does not exist, the compiler generates an error and aborts compilation.
C51INC	\C51\INC	path to the folder for Cx51 include files.
C51LIB	\C51\LIB	path to the folder for Cx51 library files.

For Windows NT, Windows 2000 and Windows XP these environment variables are entered under **Control Panel – System – Advanced – Environment Variables**.

For Windows 95, Windows 98 and Windows ME the settings are placed in **AUTOEXEC.BAT** using the following commands:

```
PATH=C:\KEIL\C51\BIN;%PATH%
SET TMP=D:\
```

```
SET C51INC=C:\KEIL\C51\INC
SET C51LIB=C:\KEIL\C51\LIB
```

Running Cx51 from the Command Prompt

To invoke the C51 or CX51 compiler, enter **C51** or **CX51** at the command 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 **Cx51** command line is:

```
C51 sourcefile [directives...]
CX51 sourcefile [directives...]
```

or:

```
C51 @commandfile
CX51 @commandfile
```

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 20 for a detailed list of the available directives.
- commandfile** is the name of a command input file that may contain *sourcefile* and *directives*. A *commandfile* is used, when the Cx51 invocation line gets complex and exceeds the limits of the Windows command prompt.

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

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

The **Cx51** compiler displays the following information upon successful compilation.

```
C51 COMPILER V6.10
C51 COMPILATION COMPLETE. 0 WARNING(S), 0 ERROR(S)
```

ERRORLEVEL

After compilation, the number of errors and warnings detected is output to the screen. The **Cx51** compiler then sets the **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 batch files. Refer to the Windows command index or to batch commands in the Windows on-line help for more information on **ERRORLEVEL** or batch files.

Cx51 Output Files

The **Cx51** compiler generates a number of output files during compilation. By default, each of these output files shares the same *filename* 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
<i>filename.LST</i>	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 symbols used and the assembly code generated. For more information, refer to the PRINT directive in the following sections.
<i>filename.OBJ</i>	Files with this extension are object modules that contain relocatable object code. Object modules may be linked to an absolute object module by the Lx51 Linker/Locator.
<i>filename.I</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. For more information, refer to the PREPRINT directive in the following sections.
<i>filename.SRC</i>	Files with this extension are assembly source files generated from your C source code. These files can be assembled with the A51 assembler. For more information, refer to the SRC directive in the following sections.

Control Directives

The Cx51 compiler offers a number of control directives that you may use to control compilation. 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. For example:

```
C51 testfile.c SYMBOLS CODE DEBUG
#pragma SYMBOLS CODE DEBUG
```

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

NOTE

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

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

Directive Categories

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

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

The following table is an alphabetical list of the control directives. The underlined characters denote the abbreviation of the directive.

Directive	Class	Description
<u>A</u> REGS, <u>N</u> OAREGS	Object	Enable or disable absolute register (ARn) addressing.
<u>A</u> SM, <u>E</u> NDASM	Source	Marks the beginning and the end of an inline assembly block.
<u>B</u> ROWSE †	Object	Enable generation of browser information.
<u>C</u> ODE †	Listing	Add an assembly listing to the listing file.
<u>C</u> OMPACT †	Object	Select COMPACT memory model.
<u>C</u> OND, <u>N</u> OCOND †	Listing	Include or exclude source lines skipped by the preprocessor.
<u>D</u> EBUG †	Object	Include debugging information in the object file.
<u>D</u> EFINE	Source	Define preprocessor names in the Cx51 invocation line.
<u>D</u> ISABLE	Object	Disables interrupts for the duration of a function.
<u>E</u> JECT	Listing	Inserts a form feed character into the listing file.
<u>F</u> LOAT <u>F</u> UZZY	Object	Specify number of bits rounded during floating compare.
<u>I</u> NC <u>D</u> IR †	Source	Specify additional path names for include files.
<u>I</u> NT <u>ER</u> VAL †	Object	Specify the interval for interrupt vectors for SIECO chips.
<u>I</u> NT <u>P</u> ROMOTE, <u>N</u> OINT <u>P</u> ROMOTE †	Object	Enable or disable ANSI integer promotion.
<u>I</u> NT <u>V</u> ECTOR, <u>N</u> OINT <u>V</u> ECTOR †	Object	Specify base address for interrupt vectors or disable vectors.
<u>L</u> ARGE †	Object	Select LARGE memory model.
<u>L</u> IST <u>I</u> NC <u>L</u> UDE	Listing	Display contents of include files in the listing file.
<u>M</u> AX <u>A</u> RGS †	Object	Specify size of variable argument lists.
<u>M</u> OD517, <u>N</u> OMOD517	Object	Enable or disable code to support the additional hardware features of the 80C517 and derivatives.
<u>M</u> OD <u>A</u> 2, <u>N</u> OMOD <u>A</u> 2	Object	Enable or disable dual DPTR register support for Atmel 82x8252 and variants.
<u>M</u> OD <u>A</u> B2, <u>N</u> OMOD <u>A</u> B2	Object	Enable or disable dual DPTR register support for Analog Device MicroConverter ADuC B2 series.
<u>M</u> OD <u>D</u> A, <u>N</u> OMOD <u>D</u> A	Object	Enable or disable code to support the arithmetic accelerator available in Dallas 80C390, 80C400, and 5240.
<u>M</u> OD <u>D</u> P2, <u>N</u> OMOD <u>D</u> P2	Object	Enable or disable dual DPTR register support for Dallas Semiconductor 320, 520, 530, 550 and variants.
<u>M</u> OD <u>P</u> 2, <u>N</u> OMOD <u>P</u> 2	Object	Enable or disable dual DPTR register support for Philips and AtmelWMM derivatives.
<u>N</u> O <u>A</u> MAKE †	Object	Disable information records for μVision2 Update function.
<u>N</u> O <u>E</u> XTEND †	Source	Disable Cx51 extensions to ANSI C.
<u>O</u> B <u>J</u> ECT, <u>N</u> O <u>O</u> B <u>J</u> ECT †	Object	Specify a name for the object file or suppress the object file.
<u>O</u> B <u>J</u> ECT <u>E</u> XTEND †	Object	Include additional variable type information in the object file.
<u>O</u> N <u>E</u> REG <u>B</u> ANK	Object	Assume that only registerbank 0 is used in interrupt code.

Directive	Class	Description
<u>OMF2</u> †	Object	Generate OMF2 output file format.
<u>OPTIMIZE</u>	Object	Specify the level of optimization performed by the compiler.
<u>ORDER</u> †	Object	Allocate variables in the order of their appearance in the source file.
<u>PAGELNGTH</u> †	Listing	Specify the number of rows on the page.
<u>PAGEWIDTH</u> †	Listing	Specify the number of columns on the page.
<u>PREPRINT</u> †	Listing	Produce a preprocessor listing file where all macros are expanded.
<u>PRINT</u>, <u>NOPRINT</u> †	Listing	Specify a name for the listing file or disable the listing file.
<u>REGFILE</u> †	Object	Specify a register definition file for global register optimization.
<u>REGISTERBANK</u>	Object	Select the register bank to use for absolute register accesses.
<u>REGPARMS</u>, <u>NOREGPARMS</u>	Object	Enable or disable register parameter passing.
<u>RET_PSTK</u> †, <u>RET_XSTK</u> †	Object	Use reentrant stack for saving return addresses.
<u>ROM</u> †	Object	Control generation of AJMP/ACALL instructions.
<u>SAVE</u>, <u>RESTORE</u>	Object	Saves and restores settings for AREGS, REGPARMS and OPTIMIZE directives.
<u>SMALL</u> †	Object	Select SMALL memory model. (Default.)
<u>SRC</u> †	Object	Create an assembler source file instead of an object module.
<u>STRING</u> †	Object	Locate implicit string constants to xdata or far memory.
<u>SYMBOLS</u> †	Listing	Include a list of all symbols used within the module in the listing file.
<u>USERCLASS</u> †	Object	Renames memory class names for flexible variable location.
<u>VARBANKING</u> †	Object	Enable far memory type variables.
<u>WARNINGLEVEL</u> †	Listing	Select the level of Warning detection.
<u>XCROM</u> †	Object	Assume ROM space for const xdata variables.

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

Control directives and their arguments, with the exception of arguments specified with the **DEFINE** directive, are not case sensitive.

Reference

The remainder of this chapter describes each of the available **Cx51** compiler control directives listed in alphabetical order. They are 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.
- μVision2 Control:** Lists how to specify the directive.
- Description:** Provides a detailed description of the directive and how to use it.
- See Also:** Names related directives.
- Example:** Shows you an example of how to use and, sometimes, the effects of the directive.

AREGS / NOAREGS

Abbreviation: None.

Arguments: None.

Default: **AREGS**

µVision2 Control: Options – C51 – Don't use absolute register accesses.

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

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

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

NOTE

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

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

```

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

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

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

```

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

```

MOV   A,R7
PUSH  ACC

```

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

```

PUSH  AR7

```

ASM / ENDASM

Abbreviation: None.

Arguments: None.

Default: None.

µVision2 Control: This directive may not be specified on the command line.

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

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

In µVision2 you may set a file specific option for C source files that contain **ASM/ENDASM** sections as follows:

- Right click on the file in the Project Window – Files tab
- Choose **Options for...** to open Options – Properties page
- Enable **Generate Assembler SRC file**
- Enable **Assemble SRC file**.

With this setting, µVision2 generates an assembler source file (**.SRC**) and translates this file with the Assembler to an Object file (**.OBJ**).

The **ENDASM** directive signals the end of the source text block.

NOTE

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

Example:

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

The following C source file:

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

generates the following .SRC file.

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

BROWSE

Abbreviation: BR

Arguments: None.

Default: No browse information is created

µVision2 Control: Options – Output – Browse Information

Description: With **BROWSE**, the compiler creates browse information. The browse information covers identifiers (including preprocessor symbols), their memory space, type, definition- and reference lists.

This information can be displayed within µVision2. Select **View - Source Browser** to open the µVision2 Source Browser. Refer to the *µVision2 Getting Started User's Guide, Chapter 4, µVision2 Utilities, Source Browser* for more information.

Example:

```
C51 SAMPLE.C BROWSE
#pragma browse
```

CODE

Abbreviation: CD

Arguments: None.

Default: No assembly code listing is generated.

µVision2 Control: Options – Listing – C Compiler Listing – 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:

```
C51 SAMPLE.C CD
#pragma code
```

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

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

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

COMPACT

Abbreviation: CP

Arguments: None.

Default: SMALL

µVision2 Control: Options – Target – Memory Model

Description: This directive selects the **COMPACT** memory model.

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

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

NOTE

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

See Also: SMALL, LARGE, ROM

Example:

```
C51 SAMPLE.C COMPACT
#pragma compact
```

COND / NOCOND

Abbreviation: CO

Arguments: None.

Default: COND

µVision2 Control: Options – Listing – C Compiler Listing – Conditional.

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 includes lines omitted from compilation in the listing file. Line numbers and nesting levels are not output, making the file easier to read.

The effect of this directive takes place one line after the preprocessor detects it.

The **NOCOND** directive excludes lines omitted from compilation from 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.

µVision2 Control: Options – Output – 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 the symbolic testing of programs. This information contains both global and local variable definitions and their addresses, as well as function names and their line numbers. Debug information contained in each object module remains valid through the Link/Locate procedure. This information may be used by the µVision2 debugger or by any of the Intel-compatible emulators.

NOTE

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

See Also: **OBJECTTEXTEND**

Example:

```
C51 SAMPLE.C DEBUG
#pragma db
```

DEFINE

Abbreviation: DF

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

Default: None.

µVision2 Control: Enter names to define at Options – Cx51 – Define.

Description: The **DEFINE** directive defines names on the invocation line which the preprocessor may query 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 assigned a value.

NOTE

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

Example:

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

DISABLE

Abbreviation: None.

Arguments: None.

Default: None.

µVision2 Control: This directive may not be specified on the command line. It may be specified only in the source file.

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 on 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:

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

```

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

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

```

EJECT

Abbreviation: EJ

Arguments: None.

Default: None.

µVision2 Control: This directive may not be specified on the command line. It may be specified only in the source file.

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

NOTE

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

Example: `#pragma eject`

FLOATFUZZY

Abbreviation: FF

Arguments: A number between 0 and 7.

Default: **FLOATFUZZY (3)**

µVision2 Control: Options – Cx51 – 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:

```
C51 MYFILE.C FLOATFUZZY (2)
#pragma FF (0)
```

INCDIR

Abbreviation: None.

Arguments: Path specifications for include files enclosed in parentheses.

Default: None.

µVision2 Control: Options – Cx51 – Include Paths.

Description: The **INCDIR** directive specifies the location of the **Cx51** compiler include files. The compiler accepts a maximum of 50 path declarations. If more than one path declaration is required, the path names must be separated by semicolons. If you specify `#include "filename.h"`, the **Cx51** Compiler searches first the current directory and then the source file directory. When this search fails or the `#include <filename.h>` is used, the paths specified by the **INCDIR** directive are searched. When these searches still fail, the paths specified by the **C51INC** environment variable are used.

Example:

```
C51 SAMPLE.C INCDIR(C:\KEIL\C51\MYINC;C:\CHIP_DIR)
```

INTERVAL

Abbreviation: None

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

Default: **INTERVAL (8)**

µVision2 Control: Options – Cx51 – Misc controls: enter the directive.

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

$$(\mathit{interval} \times \mathit{n}) + \mathit{offset} + 3,$$

where:

interval is the argument of the **INTERVAL** directive (default 8).

n is the interrupt number.

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

See Also: **INTVECTOR / NOINTVECTOR**

Example:

```
C51 SAMPLE.C INTERVAL(3)
#pragma interval(3)
```


INTPROMOTE / NOINTPROMOTE

Abbreviation: IP / NOIP

Arguments: None.

Default: INTPROMOTE

µVision2 Control: Options – Cx51 – Enable ANSI integer promotion rules.

Description: The **INTPROMOTE** directive enables ANSI integer promotion rules. Expressions used in if statements are promoted from smaller types to integer expressions before comparison. This allows Microsoft C and Borland C programs to be ported to **Cx51** with fewer modifications.

Since the 8051 is an 8-bit processor, use of the **INTPROMOTE** directive may generate less efficient code in some applications.

The **NOINTPROMOTE** directive disables automatic integer promotions. Integer promotions are normally enabled to provide the greatest compatibility between **Cx51** and other ANSI compilers. However, integer promotions can yield inefficient code on the 8051.

Example:

```
C51 SAMPLE.C INTPROMOTE
#pragma intpromote
C51 SAMPLE.C NOINTPROMOTE
```

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

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

Code generated with INTPROMOTE

```

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

```

CODE SIZE = 63 Bytes

Code generated with NOINTPROMOTE

```

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

```

CODE SIZE = 49 Bytes

INTVECTOR / NOINTVECTOR

Abbreviation: IV / NOIV

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

Default: INTVECTOR (0)

µVision2 Control: Options – Cx51 – Misc controls: enter the directive.

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

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

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

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

$$(\textit{interval} \times \textit{n}) + \textit{offset} + 3,$$

where:

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

See Also: INTERVAL

Example:

```
C51 SAMPLE.C INTVECTOR(0x8000)
#pragma iv(0x8000)
C51 SAMPLE.C NOINTVECTOR
#pragma noiv
```

LARGE

Abbreviation: LA

Arguments: None.

Default: SMALL

µVision2 Control: Options – Target – Memory Model

Description: This directive selects the **LARGE** memory model.

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

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

NOTE

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

See Also: SMALL, COMPACT, ROM

Example:

```
C51 SAMPLE.C LARGE
#pragma large
```

LISTINCLUDE

Abbreviation: LC

Arguments: None.

Default: NOLISTINCLUDE

µVision2 Control: Options – Listing – C Compiler Listing – #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:

```
C51 SAMPLE.C LISTINCLUDE
#pragma listinclude
```

MAXARGS

Abbreviation: None.

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

µVision2 Control: Options – Cx51 – Misc controls: enter the directive.

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

MAXARGS(40) for large model.

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

Example:

```
C51 SAMPLE.C MAXARGS(20)

#pragma maxaregs (4) /* allow 4 bytes for parameters */
#include <stdarg.h>

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

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

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

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


MOD517 / NOMOD517

Abbreviation: None.

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

Default: **NOMOD517**

µVision2 Control: Options – Target – Use On-Chip Arithmetic Unit
Options – Target – Use multiple DPTR registers

Description: The **MOD517** directive instructs the **Cx51** compiler to produce code for the additional hardware components (the arithmetic processor and the additional data pointers) of the Infineon C517 or variants. This feature improves the performance of integer, long, and floating-point math operations, as well as functions that make use of the additional data pointers.

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

Library functions that take advantage of the arithmetic processor have a **517** suffix. (Refer to “Chapter 8. Library Reference” on page 209 for details on these functions.)

Additional parameters may be specified with **MOD517** to control **Cx51** support of the individual components of the Infineon device. When specified, the parameters must appear within parentheses immediately following the **MOD517** directive. Parentheses are not required if none of these additional parameters is specified.

Directive	Description
NOAU	When specified, the Cx51 Compiler uses only the additional data pointers of the Infineon device. The arithmetic processor is not used. The NOAU parameter is useful for functions that are called by an interrupt while the arithmetic processor is already being used.
NODP8	When specified, the Cx51 Compiler uses only the arithmetic processor. The additional data pointers are not used. The NODP8 parameter is useful for interrupt functions declared without the using function attribute. In this case, the extra data pointers are not used and, therefore, do not need to be saved on the stack during the interrupt.

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

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

NOTE

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

See Also:

MODA2, MODAD2, MODDA, MODDP2, MODP2

Example:

```
C51 SAMPL517.C MOD517
#pragma MOD517 (NOAU)
#pragma MOD517 (NODP8)
#pragma MOD517 (NODP8, NOAU)
C51 SAMPL517.C NOMOD517
#pragma NOMOD517
```

MODA2 / NOMODA2

Abbreviation: None.

Arguments: None.

Default: **NOMODA2**

µVision2 Control: Options – Target – Use multiple DPTR registers

Description: The **MODA2** directive instructs the **Cx51** compiler to produce code for the additional hardware components (specifically, the additional CPU data pointers) available in the Atmel 80x8252 or variants and compatible derivatives. Using additional data pointers can improve the performance of the following library functions: **memcpy**, **memmove**, **memcpy**, **strcpy**, and **strcmp**.

The **NOMODA2** directive disables generation of code that utilizes the additional CPU data pointers.

See Also: **MOD517, MODAB2, MODDP2, MODP2**

Example:

```
C51 SAMPLE.C MODA2
#pragma moda2

C51 SAMPLE.C NOMODA2
#pragma nomoda2
```

MODAB2 / NOMODAB2

Abbreviation: None.

Arguments: None.

Default: **NOMODAB2**

µVision2 Control: Options – Target – Use multiple DPTR registers

Description: The **MODAB2** directive instructs the **Cx51** compiler to produce code for the additional hardware components (specifically, the additional CPU data pointers) available in the Analog Devices B2 series of MicroConverters. Using additional data pointers can improve the performance of the following library functions: **memcpy**, **memmove**, **memcpy**, **strcpy**, and **strcmp**.

The **NOMODAB2** directive disables generation of code that utilizes the additional CPU data pointers.

See Also: **MOD517, MODA2, MODDP2, MODP2**

Example:

```
C51 SAMPLE.C MODAB2
#pragma moda2

C51 SAMPLE.C NOMODAB2
#pragma nomoda2
```

MODDA2 / NOMODDA2

Abbreviation: None.

Arguments: None.

Default: **NOMODDA2**

µVision2 Control: Options – Target – Use On-Chip Arithmetic Accelerator

Description: The **MODDA2** directive instructs the **Cx51** compiler to produce code for the additional hardware components (the arithmetic accelerator) of the Dallas Semiconductor DS80C390, DS80C400 and DS5240. This feature improves the performance of integer, and long operations.

The **NOMODDA** directive disables generation of code that utilizes the on-chip Arithmetic Accelerator.

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

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

See Also: **MOD517**

Example:

```
C51 SAMPL390.C MODDA
#pragma modda

C51 SAMPL390.C NOMODDA
#pragma nomodda
```

MODDP2 / NOMODDP2

Abbreviation: None.

Arguments: None.

Default: **NOMODDP2**

µVision2 Control: Options – Target – Use multiple DPTR registers

Description: The **MODDP2** directive instructs the **Cx51** compiler to produce code for the additional hardware components (specifically, the additional CPU data pointers) available in the Dallas 80C320, C520, C530, C550, or variants and compatible derivatives. Using additional data pointers can improve the performance of the following library functions: **memcpy**, **memmove**, **memcmp**, **strcpy**, and **strcmp**.

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

See Also: **MOD517**, **MODA2**, **MODP2**

Example:

```
C51 SAMPL320.C MODDP2
#pragma moddp2

C51 SAMPL320.C NOMODDP2
#pragma nomoddp2
```

MODP2 / NOMODP2

Abbreviation: None.

Arguments: None.

Default: NOMODP2

µVision2 Control: Options – Target – Use multiple DPTR registers

Description: The **MODP2** directive instructs the **Cx51** compiler to use the additional DPTR registers (dual data pointers) that are available in some 8051 variants from Philips or Atmel/WM. Using additional data pointers can improve the performance of the following library functions: **memcpy**, **memmove**, **memcpy**, **strcpy**, and **strcmp**.

The **NOMODP2** directive disables generation of code that utilizes the dual DPTR registers.

See Also: **MOD517**, **MODA2**, **MODAB2**, **MODDP2**

Example:

```
C51 SAMPLE.C MODP2
#pragma modp2

C51 SAMPLE.C NOMODP2
#pragma nomodp2
```

NOAMAKE

Abbreviation: NOAM

Arguments: None.

Default: AutoMAKE information is generated.

µVision2 Control: This directive may not be used with µVision2.

Description: **NOAMAKE** disables the AutoMAKE project information records produced by the **Cx51** compiler. It also disables the register optimization information.

Use **NOAMAKE** to generate object files that can be used with older versions of the 8051 development tool chain.

Example:

```
C51 SAMPLE.C NOAMAKE
#pragma NOAM
```


NOEXTEND

Abbreviation: None.

Arguments: None.

Default: All language extensions are enabled.

µVision2 Control: Enter **NOEXTEND** at Options – C51 – Misc Controls

Description: The **NOEXTEND** control directs the compiler to process only ANSI C language constructs. The **Cx51** language extensions are disabled. Reserved keywords such as **bit**, **reentrant** and **using** are not recognized and generate compilation errors or warnings.

Example:

```
C51 SAMPLE.C NOEXTEND
#pragma NOEXTEND
```

OBJECT / NOOBJECT

Abbreviation: OJ / NOOJ

Arguments: An optional filename enclosed in parentheses.

Default: OBJECT (*filename.OBJ*)

µVision2 Control: Options – Output – Select Folder for Objects

Description: The **OBJECT**(*filename*) directive changes the name of the object file to the file name provided. By default, the object files are created using the source file name and the **OBJ** extension.

The **NOOBJECT** control prevents an object file from being created.

Example:

```
C51 SAMPLE.C OBJECT(sample1.obj)
#pragma oj(sample_1.obj)
C51 SAMPLE.C NOOBJECT
#pragma nooj
```

OBJECTADVANCE

Abbreviation: OA

Arguments: None.

Default: None.

µVision2 Control: Options – C51 – Code Optimization – Linker Code Packing

Description: The **OBJECTADVANCED** directive instructs the compiler to include information in the object file for linker-level program optimizations. This directive is used in conjunction with the **OPTIMIZE** directive to shrink program size and decrease execution speed.

When enabled, the **OBJECTADVANCED** directive instructs the LX51 linker/locater to perform the following optimizations:

OPTIMIZE Level	Linker Optimizations Performed
0 – 7	Maximize AJMP / ACALL: The linker rearranges code segments to maximize AJMP and ACALL instructions which are shorter than LJMP and LCALL instructions.
8	Reuse of Common Entry Code: Setup code may be reused when multiple calls are made to a single function. Reusing common entry code reduces program size. This optimization is performed on the complete application.
9	Common Block Subroutines: Recurring instruction sequences are converted into subroutines. This reduces program size but slightly increases execution speed. This optimization is performed on the complete application.
10	Rearrange Code: When detecting common block subroutines, code is rearranged to obtain larger recurring sequences.
11	Reuse of Common Exit Code: Identical exit sequences are reused. This may reduce the size of common block subroutines even further. This optimization level generates the most compact program code possible.

See Also: **OPTIMIZE, OMF2**

Example: `C51 SAMPLE.C OBJECTADVANCED DEBUG`

OBJECTEXTEND

Abbreviation: OE

Arguments: None.

Default: None.

µVision2 Control: Options – Output – Debug Information

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

NOTE

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

See Also: DEBUG, OMF2

Example:

```
C51 SAMPLE.C OBJECTEXTEND DEBUG
#pragma oe db
```

ONEREBANK

Abbreviation: OB

Arguments: None

Default: None

µVision2 Control: Enter **ONEREBANK** at Options – C51 – Misc controls

Description: Cx51 selects registerbank 0 on entry to interrupts that do not specify the **using** attribute. This is done at the beginning of the interrupt service routine with the **MOV PSW,#0** instruction. This ensures that high-priority interrupts that do not use the **using** attribute can interrupt lower priority interrupts that use a different registerbank.

If your application uses only one registerbank for interrupts, you may use the **ONEREBANK** directive. This eliminates the **MOV PSW,#0** instruction.

Example:

```
C51 SAMPLE.C ONEREBANK
#pragma OB
```

OMF2

Abbreviation: O2

Arguments: None

Default: The C51 compiler generates by default the Intel OMF51 file format. The OMF2 file format is default for the **Cx51** compiler.

µVision2 Control: Project – Select Device – Use LX51 instead of BL51.

Description: The **OMF2** directive enables the OMF2 file format which provides detailed symbol type checking across modules and eliminates the historic limitations of the Intel OMF51 file format.

The OMF2 file format is required when you want to use one of the following features of the **Cx51** compiler:

- Variable Banking: The **VARBANKING** directive enables use of the **far** memory type.
- XDATA ROM: Using the **const xdata** memory type specifies that XDATA variables are located in ROM.
- RAM Strings: The **STRING** directive specifies that string constants are located in xdata or far space.
- Contiguous Mode: The **ROM (D512K)** and **ROM (D16M)** directives enable the contiguous mode of the Dallas Semiconductor 390 and variants.

The OMF2 file format requires the extended **LX51** linker/locater and cannot be used with the BL51 linker/locater.

See Also: **OBJECTTEXTEND**

Example:

```
C51 SAMPLE.C OMF2
#pragma O2
```

OPTIMIZE

Abbreviation: OT

Arguments: A decimal number between 0 and 9 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 (8, SPEED)**

µVision2 Control: Options – C51 – Code Optimization

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	<p>Constant Folding: The compiler performs calculations that reduce expressions to numeric constants, where possible. This includes calculations of run-time addresses.</p> <p>Simple Access Optimizing: The compiler optimizes access of internal data and bit addresses in the 8051 system.</p> <p>Jump Optimizing: The compiler always extends jumps to the final target. Jumps to jumps are deleted.</p>
1	<p>Dead Code Elimination: Unused code fragments and artifacts are eliminated.</p> <p>Jump Negation: Conditional jumps are closely examined to see if they can be streamlined or eliminated by the inversion of the test logic.</p>
2	<p>Data Overlaying: Data and bit segments suitable for static overlay are identified and internally marked. The BL51 Linker/Locator has the capability, through global data flow analysis, of selecting segments which can then be overlaid.</p>
3	<p>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.</p>

Level	Description
4	<p>Register Variables: Automatic variables and function arguments are located in registers when possible. Reservation of data memory for these variables is omitted.</p> <p>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.</p> <p>Local Common Subexpression Elimination: If the same calculations are performed repetitively in an expression, the result of the first calculation is saved and used further whenever possible. Superfluous calculations are eliminated from the code.</p> <p>Case/Switch Optimizing: Code involving switch and case statements is optimized as jump tables or jump strings.</p>
5	<p>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.</p> <p>Simple Loop Optimizing: Program loops that fill a memory range with a constant are converted and optimized.</p>
6	<p>Loop Rotation: Program loops are rotated if the resulting program code is faster and more efficient.</p>
7	<p>Extended Index Access Optimizing: Uses the DPTR for register variables where appropriate. Pointer and array access are optimized for both execution speed and code size.</p>
8	<p>Common Tail Merging: When there are multiple calls to a single function, some of the setup code can be reused, thereby reducing program size.</p>
9	<p>Common Block Subroutines: Detects recurring instruction sequences and converts them into subroutines. Cx51 even rearranges code to obtain larger recurring sequences.</p>

OPTIMIZE level 9 includes all optimizations of levels 0 to 8.

Example:

```
C51 SAMPLE.C OPTIMIZE (9)
C51 SAMPLE.C OPTIMIZE (0)
#pragma ot(6, SIZE)
#pragma ot(size)
```


ORDER

Abbreviation: OR

Arguments: None.

Default: The variables are not ordered.

µVision2 Control: Options – C51 – Keep Variables in Order

Description: The **ORDER** directive instructs the **Cx51** 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. This directive causes the **Cx51** to compile more slowly.

Example:

```
C51 SAMPLE.C ORDER
#pragma OR
```

PAGELength

Abbreviation: PL

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

Default: **PAGELength (60)**

µVision2 Control: Options – Listing – Page Length

Description: The **PAGELength** directive specifies the number of lines printed per page in the listing file. The default is 60 lines per page, including headers and empty lines.

See Also: **PAGEWIDTH**

Example:

```
C51 SAMPLE.C PAGELength (70)
#pragma pl (70)
```

PAGEWIDTH

Abbreviation: PW

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

Default: PAGEWIDTH (132)

µVision2 Control: Options – Listing – Page Width

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

See Also: PAGELENGTH

Example:

```
C51 SAMPLE.C PAGEWIDTH(79)
#pragma pw(79)
```

PREPRINT

Abbreviation: PP

Arguments: An optional filename enclosed in parentheses.

Default: No preprocessor listing is generated.

µVision2 Control: Options – C51 – C Preprocessor Listing

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 used. By default, the **Cx51** 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 using the **#pragma** directive.*

Example:

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

PRINT / NOPRINT

Abbreviation: PR / NOPR

Arguments: An optional filename enclosed in parentheses.

Default: PRINT (*filename.LST*)

µVision2 Control: Options – Listing – Select Folder for List Files

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

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

Example:

```
C51 SAMPLE.C PRINT(CON:)
#pragma pr (\usr\list\sample.lst)
C51 SAMPLE.C NOPRINT
#pragma nopr
```

REGFILE

Abbreviation: RF

Arguments: A file name enclosed in parentheses.

Default: None.

µVision2 Control: Options – C51 – Global Register Coloring

Description: The **REGFILE** directive instructs the **Cx51** compiler to use a register definition file for global register optimization. The register definition file specifies the register usage of external functions. Using this information, the **Cx51** compiler can optimize the use of the general purpose registers. This feature enables global program-wide register optimization.

Example:

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

REGISTERBANK

Abbreviation: RB

Arguments: A number in range 0-3 enclosed in parentheses.

Default: REGISTERBANK (0)

µVision2 Control: Enter the REGISTERBANK directive at Options – C51 – Misc controls.

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

NOTE

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

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

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

Example:

```
C51 SAMPLE.C REGISTERBANK(1)
#pragma rb(3)
```

REGPARMS / NOREGPARMS

Abbreviation: None.

Arguments: None.

Default: **REGPARMS**

µVision2 Control: Enter the **REGPARMS** directive at Options – C51 – Misc controls.

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

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

NOTE

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

*Use **NOREGPARMS** to access older assembler functions or library files without having to reassemble or recompile them. This is illustrated in the following example program.*

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

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

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

Example:

```
C51 SAMPLE.C NOREGPARMS
```

RET_PSTK, RET_XSTK

Abbreviation: RP, RX

Arguments: None.

Default: None.

µVision2 Control: Enter the **RET_PSTK**, **RET_XSTK** directive at Options – C51 – Misc controls.

Description: The **RET_PSTK**, and **RET_XSTK** directives cause the **pdata** or **xdata** reentrant stacks to be used for return addresses. Normally, return addresses are stored on the 8051's hardware stack. These directives instruct the compiler to generate code that pops the return address from the hardware stack and stores it on the reentrant stack specified.

RET_PSTK Uses the compact model reentrant stack.

RET_XSTK Uses the large model reentrant stack.

NOTE

*You may use the **RET_xSTK** directives to unload return addresses from the on-chip or hardware stack. These directives may be selectively used on the modules that contain the deepest stack nesting.*

If you use one of these directives you must initialize the reentrant stack pointer defined in the startup code. Refer to "STARTUP.A51" on page 151 for more information on how to initialize the reentrant stacks.

```
1      #pragma RET_XSTK
2      extern void func2 (void);
3
4      void func (void) {
5 1      func2 ();
6 1      }

ASSEMBLY LISTING OF GENERATED OBJECT CODE
; FUNCTION func (BEGIN)
0000 120000      E      LCALL  ?C?CALL_XBP
; SOURCE LINE # 5
0003 120000      E      LCALL  func2
; SOURCE LINE # 6
0006 020000      E      LJMP   ?C?RET_XBP
; FUNCTION func (END)
```

Example:

```
C51 SAMPLE.C RET_XSTK
```

ROM

Abbreviation: None.

Arguments: (SMALL), (COMPACT), (LARGE), (D512K), or (D16M)

Default: ROM (LARGE)

µVision2 Control: Options – Target – Code Rom Size

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.

Option	Description
SMALL	CALL and JMP instructions are coded as ACALL and AJMP . The maximum program size may be 2 KBytes. The entire program must be allocated within the 2 KByte program memory space.
COMPACT	CALL instructions are coded as LCALL . JMP instructions are coded as AJMP within a function. The size of a function must not exceed 2 KBytes. The entire program may, however, comprise a maximum of 64 KBytes. The type of application determines whether or not ROM (COMPACT) is more advantageous than ROM (LARGE) . Any code space saving advantages in using ROM (COMPACT) must be empirically determined.
LARGE	CALL and JMP instructions are coded as LCALL and LJMP . This allows you to use the entire address space without any restrictions. Program size is limited to 64 KBytes. Function size is also limited to 64 KBytes.
D512K † (Dallas 390 & variants)	19-bit ACALL and AJMP instructions are generated. The maximum program size may be 512 KBytes. This mode is available only for the Dallas 390 and compatible devices.
D16M † (Dallas 390 & variants)	24-bit LCALL instructions and 19-bit AJMP instructions are generated. The maximum program size may be 16MBytes. This mode is available only for the Dallas 390 and compatible devices.

† The **D512K** and **D16M** options require the OMF2 directive.

See Also: **SMALL, COMPACT, LARGE**

Example:

```
C51 SAMPLE.C ROM (SMALL)
#pragma ROM (SMALL)
```

SAVE / RESTORE

Abbreviation: None.

Arguments: None.

Default: None.

µVision2 Control: This directive cannot be specified on the command line.

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

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

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

NOTE

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

Example:

```
#pragma save
#pragma noregparms

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

#pragma restore
```

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

SMALL

Abbreviation: SM

Arguments: None.

Default: SMALL

µVision2 Control: Options – Target – Memory Model

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

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

NOTE

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

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

See Also: COMPACT, LARGE, ROM

Example:

```
C51 SAMPLE.C SMALL
#pragma small
```

SRC

Abbreviation: None.

Arguments: An optional filename in parentheses.

Default: None.

µVision2 Control: Can be set under µVision2 as follows:

- Right click on the file in the Project Window – Files tab
- Choose **Options for...** to open Options – Properties page
- Enable **Generate Assembler SRC file**

Description: Use the **SRC** directive to create an assembler source file instead of an object file. This source file may be assembled with the A51 assembler.

If a filename is not specified in parentheses, the base name and path of the C source file are used with the **.SRC** extension.

NOTE

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

See Also: **ASM, ENDASM**

Example:

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

STRING

Abbreviation: ST

Arguments: (CODE), (XDATA), or (FAR)

Default: STRING (CODE)

µVision2 Control: Enter the directive **STRING** at Options – C51 – Misc controls.

Description: The **STRING** directive allows you to specify the memory type used for implicit strings. By default, strings are implicitly located in code memory. For example, “*hello world*” is located in code memory in the following:

```
void main (void) {
    printf ("hello world\n");
}
```

By using the **STRING** directive you can change the location of such strings. This option must be used carefully, since existing programs might use memory typed pointers to access strings. By allocating strings into the **xdata** or **far** memory space, you may avoid the use of code banking in your application. This option is useful especially for extended 8051 devices like the Philips 80C51MX.

Option	Description
CODE	Implicit strings are located in code space. This is the default setting of the Cx51 compiler.
XDATA †	Implicit strings are located in const xdata space.
FAR †	Implicit strings are located in const far space.

† The option **XDATA** and **FAR** require the OMF2 directive.

See Also: OMF2, XCROM

Example:

```
C51 SAMPLE.C STRING (XDATA)
#pragma STRING (FAR)
```


SYMBOLS

Abbreviation: SB

Arguments: None.

Default: No list of symbols is generated.

µVision2 Control: Options – Listing – C Compiler Listing - Symbols

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

Example:

```
C51 SAMPLE.C SYMBOLS
#pragma SYMBOLS
```

The following listing file excerpt shows the symbol listing:

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

USERCLASS

Abbreviation: UCL

Arguments: (*m*space = *user_classname*)

*m*space refers to the default memory space used for variables or program code and is explained below:

<i>m</i> space	Description
CODE	Program code.
CONST	Variables in code space (CONST class).
XCONST	Constants in const xdata space (XDATA class).
XDATA	Variables in xdata space (XDATA class).
HDATA	Variables in extended far space (HDATA class).
HCONST	Constants in extended const far space (HCONST class).

user_classname is the name for a memory class. You can supply any valid identifier for a class name.

Default: Segments receive the default class name.

µVision2 Control: Enter the directive **USERCLASS** at Options – C51 – Misc controls.

Description: The **USERCLASS** directive assigns a user defined class name for a compiler generated segment. By default, the **Cx51** compiler uses the basic class name for segment definitions. The user class name may be referenced at the extended LX51 linker/locater level to locate all segments with a class name, such as **HDATA_FLASH**, to a specific memory section. The **USERCLASS** directive renames the basic class name for a complete module, but not overlayable segments.

Memory classes are only available when you use the OMF2 format and the extended LX51 linker/locater.

Example:

```
C51 UCL.C
#pragma userclass (xdata = flash)
#pragma userclass (hconst = patch)

    int  xdata x1 [10];           // XDATA_FLASH
const char far  tst[] = "Hello"; // HCONST_PATCH
```

VARBANKING

Abbreviation: VB

Arguments: None No modification of the interrupt code.
(1) Save address extension SFR in interrupt code.

Default: The standard C51 library is used.

µVision2 Control: Options – Target – ‘far’ memory type support.
Options – Target – save address extension SFR in interrupts.

Description: The **VARBANKING** directive allows you to use **far** memory on classic 8051 devices. When enabled, a different set of library functions that support **far** memory are selected.

The access functions for **far** variables are configured in **XBANKING.A51**. Refer to “XBANKING.A51” on page 154 for more information.

VARBANKING (1) adds save and restore code for the address extension SFR to interrupt functions. The symbol **?C?XPAGE1SFR** defined in **XBANKING.A51** specifies the address of the address extension SFR. The initial value of this SFR is specified with the symbol **?C?XPAGE1RST**. At the beginning of the interrupt function the instruction **MOV ?C?XPAGE1SFR,?C?XPAGE1RST** is inserted.

NOTE

*Only interrupt functions in C modules that are translated with the **VARBANKING (1)** directive save and restore the address extension SFR. If your application contains other interrupt functions in assembly modules or libraries, you must check these functions carefully.*

The examples in `\KEIL\C51\EXAMPLES\FARMEMORY\` show how to use the C51 far memory type on classic 8051 devices.

Example:

```
C51 SAMPLE.C VARBANKING
C51 MYFILE.C VARBANKING (1)
```

WARNINGLEVEL

Abbreviation: WL

Arguments: A number from 0-2.

Default: WARNINGLEVEL (2)

µVision2 Control: Options – C51 – Warnings

Description: The **WARNINGLEVEL** directive allows you to suppress compiler warnings. Refer to “Chapter 7. Error Messages” on page 189 for a full list of the compiler warnings.

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

Example:

```
C51 SAMPLE.C WL (1)
#pragma WARNINGLEVEL (0)
```

XCROM

Abbreviation: XC

Arguments: None.

Default: All xdata variables are initialized during the execution of the startup code.

µVision2 Control: Enter the directive **XCROM** at Options – C51 – Misc controls.

Description: The **XCROM** directive directs the compiler to store constant variables in **xdata** memory rather than **code** memory. These variables must be declared using **const xdata**. This frees up code memory for your application program.

Some new 8051 devices provide a memory management unit which allows you to map ROM space into the xdata memory area. For classic 8051 devices you may use a ROM device instead of RAM for the xdata space.

See Also: **OMF2, STRING**

Example:

```
#pragma XCROM // Enable const xdata ROM

/*
 * The following text will be in a ROM that
 * is addressed in the XDATA space.
 */

const char xdata text [] = "Hello World\n";

void main (void) {
    printf (text);
}
```


Chapter 3. Language Extensions

The **Cx51** compiler provides several extensions to ANSI Standard C to support the elements of the 8051 architecture. These include extensions for:

- Memory Areas
- Memory Types
- Memory Models
- Memory Type Specifiers
- Variable Data Type Specifiers
- Bit Variables and Bit-addressable Data
- Special Function Registers
- Pointers
- Function Attributes

3

The following sections describe each of these in detail.

Keywords

To facilitate many of the features of the 8051, the **Cx51** compiler adds a number of new keywords to the scope of the C language:

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

You may disable these extensions using the **NOEXTEND** control directive. Refer to “Chapter 2. Compiling with the Cx51” on page 17 for more information.

Memory Areas

The 8051 architecture supports several physically separate memory areas or memory spaces for program and data. Each memory area offers certain advantages and disadvantages. There are memory spaces that may be:

- read from but not written to.
- read from or written to.
- read from or written to more quickly than other memory spaces.

This wide variety of memory space is quite different from most mainframe, minicomputer, and microcomputer architectures where the program, data, and constants are all loaded into the same physical memory space within the computer. Refer to the *Intel 8-Bit Embedded Controllers* handbook or other 8051 data books for more information about the 8051 memory architecture.

Program Memory

Program (CODE) memory is read only; it cannot be written to. Program memory may reside within the 8051 CPU, it may be external, or it may be both, depending upon the 8051 derivative and the hardware design.

There may be up to 64 KBytes of program memory. Program code, including all functions and library routines, is stored in program memory. Constant variables may also be stored in program memory. The 8051 executes programs stored in program memory only.

Program memory may be accessed using the **code** memory type specifier in the **Cx51** compiler.

Internal Data Memory

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

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

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

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

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

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

External Data Memory

External data memory is read/write. Access to external data is slower than access to internal data memory because the external data memory is indirectly accessed through a data pointer register which must be loaded with an address.

Several 8051 devices provide on-chip XRAM space that is accessed with the same instructions as the traditional external data space. This XRAM space is typically enabled via dedicated chip configuration SFR registers and overlaps the external memory space.

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

The **Cx51** Compiler offers two different memory types that 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 one (1) page or 256 bytes of external data memory. See “Compact Model” on page 95 for more information on **pdata**.

Far Memory

Far memory refers to the extended address space of many new 8051 variants. The **Cx51** Compiler uses generic 3-byte pointers to access extended memory spaces. Two **Cx51** memory types, **far** and **const far**, access variables in extended RAM space and constants in extended ROM space.

The **Philips 51MX Architecture** provides hardware support for 8MB **code** and **xdata** space using universal pointers. The new instructions of the 80C51MX architecture are used by the **Cx51** compiler to access **far** and **const far** variables.

The **Dallas 390 Architecture** supports an extended **code** and **xdata** address space in contiguous mode with a 24-bit DPTR register and the traditional MOVX and MOVC instructions. Variables defined with **far** and **const far** are located in these extended **xdata** and **code** address spaces.

Classic 8051 devices may also use **far** and **const far** variables if you configure **XBANKING.A51** for your target hardware. This is useful for devices that provide an address extension SFR or additional memory spaces that can be mapped into the **xdata** space. You may also use **xdata** banking hardware to extend the address space of a classic 8051 device. Refer to “XBANKING.A51” on page 154 for more information.

NOTE

*You need to specify the C51 directive OMF2 and the extended LX51 linker/locator to use the **far** and **far const** memory types.*

Special Function Register Memory

The 8051 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 101 for more information on SFRs.

Memory Models

The memory model determines the 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 **Cx51** compiler command line using the **SMALL**, **COMPACT** and **LARGE** control directives. Refer to “Control Directives” on page 20 for more information about these directives.

NOTE

*Except in very special select applications, the **SMALL** memory model generates the fastest, most efficient code.*

By explicitly declaring a variable with a memory type specifier, you may override the default memory type imposed by the memory model. Refer to “Memory Types” on page 95 for more information.

Small Model

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

Compact Model

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

When using the compact model, the **Cx51** compiler accesses external memory with instructions that utilize the @R0 and @R1 operands. R0 and R1 are byte registers and provide only the low-order byte of the address. If the compact model is used with more than 256 bytes of external memory, the high-order address byte (or page) is provided by Port 2 on the 8051. In this case, you must initialize Port 2 with the proper external memory page to use. This can be done in the startup code. You must also specify the starting address for **PDATA** to the linker.

Refer to “STARTUP.A51” on page 151 for more information on configuring P2 for the compact model.

Large Model

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

Memory Types

The **Cx51** compiler explicitly supports the architecture of the 8051 and its derivatives and provides access to all memory areas of the 8051. Each variable may be explicitly assigned to a specific memory space.

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

internal data memory. Place larger, less frequently used variables in external data memory.

Explicitly Declared Memory Types

You may specify where variables are stored by including a memory type specifier in the variable declaration.

The following table summarizes the available memory type specifiers.

Memory Type	Description
code	Program memory (64 KBytes); accessed by opcode <code>MOVC @A+DPTR</code> .
data	Directly addressable internal data memory; fastest access to variables (128 bytes).
idata	Indirectly addressable internal data memory; accessed across the full internal address space (256 bytes).
bdata	Bit-addressable internal data memory; supports mixed bit and byte access (16 bytes).
xdata	External data memory (64 KBytes); accessed by opcode <code>MOVX @DPTR</code> .
far	Extended RAM and ROM memory spaces (up to 16MB); accessed by user defined routines or specific chip extensions (Philips 80C51MX, Dallas 390).
pdata	Paged (256 bytes) external data memory; accessed by opcode <code>MOVX @Rn</code> .

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

Example:

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

NOTE

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

`data char x;` is equivalent to `char data x;`

Nonetheless, this feature should not be used in new programs because it may not be supported in future versions of the Cx51 compiler.

Be careful when you are using the old C51 syntax together with memory-specific pointers. In this case, the definition:

*`data char *x;` is equivalent to `char *data x;`*

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 that cannot be located in registers are also stored in the default memory area.

The default memory type is determined by the **SMALL**, **COMPACT** and **LARGE** compiler control directives. Refer to “Memory Models” on page 94 for more information.

3

Data Types

The Cx51 compiler provides you with a number of basic data types to use in your C programs. The Cx51 compiler supports the standard C data types as well as several data types that are unique to the 8051 platform. The following table lists the available data types.

Data Types	Bits	Bytes	Value Range
bit †	1		0 to 1
signed char	8	1	-128 to +127
unsigned char	8	1	0 to 255
enum	8 / 16	1 or 2	-128 to +127 or -32768 to +32767
signed short	16	2	-32768 to +32767
unsigned short	16	2	0 to 65535
signed int	16	2	-32768 to +32767
unsigned int	16	2	0 to 65535
signed long	32	4	-2147483648 to +2147483647
unsigned long	32	4	0 to 4294967295
float	32	4	±1.175494E-38 to ±3.402823E+38
sbit †	1		0 or 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 the Cx51 compiler. These data types are described in detail in the following sections.

Bit Types

The **Cx51** compiler provides a **bit** data type that may be used for variable declarations, argument lists, and function-return values. A **bit** variable is declared like other C data types. For example:

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

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

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

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

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

- Functions that disable interrupts (**#pragma disable**) and functions that are declared using an explicit register bank (**using n**) cannot return a bit value. The **Cx51** compiler generates an error message for functions of this type that attempt to return a **bit** type.
- A bit cannot be declared as a pointer. For example:

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

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

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

Bit-addressable Objects

Bit-addressable objects are objects that can be addressed as words or as bits. Only data objects that occupy the bit-addressable area of the 8051 internal memory fall into this category. The **Cx51** compiler places variables declared with the **bdata** memory type into this bit-addressable area. Furthermore, variables declared with the **bdata** memory type must be global. You may declare these variables as shown below:

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

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

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

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

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

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

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

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

Declarations involving the **sbit** type require that the base object be declared with the memory type **bdata**. The only exceptions are the variants for special function bits. Refer to “Special Function Registers” on page 101 for more information.

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

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

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

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

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

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

sbit tcpf31 = tcp.u.ml ^ 31; /* bit 31 of float */
sbit tcpml0 = tcp.ml ^ 0;
sbit tcpml7 = tcp.ml ^ 7;
```

NOTE

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

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

Special Function Registers

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

Within the 8051 family, the number and type of SFRs vary. Note that no SFR names are predefined by the **Cx51** compiler. However, declarations for SFRs are provided in include files.

The **Cx51** compiler provides you with a number of include files for various 8051 derivatives. Each file contains declarations for the SFRs available on that derivative. See “8051 Special Function Register Include Files” on page 228 for more information about include files.

The **Cx51** 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.) Classic 8051 devices support the SFR address range 0x80 to 0xFF. The Philips 80C51MX provides an additional extended SFR space with the address range 0x180 to 0x1FF.

sfr16

Many of the newer 8051 derivatives use two SFRs with consecutive addresses to specify 16-bit values. For example, the 8052 uses addresses 0xCC and 0xCD for the low and high bytes of timer/counter 2. The **Cx51** 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 8051 applications, it is often necessary to access individual bits within an SFR. The **Cx51** compiler makes this possible with the **sbit** data type which provides access to bit-addressable SFRs and other bit-addressable objects. For example:

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

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

NOTE

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

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

Variant 1: **sfr_name ^ int_constant**

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

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

Variant 2: **int_constant ^ int_constant**

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

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

Variant 3: **int_constant**

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

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

NOTE

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

*The **sbit** data type declaration may be used to access individual bits of variables declared with the **bdta** memory type specifier. Refer to “Bit-addressable Objects” on page 99 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_space] variable_name _at_ constant;
```

where:

memory_space is the memory space for the variable. If missing from the declaration, the default memory space is used. Refer to “Memory Models” on page 94 for more information about the default memory space.

type is the variable type.

variable_name is the variable name.

constant is the address at which to locate the variable.

The absolute address following `_at_` must conform to the physical boundaries of the memory space for the variable. The Cx51 compiler checks for invalid address specifications.

NOTE

*If you use the `_at_` keyword to declare a variable that accesses an XDATA peripheral, you may require the **volatile** keyword to ensure that the C compiler does not optimize out necessary memory accesses.*

The following restrictions apply to absolute variable location:

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

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

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

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

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

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

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

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

Pointers

The Cx51 compiler supports the declaration of variable pointers using the * character. Cx51 pointers can be used to perform all operations available in standard C. However, because of the unique architecture of the 8051 and its derivatives, the Cx51 compiler provides two different types of pointers: generic pointers and memory-specific pointers. Each of these pointer types and pointer conversion methods are discussed in the following sections.

Generic Pointers

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

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

Generic pointers are always stored using three bytes. The first byte is the memory type, the second is the high-order byte of the offset, and the third is the low-order byte of the offset. Generic pointers may be used to access any variable regardless of its location in 8051 memory space. Many of the Cx51 library routines use these pointer types for this reason. By using these generic pointers, a function can access data regardless of the memory in which it is stored.

NOTE

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

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

```
stmt level  source
  1      char *c_ptr;      /* char ptr */
  2      int *i_ptr;       /* int ptr */
  3      long *l_ptr;      /* long ptr */
```

```

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

```

ASSEMBLY LISTING OF GENERATED OBJECT CODE

```

; FUNCTION main (BEGIN)
; SOURCE LINE # 5
; SOURCE LINE # 6
; SOURCE LINE # 20
0000 750000 R   MOV     c_ptr,#00H
0003 750000 R   MOV     c_ptr+01H,#HIGH dj
0006 750000 R   MOV     c_ptr+02H,#LOW dj
; SOURCE LINE # 21
0009 750000 R   MOV     i_ptr,#00H
000C 750000 R   MOV     i_ptr+01H,#HIGH dk
000F 750000 R   MOV     i_ptr+02H,#LOW dk
; SOURCE LINE # 22
0012 750000 R   MOV     l_ptr,#00H
0015 750000 R   MOV     l_ptr+01H,#HIGH dl
0018 750000 R   MOV     l_ptr+02H,#LOW dl
; SOURCE LINE # 24
001B 750001 R   MOV     c_ptr,#01H
001E 750000 R   MOV     c_ptr+01H,#HIGH xj
0021 750000 R   MOV     c_ptr+02H,#LOW xj
; SOURCE LINE # 25
0024 750001 R   MOV     i_ptr,#01H
0027 750000 R   MOV     i_ptr+01H,#HIGH xk
002A 750000 R   MOV     i_ptr+02H,#LOW xk
; SOURCE LINE # 26
002D 750001 R   MOV     l_ptr,#01H
0030 750000 R   MOV     l_ptr+01H,#HIGH xl
0033 750000 R   MOV     l_ptr+02H,#LOW xl
; SOURCE LINE # 28
0036 7500FF R   MOV     c_ptr,#0FFH

```

```

0039 750000 R   MOV    c_ptr+01H,#HIGH cj
003C 750000 R   MOV    c_ptr+02H,#LOW  cj
           ; SOURCE LINE # 29
003F 7500FF R   MOV    i_ptr,#0FFH
0042 750000 R   MOV    i_ptr+01H,#HIGH ck
0045 750000 R   MOV    i_ptr+02H,#LOW  ck
           ; SOURCE LINE # 30
0048 7500FF R   MOV    l_ptr,#0FFH
004B 750000 R   MOV    l_ptr+01H,#HIGH cl
004E 750000 R   MOV    l_ptr+02H,#LOW  cl
           ; SOURCE LINE # 31
0051 22        RET
           ; FUNCTION main (END)

```

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

```

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

```

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

Memory-specific Pointers

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

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

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

NOTE

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

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

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

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

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

```

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

```

ASSEMBLY LISTING OF GENERATED OBJECT CODE

```

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

```

Pointer Conversions

The **Cx51** compiler may convert between memory-specific pointers and generic pointers. Pointer conversions can be forced by explicit program code using type casts or can be coerced by the compiler implicitly.

The **Cx51** 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 functions such as **printf**, **sprintf**, and **gets** which use generic pointers as arguments. For example:

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

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

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

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

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

NOTE

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

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

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

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

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

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

```

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

```

ASSEMBLY LISTING OF GENERATED OBJECT CODE

```

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

```

Abstract Pointers

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

Abstract pointer types are described here using code examples with the following variables.

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

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

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

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

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

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

The following example casts a pointer to a **char** in **xdata** to a pointer to a **char** in **idata**. Since **xdata** pointers occupy 2 bytes and **idata** pointers occupy 1 byte, this pointer conversion may not yield the desired results since the upper byte of the **xdata** pointer is ignored. Refer to “Pointer Conversions” on page 111 for more information about converting between different pointer types.

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

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

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

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

Source	<code>i = ((int (code *) (void)) 0xFF00) ();</code>			
Object	0000 12FF00		LCALL	0FF00H
	0003 8E00	R	MOV	i, R6
	0005 8F00	R	MOV	i+01H, R7

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

Source	<code>c = *((char code *) 0x8000);</code>			
Object	0000 908000		MOV	DPTR, #08000H
	0003 E4		CLR	A
	0004 93		MOVC	A, @A+DPTR
	0005 F500	R	MOV	c, A

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

Source	<code>c += *((char xdata *) 0xFF00);</code>			
Object	0000 90FF00		MOV	DPTR, #0FF00H
	0003 E0		MOVX	A, @DPTR
	0004 2500	R	ADD	A, c
	0006 F500	R	MOV	c, A

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

Source	<code>c += *((char idata *) 0xF0);</code>			
Object	0000 78F0		MOV	R0, #0F0H
	0002 E6		MOV	A, @R0
	0003 2500	R	ADD	A, c
	0005 F500	R	MOV	c, A

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

Source	c += *((char pdata *) 0xE8);		
Object	0000 78E8	MOV	R0, #0E8H
	0002 E2	MOVX	A, @R0
	0003 2500	R ADD	A, c
	0005 F500	R MOV	c, A

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

Source	i = *((int code *) 0x2100);		
Object	0000 902100	MOV	DPTR, #02100H
	0003 E4	CLR	A
	0004 93	MOVC	A, @A+DPTR
	0005 FE	MOV	R6, A
	0006 7401	MOV	A, #01H
	0008 93	MOVC	A, @A+DPTR
	0009 8E00	R MOV	i, R6
	000B F500	R MOV	i+01H, A

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

Source	px = *((char xdata * xdata *) 0x4000);		
Object	0000 904000	MOV	DPTR, #04000H
	0003 E0	MOVX	A, @DPTR
	0004 FE	MOV	R6, A
	0005 A3	INC	DPTR
	0006 E0	MOVX	A, @DPTR
	0007 8E00	R MOV	px, R6
	0009 F500	R MOV	px+01H, A

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

Source	px = ((char xdata * xdata *) 0x4000) [0];		
Object	0000 904000	MOV	DPTR, #04000H
	0003 E0	MOVX	A, @DPTR
	0004 FE	MOV	R6, A
	0005 A3	INC	DPTR
	0006 E0	MOVX	A, @DPTR
	0007 8E00	R MOV	px, R6
	0009 F500	R MOV	px+01H, A

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

Source	px = ((char xdata * xdata *) 0x4000) [1];		
Object	0000 904002	MOV	DPTR, #04002H
	0003 E0	MOVX	A, @DPTR
	0004 FE	MOV	R6, A
	0005 A3	INC	DPTR
	0006 E0	MOVX	A, @DPTR
	0007 8E00	R MOV	px, R6
	0009 F500	R MOV	px+01H, A

Function Declarations

The **Cx51** compiler provides a number of extensions for standard C function declarations. These extensions allow you to:

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

You may include these extensions or attributes (many of which may be combined) in the function declaration. Use the following standard format for your **Cx51** function declarations.

```
[return_type] funcname ([args])                [ {small | compact | large} ]
                                                [reentrant] [interrupt n] [using n]
```

where:

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

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

Function Parameters and the Stack

The stack pointer on the classic 8051 accesses internal data memory only. The **Cx51** compiler locates the stack area immediately following all variables in the internal data memory. The stack pointer accesses internal memory indirectly and can use all of the internal data memory up to the 0xFF limit.

The total stack space of the classic 8051 is limited: only 256 bytes maximum. Rather than consume stack space with function parameters or arguments, The **Cx51** compiler assigns a fixed memory location for each function parameter. When a function is called, the caller must copy the arguments into the assigned memory locations before transferring control to the desired function. The function then extracts its parameters, as needed, from these fixed memory locations. Only the return address is stored on the stack during this process. Interrupt functions require more stack space because they must switch register banks and save the values of a few registers on the stack.

NOTE

*The **Cx51** compiler uses extended stack areas that are available in some enhanced 8051 variants. In this way the stack space can be increased to several KiloBytes.*

By default, the **Cx51** compiler passes up to three function arguments in registers. This enhances speed performance. For more information, refer to “Passing Parameters in Registers” on page 120.

NOTE

Some 8051 derivatives provide only 64 bytes of on-chip data memory; most devices have just 256 bytes. Take this into consideration when determining which memory model to use, because the amount of on-chip data and idata memory used directly affects the amount of stack space.

Passing Parameters in Registers

The **Cx51** compiler allows up to three function arguments to be passed in CPU registers. This mechanism significantly improves system performance as arguments do not have to be written to and read from memory. Argument or parameter passing can be controlled by the **REGPARMS** and **NOREGPARMS** control directives defined in the previous chapter.

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

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

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

Function Return Values

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

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

NOTE

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

Specifying the Memory Model for a Function

A function's arguments and local variables are stored in the default memory space specified by the memory model. Refer to "Memory Models" on page 94 for more information.

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

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

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

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

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

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

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

Specifying the Register Bank for a Function

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

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

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

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

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

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

```

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

ASSEMBLY LISTING OF GENERATED OBJECT CODE

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

```

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

NOTE

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.

The **using** attribute is most useful in **interrupt** functions. Usually a different register bank is specified for each interrupt priority level. Therefore, you could assign one register bank for all non-interrupt code, a second register bank for the high-level interrupt, and a third register bank for the low-level interrupt.

Register Bank Access

The **Cx51** compiler defines the default register bank in a function. The **REGISTERBANK** control directive specifies which default register bank to use for all functions in a source file. This directive, however, does not generate code to switch the register bank.

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

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

By default, the **Cx51** compiler generates code that accesses the registers R0—R7 using absolute addresses. This is done for maximum performance. Absolute register accesses are controlled by the **AREGS** and **NOAREGS** control directives.

Functions which employ absolute register accesses must not be called from another function that uses a different register bank. Doing so causes unpredictable results because the called function assumes that a different register bank is selected.

To make a function insensitive to the current register bank, the function must be compiled using the **NOAREGS** control directive. This would be useful for a function that was called from the main program and also from an interrupt function that uses a different register bank.

NOTE

*The **Cx51** compiler does not and cannot detect a register bank mismatch between functions. Therefore, make sure that functions using alternate register banks call only other functions that do not assume a default register bank.*

Refer to “Chapter 2. Compiling with the Cx51” on page 17 for more information regarding the **REGISTERBANK**, **AREGS**, and **NOAREGS** directives.

Interrupt Functions

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

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

As 8051 vendors create new parts, more interrupts are added. The **Cx51** compiler supports interrupt functions for 32 interrupts (0-31). Use the interrupt vector address in the following table to determine the interrupt number.

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

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

The **interrupt** function attribute, when included in a declaration, specifies that the associated function is an interrupt function. For example:

```
unsigned int  interruptcnt;
unsigned char second;

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

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

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

In addition, the **Cx51** compiler generates the interrupt vector automatically.

The following sample program demonstrates 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 used to select a register bank different from that of the non-interrupt program code. However, because no working registers are needed in this function, the code generated to switch the register bank is eliminated by the optimizer.

```

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

```

ASSEMBLY LISTING OF GENERATED OBJECT CODE

```

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

```

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

The following rules apply to interrupt functions.

- No function arguments may be specified for an interrupt function. The compiler emits an error message if an interrupt function is declared with any arguments.
- Interrupt function declarations may not include a return value. They must be declared as void (see the above examples). The compiler emits an error message if any attempt is made to define a return value for the interrupt function. The implicit **int** return value, however, is ignored by the compiler.
- The compiler recognizes direct calls to interrupt functions and rejects them. It is pointless to call interrupt procedures directly, because exiting the procedure causes execution of the **RETI** instruction which affects the hardware interrupt system of the 8051 chip. Because no interrupt request on the part of the hardware existed, the effect of this instruction is indeterminate and usually fatal. Do not call an interrupt function indirectly through a function pointer.
- The compiler generates an interrupt vector for each interrupt function. The code generated for the vector is a jump to the beginning of the interrupt function. Generation of interrupt vectors can be suppressed by including the **NOINTVECTOR** control directive in the **Cx51** command line. In this case, you must provide interrupt vectors from separate assembly modules. Refer to the **INTVECTOR** and **INTERVAL** control directives for more information about the interrupt vector table.
- The **Cx51** compiler allows **interrupt** numbers within the 0-31 range. Refer to your 8051 derivative document to determine which interrupts are available.
- Functions called from an interrupt procedure must function with the same register bank as the interrupt procedure. When the **NOAREGS** directive is not explicitly specified, the compiler may generate absolute register accesses using the register bank selected (by the **using** attribute or by the **REGISTERBANK** control) for that function. Unpredictable results may occur when a function assumes a register bank other than the one currently selected. Refer to “Register Bank Access” on page 124 for more information.

Reentrant Functions

A reentrant function can be shared by several processes at the same time. When a reentrant function is executing, another process can interrupt the execution and then begin to execute that same reentrant function. Normally, functions in the **Cx51** compiler cannot be called recursively or in a fashion which causes reentrancy. The reason for this limitation is that function arguments and local variables are stored in fixed memory locations. The **reentrant** function attribute allows you to declare functions that may be reentrant and, therefore, may be called recursively. For example:

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

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

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

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

Reentrant functions use the default memory model to determine which memory space to use for the reentrant stack. You may specify (with the **small**, **compact**, and **large** function attributes) which memory model to use for a function. Refer to “Specifying the Memory Model for a Function” on page 121 for more information about memory models and function declarations.

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

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

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

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

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

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

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

The simulated stack and stack pointers are declared and initialized in the **Cx51** startup code in **STARTUP.A51** which can be found in the **LIB** subdirectory. You must modify the startup code to specify which simulated stack(s) to initialize in order to use reentrant functions. You can also modify the starting address for the top of the simulated stack(s) in the startup code. Refer to “STARTUP.A51” on page 151 for more information on reentrant function stack areas.

Alien Function (PL/M-51 Interface)

You may call routines written in PL/M-51 from your C programs to access PL/M-51 routines from C, declare them external with the **alien** function type specifier. For example:

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

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

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

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

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

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

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

Real-time Function Tasks

The **Cx51** compiler provides support for the **RTX51 Full** and **RTX51 Tiny** real-time multitasking operating systems through use of the `_task_` and `_priority_` keywords. The `_task_` keyword defines a function as a real-time task. The `_priority_` keyword specifies the priority for the task.

For example:

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

where:

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

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

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

3

Chapter 4. Preprocessor

The preprocessor built into the **Cx51** compiler handles directives found in the source file. The **Cx51** compiler supports all of the ANSI Standard C directives. This chapter gives a brief overview of the preprocessor.

Directives

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

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

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

Directive	Description
define	Defines a preprocessor macro or constant.
elif	Initiates an alternative branch of the if condition, when the previous if, ifdef, ifndef, or elif branch was not taken.
else	Initiates an alternative branch when the previous if, ifdef, or ifndef branch was not taken.
endif	Ends an if , ifdef , ifndef , elif , or else block.
error	Outputs an error message defined by the user. This directive instructs the compiler to emit the specified error message.
ifdef	Evaluates an expression for conditional compilation. The argument to be evaluated is the name of a definition.
ifndef	Same as ifdef but the evaluation succeeds if the definition is not defined.
if	Evaluates an expression for conditional compilation.
include	Reads source text from an external file. The notation sequence determines the search sequence of the included files. Cx51 searches for include files specified with less-than/greater-than symbols ('<' '>') in the include file directory. Cx51 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 directives that may be included on the C51 command line. Pragmas may contain the same 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)
```

This example 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);
```

This example 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 **Cx51** 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
<code>__C51__</code>	Version number of the C51 compiler (for example, 610 for version 6.10).
<code>__CX51__</code>	Version number of the CX51 compiler (for example, 610 for version 6.10).
<code>__DATE__</code>	Date when the compilation was started in ANSI format (month dd yyyy).
<code>__DATE2__</code>	Date when the compilation in short form (mm/dd/yy).
<code>__FILE__</code>	Name of the file being compiled.
<code>__LINE__</code>	Current line number in the file being compiled.
<code>__MODEL__</code>	Memory model selected (0 for SMALL , 1 for COMPACT , 2 for LARGE).
<code>__TIME__</code>	Time when the compilation was started.
<code>__STDC__</code>	Defined to 1 to indicate full conformance with the ANSI C Standard.

Chapter 5. 8051 Derivatives

A number of 8051 devices provide enhanced performance while remaining compatible with the 8051 core. These derivatives provide additional data pointers, very fast math operations, extended or reduced instruction sets.

The **Cx51** compiler directly supports the enhanced features of the following 8051-based microcontrollers:

- Analog Devices ADuC MicroConverter B2 series (2 data pointers and extended stack space).
- Atmel 89x8252 and variants (2 data pointers).
- Dallas 80C320, 80C420, 80C520, 80C530, 80C550 and variants (2 data pointers).
- Dallas 80C390, 5240 and variants (contiguous address mode, extended stack space, and arithmetic accelerator).
- Infineon C517, C517A, C509, and variants (high-speed 32-bit and 16-bit binary arithmetic operations, 8 data pointers).
- Philips 8xC750, 8xC751, and 8xC752 (maximum code space of 2 KBytes, no **LCALL** or **LJMP** instructions, 64 bytes internal, no external data memory).
- Philips 80C51MX architecture with extended instructions and memory space.
- Philips and AtmelWM support on several device variants 2 data pointers.

The **Cx51** compiler provides you with support for these CPUs through the use of special libraries, library routines, or additional directives that enable the **Cx51** compiler to generate object code that takes advantage of the device enhancements mentioned above. Refer to “Chapter 2. Compiling with the Cx51” on page 17 for more information about these additional directives.

Analog Devices MicroConverter B2 Series

The Analog Devices B2 series of MicroConverters provide 2 data pointers which can be used for memory access. Using multiple data pointers can improve the speed of library functions like **memcpy**, **memmove**, **memcmp**, **strcpy**, and **strcmp**.

The **MODAB2** directive instructs the the **Cx51** compiler compiler to generate code that uses both data pointers in your program.

The **Cx51** compiler uses at least one data pointer in an interrupt function. If an interrupt function is compiled using the **MODAB2** directive, both data pointers are saved on the stack. This happens even if the interrupt function uses only one data pointer.

To conserve stack space, you may compile interrupt functions with the **NOMODAB2** directive. The **Cx51** compiler does not use the second data pointer when this directive is used.

These devices offer also an extended stack space that is configured in startup file **START_AD.A51**.

Atmel 89x8252 and Variants

The Atmel 89x8252 and variants provide 2 data pointers which can be used for memory access. Using multiple data pointers can improve the speed of library functions like **memcpy**, **memmove**, **memcpy**, **strcpy**, and **strcmp**.

The **MODA2** directive instructs the the **Cx51** compiler compiler to generate code that uses both data pointers in your program.

The **Cx51** compiler uses at least one data pointer in an interrupt function. If an interrupt function is compiled using the **MODA2** directive, both data pointers are saved on the stack. This happens even if the interrupt function uses only one data pointer.

To conserve stack space, you may compile interrupt functions with the **NOMODA2** directive. The **Cx51** compiler does not use the second data pointer when this directive is used.

Dallas 80C320, 420, 520, and 530

The Dallas Semiconductor 80C320, 80C420, 80C520, and 80C530 provide 2 data pointers which may be used for memory access. Using multiple data pointers can improve the speed of library functions like **memcpy**, **memmove**, **memcpy**, **strcpy**, and **strcmp**.

The **MODDP2** directive instructs the **Cx51** compiler to generate code that uses both data pointers in your program.

The **Cx51** compiler uses at least one data pointer in an interrupt function. If an interrupt function is compiled using the **MODDP2** directive, both data pointers are saved on the stack—even if the interrupt function uses only one data pointer.

To conserve stack space, you may compile interrupt functions with the **NOMODDP2** directive. The **Cx51** compiler does not use the second data pointer when this directive is specified.

The DS80C420 provides auto toggle, decrement, and auto increment features for the dual data pointers. The library `\KEIL\C51\LIB\C51DS2A.LIB` contains accelerated versions of the **memcpy**, **memmove**, **memcpy**, **strcpy**, and **strcmp** functions that use these features. Add this library to your project when you use the dual DPTR feature on this device.

The DS80C550, DS80C390, and DS5240 provide auto toggle and decrement features for the dual data pointers. The library `\KEIL\C51\LIB\C51DS2T.LIB` contains accelerated versions of the **memcpy**, **memmove**, **memcpy**, **strcpy**, and **strcmp** functions that use these features. Add this library to your project when you use the dual DPTR feature on these devices.

Dallas 80C390, 80C400, 5240, and Variants

The Dallas Semiconductor 80C390, 80C400, 5240, and variants provide additional CPU modes that are fully supported by the Keil compiler.

Contiguous mode allows you to create large programs that exceed the classic 8051's 64K limit. The **ROM(D512K)** and **ROM(D16M)** directives instruct the **Cx51** compiler to generate code for the contiguous mode. The **far** memory type is used to access variables and constants using 24-bit DPTR addressing mode (in contiguous mode).

NOTE

The contiguous mode requires the extended LX51 linker/locater and the extended AX51 macro assembler that are available only in the PK51 Professional Developers Kit.

In addition to the extended address space, the DS80C390, DS80C400, and DS5240 provide auto toggle and decrement features for the dual data pointers. The `\KEIL\C51\LIB\C51DS2T.LIB` library contains accelerated versions of the **memcpy**, **memmove**, **memcpy**, **strcpy**, and **strcmp** functions that use these features. For non-contiguous mode (classic 8051 mode) applications, you must add this library to your project to use the dual DPTR of these devices. The contiguous mode C library already contains the library routines for the auto toggle and decrement features.

The DS80C390, DS80C400, and DS5240 offer an extended stack space that is configured in the **START390.A51** startup file.

Arithmetic Accelerator

The **Cx51** compiler uses the 32-bit and 16-bit arithmetic operations of the DS80C390, DS80C400 and DS5240 to improve performance of a number of math-intensive operations. C language programs execute considerably faster when using either of these CPUs.

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

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

Infineon C517, C509, 80C537, and Variants

The Infineon C517, C517A, and C509 perform high-speed 32-bit and 16-bit arithmetic operations which improve of many **int**, **long**, and **float** operations.

The C517, C517A, C509, and C515C provide 8 data pointers that may be used to increase the speed of memory to memory operations.

The **MOD517** directive instructs the **Cx51** compiler to generate code that utilizes these advanced features.

Data Pointers

The Infineon C515C, C517, C517A, and C509 provide 8 data pointers which may be used to speed-up memory access. Using multiple data pointers can improve the execution of library functions such as: **memcpy**, **memmove**, **memcmp**, **strcpy**, and **strcmp**. The 8 data pointers of the C515C, C517, C517 and C509 may also reduce the stack load of interrupt functions.

The **Cx51** compiler uses only 2 of the 8 data pointers at a time. In order to keep the stack load in the interrupt routines low, **Cx51** switches to 2 unused data pointers when switching the register bank. The contents of the register **DPSSEL** are saved on the stack and a new pair of data pointers is selected. Saving the data pointers on the stack is no longer required.

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

High-speed Arithmetic

The **Cx51** compiler uses the 32-bit and 16-bit arithmetic operations of the C517, C517A, and C509 to improve performance of a number of math-intensive operations. C language programs execute considerably faster when using either of these CPUs.

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

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

Library Routines

The extra features of the C517, C517A, and C509 are used in several library routines to enhance performance. These routines are listed below and are described in detail in “Chapter 8. Library Reference” on page 209.

acos517	log10517	sqrt517
asin517	log517	sscanf517
atan517	printf517	strtod517
atof517	scanf517	tan517
cos517	sin517	
exp517	sprintf517	

Philips 8xC750, 8xC751, and 8xC752

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

- A special library, **80C751.LIB**, which does not use these instructions is necessary for these devices.
- The **Cx51** compiler must be set to avoid using **LJMP** and **LCALL** instructions. This is accomplished using the **ROM(SMALL)** directive.

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

- Stream functions such as **printf** and **putchar** may not be used. These functions are usually not necessary for this chip because it is only equipped with a maximum of 2 KBytes and has no serial interface.
- Floating-point operations may not be used. Only operations using **char**, **unsigned char**, **int**, **unsigned int**, **long**, **unsigned long**, and **bit** data types are allowed.
- The **Cx51** compiler must be invoked with the **ROM(SMALL)** directive. This control statement instructs the C51 compiler to use only **AJMP** and **ACALL** instructions.
- The library file **80C751.LIB** must be included in the input module list of the linker. For example:

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

- A special startup module, **START751.A51**, is required. This file contains startup code that is comparable to that found in **STARTUP.A51**, but contains no **LJMP** or **LCALL** instructions. Refer to “Customization Files” on page 150 for more information.

Philips 80C51MX Architecture

The Philips 80C51MX architecture provides an extended instruction set and extended addressing modes to support up to 16MB memory space. The universal pointer registers and the related instructions give you hardware support for generic pointers. You may use the **far** memory type to place variables anywhere in the extended memory space. Programming examples for the Philips 80C51MX architecture are found in the folder `C51\EXAMPLES\PHILIPS 80C51MX`.

The Philips 80C51MX architecture is supported with the extended CX51 compiler, LX51 linker/locator, and AX51 macro assembler. These additional components are available in the PK51 Professional Developers Kit.

Philips and Atmel WM Dual DPTR

Philips Semiconductors and Atmel Wireless and Microcontrollers provide several compatible 8051 variants with dual data pointers. Using multiple data pointers can improve the speed of library functions like **memcpy**, **memmove**, **memcmp**, **strcpy**, and **strcmp**.

The **MODP2** directive instructs the **Cx51** compiler to generate code that uses both data pointers in your program.

The **Cx51** compiler uses at least one data pointer in an interrupt function. If an interrupt function is compiled using the **MODP2** directive, both data pointers are saved on the stack - this happens even if the interrupt function uses only one data pointer.

To conserve stack space, you may compile interrupt functions with the **NOMODP2** directive to prevent the **Cx51** compiler from using the second data pointer.

Chapter 6. Advanced Programming Techniques

This chapter describes advanced programming information that experienced software engineers will find invaluable. Knowledge of most of these topics is not necessary to successfully create an embedded 8051 target program using the **Cx51** compiler. However, the following sections provide insight into how many non-standard procedures can be accomplished (for example, interfacing to PL/M-51).

This chapter discusses the following topics:

- Files you may alter to customize the startup procedures
- Files you may alter to customize run-time execution of library routines
- The conventions the **Cx51** compiler uses to name code and data segments
- How to interface **Cx51** functions to assembly and PL/M-51 routines
- Data storage formats for the different **Cx51** data types
- Different optimizing features of the **Cx51** optimizing compiler

Customization Files

The **Cx51** compiler provides a number of source files you can modify to adapt your target program to a specific hardware platform. These files contain:

- Code that is executed upon startup (**STARTUP.A51**)
- Code that is used to initialize static variables (**INIT.A51**)
- Code that is used to perform low-level stream I/O
- Code for memory allocation

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

You may customize these files to adjust them to your requirements. If you are working with the μ Vision2 IDE, we recommend that you copy the customization file in your project folder to make modifications. The modified version of the file can be added the same way as other source files to your project.

When you are working with command-line tools, you must include the object files of the modified customization files in the linker command line. The following example shows you how to include custom replacement files for **STARTUP.A51** and **PUTCHAR.C**:

```
Lx51 MYMODUL1.OBJ, MYMODUL2.OBJ, STARTUP.OBJ, PUTCHAR.OBJ
```

The file **XBANKING.A51** allows you to change the configuration of the extended far memory access routines.

STARTUP.A51

The `STARTUP.A51` file contains the startup code for a **Cx51** target program. This source file is located in the `LIB` directory. Include a copy of this file in each 8051 project that needs custom startup code.

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

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

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

Constant Name	Description
<code>IDATALEN</code>	Indicates the number of bytes of idata that are to be initialized to 0. The default is 80h because most 8051 derivatives contain at least 128 bytes of internal data memory. Use a value of 100h for the 8052 and other derivatives that have 256 bytes of internal data memory.
<code>XDATASTART</code>	Specifies the xdata address to start initializing to 0.
<code>XDATALEN</code>	Indicates the number of bytes of xdata to be initialized to 0. The default is 0.
<code>PDATASTART</code>	Specifies the pdata address to start initializing to 0.
<code>PDATALEN</code>	Indicates the number of bytes of pdata to be initialized to 0. The default is 0.
<code>IBPSTACK</code>	Indicates whether or not the small model reentrant stack pointer (<code>?C_IBP</code>) 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.
<code>IBPSTACKTOP</code>	Specifies the top start address of the small model reentrant stack area. The default is 0xFF in idata memory. The Cx51 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.

Constant Name	Description
XBPSTACK	Indicates whether or not the large model reentrant stack pointer (<code>?C_XBP</code>) 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 Cx51 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.
PBPSTACK	Indicates whether the compact model reentrant stack pointer (<code>?C_PBP</code>) should be initialized. A value of 1 causes this pointer to be initialized. A value of 0 prevents initialization of this pointer. The default is 0.
PBPSTACKTOP	Specifies the top start address of the compact model reentrant stack area. The default is 0xFF in pdata memory. The Cx51 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.
PPAGEENABLE	Enables (a value of 1) or disables (a value of 0) the initialization of port 2 of the 8051 device. The default is 0. The addressing of port 2 allows the mapping of 256 byte variable memory in any arbitrary xdata page.
PPAGE	Specifies the value to write to Port 2 of the 8051 for pdata memory access. This value represents the xdata memory page to use for pdata. This is the upper 8 bits of the absolute address range to use for pdata. For example, if the pdata area begins at address 1000h (page 10h) in the xdata memory, PPAGEENABLE should be set to 1, and PPAGE should be set to 10h. The BL51 Linker/Locator must contain a value between 1000h and 10FFh in the PDATA directive. For example: BL51 <input modules> PDATA (1050H) Neither BL51 nor Cx51 checks to see if the PDATA directive and the PPAGE assembler constant are correctly specified. You must ensure that these parameters contain suitable values.

6

There are numerous devices in the 8051 family that require special startup code. The following list provides an overview of the various startup versions:

Startup File	Description
STARTUP.A51	Standard startup code for classic 8051 devices.
START_AD.A51	Startup code for Analog Devices MicroConverters B2 series variants.
STARTLPC.A51	Startup code for Philips LPC variants.
START390.A51	Startup code for Dallas 80C390, 80C400, 5240 contiguous mode.
START_MX.A51	Startup code for Philips 80C51MX architecture.
START751.A51	Startup code for Philips 80C75x variants.

INIT.A51

The `INIT.A51` file contains the initialization routine for variables that were explicitly initialized. If your system is equipped with a watchdog timer, you can integrate a watchdog refresh into the initialization code using the `watchdog` macro. This macro needs to be defined only if the initialization takes longer than the watchdog cycle time. For example, if you are using an Infineon C515, the macro could be defined as follows:

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

The `INIT_TNY.A51` file is a reduced version of `INIT.A51` that may be used for projects that do not contain XDATA memory. You should use this file when you write code for single-chip devices, like the Philips LPC series, that contain variable initializations in data space.

XBANKING.A51

This file provides routines for **far** (HDATA) and **const far** (HCONST) memory type support. The extended LX51 linker/locator manages the extended address spaces HDATA and HCONST that are addressed with **far** and **const far**. The **Cx51** Compiler uses a 3-byte generic pointer to access these memory areas. Variables defined with the **far** memory type are placed in the memory class **HDATA**. Variables defined with **const far** get the memory class **HCONST**. The LX51 linker/locator allows you to locate these memory classes in the physical 16MB code or 16MB xdata spaces. To use **far** memory with the C51 Compiler for classic 8051 devices you must use the “VARBANKING” directive described on page 84.

The memory types **far** and **const far** provide support for the large code/xdata spaces of new 8051 devices. If the CPU you are using provides an extended 24-bit DPTR register, you may adapt the default version of the file **XBANKING.A51** and define the symbols listed in the following table.

Constant Name	Description
?C?XPAGE1SFR	SFR address of DPTR page register that contains DPTR bit 16-23.
?C?XPAGE1RST	Reset value of the ?C?XPAGE1SFR to address the X:0 region. This setting used by the C51 compiler when you are using the VARBANKING(1) directive. With VARBANKING(1) the C51 compiler saves the ?C?XPAGE1SFR at the beginning of interrupt functions and sets this register to the ?C?XPAGE1RST value

6

The **far** memory type allows you to address special memory areas like EEPROM space or strings in code banking ROM. Your application accesses these memory areas as if they are a part of the standard 8051 memory space. Example programs in the folder **C51\EXAMPLES\FARMEMORY** show how to use the C51 **far** memory type on classic 8051 devices. If an example that fulfills your requirements is not provided, you may adapt the access routines listed in the table below.

Access Routine	Description
?C?CLDXPTR, ?C?CSTXPTR	load/store a BYTE (char) in extended memory.
?C?ILDXPTR, ?C?ISTXPTR	load/store a WORD (int) in extended memory.
?C?PLDXPTR, ?C?PSTXPTR	load/store a 3-BYTE pointer in extended memory.
?C?LLDXPTR, ?C?LSTXPTR	load/store a DWORD (long) in extended memory.

Each access routine gets as a parameter the memory address in a 3-byte pointer representation in the CPU registers R1/R2/R3. The register R3 holds the memory type value. For classic 8051 devices, the **Cx51** compiler uses the following memory type values:

R3 Value	Memory Type	Memory Class	Address Range
0x00	data / idata	DATA / IDATA	I:0x00-I:0xFF
0x01	xdata	XDATA	X:0x0000-X:0xFFFF
0x02-0x7F	far	HDATA	X:0x010000-X:0x7E0000
0x80-0xFD	far const	HCONST	C:0x800000-C:0xFD0000 (far const is mapped into the banked memory areas)
0xFE	pdata	XDATA	one 256-byte page in XDATA memory
0xFF	code	CODE / CONST	C:0x0000-C:0xFFFF

The R3 values 0x00, 0x01, 0xFE and 0xFF are already handled within the run-time library. Only the values 0x02 - 0xFE are passed to the XPTR access routines described above. The AX51 macro assembler provides the MBYTE operator that calculates the R3 value that needs to be passed to the XPTR access function. Below is an AX51 Assembler example for using XPTR access functions:

```
MOV R1,#LOW (variable) ; gives LSB address byte of variable
MOV R1,#HIGH (variable) ; gives MSB address byte of variable
MOV R1,#MBYTE (variable) ; gives memory type byte of variable
CALL ?C?CLDXPTR ; load BYTE variable into A
```

Basic I/O Functions

The following files contain the source code for the low-level stream I/O routines. When you use the μ Vision2 IDE, you can simply add the modified versions to the project.

C Source File	Description
PUTCHAR.C	Used by all stream routines that output characters. You may adapt this routine to your individual hardware (for example, LCD or LED displays). The default version outputs characters via the serial interface. An XON/XOFF protocol is used for flow control. Linefeed characters (<code>'\n'</code>) are converted into carriage return/linefeed sequences (<code>'\r\n'</code>).
GETKEY.C	Used by all stream routines that input characters. You may adapt this routine to your individual hardware (for example, for matrix keyboards). The default version reads a character via the serial interface. No data conversions are performed.

Memory Allocation Functions

The following files contain the source code for the memory allocation routines.

C Source File	Description
CALLOC.C	Allocates memory for an array from the memory pool.
FREE.C	Returns a previously allocated memory block to the memory pool.
INIT_MEM.C	Specifies the location and size of a memory pool from which memory may be allocated using the malloc , calloc , and realloc functions.
MALLOC.C	Allocates memory from the memory pool.
REALLOC.C	Resizes a previously allocated memory block.

Optimizer

The **Cx51** compiler is an optimizing compiler. This means that the compiler takes certain steps to ensure that the code generated and output to the object file is the most efficient (smallest and/or fastest) code possible. The compiler analyzes the generated code to produce more efficient instruction sequences. This ensures that your **Cx51** compiler program runs as quickly as possible.

The **Cx51** compiler provides several different levels of optimizing. Refer to “OPTIMIZE” on page 63 for detailed information.

General Optimizations

Optimization	Description
Constant Folding	Several constant values occurring in an expression or address calculation are combined as a constant.
Jump Optimizing	Jumps are inverted or extended to the final target address when the program efficiency is thereby increased.
Dead Code Elimination	Code which cannot be reached (dead code) is removed from the program.
Register Variables	Automatic variables and function arguments are located in registers when possible. Reservation of data memory for these variables is omitted.
Parameter Passing Via Registers	A maximum of three function arguments can be passed in registers.
Global Common Subexpression Elimination	Identical subexpressions or address calculations that occur multiple times in a function are recognized and calculated only once when possible.
Reuse of Common Entry Code	When there are multiple calls to a single function, some of the setup code can be reused, thereby reducing program size.
Common Block Subroutines	Detects recurring instruction sequences and converts them into subroutines. The compiler even rearranges code to obtain larger recurring sequences.

8051-Specific Optimizations

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

Options for Code Generation

Optimization	Description
OPTIMIZE(SIZE)	Common C operations are replaced by subprograms. Program code is thereby reduced.
NOAREGS	The Cx51 compiler no longer uses absolute register access. Program code is independent of the register bank.
NOREGPARDS	Parameter passing is always performed in local data segments. The program code is compatible to earlier versions of Cx51 .

Segment Naming Conventions

Objects generated by the Cx51 compiler (program code, program data, and constant data) are stored in segments which are units of code or data memory. A segment may be relocatable or may be absolute. Each relocatable segment has a type and a name. This section describes the conventions used by the Cx51 compiler for naming these segments.

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

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

Segment Prefix	Memory Type	Description
?PR?	program	Executable program code
?CO?	code	Constant data in program memory
?BI?	bit	Bit data in internal data memory
?BA?	bdata	Bit-addressable data in internal data memory
?DT?	data	Internal data memory
?FD?	far	Far memory (RAM space)
?FC?	const far	Far memory (constant ROM space)
?ID?	idata	Indirectly-addressable internal data memory
?PD?	pdata	Paged data in external data memory
?XD?	xdata	Xdata memory (RAM space)
?XC?	const xdata	Xdata memory (constant ROM space)

Data Objects

Data objects are the variables and constants you declare in your C programs. The **Cx51** compiler generates a separate segment for each memory type for which a variable is declared. The following table lists the segment names generated for different variable data objects.

Segment Name	Description
?BA?module_name	Bit-addressable data objects
?BI?module_name	Bit objects
?CO?module_name	Constants (strings and initialized variables)
?DT?module_name	Objects declared in data
?FC?module_name	Objects declared in const far (requires OMF2 directive)
?FD?module_name	Objects declared in far (requires OMF2 directive)
?ID?module_name	Objects declared in idata
?PD?module_name	Objects declared in pdata
?XC?module_name	Objects declared in const xdata (requires OMF2 directive)
?XD?module_name	Objects declared in xdata

Program Objects

Program objects include the code generated for C program functions by the **Cx51** compiler. Each function in a source module is assigned a separate code segment using the `?PR?function_name?module_name` naming convention. For example, the function **error_check** in the file **SAMPLE.C** would result in a segment name of `?PR?ERROR_CHECK?SAMPLE`.

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

Function arguments were historically passed using fixed memory locations. This is still true for routines written in PL/M-51. However, **Cx51** can pass up to 3 function arguments in registers. Other arguments are passed using the traditional fixed memory areas. Memory space is reserved for all function arguments regardless of whether or not some of these arguments may be passed in registers. The parameter areas must be publicly known to any calling module. So, they are publicly defined using the following segment names:

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

For example, if **func1** is a function that accepts both **bit** arguments as well as arguments of other data types, the **bit** arguments are passed starting at `?FUNC1?BIT`, and all other parameters are passed starting at `?FUNC1?BYTE`. Refer to “Interfacing C Programs to Assembler” on page 163 for examples of the function argument segments.

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

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

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

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

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

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

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

Interfacing C Programs to Assembler

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

There are several reasons 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
- You may want to manipulate SFRs or memory-mapped I/O devices directly from assembly

This section describes how to write assembly routines that can be directly interfaced to C programs.

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

Function Parameters

By default, C functions pass up to three parameters in registers. The remaining parameters are passed in fixed memory locations. You may use the directive **NOREGPARDS** to disable parameter passing in registers. Parameters are passed in fixed memory locations if parameter passing in registers is disabled or if there are too many parameters to fit in registers. Functions that pass parameters in registers are flagged by the **Cx51** compiler with an underscore character ('_') prefixed to the function name at code generation time. Functions that pass parameters only in fixed memory locations are not prefixed with an underscore. Refer to “Using the SRC Directive” on page 166 for an example.

Parameter Passing in Registers

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

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

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

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

Parameter Passing in Fixed Memory Locations

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

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

Function Return Values

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

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

Using the SRC Directive

The **Cx51** compiler can create assembly source files you assemble with the A51 Assembler. These files may be useful when you want to determine the argument passing conventions between C and assembly.

To create an assembly source file, you must use the **SRC** directive with the **Cx51** compiler. For example:

```
#pragma SRC
#pragma SMALL

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

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

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

NAME      ASM1

?PR?_asmfunc1?ASM1  SEGMENT CODE
PUBLIC _asmfunc1
; #pragma SRC
; #pragma SMALL
;
; unsigned int asmfunc1 (
                RSEG  ?PR?_asmfunc1?ASM1
                USING  0
_asmfunc1:
;---- Variable 'arg?00' assigned to Register 'R6/R7' ----
                ; SOURCE LINE # 4
                ; SOURCE LINE # 6
; return (1 + arg);
                ; SOURCE LINE # 7
                MOV   A,R7
                ADD   A,#01H
                MOV   R7,A
                CLR   A
                ADDC  A,R6
                MOV   R6,A
; }
                ; SOURCE LINE # 8
?C0001:
                RET
; END OF _asmfunc1

                END
```

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

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

```

; ASM2.SRC generated from: ASM2.C

NAME      ASM2

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

                RSEG    ?DT?asmfunc1?ASM2
?asmfunc1?BYTE:
arg?00:        DS    2
; #pragma SRC
; #pragma SMALL
; #pragma NOREGPARMS
;
; unsigned int asmfunc1 (
                RSEG    ?PR?asmfunc1?ASM2
                USING  0
asmfunc1:
                ; SOURCE LINE # 5
                ; SOURCE LINE # 7
; return (1 + arg);
                ; SOURCE LINE # 8
                MOV     A,arg?00+01H
                ADD     A,#01H
                MOV     R7,A
                CLR     A
                ADDC    A,arg?00
                MOV     R6,A
; }
                ; SOURCE LINE # 9
?C0001:
                RET
; END OF asmfunc1

                END

```

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

Register Usage

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

Overlaying Segments

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

- All segment names must be created using the **Cx51** compiler segment naming conventions.
- Each assembler function with local variables must be assigned its own data segment. This data segment may be accessed by other functions only for passing parameters. Parameters must be passed in order.

Example Routines

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

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


Small Model Example

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

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

Calling a C function from assembly.

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

Assembly code for the example function:

```

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

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

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

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

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

```

Compact Model Example

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

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

Calling a C function from assembly.

```

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

```

Assembly code for the example function:

```

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

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

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

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

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

```

Large Model Example

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

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

Calling a C function from assembly.

```

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

```

Assembly code for the example function:

```

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

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

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

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

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

```

Interfacing C Programs to PL/M-51

Intel's PL/M-51 is a popular programming language that is similar to C in many ways. You can easily interface the **Cx51** compiler to routines written in PL/M-51.

- You can access PL/M-51 functions from C by declaring them with the **alien** function type specifier.
- Public variables declared in the PL/M-51 module are available to your C programs.
- The PL/M-51 compiler generates object files in the OMF-51 format.

The **Cx51** compiler can generate code using the PL/M-51 parameter passing conventions. The **alien** function type specifier is used to declare public or external functions that are compatible with PL/M-51 in any memory model. For example:

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

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

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

NOTE

*PL/M-51 does not support variable-length argument lists. Therefore, functions declared using the **alien** type specifier must have a fixed number of arguments. The ellipsis notation used for variable-length argument lists is not allowed for **alien** functions and causes the **Cx51** compiler to generate an error message.*

For example:

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

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

Data Storage Formats

This section describes the storage formats of the data types available in the **Cx51** compiler. The **Cx51** compiler offers 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	8 / 16	1 or 2	-128 to +127 or -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	$\pm 1.175494\text{E-}38$ to $\pm 3.402823\text{E+}38$
data * , idata * , pdata *	8	1	0x00 to 0xFF
code* , xdata *	16	2	0x0000 to 0xFFFF
generic pointer	24	3	Memory type (1 byte); Offset (2 bytes) 0 to 0xFFFF

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

6

Bit Variables

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

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

Scalars of type **char** are stored in a single byte (8 bits). Memory-specific pointers that reference **data**, **idata**, and **pdata** are also stored using a single byte (8 bits). If an **enum** can be represented with an 8 bit value, the enum is also stored in a single byte.

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

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

Address	+0	+1
Contents	0x12	0x34

Signed and Unsigned Long Integers

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

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

Generic and Far Pointers

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

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

Depending on the compiler version that you are using, the memory type byte has the following values:

Memory Type	idata / data / bdata	Xdata	pdata	code
C51 Compiler (8051 devices)	0x00	0x01	0xFE	0xFF
CX51 Compiler (Philips 80C51MX)	0x7F	0x00	0x00	0x80

The Philips 80C51MX architecture supports new CPU instructions that operate on a universal pointer. Universal pointers are identical with Cx51 generic pointers.

The format of the generic pointers is also used for pointers with the memory type **far**. Therefore, any other memory type values are used to address **far** memory space.

The following example shows the memory storage of a generic pointer (on the C51 compiler) that references address 0x1234 in the **xdata** memory area.

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

Floating-point Numbers

Scalars of type **float** are stored using four bytes (32-bits). The format used follows the IEEE-754 standard.

A floating-point number is expressed as the product of two parts: the mantissa and a power of two. For example:

$$\pm \text{mantissa} \times 2^{\text{exponent}}$$

The mantissa represents the actual binary digits of the floating-point number.

The power of two is represented by the exponent. The stored form of the exponent is an 8-bit value from 0 to 255. The actual value of the exponent is calculated by subtracting 127 from the stored value (0 to 255) giving a range of –127 to +128.

The mantissa is a 24-bit value (representing about seven decimal digits) whose most significant bit (MSB) is always 1 and is, therefore, not stored. There is also a sign bit that indicates whether the floating-point number is positive or negative.

Floating-point numbers are stored on byte boundaries in 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 exponent with an offset of 127.
- M** is the 24-bit mantissa (stored in 23 bits).

Zero is a special value denoted with an exponent field of 0 and a mantissa of 0.

The floating-point number -12.5 is stored as a hexadecimal value of 0xC1480000. In memory, this value appears as follows:

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

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

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

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

From this illustration, you can determine the following information:

- The sign bit is 1, indicating a negative number.
- The exponent value is 10000010 binary or 130 decimal. Subtracting 127 from 130 leaves 3, which is the actual exponent.
- The mantissa appears as the following binary number:
1001000000000000000000

There is an understood binary point at the left of the mantissa that is always preceded by a 1. This digit is omitted from the stored form of the floating-point number. Adding 1 and the binary point to the beginning of the mantissa gives the following value:

```
1.100100000000000000000000
```

Next, 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 three, the mantissa is adjusted as follows:

```
1100.1000000000000000000000
```

The result is a binary floating-point number. Binary digits 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 is 12.

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

The sum of these values is 12.5. Because the sign bit was set, this number should be negative. So, the hexadecimal value 0xC1480000 is -12.5.

Floating-point Errors

The 8051 does not contain an interrupt vector to trap floating-point errors; therefore, your software must appropriately respond to these error conditions.

In addition to the normal floating-point values, a floating-point number may contain a binary error value. These values are defined as a part of the IEEE standard and are used whenever an error occurs during normal processing of floating-point operations. Your code should check for possible arithmetic errors at the end of each floating-point operation.

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

NOTE

The *Cx51* library function `_chkfloat_` lets you quickly check floating-point status.

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

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

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

For example:

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

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

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

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

Accessing Absolute Memory Locations

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

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

Each of these three methods is described below.

Absolute Memory Access Macros

First, you may use the absolute memory access macros provided as part of the **Cx51** library. Use the following macros to directly access the memory areas of the 8051.

CBYTE	FCVAR	CWORD
DBYTE	FVAR	DWORD
FARRAY	PBYTE	PWORD
FCARRAY	XBYTE	XWORD

Refer to “Absolute Memory Access Macros” on page 212 for definitions of these macros.

Linker Location Controls

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

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

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

xdata struct alarm_st alarm_control;
.
.
.
```

The **Cx51** compiler generates an object file for `ALMCTRL.C` and includes a segment for variables in the `xdata` memory area. Because it is the only variable declared in this module, `alarm_control` is the only variable in that segment. The name of the segment is `?XD?ALMCTRL`. The **Lx51** Linker/Locator allows you to specify the base address of any segment by using the location directives.

For BL51 you must use the `XDATA` directive, since the `alarm_control` variable was declared to reside in `xdata`:

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

For LX51 the `SEGMENTS` directive is used to locate the segment in `xdata` space:

```
LX51 ... almctrl.obj SEGMENTS(?XD?ALMCTRL(X:0x2000)) ...
```

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

In the same way you may also locate segments in the other memory areas like **code**, **xdata**, **pdata**, **idata**, and **data**. Refer to the *A51 Macro Assembler User's Guide* for more information about the Linker/Locator.

The `_at_` Keyword

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

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

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

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

Refer to “Absolute Variable Location” on page 104 for more information about the `_at_` keyword.

NOTE

*If you use the `_at_` keyword to declare a variable that accesses an XDATA peripheral, you may require the **volatile** keyword to ensure that the C compiler does not optimize out necessary memory accesses.*

Debugging

When you are using the μ Vision2 IDE and the μ Vision2 Debugger, you will get complete debug information when you enable **Options for Target – Output – Debug Information**. For command line tools the following rules apply.

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

The **Cx51** compiler uses the OMF2 object file format. The OMF2 format is also used by the **Cx51** compiler when the directive OMF2 is active. The OMF2 format requires the extended LX51 linker/locater and cannot be used with the BL51 linker/locater. The OMF2 object file format provides extensive debug information and is supported by the μ Vision2 debugger and some emulators.

Chapter 7. Error Messages

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

Fatal Errors

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

Fatal error messages conform to one of the following formats:

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

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

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

Actions

ALLOCATING MEMORY

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

CREATING LIST-FILE / OBJECT-FILE / WORKFILE

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

GENERATING INTERMEDIATE CODE

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

OPENING INPUT-FILE

The compiler failed to find or open the selected source or include file.

PARSING INVOKE-/#PRAGMA-LINE

An error was detected while evaluating arguments on the command line or while evaluating parameters in a **#pragma** statement.

PARSING SOURCE-FILE / ANALYZING DECLARATIONS

The source file contains too many external references. Reduce the number of external variables and functions accessed by the source file.

WRITING TO FILE

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

Errors

' (' AFTER CONTROL EXPECTED

Some control parameters need an argument enclosed in parentheses. This message is displayed when the left parenthesis is missing.

') ' AFTER PARAMETER EXPECTED

This message indicates that the right parenthesis of the enclosed argument is missing.

BAD DIGIT IN NUMBER

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

CAN'T CREATE FILE

The filename defined on the `FILE` line cannot be created.

CAN'T HAVE GENERAL CONTROL IN INVOCATION LINE

General controls (for example, `EJECT`) cannot be included on the command line. Place these controls in the source file using the `#pragma` statement.

FILE DOES NOT EXIST

The filename defined on the `FILE` line, cannot be found.

FILE WRITE-ERROR

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

IDENTIFIER EXPECTED

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

MEMORY SPACE EXHAUSTED

The compiler could not allocate enough memory to compile the specified source file. If you receive this message consistently, you should split the source file into two or more smaller files and re-compile.

MORE THAN 100 ERRORS IN SOURCE-FILE

During the compilation more than 100 errors were detected. This causes the termination of the compiler.

MORE THAN 256 SEGMENTS/EXTERNALS

More than 256 total references were encountered in a source file. A single source file cannot contain more than 256 functions or external references. This is a historical restriction mandated by the Intel Object Module Format (OMF-51). Functions which contain scalar and/or `bit` declarations produce two and sometimes three segment definitions in the object file.

NON-NULL ARGUMENT EXPECTED

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

OUT OF RANGE NUMBER

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

PARSE STACK OVERFLOW

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

PREPROCESSOR: LINE TOO LONG (32K)

An intermediate expansion exceeded 32K characters in length.

PREPROCESSOR: MACROS TOO NESTED

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

RESPECIFIED OR CONFLICTING CONTROL

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

SOURCE MUST COME FROM A DISK-FILE

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

UNKNOWN CONTROL

The selected control parameter is unrecognized by the compiler.

Syntax and Semantic Errors

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

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

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

where:

number is the error number.

line corresponds to the line number in the source file or include file.

file is the name of the source or include file in which the error was detected.

error message is descriptive text and is dependent upon the type of error encountered.

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

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

Number	Error Message and Description
126	Type-stack underflow The type declaration stack has underflowed. This error is usually a side-effect of error 125.
127	Invalid storage class An object was declared with an invalid memory space specification. This occurs if an object is declared with storage class of auto or register outside of a function.
129	Missing ‘;’ before ‘token’ This error usually indicates that a semicolon is missing from the previous line. When this error occurs, the compiler may generate an excess of error messages.
130	Value out of range The numerical argument after a using or interrupt specifier is invalid. The using specifier requires a register bank number between 0 and 3. The interrupt specifier requires an interrupt vector number between 0 and 31.
131	Duplicate function-parameter A formal parameter name exists more than once within a function. The formal parameter names must be unique in function declarations.
132	Not in formal parameter list The parameter declarations inside a function use a name not present in the parameter name list. For example: <pre>char function (v0, v1, v2) char *v0, *v1, *v5; /* 'v5' is unknown in the formal list */ { /* ... */ }</pre>
134	xdata/idata/pdata/data on function not permitted Functions always reside in code memory and cannot be executed out of other memory areas. Functions are implicitly defined as memory type code .
135	Bad storage class for bit Declarations of bit scalars may include one of the static or extern storage classes. The register or alien classes are invalid.
136	‘void’ on variable The type void is allowed only as a non-existent return value or an empty argument list for functions (void func (void)), or in combination with a pointer (void *).
138	Interrupt() may not receive or return value(s) An interrupt function was defined with one or more formal parameters or with a return value. Interrupt functions may not contain invocation parameters or return values.
140	Bit in illegal memory-space Definitions of bit scalars may contain the optional memory type data . If the memory type is missing then the type data is assumed, because bits always reside in the internal data memory. This error can occur when an attempt is made to use another data type with a bit scalar definition.
141	Syntax error near token: expected other_token, ... The token seen by the compiler is wrong. Depending upon the context the expected token is displayed.
142	Invalid base address The base-address of an sfr or sbit declaration is in error. Valid bases are values in the 0x80 to 0xFF range. If the declaration uses the notation base^pos , then the base address must also be a multiple of eight.

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

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

Number	Error Message and Description
180	Can't cast to 'function' Type casts to function types are invalid. Try casting to a pointer to a function.
181	Incompatible operand At least one operand type is not valid with the given operator (for example, <code>~float_type</code>).
183	Unmodifiable lvalue The object to be changed resides in code memory or has const attribute and therefore cannot be modified.
184	Sizeof: illegal operand The sizeof operator cannot determine the size of a function or bit-field.
185	Different memory space The memory space of an object declaration differs from the memory space of a prior declaration for the same object.
186	Invalid dereference This error message may be caused by an internal compiler problem. Please contact technical support if this error is repeated.
187	Not an lvalue The needed argument must be the address of an object that can be modified.
188	Unknown object size The size of an object cannot be computed because of a missing dimension size on an array or indirection via a void pointer.
189	'&' on bit/sfr illegal The address-of operator ('&') is not allowed on bit objects or special function registers (sfr).
190	'&': not an lvalue An attempt was made to construct a pointer to an anonymous object.
193	Illegal op-type(s)
193	Illegal add/sub on ptr
193	Illegal operation on bit(s)
193	Bad operand type This error results when an expression uses illegal operand-types with the given operator. Examples of invalid expressions are bit * bit , ptr + ptr , or ptr * anything . The error message includes the operator which caused the error. The following operations may be executed with bit-type operands: <ul style="list-style-type: none"> ■ Assignment (=) ■ OR / Compound OR (, =) ■ AND / Compound AND (&, &=) ■ XOR / Compound XOR (^, ^=) ■ Compare bit with bit or constant (==, !=) ■ Negation (~) bit operands may be used in expressions with other data types. In this case a type cast is automatically performed.
194	** indirection to object of unknown size The indirection operator * may not be used with void pointers because the object size, which the pointer refers to, is unknown.
195	** illegal indirection The * operator may not be applied on non-pointer arguments.

Number	Error Message and Description
196	Mspace probably invalid The conversion of a constant to a pointer constant yields an invalid memory space, for example <code>char *p = 0x91234</code> .
198	Sizeof returns zero The <code>sizeof</code> operator returns a zero value.
199	Left side of '->' requires struct/union pointer The argument on the left side of the <code>-></code> operator must be a struct pointer or a union pointer.
200	Left side of '.' requires struct/union The argument on the left side of the <code>.</code> operator must have type struct or union .
201	Undefined struct/union tag The given struct or union tag name is unknown.
202	Undefined identifier The given identifier is undefined.
203	Bad storage class (nameref) This error indicates a problem within the compiler. Please contact technical support if this error is repeated.
204	Undefined member The given member name in a struct or union reference is undefined.
205	Can't call an interrupt function An interrupt function should not be called like a normal function. The entry and exit code for these functions is specially coded for interrupts.
207	Declared with 'void' parameter list A function declared with a void parameter list cannot receive parameters from the caller.
208	Too many actual parameters The function call includes more parameters than previously declared.
209	Too few actual parameters Too few actual parameters were included in a function call.
210	Too many nested calls Function calls can be nested at most 10 levels deep.
211	Call not to a function The term of a function call does not evaluate to a function or pointer to function.
212	Indirect call: parameters do not fit within registers An indirect function call through a pointer cannot contain actual parameters. An exception to this rule is when all parameters can be passed in registers. This is due to the method of parameter passing employed by Cx51. The name of the called function must be known because parameters are written into the data segment of the called function. For indirect calls, however, the name of the called function is not known.
213	Left side of asn-op not an lvalue The address of a changeable object is required on the left side of the assignment operator.
214	Illegal pointer conversion Objects of type <code>bit</code> , <code>float</code> or aggregates cannot be converted to pointers.
215	Illegal type conversion <code>Struct/union/void</code> cannot be converted to any other types.

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

Number	Error Message and Description						
236	Different length of parameter lists The number of parameters in the function declaration is different from the number of parameters in the function prototype.						
237	Function already has a body An attempt was made to declare a body for a function twice.						
238	Duplicate member						
239	Duplicate parameter An attempt was made to define an already defined struct member or function parameter.						
240	More than 128 local bit's No more than 128 bit -scalars may be defined inside a function.						
241	Auto segment too large The required space for local objects exceeds the model-dependent maximum. The maximum segment sizes are defined as follows: <table data-bbox="482 638 771 715" style="margin-left: 40px;"> <tr> <td>SMALL</td> <td>128 bytes</td> </tr> <tr> <td>COMPACT</td> <td>256 bytes</td> </tr> <tr> <td>LARGE</td> <td>65535 bytes</td> </tr> </table>	SMALL	128 bytes	COMPACT	256 bytes	LARGE	65535 bytes
SMALL	128 bytes						
COMPACT	256 bytes						
LARGE	65535 bytes						
242	Too many initializers The number of initializers exceeded the number of objects to be initialized.						
243	String out of bounds The number of characters in the string exceeds the number of characters required to initialize the array of characters.						
244	Can't initialize, bad type or class An attempt was made to initialize a bit or an sfr .						
245	Unknown pragma, line ignored The #pragma statement is unknown so, the entire line is ignored.						
246	Floating-point error This error occurs when a floating-point argument lies outside of the valid range for 32-bit floating values. The numeric range of the 32-bit IEEE values is: $\pm 1.175494E-38$ to $\pm 3.402823E+38$.						
247	Non-address/constant initializer A valid initializer expression must evaluate to a constant value or the name of an object plus or minus a constant.						
248	Aggregate initialization needs curly braces The braces ({ }) around the given struct or union initializer were missing.						
249	Segment <name>: Segment too large The compiler detected a data segment that was too large. The maximum size of a data segment depends on memory space.						
250	'\esc'; value exceeds 255 An escape sequence in a string constant exceeds the valid value range. The maximum value is 255.						
251	Illegal octal digit The specified character is not a valid octal digit.						
252	Misplaced primary control, line ignored Primary controls must be specified at the start of the C module before any #include directives or declarations.						

Number	Error Message and Description
253	<p>Internal error (ASMGEN\CLASS) This error can occur under the following circumstances:</p> <ul style="list-style-type: none"> ■ An intrinsic function (for example, <code>_testbit_</code>) was activated incorrectly. This is the case when no prototype of the function exists and the number of actual parameters or their type is incorrect. For this reason, the appropriate declaration files must always be used (<code>INTRINS.H</code>, <code>STRING.H</code>). See Chapter 8 for more information on intrinsic functions. ■ Cx51 recognized an internal consistency problem. Please contact technical support if this error occurs.
255	<p>Switch expression has illegal type The expression in a switch statement does not have a legal data type.</p>
256	<p>Conflicting memory model A function which contains the alien attribute may contain only the model specification small. The parameters of the function must lie in internal data memory. This applies to all external alien declarations and alien functions. For example:</p> <pre>alien plm_func (char c) large { ... }</pre> <p>generates error 256.</p>
257	<p>Alien function cannot be reentrant A function that contains the alien attribute cannot simultaneously contain the attribute reentrant. The parameters of the function cannot be passed via the virtual stack. This applies to all external alien declarations and alien functions.</p>
258	<p>Mspace illegal on struct/union member Mspace on parameter ignored A member of a structure or a parameter may not contain the specification of a memory type. The object to which the pointer refers may, however, contain a memory type. For example:</p> <pre>struct vp { char code c; int xdata i; };</pre> <p>generates error 258.</p> <pre>struct v1 { char c; int xdata *i; };</pre> <p>is the correct declaration for the struct.</p>
259	<p>Pointer: different mspace A spaced pointer has been assigned another spaced pointer with a different memory space. For example:</p> <pre>char xdata *p1; char idata *p2; p1 = p2; /* different memory spaces */</pre>
260	<p>Pointer truncation A spaced pointer has been assigned some constant value which exceeds the range covered by the pointers memory space. For example:</p> <pre>char idata *p1 = 0x1234; /* result is 0x34 */</pre>

Number	Error Message and Description
261	<p>Bit(s) in reentrant () A function with the attribute <code>reentrant</code> cannot have bit objects declared inside the function. For example:</p> <pre>int func1 (int i1) reentrant { bit b1, b2; /* not allowed ! */ return (i1 - 1); }</pre>
262	<p>'using/disable': can't return bit value Functions declared with the <code>using</code> attribute and functions which rely on disabled interrupts (<code>#pragma disable</code>) cannot return a <code>bit</code> value to the caller. For example:</p> <pre>bit test (void) using 3 { bit b0; return (b0); }</pre> <p>produces error 262.</p>
263	<p>Save/restore: save-stack overflow/underflow The maximum nesting depth <code>#pragma save</code> comprises eight levels. The pragmas <code>save</code> and <code>restore</code> work with a stack according to the LIFO (last in, first out) principal.</p>
264	<p>Intrinsic '<intrinsic_name>': declaration/activation error This error indicates that an <code>intrinsic</code> function was defined incorrectly (parameter number or ellipsis notation). This error should not occur if you are using the standard <code>.H</code> files. Make sure that you are using the <code>.H</code> files that were included with Cx51. Do not try to define your own prototypes for intrinsic library functions.</p>
265	<p>Recursive call to non-reentrant function Non reentrant functions cannot be called recursively since such calls would overwrite the parameters and local data of the function. If you need recursive calls, you should declare the function with the <code>reentrant</code> attribute.</p>
267	<p>Funcdef requires ANSI-style prototype A function was invoked with parameters but the declaration specifies an empty parameter list. The prototype should be completed with the parameter types in order to give the compiler the opportunity to pass parameters in registers and have the calling mechanism consistent over the application.</p>
268	<p>Bad taskdef (taskid/priority/using) The task declaration is incorrect.</p>
271	<p>Misplaced 'asm/endasm' control The <code>asm</code> and <code>endasm</code> statements may not be nested. <code>Endasm</code> requires that an <code>asm</code> block be opened by a previous <code>asm</code> statement. For example:</p> <pre>#pragma asm . . . assembler instruction(s) . . . #pragma endasm</pre>

Number	Error Message and Description
272	'asm' requires SRC control to be active The use of asm and endasm in a source file requires that the file be compiled using the SRC directive. The compiler then generates an assembly source file which may then be assembled with A51.
273	'asm/endasm' not allowed in include file The use of asm and endasm is not permitted within include files. For debug reasons executable code should be avoided in include files anyway.
274	Absolute specifier illegal The absolute address specification is not allowed on bit objects, functions, and function locals. The address must conform to the memory space of the object. For example, the following declaration is invalid because the range of the indirectly addressable data space is 0x00 to 0xFF. <pre>idata int _at_ 0x1000;</pre>
278	Constant too big This error occurs when a floating-point argument lies outside of the valid range for 32-bit floating values. The numeric range of the 32-bit IEEE values is: $\pm 1.175494E-38$ to $\pm 3.402823E+38$.
279	Multiple initialization An attempt has been made to initialize some object more than once.
280	unreferenced symbol/label/parameter A symbol, label, or parameter that was declared in a function and was not used.
281	non-pointer type converted to pointer The referenced program object cannot be converted to a pointer.
282	not a sfr reference This function invocation requires a SFR location as parameter.
283	asmparms: parameters do not fit within registers Parameters do not fit within the available CPU registers.
284	<name>: in overlayable space, function no longer reentrant A reentrant function contains explicit memory type specifiers for local variables. The function will no longer be fully reentrant.
300	Unterminated comment This message occurs when a comment does not have a closing delimiter (*).
301	Identifier expected The syntax of a preprocessor directive expects an identifier.
302	Misused # operator This message occurs if the stringize operator '#' is not followed by an identifier.
303	Formal argument expected This message occurs if the stringize operator '#' is not followed by an identifier representing a formal parameter name of the macro currently being defined.
304	Bad macro parameter list The macro parameter list does not represent a brace enclosed, comma separated list of identifiers.
305	Unterminated string/char constant A string or character constant is invalid. Typically, this error is encountered if the closing quote is missing.
306	Unterminated macro call The end of the input file was reached while the preprocessor was collecting and expanding actual parameters of a macro call.

Number	Error Message and Description
307	Macro 'name': parameter count mismatch The number of actual parameters in a macro call does not match the number of parameters of the macro definition. This error indicates that too few parameters were specified.
308	Invalid integer constant expression The numerical expression of an if/elif directive contains a syntax error.
309	Bad or missing file name The filename argument in an include directive is invalid or missing.
310	Conditionals too nested(20) The source file contains too many nested directives for conditional compilation. The maximum nesting level allowed is 20.
311	Misplaced elif/else control
312	Misplaced endif control The directives elif, else, and endif are legal only within an if, ifdef, or ifndef directive.
313	Can't remove predefined macro 'name' An attempt was made to remove a predefined macro. User-defined macros may be deleted using the #undef directive. Predefined macros cannot be removed.
314	Bad # directive syntax In a preprocessor directive, the character '#' must be followed by either a newline character or the name of a preprocessor command (for example, if/define/ifdef, ...).
315	Unknown # directive 'name' The name of the preprocessor directive is not known to the compiler.
316	Unterminated conditionals The number of endifs does not match the number of if or ifdefs after the end of the input file.
318	Can't open file 'filename' The given file could not be opened.
319	'File' is not a disk file The given file is not a disk file. Files other than disk files are not legal for compilation.
320	User_error_text This error number is reserved for errors introduced with the #error directive of the preprocessor. The #error directive emits the user error text to come up with error 320 which prevents the compiler from generating code.
321	Missing <character> In the filename argument of an include directive, the closing character is missing. For example: <code>#include <stdio.h</code>
325	Duplicate formal parameter 'name' A formal parameter of a macro may be defined only once.
326	Macro body cannot start or end with '##' The concat operator ('##') cannot be the first or last token of a macro body.
327	Macro 'macroname': more than 50 parameters The number of parameters per macro is limited to 50.

Warnings

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

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

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

where:

number is the error number.

line corresponds to the line number in the source file or include file.

file is the name of the source or include file in which the error was detected.

warning message is descriptive text that is dependent upon the type of warning encountered.

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

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

Number	Warning Message and Description
206	<p>Missing function prototype The called function is unknown because no prototype declaration exists. Calls to unknown functions are always at risk that the number of parameters does not correspond to the actual requirements. If this is the case, the function is called incorrectly.</p> <p>Without function prototypes, the compiler has no way to check for missing or excessive parameters or their types. To avoid this warning, include prototypes of the functions used in your program.</p> <p>Function prototypes must be specified before the function is called. Note that function definitions automatically produce prototypes.</p>
209	<p>Too few actual parameters Too few actual parameters were included in a function call.</p>
219	<p>Long constant truncated to int The value of a constant expression must be capable of being represented by an int type.</p>
245	<p>Unknown pragma, line ignored The #pragma statement is unknown, so the entire pragma line is ignored.</p>
258	<p>Mspace illegal on struct/union member Mspace on parameter ignored A member of a structure or a parameter may not contain the specification of a memory type. The object to which the pointer refers may, however, contain a memory type. For example:</p> <pre>struct vp { char code c; int xdata i; };</pre> <p>generates error 258.</p> <pre>struct vl { char c; int xdata *i; };</pre> <p>is the correct declaration for the struct.</p>
259	<p>Pointer: different mspace This warning is generated when two pointers that do not refer to the same memory type of object are compared.</p>
260	<p>Pointer truncation This error or warning occurs when converting a pointer to a pointer with a smaller offset area. The conversion takes place, but the offset of the larger pointer is truncated to fit into the smaller pointer.</p>
261	<p>Bit in reentrant function A reentrant function cannot contain bits because bit scalars cannot be stored on the virtual stack.</p>
265	<p>'name': recursive call to non-reentrant function A direct recursion to a non-reentrant function was discovered. This can be intentional but should be functionally checked (through the generated code) for each individual case. Indirect recursions are discovered by the linker/locator.</p>

Number	Warning Message and Description
271	<p>Misplaced 'asm/endasm' control The asm and endasm statements may not be nested. Endasm requires that an asm block be opened by a previous asm statement. For example:</p> <pre data-bbox="444 314 748 531">#pragma asm . . . assembler instruction(s) . . . #pragma endasm</pre>
275	<p>Expression with possibly no effect The compiler detected an expression which does not generate code. For example:</p> <pre data-bbox="444 621 1029 739">void test (void) { int i1, i2, i3; i1, i2, i3; /* dead expression */ i1 << 3; /* result is not used */ }</pre>
276	<p>Constant in condition expression The compiler detected a conditional expression with a constant value. In most cases this is a typing mistake. For example:</p> <pre data-bbox="444 855 1105 973">void test (void) { int i1, i2, i3; if (i1 = 1) i2 = 3; /* const assigned with = */ while (i3 = 2); /* const assigned with = */ }</pre>
277	<p>Different mspaces to pointer A typedef declaration has a conflict of the memory spaces. For example:</p> <pre data-bbox="444 1064 1068 1112">typedef char xdata XCC; /* mspace xdata */ typedef XCC idata PICC; /* mspace collision */</pre>
280	<p>Unreferenced symbol/label This message identifies a symbol or label which has been defined but not used.</p>
307	<p>Macro 'name': parameter count mismatch The number of actual parameters in a macro call does not match the number of parameters of the macro definition. This warning indicates that too many parameters were used. Excess parameters are ignored.</p>
317	<p>Macro 'name': invalid redefinition A predefined macro cannot be redefined or removed. Refer to "Predefined Macro Constants" on page 138 for more information.</p>
322	<p>Unknown identifier The identifier in an #if directive line is undefined (evaluates to FALSE).</p>
323	<p>Newline expected, extra characters found A #directive line is correct but contains extra non commented characters. For example:</p> <pre data-bbox="444 1541 739 1564">#include <stdio.h> foo</pre>

Number	Warning Message and Description
324	Preprocessor token expected A preprocessor token was expected but a newline character was found in input. For example: #line where the arguments to the #line directive are missing.

Chapter 8. Library Reference

The **Cx51** run-time library provides you with more than 100 predefined functions and macros to use in your 8051 C programs. This library makes embedded software development easier by providing you with routines that perform common programming tasks such as string and buffer manipulation, data conversion, and floating-point math operations.

Typically, the routines in this library conform to the ANSI C Standard. However, some functions differ slightly in order to take advantage of the features found in the 8051 architecture. For example, the function **isdigit** returns a **bit** value as opposed to an **int**. Where possible, function return types and argument types are adjusted to use the smallest possible data type. In addition, unsigned data types are favored over signed types. These alterations to the standard library provide a maximum of performance while also reducing program size.

All routines in this library are implemented to be independent of and to function using any register bank.

Intrinsic Routines

The **Cx51** compiler supports a number of intrinsic library functions. Non-intrinsic functions generate **ACALL** or **LCALL** instructions to perform the library routine. Intrinsic functions generate in-line code to perform the library routine. The generated in-line code is much faster and more efficient than a called routine would be. The following functions are available in intrinsic form:

<u>_crol_</u>	<u>_iror_</u>	<u>_nop_</u>
<u>_cror_</u>	<u>_lrol_</u>	<u>_testbit_</u>
<u>_irol_</u>	<u>_lror_</u>	

These routines are described in detail in the following sections.

Library Files

The **Cx51** library includes six different compile-time libraries which are optimized for various functional requirements. These libraries support most of the ANSI C function calls.

Library File	Description
C51S.LIB	Small model library without floating-point arithmetic
C51FPS.LIB	Small model floating-point arithmetic library
C51C.LIB	Compact model library without floating-point arithmetic
C51FPC.LIB	Compact model floating-point arithmetic library
C51L.LIB	Large model library without floating-point arithmetic
C51FPL.LIB	Large model floating-point arithmetic library
80C751.LIB	Library for use with the Signetics 8xC751 and derivatives.

The Philips 80C51MX, Dallas 390 contiguous mode and variable code banking requires a different set of Cx51 run-time libraries. The LX51 linker/locator automatically adds the correct library set to your project.

Several library modules are provided in source code form. These routines are used to perform low-level hardware-related I/O for the stream I/O functions. You can find the source for these routines in the **LIB** directory. You may modify these source files and substitute them for the library routines. By using these routines, you can quickly adapt the library to perform (using any hardware I/O device available in your target) stream I/O. Refer to “Stream Input and Output” on page 224 for more information.

Standard Types

The **Cx51** standard library contains definitions for a number of standard types which may be used by the library routines. These standard types are declared in include files which you may access from your C programs.

jmp_buf

The **jmp_buf** type is defined in **SETJMP.H** and specifies the buffer used by the **setjmp** and **longjmp** routines to save and restore the program environment. The **jmp_buf** type is defined as follows:

```
#define _JBLEN 7
typedef char jmp_buf[_JBLEN];
```

va_list

The **va_list** array type is defined in **STDARG.H**. This type holds data required by the **va_arg** and **va_end** routines. The **va_list** type is defined as follows:

```
typedef char *va_list;
```

Absolute Memory Access Macros

The Cx51 standard library contains definitions for a number of macros that allow you to access explicit memory addresses. These macros are defined in **ABSACC.H**. Each of these macros is defined to be used like an array.

CBYTE

The **CBYTE** macro allows you to access individual bytes in the program memory of the 8051. You may use this macro in your programs as follows:

```
rval = CBYTE [0x0002];
```

to read the contents of the byte in program memory at address 0002h.

CWORD

The **CWORD** macro allows you to access individual words in the program memory of the 8051. You may use this macro in your programs as follows:

```
rval = CWORD [0x0002];
```

to read the contents of the word in program memory at address 0004h
($2 \times \text{sizeof}(\text{unsigned int}) = 4$).

NOTE

The index used with this macro does not represent the memory address of the integer value. To obtain the memory address, you must multiply the index by the size of an integer (2 bytes).

DBYTE

The **DBYTE** macro allows you to access individual bytes in the internal data memory of the 8051. You may use this macro in your programs as follows:

```
rval = DBYTE [0x0002];  
DBYTE [0x0002] = 5;
```

to read or write the contents of the byte in internal data memory at address 0002h.

DWORD

The **DWORD** macro allows you to access individual words in the internal data memory of the 8051. You may use this macro in your programs as follows:

```
rval = DWORD [0x0002];  
DWORD [0x0002] = 57;
```

to read or write the contents of the word in internal data memory at address 0004h ($2 \times \text{sizeof}(\text{unsigned int}) = 4$).

NOTE

The index used with this macro does not represent the memory address of the integer value. To obtain the memory address, you must multiply the index by the size of an integer (2 bytes).

FARRAY, FCARRAY

The **FARRAY** and **FCARRAY** macros can be used to access an array of type *object* in the **far** and **const far** memory areas. **FARRAY** provides access to the **far** space (memory class HDATA). **FCARRAY** provides access to the **const far** space (memory class HCONST). You can use this macros in your programs as follows:

```
int i;
long l;

l = FARRAY (long, 0x8000) [i];
FARRAY (long, 0x8000) [10] = 0x12345678;

#define DualPortRam FARRAY (int, 0x24000)
DualPortRam [i] = 0x1234;

l = FCARRAY (long, 0x18000) [5];
```

The **FARRAY** and **FCARRAY** macros scales the index by the size of type *object* and adds the result to *addr*. The final address is then used to access the memory.

NOTE

The absolute addressed object cannot cross a 64KB segment boundary. For example, you cannot access a long array that has 10 elements and starts at address 0xFFF8.

FVAR, FCVAR,

The **FVAR** and **FCVAR** macro definitions may be used to access absolute memory addresses in the **far** and **const far** memory areas. **FVAR** provides access to the **far** space (memory class **HDATA**). **FCVAR** provides access to the **const far** space (memory class **HCONST**). You can use these macros in your programs as follows:

```
#define IOVAL FVAR (long, 0x14FFE) // long at HDATA address 0x14FFE
var = IOVAL;                    /* read */
IOVAL = 0x10;                   /* write */

var = FCVAR (int, 0x24002) /* read int from HCONST address 0x24002 */
```

The **HVAR** macro uses the **huge** modifier to access the memory by segment and offset, as opposed to **MVAR**'s page and offset.

NOTE

The absolute addressed object cannot cross a 64KB segment boundary. For example, you cannot access a long variable at address 0xFFFE.

PBYTE

The **PBYTE** macro allows you to access individual bytes in one page of the external data memory of the 8051. You may use this macro in your programs as follows:

```
rval = PBYTE [0x0002];  
PBYTE [0x0002] = 38;
```

to read or write the contents of the byte in **pdata** memory at address 0002h.

PWORD

The **PWORD** macro allows you to access individual words in one page of the external data memory of the 8051. You may use this macro in your programs as follows:

```
rval = PWORD [0x0002];  
PWORD [0x0002] = 57;
```

to read or write the contents of the word in **pdata** memory at address 0004h ($2 \times \text{sizeof}(\text{unsigned int}) = 4$).

NOTE

The index used with this macro does not represent the memory address of the integer value. To obtain the memory address, you must multiply the index by the size of an integer (2 bytes).

XBYTE

The **XBYTE** macro allows you to access individual bytes in the external data memory of the 8051. You may use this macro in your programs as follows:

```
rval = XBYTE [0x0002];  
XBYTE [0x0002] = 57;
```

to read or write the contents of the byte in external data memory at address 0002h.

XWORD

The **XWORD** macro allows you to access individual words in the external data memory of the 8051. You may use this macro in your programs as follows:

```
rval = XWORD [2];  
XWORD [2] = 57;
```

to read or write the contents of the word in external data memory at address 0004h ($2 \times \text{sizeof}(\text{unsigned int}) = 4$).

NOTE

The index used with this macro does not represent the memory address of the integer value. To obtain the memory address, you must multiply the index by the size of an integer (2 bytes).

Routines by Category

This sections gives a brief overview of the major categories of routines available in the **Cx51** standard library. Refer to “Reference” on page 232 for a complete description of routine syntax and usage.

NOTE

Many of the routines in the Cx51 standard library are reentrant, intrinsic, or both. These specifications are listed under attributes in the following tables. Unless otherwise noted, routines are non-reentrant and non-intrinsic.

Buffer Manipulation

Routine	Attributes	Description
memccpy		Copies data bytes from one buffer to another until a specified character or specified number of characters has been copied.
memchr	reentrant	Returns a pointer to the first occurrence of a specified character in a buffer.
memcmp	reentrant	Compares a given number of characters from two different buffers.
memcpy	reentrant	Copies a specified number of data bytes from one buffer to another.
memmove	reentrant	Copies a specified number of data bytes from one buffer to another.
memset	reentrant	Initializes a specified number of data bytes in a buffer to a specified character value.

The buffer manipulation routines are used to work on memory buffers on a character-by-character basis. A buffer is an array of characters like a string, however, a buffer is usually not terminated with a null character (`'\0'`). For this reason, these routines require a buffer length or count argument.

All of these routines are implemented as functions. Function prototypes are included in the `STRING.H` include file.

Character Conversion and Classification

Routine	Attributes	Description
isalnum	reentrant	Tests for an alphanumeric character.
isalpha	reentrant	Tests for an alphabetic character.
iscntrl	reentrant	Tests for a Control character.
isdigit	reentrant	Tests for a decimal digit.
isgraph	reentrant	Tests for a printable character with the exception of space.
islower	reentrant	Tests for a lowercase alphabetic character.
isprint	reentrant	Tests for a printable character.
ispunct	reentrant	Tests for a punctuation character.
isspace	reentrant	Tests for a whitespace character.
isupper	reentrant	Tests for an uppercase alphabetic character.
isxdigit	reentrant	Tests for a hexadecimal digit.
toascii	reentrant	Converts a character to an ASCII code.
toint	reentrant	Converts a hexadecimal digit to a decimal value.
tolower	reentrant	Tests a character and converts it to lowercase if it is uppercase.
_tolower	reentrant	Unconditionally converts a character to lowercase.
toupper	reentrant	Tests a character and converts it to uppercase if it is lowercase.
_toupper	reentrant	Unconditionally converts a character to uppercase.

The character conversion and classification routines allow you to test individual characters for a variety of attributes and convert characters to different formats.

The **_tolower**, **_toupper**, and **toascii** routines are implemented as macros. All other routines are implemented as functions. All macro definitions and function prototypes are found in the **CTYPE.H** include file.

Data Conversion

Routine	Attributes	Description
abs	reentrant	Generates the absolute value of an integer type.
atof / atof517		Converts a string to a float.
atoi		Converts a string to an int.
atol		Converts a string to a long.
cabs	reentrant	Generates the absolute value of a character type.
labs	reentrant	Generates the absolute value of a long type.
strtod / strtod517		Converts a string to a float.
strtol		Converts a string to a long.
strtoul		Converts a string to an unsigned long.

The data conversion routines convert strings of ASCII characters to numbers. All of these routines are implemented as functions and most are prototyped in the include file `STDLIB.H`. The **abs**, **cabs**, and **labs** functions are prototyped in the `MATH.H` include file. The **atof517**, and **strtod517** function are prototyped in the include file `80C517.H`.

Math Routines

Routine	Attributes	Description
acos / acos517		Calculates the arc cosine of a specified number.
asin / asin517		Calculates the arc sine of a specified number.
atan / atan517		Calculates the arc tangent of a specified number.
atan2		Calculates the arc tangent of a fraction.
ceil		Finds the integer ceiling of a specified number.
cos / cos517		Calculates the cosine of a specified number.
cosh		Calculates the hyperbolic cosine of a specified number.
exp / exp517		Calculates the exponential function of a specified number.
fabs	reentrant	Finds the absolute value of a specified number.
floor		Finds the largest integer less than or equal to a specified number.
fmod		Calculates the floating-point remainder.
log / log517		Calculates the natural logarithm of a specified number.
log10 / log10517		Calculates the common logarithm of a specified number.
modf		Generates integer and fractional components of a specified number.
pow		Calculates a value raised to a power.
rand	reentrant	Generates a pseudo random number.
sin / sin517		Calculates the sine of a specified number.
sinh		Calculates the hyperbolic sine of a specified number.
srand		Initializes the pseudo random number generator.
sqrt / sqrt517		Calculates the square root of a specified number.
tan / tan517		Calculates the tangent of a specified number.
tanh		Calculates the hyperbolic tangent of a specified number.
chkfloat	intrinsic, reentrant	Checks the status of float numbers.
crol	intrinsic, reentrant	Rotates an unsigned char left a specified number of bits.
cror	intrinsic, reentrant	Rotates an unsigned char right a specified number of bits.
irol	intrinsic, reentrant	Rotates an unsigned int left a specified number of bits.
iror	intrinsic, reentrant	Rotates an unsigned int right a specified number of bits.
lrol	intrinsic, reentrant	Rotates an unsigned long left a specified number of bits.
lror	intrinsic, reentrant	Rotates an unsigned long right a specified number of bits.

The math routines perform common mathematical calculations. Most of these routines work with floating-point values and therefore include the floating-point libraries and support routines.

All of these routines are implemented as functions. Most are prototyped in the include file `MATH.H`. Functions which end in 517 (`acos517`, `asin517`, `atan517`, `cos517`, `exp517`, `log517`, `log10517`, `sin517`, `sqrt517`, and `tan517`) are prototyped in the `80C517.H` include file. The `rand` and `srand` functions are prototyped in the `STDLIB.H` include file.

The `_chkfloat_`, `_crol_`, `_cror_`, `_irol_`, `_iror_`, `_lrol_`, and `_lror_` functions are prototyped in the `INTRINS.H` include file.

Memory Allocation Routines

Routine	Attributes	Description
calloc		Allocates storage for an array from the memory pool.
free		Frees a memory block that was allocated using calloc , malloc , or realloc .
init_mempool		Initializes the memory location and size of the memory pool.
malloc		Allocates a block from the memory pool.
realloc		Reallocates a block from the memory pool.

The memory allocation functions provide you with a means to specify, allocate, and free blocks of memory from a memory pool. All memory allocation functions are implemented as functions and are prototyped in the **STDLIB.H** include file.

Before using any of these functions to allocate memory, you must first specify, using the **init_mempool** routine, the location and size of a memory pool from which subsequent memory requests are satisfied.

The **calloc** and **malloc** routines allocate blocks of memory from the pool. The **calloc** routine allocates an array with a specified number of elements of a given size and initializes the array to 0. The **malloc** routine allocates a specified number of bytes.

The **realloc** routine changes the size of an allocated block, while the **free** routine returns a previously allocated memory block to the memory pool.

Stream Input and Output Routines

Routine	Attributes	Description
getchar	reentrant	Reads and echoes a character using the <code>_getkey</code> and <code>putchar</code> routines.
_getkey		Reads a character using the 8051 serial interface.
gets		Reads and echoes a character string using the <code>getchar</code> routine.
printf / printf517		Writes formatted data using the <code>putchar</code> routine.
putchar		Writes a character using the 8051 serial interface.
puts	reentrant	Writes a character string and newline ('\n') character using the <code>putchar</code> routine.
scanf / scanf517		Reads formatted data using the <code>getchar</code> routine.
sprintf / sprintf517		Writes formatted data to a string.
sscanf / sscanf517		Reads formatted data from a string.
ungetchar		Places a character back into the <code>getchar</code> input buffer.
vprintf		Writes formatted data using the <code>putchar</code> function.
vsprintf		Writes formatted data to a string.

The stream input and output routines allow you to read and write data to and from the 8051 serial interface or a user-defined I/O interface. The default `_getkey` and `putchar` functions found in the **Cx51** library read and write characters using the 8051 serial interface. You can find the source for these functions in the **LIB** directory. You may modify these source files and substitute them for the library routines. When this is done, other stream functions then perform input and output using the new `_getkey` and `putchar` routines.

If you want to use the existing `_getkey` and `putchar` functions, you must first initialize the 8051 serial port. If the serial port is not properly initialized, the default stream functions do not function. Initializing the serial port requires manipulating special function registers SFRs of the 8051. The include file **REG51.H** contains definitions for the required SFRs.

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

```
.
.
.
#include <reg51.h>
.
.
.
SCON = 0x50;          /* Setup serial port control register */
                    /* Mode 1: 8-bit uart var. baud rate */
                    /* REN: enable receiver */

PCON &= 0x7F;        /* Clear SMOD bit in power ctrl reg */
                    /* This bit doubles the baud rate */

TMOD &= 0xCF         /* Setup timer/counter mode register */
                    /* Clear M1 and M0 for timer 1 */
TMOD |= 0x20;        /* Set M1 for 8-bit autoreload timer */

TH1 = 0xFD;          /* Set autoreload value for timer 1 */
                    /* 9600 baud with 11.0592 MHz xtal */

TR1 = 1;             /* Start timer 1 */

TI = 1;              /* Set TI to indicate ready to xmit */
.
.
.
```

The stream routines treat input and output as streams of individual characters. There are routines that process characters as well as functions that process strings. Choose the routines that best suit your requirements.

All of these routines are implemented as functions. Most are prototyped in the **STDIO.H** include file. The **printf517**, **scanf517**, **sprintf517**, and **sscanf517** functions are prototyped in the **80C517.H** include file.

String Manipulation Routines

Routine	Attributes	Description
strcat		Concatenates two strings.
strchr	reentrant	Returns a pointer to the first occurrence of a specified character in a string.
strcmp	reentrant	Compares two strings.
strcpy	reentrant	Copies one string to another.
strcspn		Returns the index of the first character in a string that matches any character in a second string.
strlen	reentrant	Returns the length of a string.
strncat		Concatenates up to a specified number of characters from one string to another.
strncmp		Compares two strings up to a specified number of characters.
strncpy		Copies up to a specified number of characters from one string to another.
strpbrk		Returns a pointer to the first character in a string that matches any character in a second string.
strpos	Reentrant	Returns the index of the first occurrence of a specified character in a string.
strrchr	Reentrant	Returns a pointer to the last occurrence of a specified character in a string.
strrpbrk		Returns a pointer to the last character in a string that matches any character in a second string.
strrpos	Reentrant	Returns the index of the last occurrence of a specified character in a string.
strspn		Returns the index of the first character in a string that does not match any character in a second string.
strstr		Returns a pointer in a string that is identical to a second sub-string.

The string routines are implemented as functions and are prototyped in the **STRING.H** include file. They perform the following operations:

- Copying strings
- Appending one string to the end of another
- Comparing two strings
- Locating one or more characters from a specified set in a string

All string functions operate on null-terminated character strings. To work on non-terminated strings, use the buffer manipulation routines described earlier in this section.

Variable-length Argument List Routines

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 Routines

Routine	Attributes	Description
setjmp	reentrant	Saves the current stack condition and program address.
longjmp	reentrant	Restores the stack condition and program address.
nop	intrinsic, reentrant	Inserts an 8051 NOP instruction.
testbit	intrinsic, reentrant	Tests the value of a bit and clears it to 0.

Routines found in the miscellaneous category do not fit easily into any other library routine category. The **setjmp** and **longjmp** routines are implemented as functions and are prototyped in the `STDJMP.H` include file.

The **_nop_** and **_testbit_** routines direct the compiler to generate an **NOP** instruction and a **JBC** instruction respectively. These routines are prototyped in the `INTRINS.H` include file.

Include Files

The include files that are provided with the **Cx51** standard library are found in the **INC** subdirectory. These files contain constant and macro definitions, type definitions, and function prototypes. The following sections describe the use and contents of each include file. Macros and functions included in the file are listed as well.

8051 Special Function Register Include Files

The **Cx51** compiler package provides you with a number of include files that define manifest constants for the special function registers found on many 8051 derivatives. These files can be found in the folder **KEIL\C51\MNC** and the sub-folders. For example, the Special Function Registers (SFR) of the Philips 80C554 device are defined in the file **KEIL\C51\INC\PHILIPS\REG554.H**.

Within the μ Vision2 editor context menu that opens on a right mouse click in an editor window, you can insert the SFR definition that matches the selected device.

SFR definition files for all 8051 variants can be downloaded from www.keil.com. The device database available on this web page contains the header file for the Special Function Registers file of almost all 8051 devices.

80C517.H

The **80C517.H** include file contains routines that use the enhanced operational features of the 80C517 CPU and its derivatives. These routines are:

acos517
asin517
atan517
atof517
cos517
exp517

log10517
log517
printf517
scanf517
sin517
sprintf517

sqrt517
sscanf517
strtod517
tan517

ABSACC.H

The `ABSACC.H` include file contains definitions for macros that allow you to directly access the different memory areas of the 8051.

CBYTE	FARRAY	PBYTE
CWORD	FCARRAY	PWORD
DBYTE	FCVAR	XBYTE
DWORD	FVAR	XWORD

ASSERT.H

The `ASSERT.H` include file defines the `assert` macro you can use to create test conditions in your programs.

CTYPE.H

The `CTYPE.H` include file contains definitions and prototypes for routines which classify ASCII characters and routines which perform character conversions. The following is a list of these routines:

isalnum	isprint	toint
isalpha	ispunct	tolower
iscentrl	isspace	_tolower
isdigit	isupper	toupper
isgraph	isxdigit	_toupper
islower	toascii	

INTRINS.H

The `INTRINS.H` include file contains prototypes for routines that instruct the compiler to generate in-line intrinsic code.

chkfloat	_irol_	_lror_
crol	_iror_	_nop_
cror	_lrol_	_testbit_

MATH.H

The **MATH.H** include file contains prototypes and definitions for all routines that perform floating-point math calculations. Other math functions are also included in this file. All of the math function routines are listed below:

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

SETJMP.H

The **SETJMP.H** include file defines the **jmp_buf** type and prototypes the **setjmp** and **longjmp** routines which use it.

STDARG.H

The **STDARG.H** include file defines macros that allow you to access arguments in functions with variable-length argument lists. The macros include:

va_arg	va_end	va_start
---------------	---------------	-----------------

In addition, the **va_list** type is defined in this file.

STDDEF.H

The **STDDEF.H** include file defines the **offsetof** macro you can use to determine the offset of members of a structure.

STDIO.H

The **STDIO.H** include file contains prototypes and definitions for stream I/O routines. They are:

getchar	putchar	scanf
_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	init_mempool	strtod
atoi	malloc	strtol
atol	rand	strtoul
calloc	realloc	
free	srand	

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:

memcpy	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 **Cx51** standard library reference. The routines included in the standard library are described here in alphabetical order and each is divided into several sections:

- Summary:** Briefly describes the routine's effect, lists include file(s) containing its declaration and prototype, illustrates the syntax, and describes any arguments.
- Description:** Provides you with a detailed description of the routine and how it is used.
- Return Value:** Describes the value returned by the routine.
- See Also:** Names related routines.
- Example:** Gives a function or program fragment demonstrating proper use of the function.

abs

Summary: `#include <math.h>`
`int abs (`
 `int val);` */* number to take absolute value*
*of */*

Description: The **abs** function determines the absolute value of the integer argument *val*.

Return Value: The **abs** function returns the absolute value of *val*.

See Also: **cabs, fabs, labs**

Example:

```
#include <math.h>
#include <stdio.h>                    /* for printf */

void tst_abs (void) {
    int x;
    int y;

    x = -42;

    y = abs (x);

    printf ("ABS(%d) = %d\n", x, y);
}
```

acos / acos517

Summary: `#include <math.h>`
 float `acos (`
 float `x);` */* number to calculate arc*
 cosine of **/*

Description: The `acos` function calculates the arc cosine of the floating-point number `x`. The value of `x` must be between -1 and 1. The floating-point value returned by `acos` is a number in the 0 to π range.

The `acos517` function is identical to `acos`, but uses the arithmetic unit of the Infineon C517x, C509 to provide faster execution. When using this function, include the header file `80C517.H`. Do not use this routine with a CPU that does not support this feature.

Return Value: The `acos` function returns the arc cosine of `x`.

See Also: `asin`, `atan`, `atan2`

Example:

```
#include <math.h>
#include <stdio.h>                    /* for printf */

void tst_acos (void) {
    float x;
    float y;

    for (x = -1.0; x <= 1.0; x += 0.1) {
        y = acos (x);

        printf ("ACOS(%f) = %f\n", x, y);
    }
}
```

asin / asin517

Summary: `#include <math.h>`
 float asin (
 float x); */* number to calculate arc sine
of */*

Description: The **asin** function calculates the arc sine of the floating-point number *x*. The value of *x* must be in the range -1 to 1. The floating-point value returned by **asin** is a number in the $-\pi/2$ to $\pi/2$ range.

The **asin517** function is identical to **asin**, but uses the arithmetic unit of the Infineon C517x, C509 to provide faster execution. When using this function, include the header file 80C517.H. Do not use this routine with a CPU that does not support this feature.

Return Value: The **asin** function returns the arc sine of *x*.

See Also: **acos, atan, atan2**

Example:

```
#include <math.h>
#include <stdio.h>                    /* for printf */

void tst_asin (void) {
    float x;
    float y;

    for (x = -1.0; x <= 1.0; x += 0.1) {
        y = asin (x);

        printf ("ASIN(%f) = %f\n", x, y);
    }
}
```

assert

Summary: `#include <assert.h>`
`void assert (`
 expression`);`

Description: The **assert** macro tests *expression* and prints a diagnostic message using the **printf** library routine if it is false.

Return Value: None.

Example:

```
#include <assert.h>
#include <stdio.h>

void check_parms (
    char *string)
{
    assert (string != NULL); /* check for NULL ptr */
    printf ("String %s is OK\n", string);
}
```

atan / atan517

Summary: `#include <math.h>`
 float atan (
 float x); /* number to calculate arc
 tangent of */

Description: The **atan** function calculates the arc tangent of the floating-point number *x*. The floating-point value returned by **atan** is a number in the $-\pi/2$ to $\pi/2$ range.

The **atan517** function is identical to **atan**, but uses the arithmetic unit of the Infineon C517x, C509 to provide faster execution. When using this function, include the header file 80C517.H. Do not use this routine with a CPU that does not support this feature.

Return Value: The **atan** function returns the arc tangent of *x*.

See Also: **acos, asin, atan2**

Example:

```
#include <math.h>
#include <stdio.h>                    /* for printf */

void tst_atan (void) {
    float x;
    float y;

    for (x = -10.0; x <= 10.0; x += 0.1) {
        y = atan (x);

        printf ("ATAN(%f) = %f\n", x, y);
    }
}
```

atan2

Summary:

```
#include <math.h>
float atan2 (
    float y,          /* denominator for arc tangent */
    float x);        /* numerator for arc tangent */
```

Description:

The **atan2** function calculates the arc tangent of the floating-point ratio y/x . This function uses the signs of both x and y to determine the quadrant of the return value. The floating-point value returned by **atan2** is a number in the $-\pi$ to π range.

Return Value:

The **atan2** function returns the arc tangent of y/x .

See Also:

acos, asin, atan

Example:

```
#include <math.h>
#include <stdio.h>                /* for printf */

void tst_atan2 () {
    float x;
    float y;
    float z;

    x = -1.0;

    for (y = -10.0; y < 10.0; y += 0.1) {
        z = atan2 (y,x);

        printf ("ATAN2(%f/%f) = %f\n", y, x, z);
    }

    /* z approaches -pi as y goes from -10 to 0 */
    /* z approaches +pi as y goes from +10 to 0 */
}
```

atof / atof517

Summary: `#include <stdlib.h>`
 float `atof (`
 void `*string);` `/* string to convert */`

Description: The **atof** function converts *string* into a floating-point value. The input *string* is a sequence of characters that can be interpreted as a floating-point number. This function stops processing characters from *string* at the first one it cannot recognize as part of the number.

The **atof517** function is identical to `atof`, but uses the arithmetic unit of the Infineon C517x, C509 to provide faster execution. When using this function, include the header file `80C517.H`. Do not use this routine with a CPU that does not support this feature.

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, strtod, strtol, strtoul**

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, strtod, strtol, strtoul`

Example:

```
#include <stdlib.h>
#include <stdio.h>            /* for printf */

void tst_atoi (void) {
    int i;
    char s [] = "12345";

    i = atoi (s);
    printf ("atoi(%s) = %d\n", s, i);
}
```


atol

Summary: `#include <stdlib.h>`
`long atol (`
 `void *string);` `/* string to convert */`

Description: The `atol` function converts *string* into a long integer value. The input *string* is a sequence of characters that can be interpreted as a long. This function stops processing characters from *string* at the first one it cannot recognize as part of the number.

The `atol` function requires *string* to have the following format:

`[whitespace] [{+|-}] 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`, `strtod`, `strtol`, `strtoul`

Example:

```
#include <stdlib.h>
#include <stdio.h>            /* for printf */

void tst_atol (void) {
    long l;
    char s [] = "8003488051";

    l = atol (s);
    printf ("ATOL(%s) = %ld\n", s, l);
}
```

cabs

Summary: `#include <math.h>`
`char cabs (`
 `char val);` /* character to take absolute value of */

Description: The `cabs` function determines the absolute value of the character argument `val`.

Return Value: The `cabs` function returns the absolute value of `val`.

See Also: `abs`, `fabs`, `labs`

Example:

```
#include <math.h>
#include <stdio.h>           /* for printf */

void tst_cabs (void) {
    char x;
    char y;

    x = -23;

    y = cabs (x);

    printf ("CABS(%bd) = %bd\n", x, y);
}
```

calloc

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

Description: The **calloc** function allocates memory for an array of *num* elements. Each element in the array occupies *len* bytes and is initialized to 0. The total number of bytes of memory allocated is $num \times len$.

NOTE

*Source code is provided for this routine in the **LIB** directory. You can modify the source to customize this function for your hardware environment. Refer to “Chapter 6. Advanced Programming Techniques” on page 149 for more information.*

Return Value: The **calloc** function returns a pointer to the allocated memory or a null pointer if the memory allocation request cannot be satisfied.

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

Example:

```
#include <stdlib.h>
#include <stdio.h>                            /* for printf */

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

    p = calloc (100, sizeof (int));

    if (p == NULL)
        printf ("Error allocating array\n");
    else
        printf ("Array address is %p\n", (void *) p);
}
```

ceil

Summary: `#include <math.h>`
 float `ceil (`
 float *val*); */* number to calculate ceiling for */*

Description: The `ceil` function calculates the smallest integer value that is greater than or equal to *val*.

Return Value: The `ceil` function returns a **float** that contains the smallest integer value that is not less than *val*.

See Also: **floor**

Example:

```
#include <math.h>
#include <stdio.h>                                /* for printf */

void tst_ceil (void) {
    float x;
    float y;

    x = 45.998;
    y = ceil (x);

    printf ("CEIL(%f) = %f\n", x, y);

    /* output is "CEIL(45.998) = 46" */
}
```

chkfloat

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

Description: The `_chkfloat_` function checks the status of a floating-point number.

Return Value: The `_chkfloat_` function returns an **unsigned char** that contains the following status information:

Return Value	Meaning
0	Standard floating-point numbers
1	Floating-point value 0
2	+INF (positive overflow)
3	-INF (negative overflow)
4	NaN (Not a Number) error status

Example:

```
#include <intrins.h>
#include <stdio.h> /* for printf */

char _chkfloat_ (float);

float f1, f2, f3;

void tst_chkfloat (void) {
    f1 = f2 * f3;

    switch (_chkfloat_ (f1)) {
        case 0:
            printf ("result is a number\n"); break;
        case 1:
            printf ("result is zero\n"); break;
        case 2:
            printf ("result is +INF\n"); break;
        case 3:
            printf ("result is -INF\n"); break;
        case 4:
            printf ("result is NaN\n"); break;
    }
}
```

cos / cos517

Summary: `#include <math.h>`
`float cos (`
 `float x);` `/* number to calculate cosine`
`for */`

Description: The `cos` function calculates the cosine of the floating-point value `x`. The value of `x` must be between `-65535` and `65535`. Values outside this range result in an **NaN** error.

The `cos517` function is identical to `cos`, but uses the arithmetic unit of the Infineon C517x, C509 to provide faster execution. When using this function, include the header file `80C517.H`. Do not use this routine with a CPU that does not support this feature.

Return Value: The `cos` function returns the cosine for the value `x`.

See Also: `sin`, `tan`

Example:

```
#include <math.h>
#include <stdio.h>                    /* for printf */

void tst_cos (void) {
    float x;
    float y;

    for (x = 0; x < (2 * 3.1415); x += 0.1) {
        y = cos (x);

        printf ("COS(%f) = %f\n", x, y);
    }
}
```

cosh

Summary: `#include <math.h>`
 float cosh (
 float x); /* value for hyperbolic cos
 function */

Description: The **cosh** function calculates the hyperbolic cosine of the floating-point value *x*.

Return Value: The **cosh** function returns the hyperbolic cosine for the value *x*.

See Also: **sinh, tanh**

Example:

```
#include <math.h>
#include <stdio.h>                    /* for printf */

void tst_cosh (void) {
    float x;
    float y;

    for (x = 0; x < (2 * 3.1415); x += 0.1) {
        y = cosh (x);

        printf ("COSH(%f) = %f\n", x, y);
    }
}
```

`_crol_`

Summary:

```
#include <intrins.h>
unsigned char _crol_ (
    unsigned char c,      /* character to rotate left */
    unsigned char b);    /* bit positions to rotate */
```

Description:

The `_crol_` routine rotates the bit pattern for the character *c* left *b* bits. This routine is implemented as an intrinsic function. The code required is included in-line rather than being called.

Return Value:

The `_crol_` routine returns the rotated value of *c*.

See Also:

`_cror_`, `_irol_`, `_iror_`, `_lrol_`, `_lror_`

Example:

```
#include <intrins.h>

void tst_crol (void) {
    char a;
    char b;

    a = 0xA5;

    b = _crol_(a,3);          /* b now is 0x2D */
}
```


cror

Summary: `#include <intrins.h>`
 unsigned char _cror_ (
 unsigned char *c*, */* character to rotate right */*
 unsigned char *b*); */* bit positions to rotate */*

Description: The `_cror_` routine rotates the bit pattern for the character *c* right *b* bits. This routine is implemented as an intrinsic function. The code required is included in-line rather than being called.

Return Value: The `_cror_` routine returns the rotated value of *c*.

See Also: `_crol_`, `_irol_`, `_iror_`, `_lrol_`, `_lror_`

Example:

```
#include <intrins.h>

void tst_crdr (void) {
    char a;
    char b;

    a = 0xA5;

    b = _crol_(a,1);                    /* b now is 0xD2 */
}
```

exp / exp517

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

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

The `exp517` function is identical to `exp`, but uses the arithmetic unit of the Infineon C517x, C509 to provide faster execution. When using this function, include the header file 80C517.H. Do not use this routine with a CPU that does not support this feature.

Return Value: The `exp` function returns the floating-point value e^x .

See Also: `log`, `log10`

Example:

```
#include <math.h>
#include <stdio.h>                    /* for printf */

void tst_exp (void) {
    float x;
    float y;

    x = 4.605170186;

    y = exp (x);                      /* y = 100 */

    printf ("EXP(%f) = %f\n", x, y);
}
```

fabs

Summary: `#include <math.h>`
`float fabs (`
`float val);` /* number to calc absolute value for */

Description: The `fabs` function determines the absolute value of the floating-point number `val`.

Return Value: The `fabs` function returns the absolute value of `val`.

See Also: `abs`, `cabs`, `labs`

Example:

```
#include <math.h>
#include <stdio.h>                    /* for printf */

void tst_fabs (void) {
    float x;
    float y;

    x = 10.2;
    y = fabs (x);
    printf ("FABS(%f) = %f\n", x, y);

    x = -3.6;
    y = fabs (x);
    printf ("FABS(%f) = %f\n", x, y);
}
```

floor

Summary: `#include <math.h>`
float floor (
 float val); /* value for floor function */

Description: The **floor** function calculates the largest integer value that is less than or equal to *val*.

Return Value: The **floor** function returns a **float** that contains the largest integer value that is not greater than *val*.

See Also: **ceil**

Example:

```
#include <math.h>
#include <stdio.h>                    /* for printf */

void tst_floor (void) {
    float x;
    float y;

    x = 45.998;

    y = floor (x);

    printf ("FLOOR(%f) = %f\n", x, y);  * prints 45 */
}
```

fmod

Summary: `#include <math.h>`
`float fmod (`
 `float x, /* value to calculate modulo for */`
 `float y); /* integer portion of modulo */`

Description: The **fmod** function returns a value f such that f has the same sign as x , the absolute value of f is less than the absolute value of y , and there exists an integer k such that $k*y+f$ equals x . If the quotient x/y cannot be represented, the result is undefined.

Return Value: The **fmod** function returns the floating-point remainder of x/y .

Example:

```
#include <math.h>
#include <stdio.h>                                   /* for printf */

void tst_fmod (void) {
    float f;

    f = fmod (15.0, 4.0);
    printf ("fmod (15.0, 4.0) = %f\n", f);
}
```

free

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

Description: The `free` function returns a memory block to the memory pool. The `p` argument points to a memory block allocated with the `calloc`, `malloc`, or `realloc` functions. Once it has been returned to the memory pool by the `free` function, the block is available for subsequent allocation.

If `p` is a null pointer, it is ignored.

NOTE

Source code for this routine is located in the folder `\KEIL\C51\LIB`. You may modify the source to customize this function for your particular hardware environment. Refer to “Chapter 6. Advanced Programming Techniques” on page 149 for more information.

Return Value: None.

See Also: `calloc`, `init_mempool`, `malloc`, `realloc`

Example:

```
#include <stdlib.h>
#include <stdio.h>            /* for printf */

void tst_free (void) {
    void *mbuf;

    printf ("Allocating memory\n");
    mbuf = malloc (1000);

    if (mbuf == NULL) {
        printf ("Unable to allocate memory\n");
    }
    else {
        free (mbuf);
        printf ("Memory free\n");
    }
}
```

getchar

Summary: `#include <stdio.h>`
`char getchar (void);`

Description: The **getchar** function reads a single character from the input stream using the **_getkey** function. The character read is then passed to the **putchar** function to be echoed.

NOTE

*This function is implementation-specific and is based on the operation of the **_getkey** and/or **putchar** functions. These functions, as provided in the standard library, read and write characters using the serial port of the 8051. Custom functions may use other I/O devices.*

Return Value: The **getchar** function returns the character read.

See Also: **_getkey, putchar, ungetchar**

Example:

```
#include <stdio.h>

void tst_getchar (void) {
    char c;

    while ((c = getchar ()) != 0x1B) {
        printf ("character = %c %bu %bx\n", c, c, c);
    }
}
```

_getkey

Summary: `#include <stdio.h>`
`char _getkey (void);`

Description: The `_getkey` function waits for a character to be received from the serial port.

NOTE

This routine is implementation-specific, and its function may deviate from that described above. Source is included for the `_getkey` and `putchar` functions which may be modified to provide character level I/O for any hardware device. Refer to “Customization Files” on page 150 for more information.

Return Value: The `_getkey` routine returns the received character.

See Also: `getchar`, `putchar`, `ungetchar`

Example:

```
#include <stdio.h>

void tst_getkey (void) {
    char c;

    while ((c = _getkey ()) != 0x1B) {
        printf ("key = %c %bu %bx\n", c, c, c);
    }
}
```


gets

Summary: `#include <stdio.h>`
 `char *gets (`
 `char *string, /* string to read */`
 `int len); /* maximum characters to read`
 `*/`

Description: The **gets** function calls the **getchar** function to read a line of characters into *string*. The line consists of all characters up to and including the first newline character (`'\n'`). The newline character is replaced by a null character (`'\0'`) in *string*.

The *len* argument specifies the maximum number of characters that may be read. If *len* characters are read before a newline is encountered, the **gets** function terminates *string* with a null character and returns.

NOTE

*This function is implementation-specific and is based on the operation of the **_getkey** and/or **putchar** functions. These functions, as provided in the standard library, read and write characters using the serial port of the 8051. Custom functions may use other I/O devices.*

Return Value: The **gets** function returns *string*.

See Also: **printf, puts, scanf**

Example:

```
#include <stdio.h>

void tst_gets (void) {
    xdata char buf [100];

    do {
        gets (buf, sizeof (buf));
        printf ("Input string \"%s\"", buf);
    } while (buf [0] != '\0');
}
```

init_mempool

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

Description: The `init_mempool` function initializes the memory management routines and provides the starting address and size of the memory pool. The `p` argument points to a memory area in `xdata` which is managed using the `calloc`, `free`, `malloc`, and `realloc` library functions. The `size` argument specifies the number of bytes to use for the memory pool.

NOTE

This function must be used to setup the memory pool before any other memory management functions (`calloc`, `free`, `malloc`, `realloc`) can be called. Call the `init_mempool` function only once at the beginning of your program.

Source code is provided for this routine in the folder `\\KEIL\C51\LIB`. You can modify the source to customize this function for your hardware environment. Refer to “Chapter 6. Advanced Programming Techniques” on page 149 for more information.

Return Value: None.

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

Example:

```
#include <stdlib.h>

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

    init_mempool (&XBYTE [0x2000], 0x1000);
    /* initialize memory pool at xdata 0x2000
       for 4096 bytes */

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

`_irol_`

Summary: `#include <intrins.h>`
 `unsigned int _irol_ (`
 `unsigned int i, /* integer to rotate left */`
 `unsigned char b); /* bit positions to rotate */`

Description: The `_irol_` routine rotates the bit pattern for the integer *i* left *b* bits. This routine is implemented as an intrinsic function. The code required is included in-line rather than being called.

Return Value: The `_irol_` routine returns the rotated value of *i*.

See Also: `_cror_`, `_crol_`, `_iror_`, `_lrol_`, `_lror_`

Example:

```
#include <intrins.h>

void tst_irol (void) {
    int a;
    int b;

    a = 0xA5A5;

    b = _irol_(a,3);                    /* b now is 0x2D2D */
}
```

`_iror_`

Summary: `#include <intrins.h>`
`unsigned int _iror_ (`
 `unsigned int i, /* integer to rotate right */`
 `unsigned char b); /* bit positions to rotate */`

Description: The `_iror_` routine rotates the bit pattern for the integer *i* right *b* bits. This routine is implemented as an intrinsic function. The code required is included in-line rather than being called.

Return Value: The `_iror_` routine returns the rotated value of *i*.

See Also: `_cror_`, `_crol_`, `_irol_`, `_lrol_`, `_lror_`

Example:

```
#include <intrins.h>

void tst_iror (void) {
    int a;
    int b;

    a = 0xA5A5;

    b = _irol_(a,1);                   /* b now is 0xD2D2 */
}
```


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);
    }
}
```

isctrl

Summary: **#include <ctype.h>**
bit isctrl (
 char c); */* character to test */*

Description: The **isctrl** function tests *c* to determine if it is a control character (0x00-0x1F or 0x7F).

Return Value: The **isctrl** 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_isctrl (void) {
    unsigned char i;
    char *p;

    for (i = 0; i < 128; i++) {
        p = (isctrl (i) ? "YES" : "NO");

        printf ("isctrl (%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`, `isctrl`, `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`, `isctrl`, `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`, `isctrl`, `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`, `isctrl`, `isdigit`, `isgraph`, `islower`, `isprint`, `ispunct`, `isspace`, `isxdigit`

Example:

```
#include <ctype.h>
#include <stdio.h>                    /* for printf */

void tst_isupper (void) {
    unsigned char i;
    char *p;

    for (i = 0; i < 128; i++) {
        p = (isupper (i) ? "YES" : "NO");

        printf ("isupper (%c) %s\n", i, p);
    }
}
```

isxdigit

Summary: `#include <ctype.h>`
 bit `isxdigit (`
 char `c);` `/* character to test */`

Description: The `isxdigit` function tests `c` to determine if it is a hexadecimal digit ('A'-'Z', 'a'-'z', '0'-'9').

Return Value: The `isxdigit` function returns a value of 1 if `c` is a hexadecimal digit and a value of 0 if it is not.

See Also: `isalnum`, `isalpha`, `isctrl`, `isdigit`, `isgraph`, `islower`, `isprint`, `ispunct`, `isspace`, `isupper`

Example:

```
#include <ctype.h>
#include <stdio.h>                    /* for printf */

void tst_isxdigit (void) {
    unsigned char i;
    char *p;

    for (i = 0; i < 128; i++) {
        p = (isxdigit (i) ? "YES" : "NO");

        printf ("isxdigit (%c) %s\n", i, p);
    }
}
```

labs

Summary: `#include <math.h>`
 `long labs (`
 `long val);` */* value to calc. abs. value for */*

Description: The `labs` function determines the absolute value of the long integer `val`.

Return Value: The `labs` function returns the absolute value of `val`.

See Also: `abs, cabs, fabs`

Example:

```
#include <math.h>
#include <stdio.h>                    /* for printf */

void tst_labs (void) {
    long x;
    long y;

    x = -12345L;

    y = labs (x);

    printf ("LABS(%ld) = %ld\n", x, y);
}
```


log / log517

Summary: `#include <math.h>`
`float log (`
 `float val);` /* value to take natural logarithm of */

Description: The **log** function calculates the natural logarithm for the floating-point number *val*. The natural logarithm uses the base *e* or 2.718282...

The **log517** function is identical to **log**, but uses the arithmetic unit of the Infineon C517x, C509 to provide faster execution. When using this function, include the header file 80C517.H. Do not use this routine with a CPU that does not support this feature.

Return Value: The **log** function returns the floating-point natural logarithm of *val*.

See Also: **exp, log10**

Example:

```
#include <math.h>
#include <stdio.h>                     /* for printf */

void tst_log (void) {
    float x;
    float y;

    x = 2.71838;
    x *= x;

    y = log (x);                        /* y = 2 */

    printf ("LOG(%f) = %f\n", x, y);
}
```

log10 / log10517

Summary: `#include <math.h>`
 float log10 (
 float val); /* value to take common logarithm of */

Description: The **log10** function calculates the common logarithm for the floating-point number *val*. The common logarithm uses base 10.

The **log10517** function is identical to `log10`, but uses the arithmetic unit of the Infineon C517x, C509 to provide faster execution. When using this function, include the header file `80C517.H`. Do not use this routine with a CPU that does not support this feature.

Return Value: The **log10** function returns the floating-point common logarithm of *val*.

See Also: **exp, log**

Example:

```
#include <math.h>
#include <stdio.h>                    /* for printf */

void tst_log10 (void) {
    float x;
    float y;

    x = 1000;

    y = log10 (x);                    /* y = 3 */

    printf ("LOG10(%f) = %f\n", x, y);
}
```

longjmp

Summary:

```
#include <setjmp.h>
void longjmp (
    jmp_buf env, /* environment to restore */
    int retval); /* return value */
```

Description: The **longjmp** function restores the state which was previously stored in *env* by the **setjmp** function. The *retval* argument specifies the value to return from the **setjmp** function call.

The **longjmp** and **setjmp** functions can be used to execute a non-local goto and are usually utilized to pass control to an error recovery routine.

Local variables and function arguments are restored only if declared with the **volatile** attribute.

Return Value: None.

See Also: **setjmp**

Example:

```
#include <setjmp.h>
#include <stdio.h>                /* for printf */

jmp_buf env; /* jump environment (must be global) */
bit error_flag;

void trigger (void) {
    .
    .
    .
    /* put processing code here */
    .
    .
    if (error_flag != 0) {
        longjmp (env, 1);      /* return 1 to setjmp */
    }
    .
    .
}

void recover (void) {
    /* put recovery code here */
}

void tst_longjmp (void) {
    .
    .
    .
    if (setjmp (env) != 0) { /* setjmp returns a 0 */
        printf ("LONGJMP called\n");
        recover ();
    }
    else {
        printf ("SETJMP called\n");

        error_flag = 1;        /* force an error */

        trigger ();
    }
}
```

lrol

Summary: **#include <intrins.h>**
 unsigned long _lrol_ (
 unsigned long l, /* 32-bit integer to rotate left */
 unsigned char b); /* bit positions to rotate */

Description: The **_lrol_** routine rotates the bit pattern for the long integer *l* left *b* bits. This routine is implemented as an intrinsic function. The code required is included in-line rather than being called.

Return Value: The **_lrol_** routine returns the rotated value of *l*.

See Also: **_cror_, _crol_, _irol_, _iror_, _lror_**

Example:

```
#include <intrins.h>

void tst_lrol (void) {
    long a;
    long b;

    a = 0xA5A5A5A5;

    b = _lrol_(a,3);            /* b now is 0x2D2D2D2D */
}
```

`_lror_`

Summary: `#include <intrins.h>`
`unsigned long _lror_ (`
 `unsigned long l, /* 32-bit int to rotate right */`
 `unsigned char b); /* bit positions to rotate */`

Description: The `_lror_` routine rotates the bit pattern for the long integer `l` right `b` bits. This routine is implemented as an intrinsic function. The code required is included in-line rather than being called.

Return Value: The `_lror_` routine returns the rotated value of `l`.

See Also: `_cror_`, `_crol_`, `_irol_`, `_iror_`, `_lrol_`

Example:

```
#include <intrins.h>

void tst_lror (void) {
    long a;
    long b;

    a = 0xA5A5A5A5;

    b = _lrol_(a,1);           /* b now is 0xD2D2D2D2 */
}
```

malloc

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

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

NOTE

Source code is provided for this routine in the `\KEIL\C51\LIB` directory. You may modify the source to customize this function for your hardware environment. Refer to “Chapter 6. Advanced Programming Techniques” on page 149 for more information.

Return Value: The **malloc** function returns a pointer to the allocated block or a null pointer if there was not enough memory to satisfy the allocation request.

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

Example:

```
#include <stdlib.h>
#include <stdio.h>                /* for printf */

void tst_malloc (void) {
    unsigned char xdata *p;

    p = malloc (1000);           /* allocate 1000 bytes */

    if (p == NULL)
        printf ("Not enough memory space\n");
    else
        printf ("Memory allocated\n");
}
```

memccpy

Summary: `#include <string.h>`
 `void *memccpy (`
 `void *dest, /* destination buffer */`
 `void *src, /* source buffer */`
 `char c, /* character which ends copy */`
 `int len); /* maximum bytes to copy */`

Description: The `memccpy` function copies 0 or more characters from `src` to `dest`. Characters are copied until the character `c` is copied or until `len` bytes have been copied, whichever comes first.

Return Value: The `memccpy` function returns a pointer to the byte in `dest` that follows the last character copied or a null pointer if the last character copied was `c`.

See Also: `memchr`, `memcmp`, `memcpy`, `memmove`, `memset`

Example:

```
#include <string.h>
#include <stdio.h>                    /* for printf */

void tst_memccpy (void) {
    static char src1 [100] = "Copy this string
                           to dst1";
    static char dst1 [100];

    void *c;

    c = memccpy (dst1, src1, 'g', sizeof (dst1));

    if (c == NULL)
        printf ("'g' was not found in the src
                buffer\n");
    else
        printf ("characters copied up to 'g'\n");
}
```


memchr

- Summary:** `#include <string.h>`
`void *memchr (`
 `void *buf, /* buffer to search */`
 `char c, /* byte to find */`
 `int len); /* maximum buffer length */`
- Description:** The **memchr** function scans *buf* for the character *c* in the first *len* bytes of the buffer.
- Return Value:** The **memchr** function returns a pointer to the character *c* in *buf* or a null pointer if the character was not found.
- See Also:** **memcpy, memcmp, memcopy, memmove, memset**

Example:

```
#include <string.h>
#include <stdio.h>                        /* for printf */

void tst_memchr (void) {
    static char srcl [100] =
        "Search this string from the start";

    void *c;

    c = memchr (srcl, 'g', sizeof (srcl));

    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	<i>buf1</i> less than <i>buf2</i>
= 0	<i>buf1</i> equal to <i>buf2</i>
> 0	<i>buf1</i> greater than <i>buf2</i>

Return Value:

The **memcmp** function returns a positive, negative, or zero value indicating the relationship of *buf1* and *buf2*.

See Also:

memcmp, memchr, memmove, memset

Example:

```
#include <string.h>
#include <stdio.h>                                /* for printf */

void tst_memcmp (void) {
    static char hexchars [] = "0123456789ABCDEF";
    static char hexchars2 [] = "0123456789abcdef";

    char i;

    i = memcmp (hexchars, hexchars2, 16);

    if (i < 0)
        printf ("hexchars < hexchars2\n");

    else if (i > 0)
        printf ("hexchars > hexchars2\n");

    else
        printf ("hexchars == hexchars2\n");
}
```

memcpy

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

Description: The **memcpy** function copies *len* bytes from *src* to *dest*. If these memory buffers overlap, the **memcpy** function cannot guarantee that bytes in *src* are copied to *dest* before being overwritten. If these buffers do overlap, use the **memmove** function.

Return Value: The **memcpy** function returns *dest*.

See Also: **memcpy, 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: `memcpy`, `memchr`, `memcmp`, `memcpy`, `memset`

Example:

```
#include <string.h>
#include <stdio.h>                                /* for printf */

void tst_memmove (void) {
    static char buf [] = "This is line 1 "
                        "This is line 2 "
                        "This is line 3 ";

    printf ("buf before = %s\n", buf);

    memmove (&buf [0], &buf [16], 32);

    printf ("buf after  = %s\n", buf);
}
```

memset

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

Description: The `memset` function sets the first `len` bytes in `buf` to `c`.

Return Value: The `memset` function returns `dest`.

See Also: `memcpy`, `memchr`, `memcmp`, `memcpy`, `memmove`

Example:

```
#include <string.h>
#include <stdio.h>                            /* for printf */

void tst_memset (void) {
    char buf [10];

    memset (buf, '\0', sizeof (buf));
           /* fill buffer with null characters */
}
```

modf

Summary: `#include <math.h>`
`float modf (`
 `float val, /* value to calculate modulo for */`
 `float *ip); /* integer portion of modulo */`

Description: The `modf` function splits the floating-point number `val` into integer and fractional components. The fractional part of `val` is returned as a signed floating-point number. The integer part is stored as a floating-point number at `ip`.

Return Value: The `modf` function returns the signed fractional part of `val`.

Example:

```
#include <math.h>
#include <stdio.h>                                /* for printf */

void tst_modf (void) {
    float x;
    float int_part, frc_part;

    x = 123.456;

    frc_part = modf (x, &int_part);

    printf ("%f = %f + %f\n", x, int_part, frc_part);
}
```

`_nop_`

Summary: `#include <intrins.h>`
 `void _nop_ (void);`

Description: The `_nop_` routine inserts an 8051 NOP instruction into the program. This routine can be used to pause for 1 CPU cycle. This routine is implemented as an intrinsic function. The code required is included in-line rather than being called.

Return Value: None.

Example:

```
#include <intrins.h>
#include <stdio.h>                    /* for printf */

void tst_nop (void) {

    P1 = 0xFF;

    _nop_ ();                         /* delay for hardware */
    _nop_ ();
    _nop_ ();

    P1 = 0x00;

}
```

offsetof

Summary:

```
#include <stddef.h>
int offsetof (
    structure,           /* structure to use */
    member);           /* member to get offset for */
```

Description:

The **offsetof** macro calculates the offset of the *member* structure element from the beginning of the structure. The *structure* argument must specify the name of a structure. The *member* argument must specify the name of a member of the structure.

Return Value:

The **offsetof** macro returns the offset, in bytes, of the *member* element from the beginning of **struct** *structure*.

Example:

```
#include <stddef.h>

struct index_st
{
    unsigned char type;
    unsigned long num;
    unsigned int len;
};

typedef struct index_st index_t;

void main (void)
{
    int x, y;

    x = offsetof (struct index_st, len); /* x = 5 */
    y = offsetof (index_t, num);        /* x = 1 */
}
```


pow

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

Description: The **pow** function calculates x raised to the y th power.

Return Value: The **pow** function returns the value x^y . If $x \neq 0$ and $y = 0$, **pow** returns a value of 1. If $x = 0$ and $y \leq 0$, **pow** returns NaN. If $x < 0$ and y is not an integer, **pow** returns NaN.

See Also: **sqrt**

Example:

```
#include <math.h>
#include <stdio.h>                         /* for printf */

void tst_pow (void) {
    float base;
    float power;
    float y;

    base = 2.0;
    power = 8.0;

    y = pow (base, power);                 /* y = 256 */

    printf ("%f ^ %f = %f\n", base, power, y);
}
```

printf / printf517

Summary:

```
#include <stdio.h>
int printf (
    const char *fmtstr /* format string */
    [, arguments]...); /* additional arguments */
```

Description: The **printf** function formats a series of strings and numeric values and builds a string to write to the output stream using the **putchar** function. The *fmtstr* argument is a format string and may be composed of characters, escape sequences, and format specifications.

Ordinary characters and escape sequences are copied to the stream in the order in which they are interpreted. Format specifications always begin with a percent sign ('%') and require additional *arguments* to be included in the function call.

The format string is read from left to right. The first format specification encountered references the first argument after *fmtstr* and converts and outputs it using the format specification. The second format specification accesses the second argument after *fmtstr*, and so on. If there are more arguments than format specifications, the extra arguments are ignored. Results are unpredictable if there are not enough arguments for the format specifications.

Format specifications have the following format:

```
% [flags] [width] [.precision] [{b | B | l | L}] type
```

Each field in the format specification can be a single character or a number which specifies a particular format option.

The *type* field is a single character that specifies whether the argument is interpreted as a character, string, number, or pointer, as shown in the following table.

Character	Argument Type	Output Format
d	int	Signed decimal number
u	unsigned int	Unsigned decimal number
o	unsigned int	Unsigned octal number
x	unsigned int	Unsigned hexadecimal number using "0123456789abcdef"
X	unsigned int	Unsigned hexadecimal number using "0123456789ABCDEF"
f	float	Floating-point number using the format [-]ddd.dddd
e	float	Floating-point number using the format [-]d.ddde[-]dd
E	float	Floating-point number using the format [-]d.dddE[-]dd
g	float	Floating-point number using either e or f format, whichever is more compact for the specified value and precision
G	float	Identical to the g format except that (where applicable) E precedes the exponent instead of e
c	char	Single character
s	generic *	String with a terminating null character
p	generic *	Pointer using the format <i>t.aaaa</i> where <i>t</i> is the memory type the pointer references (c: code, i: data/idata, x: xdata, p: pdata) and <i>aaaa</i> is the hexadecimal address

The optional characters **b** or **B** and **l** or **L** may immediately precede the type character to respectively specify **char** or **long** versions of the integer types **d**, **i**, **u**, **o**, **x**, and **X**.

The *flags* field is a single character used to justify the output and to print +/- signs and blanks, decimal points, and octal and hexadecimal prefixes, as shown in the following table.

Flag	Meaning
-	Left justify the output in the specified field width.
+	Prefix the output value with a + or - sign if the output is a signed type.
blank (' ')	Prefix the output value with a blank if it is a signed positive value. Otherwise, no blank is prefixed.
#	Prefixes a non-zero output value with 0, 0x, or 0X when used with o, x, and X field types, respectively. When used with the e, E, f, g, and G field types, the # flag forces the output value to include a decimal point. The # flag is ignored in all other cases.
*	Ignore format specifier.

The *width* field is a non-negative number that specifies the minimum number of characters printed. If the number of characters in the output value is less than *width*, blanks are added on the left or right (when the - flag is specified) to pad to the minimum width. If *width* is prefixed with a '0', zeros are padded instead of blanks. The *width* field never truncates a field. If the length of the output value exceeds the specified width, all characters are output.

The *width* field may be an asterisk (**), in which case an **int** argument from the argument list provides the width value. Specifying a 'b' in front of the asterisk specifies that the argument used is an **unsigned char**.

The *precision* field is a non-negative number that specifies the number of characters to print, the number of significant digits, or the number of decimal places. The *precision* field can cause truncation or rounding of the output value in the case of a floating-point number as specified in the following table.

Type	Meaning of Precision Field
d, u, o, x, X	The <i>precision</i> field is where you specify the minimum number of digits that are included in the output value. Digits are not truncated if the number of digits in the argument exceeds that defined in the <i>precision</i> field. If the number of digits in the argument is less than the <i>precision</i> field, the output value is padded on the left with zeros.
f	The <i>precision</i> field is where you specify the number of digits to the right of the decimal point. The last digit is rounded.
e, E	The <i>precision</i> field is where you specify the number of digits to the right of the decimal point. The last digit is rounded.
g, G	The <i>precision</i> field is where you specify the maximum number of significant digits in the output value.
c, p	The <i>precision</i> field has no effect on these field types.
s	The <i>precision</i> field is where you specify the maximum number of characters in the output value. Excess characters are not output.

The *precision* field may be an asterisk (*), in which case an **int** argument from the argument list provides the value for the precision. Specifying a **'b'** in front of the asterisk specifies that the argument used is an **unsigned char**.

The **printf517** function is identical to **printf**, but uses the arithmetic unit of the Infineon C517x, C509 to provide faster execution. When using this function, include the header file 80C517.H. Do not use this routine with a CPU that does not support this feature.

NOTE

*This function is implementation-specific and is based on the operation of the **putchar** function. This function, as provided in the standard library, writes characters using the serial port of the 8051. Custom functions may use other I/O devices.*

*You must ensure that the argument type matches that of the format specification. You can use type casts to ensure that the proper type is passed to **printf**.*

*The total number of bytes that may be passed to **printf** is limited due to the memory restrictions imposed by the 8051. A maximum of 15 bytes may be passed in **SMALL** model or **COMPACT** model. A maximum of 40 bytes may be passed in **LARGE** model.*

Return Value: The **printf** function returns the number of characters actually written to the output stream.

See Also: **gets, puts, scanf, sprintf, sscanf, vprintf, vsprintf**

Example:

```
#include <stdio.h>

void tst_printf (void) {
    char a;
    int b;
    long c;
    unsigned char x;
    unsigned int y;
    unsigned long z;
    float f,g;
    char buf [] = "Test String";
    char *p = buf;

    a = 1;
    b = 12365;
    c = 0x7FFFFFFF;
    x = 'A';
    y = 54321;
    z = 0x4A6F6E00;
    f = 10.0;
    g = 22.95;

    printf ("char %bd int %d long %ld\n",a,b,c);
    printf ("Uchar %bu Uint %u Ulong %lu\n",x,y,z);
    printf ("xchar %bx xint %x xlong %lx\n",x,y,z);
    printf ("String %s is at address %p\n",buf,p);
    printf ("%f != %g\n", f, g);
    printf ("%*f != %*g\n", 8, f, 8, g);
}
```

putchar

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

Description: The **putchar** function transmits the character *c* using the 8051 serial port.

NOTE

*This routine is implementation-specific and its function may deviate from that described above. Source is included for the **_getkey** and **putchar** functions which may be modified to provide character level I/O for any hardware device. Refer to “Customization Files” on page 150 for more information.*

Return Value: The **putchar** routine returns the character output, *c*.

See Also: **getchar, _getkey, ungetchar**

Example:

```
#include <stdio.h>

void tst_putchar (void) {
    unsigned char i;

    for (i = 0x20; i < 0x7F; i++)
        putchar (i);
}
```


puts

Summary: `#include <stdio.h>`
`int puts (`
 `const char *string);` /* string to output */

Description: The **puts** function writes *string* followed by a newline character ('`\n`') to the output stream using the **putchar** function.

NOTE

*This function is implementation-specific and is based on the operation of the **putchar** function. This function, as provided in the standard library, writes characters using the serial port of the 8051. Custom functions may use other I/O devices.*

Return Value: The **puts** function returns **EOF** if an error occurred and a value of 0 if no errors were encountered.

See Also: **gets, printf, scanf**

Example:

```
#include <stdio.h>

void tst_puts (void) {

    puts ("Line #1");
    puts ("Line #2");
    puts ("Line #3");

}
```

rand

Summary: `#include <stdlib.h>`
`int rand (void);`

Description: The **rand** function generates a pseudo-random number between 0 and 32767.

Return Value: The **rand** function returns a pseudo-random number.

See Also: **srand**

Example:

```
#include <stdlib.h>
#include <stdio.h>           /* for printf */

void tst_rand (void) {
    int i;
    int r;

    for (i = 0; i < 10; i++) {
        printf ("I = %d, RAND = %d\n", i, rand ());
    }
}
```

realloc

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

Description: The **realloc** function changes the size of a previously allocated memory block. The *p* argument points to the allocated block and the *size* argument specifies the new size for the block. The contents of the existing block are copied to the new block. Any additional area in the new block, due to a larger block size, is not initialized.

NOTE

Source code is provided for this routine in the folder \KEIL\C51\LIB. You can modify the source to customize this function for your hardware environment. Refer to “Chapter 6. Advanced Programming Techniques” on page 149 for more information.

Return Value: The **realloc** function returns a pointer to the new block. If there is not enough memory in the memory pool to satisfy the memory request, a null pointer is returned and the original memory block is not affected.

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

Example:

```
#include <stdlib.h>
#include <stdio.h>                   /* for printf */

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

    p = malloc (100);
    if (p != NULL) {
        new_p = realloc (p, 200);

        if (new_p != NULL) p = new_p;
        else printf ("Reallocation failed\n");
    }
}
```

scanf

Summary:

```
#include <stdio.h>
int scanf (
    const char *fmtstr    /* format string */
    [, argument]...);    /* additional arguments */
```

Description:

The **scanf** function reads data using the **getchar** routine. Data input are stored in the locations specified by *argument* according to the format string *fmtstr*. Each *argument* must be a pointer to a variable that corresponds to the type defined in *fmtstr* which controls the interpretation of the input data. The *fmtstr* argument is composed of one or more whitespace characters, non-whitespace characters, and format specifications as defined below.

The **scanf517** function is identical to **scanf**, but uses the arithmetic unit of the Infineon C517x, C509 to provide faster execution. When using this function include the header file 80C517.H. Do not use this routine with a CPU that does not support this feature.

- Whitespace characters, blank (' '), tab ('\t'), or newline ('\n'), causes **scanf** to skip whitespace characters in the input stream. A single whitespace character in the format string matches 0 or more whitespace characters in the input stream.
- Non-whitespace characters, with the exception of the percent sign ('%'), cause **scanf** to read but not store a matching character from the input stream. The **scanf** function terminates if the next character in the input stream does not match the specified non-whitespace character.
- Format specifications begin with a percent sign ('%') and cause **scanf** to read and convert characters from the input stream to the specified type values. The converted value is stored to an *argument* in the parameter list. Characters following a percent sign that are not recognized as a format specification are treated as an ordinary character. For example, %% matches a single percent sign in the input stream.

The format string is read from left to right. Characters that are not part of the format specifications must match characters in the input stream. These characters are read from the input stream but are discarded and not stored. If a character in the input stream conflicts with the format string, **scanf** terminates. Any conflicting characters remain in the input stream.

The first format specification encountered in the format string references the first argument after *fmtstr* and converts input characters and stores the value using the format specification. The second format specification accesses the second argument after *fmtstr*, and so on. If there are more arguments than format specifications, the extra arguments are ignored. Results are unpredictable if there are not enough arguments for the format specifications.

Values in the input stream are called input fields and are delimited by whitespace characters. When converting input fields, **scanf** ends a conversion for an argument when a whitespace character is encountered. Additionally, any unrecognized character for the current format specification ends a field conversion.

Format specifications have the following format:

% [*****] [*width*] [**{b | h | l}**] *type*

Each field in the format specification can be a single character or a number which specifies a particular format option.

The *type* field is where a single character specifies whether input characters are interpreted as a character, string, or number. This field can be any one of the characters in the following table.

Character	Argument Type	Input Format
d	int *	Signed decimal number
i	int *	Signed decimal, hexadecimal, or octal integer
u	unsigned int *	Unsigned decimal number

Character	Argument Type	Input Format
o	unsigned int *	Unsigned octal number
x	unsigned int *	Unsigned hex number
e	float *	Floating-point number
f	float *	Floating-point number
g	float *	Floating-point number
c	char *	A single character
s	char *	A string of characters terminated by whitespace

An asterisk (*) as the first character of a format specification causes the input field to be scanned but not stored. The asterisk suppresses assignment of the format specification.

The *width* field is a non-negative number that specifies the maximum number of characters read from the input stream. No more than *width* characters are read from the input stream and converted for the corresponding *argument*. However, fewer than *width* characters may be read if a whitespace character or an unrecognized character is encountered first.

The optional characters **b**, **h**, and **l** may immediately precede the type character to respectively specify **char**, **short**, or **long** versions of the integer types **d**, **i**, **u**, **o**, and **x**.

NOTE

*This function is implementation-specific and is based on the operation of the **_getkey** and/or **putchar** functions. These functions, as provided in the standard library, read and write characters using the serial port of the 8051. Custom functions may use other I/O devices.*

*The total number of bytes that may be passed to **scanf** is limited due to the memory restrictions imposed by the 8051. A maximum of 15 bytes may be passed in **SMALL** model or **COMPACT** model. A maximum of 40 bytes may be passed in **LARGE** model.*

Return Value: The **scanf** function returns the number of input fields that were successfully converted. An **EOF** is returned if an error is encountered.

See Also: **gets, printf, puts, sprintf, sscanf, vprintf, vsprintf**

Example:

```
#include <stdio.h>

void tst_scanf (void) {
    char a;
    int b;
    long c;

    unsigned char x;
    unsigned int y;
    unsigned long z;

    float f,g;

    char d, buf [10];

    int argsread;

    printf ("Enter a signed byte, int, and long\n");
    argsread = scanf ("%bd %d %ld", &a, &b, &c);
    printf ("%d arguments read\n", argsread);

    printf ("Enter an unsigned byte, int, and long\n");
    argsread = scanf ("%bu %u %lu", &x, &y, &z);
    printf ("%d arguments read\n", argsread);

    printf ("Enter a character and a string\n");
    argsread = scanf ("%c %9s", &d, buf);
    printf ("%d arguments read\n", argsread);

    printf ("Enter two floating-point numbers\n");
    argsread = scanf ("%f %f", &f, &g);
    printf ("%d arguments read\n", argsread);
}
```

setjmp

Summary: `#include <setjmp.h>`
 `int setjmp (`
 `jmp_buf env);` `/* current environment */`

Description: The **setjmp** function saves the current state of the CPU in *env*. The state can be restored by a subsequent call to the **longjmp** function. When used together, the **setjmp** and **longjmp** functions provide you with a way to execute a non-local goto.

A call to the **setjmp** function saves the current instruction address as well as other CPU registers. A subsequent call to the **longjmp** function restores the instruction pointer and registers, and execution resumes at the point just after the **setjmp** call.

Local variables and function arguments are restored only if declared with the **volatile** attribute.

Return Value: The **setjmp** function returns a value of 0 when the current state of the CPU has been copied to *env*. A non-zero value indicates that the **longjmp** function was executed to return to the **setjmp** function call. In such a case, the return value is the value passed to the **longjmp** function.

See Also: **longjmp**

Example: See **longjmp**

sin / sin517

Summary: `#include <math.h>`
 `float sin (`
 `float x);` `/* value to calculate sine for */`

Description: The **sin** function calculates the sine of the floating-point value *x*. The value of *x* must be in the -65535 to +65535 range or an NaN error value is generated.

The **sin517** function is identical to **sin**, but uses the arithmetic unit of the Infineon C517x, C509 to provide faster execution. When using this function include the header file 80C517.H. Do not use this routine with a CPU that does not support this feature.

Return Value: The **sin** function returns the sine of *x*.

See Also: **cos, tan**

Example:

```
#include <math.h>
#include <stdio.h>                    /* for printf */

void tst_sin (void) {
    float x;
    float y;

    for (x = 0; x < (2 * 3.1415); x += 0.1) {
        y = sin (x);

        printf ("SIN(%f) = %f\n", x, y);
    }
}
```

sinh

Summary: `#include <math.h>`
`float sinh (`
 `float val);` /* value to calc hyperbolic sine for */

Description: The `sinh` function calculates the hyperbolic sine of the floating-point value `x`. The value of `x` must be in the -65535 to +65535 range or an NaN error value is generated.

Return Value: The `sinh` function returns the hyperbolic sine of `x`.

See Also: `cosh`, `tanh`

Example:

```
#include <math.h>
#include <stdio.h>           /* for printf */

void tst_sinh (void) {
    float x;
    float y;

    for (x = 0; x < (2 * 3.1415); x += 0.1) {
        y = sinh (x);
        printf ("SINH(%f) = %f\n", x, y);
    }
}
```

sprintf / sprintf517

Summary: `#include <stdio.h>`
`int sprintf (`
 `char *buffer, /* storage buffer */`
 `const char *fmtstr /* format string */`
 `[, argument]...); /* additional arguments */`

Description: The **sprintf** function formats a series of strings and numeric values and stores the resulting string in *buffer*. The *fmtstr* argument is a format string and has the same requirements as specified for the **printf** function. Refer to “printf / printf517” on page 290 for a description of the format string and additional arguments.

The **sprintf517** function is identical to **sprintf**, but uses the arithmetic unit of the Infineon C517x, C509 to provide faster execution. When using this function, include the header file 80C517.H. Do not use this routine with a CPU that does not support this feature.

NOTE

*The total number of bytes that may be passed to **sprintf** is limited due to the memory restrictions imposed by the 8051. A maximum of 15 bytes may be passed in **SMALL** model or **COMPACT** model. A maximum of 40 bytes may be passed in **LARGE** model.*

Return Value: The **sprintf** function returns the number of characters actually written to *buffer*.

See Also: **gets, printf, puts, scanf, sscanf, vprintf, vsprintf**

Example:

```
#include <stdio.h>

void tst_sprintf (void) {
    char buf [100];
    int n;

    int a,b;
    float pi;

    a = 123;
    b = 456;
    pi = 3.14159;

    n = sprintf (buf, "%f\n", 1.1);
    n += sprintf (buf+n, "%d\n", a);
    n += sprintf (buf+n, "%d %s %g", b, "---", pi);
    printf (buf);
}
```

sqrt / sqrt517

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

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

The `sqrt517` function is identical to `sqrt`, but uses the arithmetic unit of the Infineon C517x, C509 to provide faster execution. When using this function, include the header file 80C517.H. Do not use this routine with a CPU that does not support this feature.

Return Value: The `sqrt` function returns the positive square root of x .

See Also: `exp`, `log`, `pow`

Example:

```
#include <math.h>
#include <stdio.h>                    /* for printf */

void tst_sqrt (void) {
    float x;
    float y;

    x = 25.0;

    y = sqrt (x);                      /* y = 5 */

    printf ("SQRT(%f) = %f\n", x, y);
}
```

srand

Summary: `#include <stdlib.h>`
 void srand (
 int seed); */* random number generator seed */*

Description: The **srand** function sets the starting value *seed* used by the pseudo-random number generator in the **rand** function. The random number generator produces the same sequence of pseudo-random numbers for any given value of *seed*.

Return Value: None.

See Also: **rand**

Example:

```
#include <stdlib.h>
#include <stdio.h>                                /* for printf */

void tst_srand (void) {
    int i;
    int r;

    srand (56);

    for (i = 0; i < 10; i++) {
        printf ("I = %d, RAND = %d\n", i, rand ());
    }
}
```

sscanf / sscanf517

Summary: `#include <stdio.h>`
`int sscanf (`
 `char *buffer, /* scanf input buffer */`
 `const char *fmtstr /* format string */`
 `[, argument]...); /* additional arguments */`

Description: The `sscanf` function reads data from the string *buffer*. Data input are stored in the locations specified by *argument* according to the format string *fmtstr*. Each *argument* must be a pointer to a variable that corresponds to the type defined in *fmtstr* which controls the interpretation of the input data. The *fmtstr* argument is composed of one or more whitespace characters, non-whitespace characters, and format specifications, as defined in the `scanf` function description. Refer to “scanf” on page 300 for a complete description of the formation string and additional arguments.

The `sscanf517` function is identical to `sscanf`, but uses the arithmetic unit of the Infineon C517x, C509 to provide faster execution. When using this function, include the header file 80C517.H. Do not use this routine with a CPU that does not support this feature.

NOTE

*The total number of bytes that may be passed to `sscanf` is limited due to the memory restrictions imposed by the 8051. A maximum of 15 bytes may be passed in **SMALL** model or **COMPACT** model. A maximum of 40 bytes may be passed in **LARGE** model.*

Return Value: The `sscanf` function returns the number of input fields that were successfully converted. An **EOF** is returned if an error is encountered.

See Also: `gets`, `printf`, `puts`, `scanf`, `sprintf`, `vprintf`, `vsprintf`

Example:

```
#include <stdio.h>

void tst_sscanf (void) {
    char a;
    int b;
    long c;

    unsigned char x;
    unsigned int y;
    unsigned long z;

    float f,g;

    char d, buf [10];

    int argsread;

    printf ("Reading a signed byte, int, and long\n");
    argsread = sscanf ("1 -234 567890",
                      "%bd %d %ld", &a, &b, &c);
    printf ("%d arguments read\n", argsread);

    printf ("Reading an unsigned byte, int, and long\n");
    argsread = sscanf ("2 44 98765432",
                      "%bu %u %lu", &x, &y, &z);
    printf ("%d arguments read\n", argsread);

    printf ("Reading a character and a string\n");
    argsread = sscanf ("a abcdefg", "%c %9s", &d, buf);
    printf ("%d arguments read\n", argsread);

    printf ("Reading two floating-point numbers\n");
    argsread = sscanf ("12.5 25.0", "%f %f", &f, &g);
    printf ("%d arguments read\n", argsread);
}
```


strcat

Summary: `#include <string.h>`
 `char *strcat (`
 `char *dest, /* destination string */`
 `char *src); /* source string */`

Description: The `strcat` function concatenates or appends `src` to `dest` and terminates `dest` with a null character.

Return Value: The `strcat` function returns `dest`.

See Also: `strcpy`, `strlen`, `strncat`, `strncpy`

Example:

```
#include <string.h>
#include <stdio.h>                                    /* for printf */

void tst_strcat (void) {
    char buf [21];
    char s [] = "Test String";

    strcpy (buf, s);
    strcat (buf, " #2");

    printf ("new string is %s\n", buf);
}
```

strchr

Summary: `#include <string.h>`
`char *strchr (`
 `const char *string, /* string to search */`
 `char c); /* character to find */`

Description: The `strchr` function searches *string* for the first occurrence of *c*. The null character terminating *string* is included in the search.

Return Value: The `strchr` function returns a pointer to the character *c* found in *string* or a null pointer if no matching character is found.

See Also: `strcspn`, `strpbrk`, `strpos`, `strrchr`, `strrpbrk`, `strrpos`, `strspn`, `strstr`

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	<i>string1</i> less than <i>string2</i>
= 0	<i>string1</i> equal to <i>string2</i>
> 0	<i>string1</i> greater than <i>string2</i>

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, strncpy, strcpy`

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	<i>string1</i> less than <i>string2</i>
= 0	<i>string1</i> equal to <i>string2</i>
> 0	<i>string1</i> greater than <i>string2</i>

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, strcat**

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, strecspn, strpos, strrchr, strpbrk, 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, strecpn, strpbrk, strrchr, strpbrk, 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, strecspn, strpbrk, strpos, strrbrk, 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, strecspn, strpbrk, strpos, strrchr, strrpos, strspn**

Example:

```
#include <string.h>
#include <stdio.h>                /* for printf */

void tst_strrpbrk (void) {
    char vowels [] = "AEIOUaeiou";
    char text [] = "American National Standards
                   Institute";

    char *p;

    p = strrpbrk (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, strecpn, strpbrk, strpos, strrchr, strrpbkr, strspn, strstr

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, strecspn, strpbrk, strpos, strrchr, strpbrk, 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]);
}
```

strstr

Summary: `#include <string.h>`
`char *strstr (`
 `const char *src, /* string to search */`
 `char *sub); /* sub string to search */`

Description: The `strstr` function locates the first occurrence of the string `sub` in the string `src` and returns a pointer to the beginning of the first occurrence.

Return Value: The `strstr` function returns a pointer within `src` that points to a string identical to `sub`. If no such `sub` string exists in `src` a null pointer is returned.

See Also: `strchr`, `strpos`

Example:

```
#include <string.h>
#include <stdio.h> /* for printf */

char s1 [] = "My House is small";
char s2 [] = "My Car is green";

void tst_strstr (void) {
    char *s;

    s = strstr (s1, "House");
    printf ("substr (s1, \"House\") returns %s\n", s);
}
```


strtod / strtod517

Summary: `#include <stdlib.h>`
 unsigned long strtod (
 const char **string*, /* string to convert */
 char ***ptr*); /* ptr to subsequent characters */

Description: The **strtod** 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. Whitespace characters at the beginning of string are skipped.

The **strtod517** function is identical to `atof`, but uses the arithmetic unit of the Infineon 80C517 to provide faster execution. When using this function, include the header file `80C517.H`. Do not use this routine with a CPU that does not support this feature.

The **strtod** function requires *string* to have the following format:

$[\{+|- \}] \textit{digits} [.\textit{digits}] [\{e|E\} [\{+|- \}] \textit{digits}]$

where:

digits may be one or more decimal digits.

The value of *ptr* is set to point to the first character in the *string* immediately following the converted part of the *string*. If *ptr* is NULL, no value is assigned to *ptr*. If no conversion is possible, then *ptr* is set to the value of *string* and the value 0 is returned by **strtoul**.

Return Value: The **strtod** function returns the floating-point value that is produced by interpreting the characters in the *string* as a number.

See Also: **atof, atoi, atol, strtol, strtoul**

Example:

```
#include <stdlib.h>
#include <stdio.h>                /* for printf */

void tst_strtod (void) {
    float f;
    char s [] = "1.23";

    f = strtod (s, NULL);
    printf ("strtod(%s) = %f\n", s, f);
}
```

strtol

Summary: **#include <stdlib.h>**
long strtol (
 const char *string, /* string to convert */
 char **ptr, /* ptr to subsequent characters */
 unsigned char base); /* number base for conversion */

Description: The **strtol** function converts *string* into a long value. The input *string* is a sequence of characters that can be interpreted as an integer number. Whitespace characters at the beginning of string are skipped. An optional sign may precede the number.

The **strtol** function requires *string* to have the following format:

$[whitespace] [\{ + | - \}] digits$

where:

digits may be one or more decimal digits.

If the *base* is zero, the number should have the format of a *decimal-constant*, *octal-constant* or *hexadecimal-constant*. The radix of the number is deduced from its format. If the value of *base* is between 2 and 36, the number must consist of a non-zero sequence of letters and digits representing an integer in the specified base. The letters a through z (or A through Z) represent the values 10 through 36, respectively. Only those letters representing values less than the *base* are permitted. If the *base* is 16, the number may begin with 0x or 0X, which is ignored.

The value of *ptr* is set to point to the first character in *string* immediately following the converted part of the *string*. If *ptr* is NULL no value is assigned to *ptr*. If no conversion is possible, *ptr* is set to the value of *string* and the value 0 is returned by **strtol**.

Return Value: The **strtol** function returns the integer value that is produced by interpreting the characters in *string* as number. The

value `LONG_MIN` or `LONG_MAX` is returned in case of overflow.

See Also: `atof`, `atoi`, `atol`, `strtod`, `strtoul`

Example:

```
#include <stdlib.h>
#include <stdio.h>           /* for printf */

char s [] = "-123456789";

void tst_strtol (void) {
    long l;

    l = strtol (s, NULL, 10);
    printf ("strtol(%s) = %ld\n", s, l);
}
```

strtoul

Summary:

```
#include <stdlib.h>
unsigned long strtoul (
    const char *string, /* string to convert */
    char **ptr,         /* ptr to subsequent characters */
    unsigned char base); /* number base for conversion */
```

Description:

The **strtoul** function converts *string* into an unsigned long value. The input *string* is a sequence of characters that can be interpreted as an integer number. Whitespace characters at the beginning of string are skipped. An optional sign may precede the number.

The **strtoul** function requires string to have the following format:

[whitespace] [{+|-}] digits

where:

digits may be one or more decimal digits.

If the *base* is zero, the number should have the format of a *decimal-constant*, *octal-constant* or *hexadecimal-constant*. The radix of the number is deduced from its format. If the value of *base* is between 2 and 36, the number must consist of a non-zero sequence of letters and digits representing an integer in the specified base. The letters a through z (or A through Z) represent the values 10 through 36, respectively. Only those letters representing values less than the *base* are permitted. If the *base* is 16, the number may begin with 0x or 0X, which is ignored.

The value of *ptr* is set to point to the first character in *string* immediately following the converted part of the *string*. If *ptr* is NULL no value is assigned to *ptr*. If no conversion is possible, *ptr* is set to the value of *string* and the value 0 is returned by **strtoul**.

Return Value:

The **strtoul** function returns the integer value that is produced by interpreting the characters in *string* as a

number. The value `ULONG_MAX` is returned in case of overflow.

See Also: `atof`, `atoi`, `atol`, `strtod`, `strtol`

Example:

```
#include <stdlib.h>
#include <stdio.h>           /* for printf */

char s [] = "12345AB";

void tst_strtoul (void) {
    unsigned long ul;

    ul = strtoul (s, NULL, 16);
    printf ("strtoul(%s) = %lx\n", s, ul);
}
```

tan / tan517

Summary: `#include <math.h>`
`float tan (`
 `float x);` `/* value to calculate tangent of`
`*/`

Description: The **tan** function calculates the tangent of the floating-point value *x*. The value of *x* must be in the -65535 to +65535 range or an NaN error value is generated.

The **tan517** function is identical to **tan**, but uses the arithmetic unit of the Infineon C517x, C509 to provide faster execution. When using this function, include the header file 80C517.H. Do not use this routine with a CPU that does not support this feature.

Return Value: The **tan** function returns the tangent of *x*.

See Also: **cos, sin**

Example:

```
#include <math.h>
#include <stdio.h>                    /* for printf */

void tst_tan (void) {
    float x, y, pi;

    pi = 3.14159;

    for (x = -(pi/4); x < (pi/4); x += 0.1) {
        y = tan (x);
        printf ("TAN(%f) = %f\n", x, y);
    }
}
```

tanh

Summary: `#include <math.h>`
`float tanh (`
 `float x);` `/* value to calc hyperbolic`
 `tangent for */`

Description: The `tanh` function calculates the hyperbolic tangent for the floating-point value `x`.

Return Value: The `tanh` function returns the hyperbolic tangent of `x`.

See Also: `cosh`, `sinh`

Example:

```
#include <math.h>
#include <stdio.h>                    /* for printf */

void tst_tanh (void) {
    float x;
    float y;
    float pi;

    pi = 3.14159;

    for (x = -(pi/4); x < (pi/4); x += 0.1) {
        y = tanh (x);
        printf ("TANH(%f) = %f\n", x, y);
    }
}
```


testbit

Summary: `#include <intrins.h>`
 `bit _testbit_ (`
 `bit b);` `/* bit to test and clear */`

Description: The `_testbit_` routine produces a JBC instruction in the generated program code to simultaneously test the bit `b` and clear it to 0. This routine can be used only on directly addressable bit variables and is invalid on any type of expression. This routine is implemented as an intrinsic function. The code required is included in-line rather than being called.

Return Value: The `_testbit_` routine returns the value of `b`.

Example:

```
#include <intrins.h>
#include <stdio.h>                            /* for printf */

void tst_testbit (void){
  bit test_flag;

  if (_testbit_ (test_flag))
    printf ("Bit was set\n");

  else
    printf ("Bit was clear\n");
}
```

toascii

- Summary:** `#include <ctype.h>`
`char toascii (`
 `char c);` */* character to convert */*
- Description:** The `toascii` macro converts `c` to a 7-bit ASCII character. This macro clears all but the lower 7 bits of `c`.
- Return Value:** The `toascii` macro returns the 7-bit ASCII character for `c`.
- See Also:** `toint`

Example:

```
#include <ctype.h>
#include <stdio.h>                    /* for printf */

void tst_toascii ( char c) {
    char k;

    k = toascii (c);

    printf ("%c is an ASCII character\n", k);
}
```

toint

Summary: `#include <ctype.h>`
`char toint (`
 `char c);` `/* digit to convert */`

Description: The **toint** function interprets *c* as a hexadecimal value. ASCII characters '0' through '9' generate values of 0 to 9. ASCII characters 'A' through 'F' and 'a' through 'f' generate values of 10 to 15. If the value of *c* is not a hexadecimal digit, the function returns -1.

Return Value: The **toint** function returns the value of the ASCII hexadecimal character *c*.

See Also: **toascii**

Example:

```
#include <ctype.h>
#include <stdio.h>                    /* for printf */

void tst_toint (void) {
    unsigned long l;
    char k;

    for (l = 0; isdigit (k = getchar ());
         l *= 10) {

        l += toint (k);
    }
}
```

tolower

Summary: `#include <ctype.h>`
`char tolower (`
 `char c);` `/* character to convert */`

Description: The **tolower** function converts *c* to a lowercase character. If *c* is not an alphabetic letter, the **tolower** function has no effect.

Return Value: The **tolower** function returns the lowercase equivalent of *c*.

See Also: `_tolower`, `toupper`, `_toupper`

Example:

```
#include <ctype.h>
#include <stdio.h>                    /* for printf */

void tst_tolower (void) {
    unsigned char i;

    for (i = 0x20; i < 0x7F; i++) {
        printf ("tolower(%c) = %c\n", i, tolower(i));
    }
}
```

_tolower

- Summary:** `#include <ctype.h>`
 `char _tolower (`
 `char c);` */* character to convert */*
- Description:** The `_tolower` macro is a version of `tolower` that can be used when `c` is known to be an uppercase character.
- Return Value:** The `_tolower` macro returns a lowercase character.
- See Also:** `tolower`, `toupper`, `_toupper`

Example:

```
#include <ctype.h>
#include <stdio.h>                    /* for printf */

void tst__tolower ( char k) {
    if (isupper (k)) k = _tolower (k);
}
```

toupper

Summary: `#include <ctype.h>`
`char toupper (`
 `char c);` `/* character to convert */`

Description: The **toupper** function converts *c* to an uppercase character. If *c* is not an alphabetic letter, the **toupper** function has no effect.

Return Value: The **toupper** function returns the uppercase equivalent of *c*.

See Also: **tolower, _tolower, _toupper**

Example:

```
#include <ctype.h>
#include <stdio.h>                    /* for printf */

void tst_toupper (void) {
    unsigned char i;

    for (i = 0x20; i < 0x7F; i++) {
        printf ("toupper(%c) = %c\n", i, toupper(i));
    }
}
```

_toupper

- Summary:** `#include <ctype.h>`
`char _toupper (`
 `char c);` */* character to convert */*
- Description:** The `_toupper` macro is a version of `toupper` that can be used when `c` is known to be a lowercase character.
- Return Value:** The `_toupper` macro returns an uppercase character.
- See Also:** `tolower`, `_tolower`, `toupper`

Example:

```
#include <ctype.h>
#include <stdio.h>                    /* for printf */

void tst__toupper ( char k) {
    if (islower (k)) k = _toupper (k);
}
```

ungetchar

Summary: `#include <stdio.h>`
`char ungetchar (`
 `char c);` `/* character to unget */`

Description: The **ungetchar** function stores the character *c* back into the input stream. Subsequent calls to **getchar** and other stream input functions return *c*. Only one character may be passed to **unget** between calls to **getchar**.

Return Value: The **ungetchar** function returns the character *c* if successful. If **ungetchar** is called more than once between function calls that read from the input stream, **EOF** is returned indicating an error condition.

See Also: `_getkey`, `putchar`, `ungetchar`

Example:

```
#include <stdio.h>

void tst_ungetchar (void) {
    char k;

    while (isdigit (k = getchar ())) {
        /* stay in the loop as long as k is a digit */
    }
    ungetchar (k);
}
```


va_arg

Summary: **#include <stdarg.h>**
type va_arg (
 argptr, / optional argument list */*
 type); / type of next argument */*

Description: The **va_arg** macro is used to extract subsequent arguments from a variable-length argument list referenced by *argptr*. The *type* argument specifies the data type of the argument to extract. This macro may be called only once for each argument and must be called in the order of the parameters in the argument list.

The first call to **va_arg** returns the first argument after the *prevparm* argument specified in the **va_start** macro. Subsequent calls to **va_arg** return the remaining arguments in succession.

Return Value: The **va_arg** macro returns the value for the specified argument type.

See Also: **va_end, va_start**

Example:

```
#include <stdarg.h>
#include <stdio.h>                /* for printf */

int varfunc (char *buf, int id, ...) {
    va_list tag;

    va_start (tag, id);

    if (id == 0) {
        int arg1;
        char *arg2;
        long arg3;

        arg1 = va_arg (tag, int);
        arg2 = va_arg (tag, char *);
        arg3 = va_arg (tag, long);
    }
    else {
        char *arg1;
        char *arg2;
        long arg3;

        arg1 = va_arg (tag, char *);
        arg2 = va_arg (tag, char *);
        arg3 = va_arg (tag, long);
    }
}

void caller (void) {
    char tmp_buffer [10];

    varfunc (tmp_buffer, 0, 27, "Test Code", 100L);
    varfunc (tmp_buffer, 1, "Test", "Code", 348L);
}
```

va_end

Summary: `#include <stdarg.h>`
 `void va_end (`
 `argptr);` `/* optional argument list */`

Description: The `va_end` macro is used to terminate use of the variable-length argument list pointer `argptr` that was initialized using the `va_start` macro.

Return Value: None.

See Also: `va_arg`, `va_start`

Example: See `va_arg`.

va_start

Summary:

```
#include <stdarg.h>
void va_start (
    argptr,          /* optional argument list */
    prevparm);      /* arg preceding optional args */
```

Description:

The **va_start** macro, when used in a function with a variable-length argument list, initializes *argptr* for subsequent use by the **va_arg** and **va_end** macros. The *prevparm* argument must be the name of the function argument immediately preceding the optional arguments specified by an ellipsis (...). This function must be called to initialize a variable-length argument list pointer before any access using the **va_arg** macro is made.

Return Value:

None.

See Also:

va_arg, **va_end**

Example:

See **va_arg**.

vprintf

Summary: `#include <stdio.h>`
`void vprintf (`
 `const char * fmtstr, /* pointer to format string */`
 `char * argptr); /* pointer to argument list */`

Description: The **vprintf** function formats a series of strings and numeric values and builds a string to write to the output stream using the **putchar** function. The function is similar to the counterpart **printf**, but it accepts a pointer to a list of arguments instead of an argument list.

The *fmtstr* argument is a pointer to a format string and has the same form and function as the *fmtstr* argument for the **printf** function. Refer to “printf / printf517” on page 290 for a description of the format string. The *argptr* argument points to a list of arguments that are converted and output according to the corresponding format specifications in the format.

NOTE

*This function is implementation-specific and is based on the operation of the **putchar** function. This function, as provided in the standard library, writes characters using the serial port of the 8051. Custom functions may use other I/O devices.*

Return Value: The **vprintf** function returns the number of characters actually written to the output stream.

See Also: **gets, puts, printf, scanf, sprintf, sscanf, vsprintf**

Example:

```
#include <stdio.h>
#include <stdarg.h>

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

    va_start (arg_ptr, fmt);      /* format string */
    vprintf (fmt, arg_ptr);
    va_end (arg_ptr);
}

void tst_vprintf (void) {
    int i;
    i = 1000;
    /* call error with one parameter */
    error ("Error: '%d' number too large\n", i);
    /* call error with just a format string */
    error ("Syntax Error\n");
}
```

vsprintf

Summary:

```
#include <stdio.h>
void vsprintf (
    char *buffer,           /* pointer to storage buffer */
    const char *fmtstr,    /* pointer to format string */
    char *argptr);        /* pointer to argument list */
```

Description: The **vsprintf** function formats a series of strings and numeric values and stores the string in *buffer*. The function is similar to the counterpart **sprintf**, but it accepts a pointer to a list of arguments instead of an argument list.

The *fmtstr* argument is a pointer to a format string and has the same form and function as the *fmtstr* argument for the **printf** function. Refer to “printf / printf517” on page 290 for a description of the format string. The *argptr* argument points to a list of arguments that are converted and output according the corresponding format specifications in the format.

Return Value: The **vsprintf** function returns the number of characters actually written to the output stream.

See Also: **gets, puts, printf, scanf, sprintf, sscanf, vprintf**

Example:

```
#include <stdio.h>
#include <stdarg.h>

xdata char etxt[30];          /* text buffer */

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

    va_start (arg_ptr, fmt);  /* format string */
    vsprintf (etxt, fmt, arg_ptr);
    va_end (arg_ptr);
}

void tst_vprintf (void) {
    int i;
    i = 1000;

    /* call error with one parameter */
    error ("Error: '%d' number too large\n", i);

    /* call error with just a format string */
    error ("Syntax Error\n");
}
```


Appendix A. Differences from ANSI C

A

The **Cx51** 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 **Cx51**. ANSI provides wide characters for future support of an international character set.

■ Recursive Function Calls

Recursive function calls are not supported by default. Functions that are recursive must be declared using the **reentrant** function attribute. Reentrant functions can be called recursively because the local data and parameters are stored in a reentrant stack. In comparison, functions which are not declared using the **reentrant** attribute use static memory segments for the local data of the function. A recursive call to these functions overwrites the local data of the prior function call instance.

Library-related Differences

The ANSI C Standard Library includes a vast number of routines, most of which are included in **Cx51**. Many, however, are not applicable to an embedded application and are excluded from the **Cx51** library.

The following ANSI Standard library routines are included in the **Cx51** library:

abs	cosh	isdigit
acos	exp	isgraph
asin	fabs	islower
atan	floor	isprint
atan2	fmod	ispunct
atof	free	isspace
atoi	getchar	isupper
atol	gets	isxdigit
calloc	isalnum	labs
ceil	isalpha	log
cos	iscentrl	log10

longjmp	sin	strchr
malloc	sinh	strspn
memchr	sprintf	strstr
memcmp	sqrt	strtod
memcpy	srand	strtol
memmove	sscanf	strtoul
memset	strcat	tan
modf	strchr	tanh
pow	strcmp	tolower
printf	strcpy	toupper
putchar	strcspn	va_arg
puts	strlen	va_end
rand	strncat	va_start
realloc	strncmp	vprintf
scanf	strncpy	vsprintf
setjmp	strpbrk	

The following ANSI Standard library routines are not included in the Cx51 library:

abort	freopen	remove
asctime	frexp	rename
atexit	fscanf	rewind
bsearch	fseek	setbuf
clearerr	fsetpos	setlocale
clock	ftell	setvbuf
ctime	fwrite	signal
difftime	getc	strcoll
div	getenv	strerror
exit	gmtime	strftime
fclose	ldexp	strtok
feof	ldiv	strxfrm
ferror	localeconv	system
fflush	localtime	time
fgetc	mblen	tmpfile
fgetpos	mbstowcs	tmpnam
fgets	mbtowc	ungetc
fopen	mktime	vfprintf
fprintf	perror	wcstombs
fputc	putc	wctomb
fputs	qsort	
fread	raise	

The following routines are not found in the ANSI Standard Library but are included in the **Cx51** library:

acos517	_iror_	strpos
asin517	log10517	strrpbrk
atan517	log517	strrpos
atof517	_lrol_	strtod517
cabs	_lror_	tan517
chkfloat	memcpy	_testbit_
cos517	_nop_	toascii
crol	printf517	toint
cror	scanf517	_tolower
exp517	sin517	_toupper
_getkey	sprintf517	ungetchar
init_mempool	sqrt517	
irol	sscanf517	



Appendix B. Version Differences

This appendix lists an overview of major product enhancements and differences between the current version of the **Cx51** compiler and previous versions. The current version contains *all* enhancements listed below:

Version 6.0 Differences

B

- **EXTERNAL and SEGMENT limitations removed**
The number of external symbols and segments per module are no longer limited to 256. This historical limitation was imposed by the old Intel Object File format.
- **First 256 characters of a variable name are significant**
Now the first 256 characters of a variable name are significant. Previously, only the first 32 characters were significant.
- **Support for Philips 80C51MX and Dallas Contiguous Mode**
Cx51 provides support for the Philips 80C51MX architecture and the Dallas Contiguous Mode that is available on the Dallas 390 and variants.
- **OMF2 directive and far memory type support**
The OMF2 directive selects a new OMF file format that provides detailed symbol type checking across modules and supports up to 16MB code and xdata memory. This format is required when you use the **STRING**, **VARBANKING**, and **XCROM** directives.
- **STRING directive**
Cx51 allows you to locate constant strings into **const xdata** or **const far** space which leaves more code space available for program code.
- **USERCLASS directive**
Assigns user defined class names to compiler generated segments. Class names may be referenced by the **LX51** linker to locate all segments with a specific class name.
- **VARBANKING directive and far memory type support**
Two new memory types, **far** and **const far**, and user configurable access routines provide support for up to 16MB extended code and xdata memory. The **VARBANKING** directive enables **far** memory type support.
- **XCROM directive**
The **XCROM** directive locates constants into **xdata** ROM which frees **code** ROM space for program code.

- **Support for Analog Devices B2 series of MicroConverters**

The B2 series of ADuC devices contains dual DPTR and an extended stack.

NOTE

Only the the PK51 Professional Developers Kit supports the OMF2 output file format, Philips 80C51MX, Dallas Contiguous Mode, and VARBANKING. These options are not available in the CA51 and DK51 packages.

Version 5 Differences

- **Optimize Level 7, 8, and 9**

C51 offers three new optimizer levels. These new optimizations focus primarily on code density. Refer to “Optimizer” on page 157 for more information.

- **Directives for the dual DPTR support**

C51 provides dual DPTR support for Atmel, Atmel WM, and, Philips with the directives **MODA2** and **MODP2**.

- **data, pdata, xdata automatic variables overlayable in all memory models**

C51 now overlays all data, pdata, and xdata automatic variables regardless of the selected memory model. In previous C51 versions, only automatic variables of the default memory type are overlaid. For example, C51 Version 5 did not overlay pdata or xdata variables if a function where compiled in the SMALL memory model.

- **The data type enum adjusts automatically 8 or 16 bits.**

C51 now uses a char variable to represent an enum, if the enum range allows that.

- **modf, strtod, strtol, strtoul Library Functions**

C51 now includes the ANSI standard library functions **modf**, **strtod**, **strtol**, **strtoul**

- **BROWSE, INCDIR, ONEREBANK, RET_XSTK, and RET_PSTK Directives**

C51 supports new directives for generating Browse Information, specifying include directives, optimizing interrupt code, and using the reentrant stack for return addresses. Refer to “Chapter 2. Compiling with the Cx51” on page 17 for more information.

Version 4 Differences

- **Byte Order of Floating-point Numbers**
 Floating-point numbers are now stored in the big endian order. Previous releases of the C51 compiler stored floating-point numbers in little endian format. Refer to “Floating-point Numbers” on page 179 for more information.
- **`_chkfloat_` Library Function**
 The intrinsic function `_chkfloat_` allows for fast testing of floating-point numbers for error (NaN), ±INF, zero and normal numbers. Refer to “`_chkfloat_`” on page 245 for more information.
- **FLOATFUZZY Directive**
 C51 now supports the **FLOATFUZZY** directive. This directive controls the number of bits ignored during the execution of a floating-point compare. Refer to “**FLOATFUZZY**” on page 38 for more information.
- **Floating-point Arithmetic is Fully Reentrant**
 Intrinsic floating-point arithmetic operations (add, subtract, multiply, divide, and compare) are now fully reentrant. The C library routines **fp`save`** and **fp`restore`** are no longer needed. Several library routines are also reentrant. Refer to “Routines by Category” on page 218 for more information.
- **Long and Floating-point Operations no Longer use an Arithmetic Stack**
 The long and floating-point arithmetic is more efficient; the code generated is now totally register-based and does not use a simulated arithmetic stack. This also reduces the memory needs of the generated code.
- **Memory Types**
 The memory types have been changed to achieve better performance in the run-time library and to reflect the memory map of the MCS[®] 251 architecture.
- **Memory Type Bytes for Generic Pointers**
 The memory type bytes used in generic pointers have changed. The following table contains the memory type byte values and their associated memory type.

Memory Type	idata	data	bdata	xdata	pdata	code
C51 V5 Value	0x00	0x00	0x00	0x01	0xFE	0xFF
C51 V4 Value	0x01	0x04	0x04	0x02	0x03	0x05

- **WARNINGLEVEL Directive**
 C51 now supports the **WARNINGLEVEL** directive which lets you specify the strength of the warning detection for the C51 compiler. The C51

compiler now also checks for unused local variables, labels, and expressions. Refer to “`WARNINGLEVEL`” on page 85 for more information.

B

Version 3.4 Differences

- **_at_ Keyword**

C51 supports variable location using the **_at_** keyword. This new keyword allows you to specify the address of a variable in a declaration. Refer to “The **_at_** Keyword” on page 186 for more information.
- **NOAMAKE Directive**

C51 now supports the **NOAMAKE** directive. This directive causes C51 to generate object modules without project information and register optimization records. This is necessary only if you want to use object files with older versions of C51 tools.
- **OH51 Hex File Converter**

The OHS51 Object-Hex-Symbol Converter provided with prior versions of C51 has been replaced with OH51.
- **Optimizer Level 6**

C51 now supports optimizer level 6 which provides loop rotation. The resulting code is more efficient and executes faster.
- **ORDER Directive**

When you specify the **ORDER** directive, C51 locates variables in memory in the order in which they are declared in your source file. Refer to “ORDER” on page 65 for more information.
- **REGFILE Directive**

C51 now supports the **REGFILE** directive which lets you specify the name of the register definition file generated by the linker. This file contains information that is used to optimize the use of registers between functions in different modules. Refer to “REGFILE” on page 70 for more information.
- **vprintf and vsprintf Library Functions**

The **vprintf** and **vsprintf** library functions have been added. Refer to “vprintf” on page 349 and “vsprintf” on page 351 for more information.

Version 3.2 Differences

■ ANSI Standard Automatic Integer Promotion

The latest version of the ANSI C Standard requires that calculations use **int** values if **char** or **unsigned char** values might overflow during the calculation. This new requirement is based on the premise that **int** and **char** operations are similar on 16-bit CPUs. C51 supports this feature as the default and provides you with two new directives, **INTPROMOTE** and **NOINTPROMOTE**, to enable or disable integer promotion.

There is a big difference between 8-bit and 16-bit operations on the 8-bit 8051 in terms of code size and execution speed. For this reason, you might want to disable integer promotion by using the **NOINTPROMOTE** directive.

However, if you wish to retain maximum compatibility with other C compilers and platforms, leave integer promotions enabled.

■ Assembly Source Generation with In-Line Assembly

You may use the new directives **ASM** and **ENDASM** to include source text to output to **.SRC** files generated using the **SRC** command directive.

■ New Directives

The directives **ASM**, **ENDASM**, **INTERVAL**, **INTPROMOTE**, **INTVECTOR**, **MAXARGS**, and **NOINTPROMOTE** have been added or enhanced.

■ Offset and Interval Can Now Be Specified for Interrupt Vectors

You may now specify the offset and interval for the interrupt vector table. These features provide support for the SIECO-51 derivatives and allow you to specify a different location for the interrupt vector in situations where the interrupt table is not located at address 0000h.

■ Parameter Passing to Indirectly Called Functions

Function parameters may now be passed to indirectly called functions if all of the parameters can be passed in CPU registers. These functions do not have to be declared with the **reentrant** attribute.

■ Source Code Provided For Memory Allocation Functions

C source code for the memory allocation routines is now provided with the C51 compiler. You may now more easily adapt these functions to the hardware architecture of your embedded system.

■ Trigraphs

C51 now supports trigraph sequences.

- **Variable-length Argument Lists for All Functions**

Variable-length argument lists are now supported for all function types. Functions with a variable length argument list do not have to be declared using the **reentrant** attribute. The new command line directive **MAXARGS** determines the size of the parameter passing area.

Version 3.0 Differences

- **New Directive Added for Assembly Source File Output**

The **SRC** directive has been added to direct the compiler to generate an assembly language source file instead of an object file.

- **New Library Functions**

The library functions **calloc**, **free**, **init_mempool**, **malloc**, and **realloc** have been added.

Version 2 Differences

■ Absolute Register Addressing

C51 now generates code that performs absolute register addressing. This improves execution speed. The directives **AREGS** and **NOAREGS**, respectively, enable or disable this feature.

■ Bit-addressable Memory Type

Variable types of **char** and **int** can now be declared to reside in the bit-addressable internal memory area by using the **bdata** memory specifier.

■ Intrinsic Functions

Intrinsic functions have been added to the library to support some of the special instructions built in to the 8051.

■ Mixed Memory Models

Calls to and from functions of different memory models are now supported.

■ New Optimizer Levels

Two new levels of optimization have been added to the C51 compiler. These new levels support register variables, local common subexpression elimination, loop optimizations, and global common subexpression elimination, to name a few.

■ New Predefined Macros

The macros `__C51__` and `__MODEL__` are now defined by the preprocessor at compile time.

■ Reentrant and Recursive Functions

Individual functions may now be defined as being reentrant or recursive by using the **reentrant** function attribute.

■ Registers Used for Parameter Passing

C51 now passes up to 3 function arguments using registers. The **REGPARMS** and **NOREGPARMS** directives enable or disable this feature.

■ Support for Memory-specific Pointers

Pointers may now be defined to reference data in a particular memory area.

■ Support for PL/M-51 Functions

The **alien** keyword has been added to support PL/M-51 compatible functions and function calls.

■ Volatile Type Specifier

The **volatile** variable attribute may be used to enforce variable access and to prevent optimizations involving that variable.

B

Appendix C. Writing Optimum Code

This section lists a number of ways you can improve the efficiency (i.e., smaller code and faster execution) of the 8051 code generated by the **Cx51** 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 the small model always generates the smallest, fastest code possible. The **SMALL** directive instructs the **Cx51** compiler to use the small memory model. In the small model, all variables, unless declared otherwise, reside in the internal memory of the 8051. Memory access to internal data memory is fast (typically performed in 1 or 2 clock cycles), and the generated code is much smaller than that generated with the compact or large models. For example, the following loop:

```
for (i = 0; i < 100; i++) {
    do_nothing ();
}
```

is compiled in both the small and large models to demonstrate the difference in generated code. The following is the small model translation:

```
stmt level  source
1          #pragma small
2
3          void do_nothing (void);
4
5
6          void func (void)
7          {
8  1        unsigned char i;
9  1
10 1        for (i = 0; i < 100; i++)
11 1          {
12 2            do_nothing ();
13 2          }
14 1        }
          ; FUNCTION func (BEGIN)
          ; SOURCE LINE # 10
0000 E4          CLR    A
0001 F500        R        MOV    i,A
0003             ?C0001:
0003 E500        R        MOV    A,i
0005 C3          CLR    C
0006 9464        SUBB   A,#064H
```

```

0008 5007          JNC    ?C0004
                ; SOURCE LINE # 12
000A 120000  E      LCALL  do_nothing
                ; SOURCE LINE # 13
000D 0500   R      INC    i
000F 80F2          SJMP   ?C0001
                ; SOURCE LINE # 14

0011          ?C0004:
0011 22          RET
                ; FUNCTION func (END)

```

In the small model, the variable `i` is maintained in internal data memory. The instructions to access `i`, `MOV A,i` and `INC i`, require only two bytes each of code space. In addition, each of these instructions executes in only one clock cycle. The total size for the main function when compiled in small model is 11h or 17 bytes.

The following is the same code compiled using the large model:

```

                ; FUNCTION func (BEGIN)
                ; SOURCE LINE # 10
0000 E4          CLR    A
0001 900000  R      MOV    DPTR,#i
0004 F0          MOVX   @DPTR,A
0005          ?C0001:
0005 900000  R      MOV    DPTR,#i
0008 E0          MOVX   A,@DPTR
0009 C3          CLR    C
000A 9464          SUBB   A,#064H
000C 500B          JNC    ?C0004
                ; SOURCE LINE # 12
000E 120000  E      LCALL  do_nothing
                ; SOURCE LINE # 13
0011 900000  R      MOV    DPTR,#i
0014 E0          MOVX   A,@DPTR
0015 04          INC    A
0016 F0          MOVX   @DPTR,A
0017 80EC          SJMP   ?C0001
                ; SOURCE LINE # 14

0019          ?C0004:
0019 22          RET
                ; FUNCTION func (END)

```

In the large model, the variable `i` is maintained in external data memory. To access `i`, the compiler must first load the data pointer and then perform an external memory access (see offset 0001h through 0004h in the above listing). These two instructions alone take 4 clock cycles. The code to increment `i` is found from offset 0011h to offset 0016h. This operation consumes 6 bytes of code space and takes 7 clock cycles to execute. The total size for the main function when compiled in the small model is 19h or 25 bytes.

Variable Location

Frequently accessed data objects should be located in the internal data memory of the 8051. Accessing the internal data memory is much more efficient than accessing the external data memory. The internal data memory is shared among register banks, the bit data area, the stack, and other user defined variables with the memory type **data**.

Because of the limited amount of internal data memory (128 to 256 bytes), all your program variables may not fit into this memory area. In this case, you must locate some variables in other memory areas. There are two ways to do this.

One way is to change the memory model and let the compiler do all the work. This is the simplest method, but it is also the most costly in terms of the amount of generated code and system performance. Refer to “Memory Model” on page 367 for more information.

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

Variable Size

Members of the 8051 family are all 8-bit CPUs. Operations that use 8-bit types (like **char** and **unsigned char**) are much more efficient than operations that use **int** or **long** types. For this reason, always use the smallest data type possible.

The **Cx51** compiler directly supports all byte operations. Byte types are not promoted to integers unless required. See the **INTPROMOTE** directive for more information.

An example can be illustrated by examining a multiplication operation. The multiplication of two **char** objects is done inline with the 8051 instruction **MUL AB**. To accomplish the same operation with **int** or **long** types would require a call to a compiler library function.

Unsigned Types

The 8051 family of processors does not specifically support operations with signed numbers. The compiler must generate additional code to deal with sign extensions. Far less code is produced if unsigned objects are used wherever possible.

Local Variables

When possible, use local variables for loops and other temporary calculations. As part of the optimization process, the compiler attempts to maintain local variables in registers. Register access is the fastest type of memory access. The best effect is normally achieved with **unsigned char** and **unsigned int** variable types.

Other Sources

The quality of the compiler-generated code is more often than not directly influenced by the algorithms implemented in the program. Sometimes, you can improve the performance or reduce the code size simply by using a different algorithm. For example, a heap sort algorithm always outperforms a bubble sort algorithm.

For more information on how to write efficient programs, refer to the following books:

The Elements of Programming Style, Second Edition

Kernighan & Plauger
McGraw-Hill
ISBN 0-07-034207-5

Writing Efficient Programs

Jon Louis Bentley
Prentice-Hall Software Series
ISBN 0-13-970244-X

Efficient C

Plum & Brodie
Plum Hall, Inc.
ISBN 0-911537-05-8

Appendix D. Compiler Limits

The **Cx51** compiler embodies some known limitations that are listed below. For the most part, there are no limits 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.

- 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 may be up to 256 characters long. The C language provides for case sensitivity in regard to function and variable names. However, for compatibility reasons, all names in the object file appear in capital letters. It is therefore irrelevant if an external object name within the source program is written in capital or small letters.
- The maximum number of **case** statements in a **switch** block is not fixed. Limits are imposed only by the available memory size and the maximum size of individual functions.
- The maximum number of nested function calls in an invocation parameter list is 10.
- The maximum number of nested include files is 9. This value is independent of list files, preprocessor files, or whether or not an object file is to be generated.
- The maximum depth of directives for conditional compilation is 20. This is a preprocessor limitation.
- Instruction blocks (`{...}`) may be nested up to 15 levels deep.
- Macros may be nested up to 8 levels deep.
- A maximum of 32 parameters may be passed in a macro or function call.
- The maximum length of a line or a macro definition is 2000 characters. Even after a macro expansion, the result may not exceed 2000 characters.

D

Appendix E. Byte Ordering

Most microprocessors have a memory architecture that is composed of 8-bit address locations known as bytes. Many data items (addresses, numbers, and strings) are too long to be stored using a single byte and must be stored in a series of consecutive bytes.

When using data that are stored in multiple bytes, byte ordering becomes an issue. Unfortunately, there is not just one standard for the order in which bytes in multi-byte data are stored. There are two popular methods of byte ordering currently in widespread use.

The first method is called little endian and is often referred to as Intel order. In little endian, the least significant, or low-order byte is stored first. For example, a 16-bit integer value of 0x1234 (4660 decimal) would be stored using the little endian method in two consecutive bytes as follows:

Address	+0	+1
Contents	0x34	0x12

A 32-bit integer value of 0x57415244 (1463898692 decimal) would be stored using the little endian method as follows:

Address	+0	+1	+2	+3
Contents	0x44	0x52	0x41	0x57

A second method of accessing multi-byte data is called big endian and is often referred to as Motorola order. In big endian, the most significant, or high-order byte is stored first, and the least significant, or low-order byte is stored last. For example, a 16-bit integer value of 0x1234 would be stored using the big endian method in two consecutive bytes as follows:

Address	+0	+1
Contents	0x12	0x34

A 32-bit integer value of 0x004A4F4E would be stored using the big endian method as follows:

Address	+0	+1	+2	+3
Contents	0x00	0x4A	0x4F	0x4E

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

- The 8051 **LCALL** instruction stores the address of the next instruction on the stack. The address is pushed onto the stack low-order byte first. The address is, therefore, stored in memory in little endian format.
- All other 16-bit and 32-bit values are stored, contrary to other Intel processors, in big endian format, with the high-order byte stored first. For example, the **LJMP** and **LCALL** instructions expect 16-bit addresses that are in big endian format.
- Floating-point numbers are stored according to the IEEE-754 format and are stored in big endian format with the high-order byte stored first.

If your 8051 embedded application performs data communications with other microprocessors, it may be necessary to know the byte ordering method used by the other CPU. This is certainly true when transmitting raw binary data.

Appendix F. Hints, Tips, and Techniques

This section lists a number of illustrations and tips which commonly require further explanation. Items in this section are listed in no particular order and are merely intended to be referenced if you experience similar problems.

Recursive Code Reference Error

The following program example:

```
#pragma code symbols debug oe

void func1(unsigned char *msg ) { ; }

void func2( void ) {
    unsigned char uc;
    func1("xxxxxxxxxxxxxxxxxxx");
}

code void (*func_array[])() = { func2 };

void main( void ) {
    (*func_array[0])();
}
```

when compiled and linked using the following command lines:

```
C51 EXAMPLE1.C
BL51 EXAMPLE1.OBJ IX
```

fails and display the following error message.

```
*** WARNING 13: RECURSIVE CALL TO SEGMENT
SEGMENT: ?CO?EXAMPLE1
CALLER: ?PR?FUNC2?EXAMPLE1
```

In this program example, `func2` defines a constant string ("xxx...xxx") which is directed into the constant code segment `?CO?EXAMPLE1`. The definition `code void (*func_array[])() = { func2 };` yields a reference between segment `?CO?EXAMPLE1` (where the code table is located) and the executable code segment `?PR?FUNC2?EXAMPLE1`. Because `func2` also refers to segment `?CO?EXAMPLE1`, BL51 assumes that there is a recursive call.

To avoid this problem, link using the following command line:

```
Lx51 EXAMPLE1.OBJ IX OVERLAY &
(?CO?EXAMPLE1 ~ FUNC2, MAIN ! FUNC2)
```

?CO?EXAMPLE1 ~ FUNC2 deletes the implied call reference between `func2` and the code constant segment in the example. Then, `MAIN ! FUNC2` adds an additional call to the reference listing between `MAIN` and `FUNC2` instead. Refer to the *Ax51 Macro Assembler User's Guide* for more information.

In summary, automatic overlay analysis cannot be successfully accomplished when references are made via pointers to functions. References of this type must be manually implemented, as in the example above.

Problems Using the printf Routines

The `printf` functions are implemented using a variable-length argument list. Arguments specified after the format string are passed using their inherent data type. This can cause problems when the format specification expects a data object of a different type than was passed. For example, the following code:

```
printf ("%c %d %u %bu", 'A', 1, 2, 3);
```

does *not* print the string “A 1 2 3”. This is because the **Cx51** compiler passes the arguments `1`, `2`, and `3` all as 8-bit byte types. The format specifiers `%d` and `%u` both expect 16-bit int types.

To avoid this type of problem, you must explicitly define the data type to pass to the `printf` function. To do this, you must type cast the above values. For example:

```
printf ("%c %d %u %bu", 'A', (int) 1, (unsigned int) 2, (char) 3);
```

If you are uncertain of the size of the argument that is passed, you may cast the value to the desired size.

Uncalled Functions

It is common practice during the development process to write but not call additional functions. While the compiler permits this without error, the Linker/Locator does not treat this code casually because of the support for data overlaying, and emits a warning message.

Interrupt functions are never called, they are invoked by the hardware. An uncalled routine is treated as a potential interrupt routine by the linker. This means that the function is assigned non-overlayable data space for its local variables. This quickly exhausts all available data memory (depending upon the memory model used).

If you unexpectedly run out of memory, be sure to check for linker warnings relating to uncalled or unused routines. You can use the linker's **IXREF** directive to include a cross reference list in the linker map (**.M51**) file.

Using Monitor-51

If you want to test a C program with Monitor-51 and if the Monitor-51 is installed at code address 0, consider the following rules (the specification refers to a target system where the available code memory for user programs starts at address 8000H):

- All C modules which contain interrupt functions must be translated with the directive **INTVECTOR (0x8000)**.
- In **STARTUP.A51**, the statement **CSEG AT 0** must be replaced with **CSEG AT 8000H**. Then, this file must be assembled and linked with your program as specified in the file header.

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.

Function Pointers

Function pointers are one of the most difficult aspects of C to understand and to properly utilize. Most problems involving function pointers are caused by improper declaration of the function pointer, improper assignment, and improper dereferencing.

The following brief example demonstrates how to declare a function pointer (f), how to assign function addresses to it, and how to call the functions through the pointer. The **printf** routine is used for example purposes when running DS51 to simulate program execution.

```
#pragma code symbols debug oe

#include <reg51.h>      /* special function register declarations */
#include <stdio.h>      /* prototype declarations for I/O functions */

void func1(int d) {      /* function #1 */
    printf("In FUNC1(%d)\n", d);
}

void func2(int i) {      /* function #2 */
    printf("In FUNC2(%d)\n", i);
}

void main(void) {
    void (*f)(int i);    /* Declaration of a function pointer */
                        /* that takes one integer arguments */
                        /* and returns nothing */

    SCON = 0x50;         /* SCON: mode 1, 8-bit UART, enable rcvr */
    TMOD |= 0x20;        /* TMOD: timer 1, mode 2, 8-bit reload */
    TH1 = 0xf3;         /* TH1: reload value for 2400 baud */
    TR1 = 1;           /* TR1: timer 1 run */
    TI = 1;            /* TI: set TI to send first char of UART */

    while( 1 ) {
        f = (void *)func1; /* f points to function #1 */
        f(1);
        f = (void *)func2; /* f points to function #2 */
        f(2);
    }
}
```

NOTE

*Because of the limited stack space of the 8051, the linker overlays function variables and arguments in memory. When you use a function pointer, the linker cannot correctly create a call tree for your program. For this reason, you may have to correct the call tree for the data overlaying. Use the **OVERLAY** directive with the linker to do this. Refer to the *Ax51 Macro Assembler User's Guide* for more information.*

F

Glossary

A51

The standard 8051 Macro Assembler.

AX51

The extended 8051 Macro Assembler.

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. The first 128 characters are standardized. The remaining 128 are defined by the implementation.

batch file

An ASCII text file containing commands and programs that can be invoked from the command line.

Binary-Coded Decimal (BCD)

A BCD (Binary-Coded Decimal) is a system used to encode decimal numbers in binary form. Each decimal digit of a number is encoded as a binary value 4 bits long. A byte can hold 2 BCD digits – one in the upper 4 bits (or nibble) and one in the lower 4 bits (or nibble).

BL51

The standard 8051 linker/locator.

block

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

C51

The Optimizing C Compiler for classic 8051 and extended 8051 devices.

CX51

The Optimizing C Compiler for Philips 80C51MX architecture and the Dallas 80C390.

constant expression

Any expression that evaluates to a constant non-variable value. Constants may include character and integer constant values.

control

Command line control switch to the compiler, assembler or linker.

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

Instruction or control switch to the compiler, assembler or linker.

escape sequence

A backslash ('\') character followed by a single letter or a combination of digits that specifies a particular character value in strings and character constants.

expression

A combination of any number of operators and operands that produces a constant value.

formal parameters

The variables that receive the value of arguments passed to a function.

function

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

function body

A block containing the declarations and statements that make up a function.

function call

An expression that invokes and possibly passes arguments to a function.

function declaration

The name and return type of a function that is explicitly defined elsewhere in the program.

function definition

The name, formal parameters, return type, declarations, and statements describing what a function does.

function prototype

A function declaration that includes the list of formal parameters in parentheses following the function name.

in-circuit emulator (ICE)

A hardware device that aids in debugging embedded software by providing hardware-level single-stepping, tracing, and break-pointing. Some ICEs provide a trace buffer that stores the most recent CPU events.

include file

A text file that is incorporated into a source file.

keyword

A reserved word with a predefined meaning for the compiler or assembler.

L51

The **old** version of the 8051 linker/locator. L51 is replaced with the **BL51** linker/locator.

LX51

The extended 8051 linker/locator.

LIB51, LIBX51

The commands to manipulate library files using the 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.

manifest constant

A macro that is defined to have a constant value.

MCS[®] 51

The general name applied to the Intel family of 8051 compatible microprocessors.

memory model

Any of the models that specifies which memory areas are used for function arguments and local variables.

mnemonic

An ASCII string that represents a machine language opcode in an assembly language instruction.

MON51

An 8051 program that can be loaded into your target CPU to aid in debugging and rapid product development through rapid software downloading.

MSB

Most significant bit or byte.

newline character

A character used to mark the end of a line in a text file or the escape sequence ('**n**') to represent the newline character.

null character

ASCII character with the value 0 represented as the escape sequence ('**0**').

null pointer

A pointer that references nothing. A null pointer has the integer value 0.

object

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

object file

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

OH51, OHX51

The commands to convert absolute object files into Intel HEX file format.

opcode

Also referred to as operation code. An opcode is the first byte of a machine code instruction and is usually represented as a 2-digit hexadecimal number. The opcode indicates the type of machine language instruction and the type of operation to perform.

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 introduced by Intel at the beginning of the 1980's.

pointer

A variable containing 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 directives to the compiler.

relocatable

Object code that can be relocated and is not at a fixed address.

RTX51 Full

An 8051 Real-time Executive that provides a multitasking operating system kernel and library of routines for its use.

RTX51 Tiny

A limited version of RTX51.

scalar types

In C, integer, enumerated, floating-point, and pointer types.

scope

Sections of 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.

Special Function Register (SFR)

An SFR or Special Function Register is a register in the 8051 internal data memory space that is used to read and write to the hardware components of

the 8051. This includes the serial port, timers, counters, I/O ports, and other hardware control registers.

source file

A text file containing C program or assembly program code.

stack

An area of memory, indirectly accessed by a stack pointer, that shrinks and expands dynamically as items are pushed onto and popped off of the stack. Items in the stack are removed on a LIFO (last-in first-out) basis.

static

A storage class that, when used with a variable declaration in a function, causes variables to retain their value after exiting the block or function in which they are declared.

stream functions

Routines in the library that read and write characters using the input and output streams.

string

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

string literal

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

structure

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

structure member

One element of a structure.

token

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

two's complement

A binary notation that is used to represent both positive and negative numbers. Negative values are created by complementing all bits of a positive value and adding 1.

type

A description of the range of values associated with a variable. For example, an **int** type can have any value within its specified range (-32768 to 32767).

type cast

An operation in which an operand of one type is converted to another type by specifying the desired type, enclosed within parentheses, immediately preceding the operand.

μVision2

An integrated software development platform that supports the Keil Software development tools. μVision2 combines Project Management, Source Code Editing, and Program Debugging in one environment.

whitespace character

Characters used as delimiters in C programs such as space, tab, and newline.

wild card

One of the characters (? or *) that can be used in place of characters in a filename.

Index

#	134
##	135
#define	133
#elif	133
#else	133
#endif	133
#error	133
#if	133
#ifdef	133
#ifndef	133
#include	133
#line	133
#pragma	133
#undef	133
.I files	19
.LST files	19
.OBJ files	19
.SRC files	19
?C?xLDPTR	152
?C?XPAGE1RST	152
?C?XPAGE1SFR	152
?C?xSTXPTR	152
__C51__	136
__CX51__	136
__DATE__	136
__DATE2__	136
__FILE__	136
__LINE__	136
__MODEL__	136
__STDC__	136
__TIME__	136
_at	102,184,356
_chkfloat	219,243
_crol	207,219,246
_cror	207,219,247
_getkey	222,254
_irol	207,219,257
_iror	207,219,258
_lrol	207,219,275
_lror	207,219,276
_nop	207,225,285
_testbit	207,225,334
_tolower	217,338
_toupper	217,340
+INF	
described	180
8051 Derivatives	137
8051 Hardware Stack	117
8051 Memory Areas	88
8051 Variants	15
8051-Specific Optimizations	156
80C320/520 or variants	53
80C517 Routines	
acos517	226
asin517	226
atan517	226
atof517	226
cos517	226
exp517	226
log10517	226
log517	226
printf517	226
scanf517	226
sin517	226
sprintf517	226
sqrt517	226
sscanf517	226
strtod517	226
tan517	226
80C517.H	226
80C751.LIB	208
80x8252 or variants	50
A	
A51	
Interfacing	161
A51, defined	375
abs	218,231
ABSACC.H	227
Absolute Memory Access	
Macros	210
CBYTE	210
CWORD	210
DBYTE	211
DWORD	211
FARRAY	212

- FCARRAY 212
 - FCVAR 213
 - FVAR 213
 - PBYTE 214
 - PWORD 214
 - XBYTE 215
 - XWORD 215
 - Absolute Memory Locations 182
 - Absolute register addressing 24
 - Absolute value
 - abs 231
 - cabs 240
 - fabs 249
 - labs 270
 - Abstract Pointers 112
 - Access Optimizing 156
 - Accessing Absolute Memory Locations 182
 - acos 219,232
 - acos517 232
 - Additional items, notational conventions 5
 - Address of interrupts 123
 - ADuC 150
 - ADuC B2 Series 51,138
 - Advanced Programming Techniques 147
 - alien 130
 - Analog Devices 51,138,150
 - ANSI
 - Differences 349
 - Include Files 226
 - Library 207
 - Standard C Constant 136
 - ANSI, defined 375
 - Arc
 - cosine 232
 - sine 233
 - tangent 235,236
 - AREGS 24
 - Argument lists, variable-length ... 47,225
 - argument, defined 375
 - Arithmetic Accelerator 142
 - arithmetic types, defined 375
 - array, defined 375
 - ASCII, defined 375
 - asin 219,233
 - asin517 233
 - ASM 26
 - Assembly code in-line 26
 - Assembly listing 29
 - Assembly source file generation 78
 - assert 234
 - ASSERT.H 227
 - atan 219,235
 - atan2 219,236
 - atan517 235
 - Atmel
 - 89x8252 and variants 139
 - Atmel 80x8252 or variants 50
 - Atmel WM
 - dual DPTR support 146
 - AtmelWM dual DPTR 54
 - atof 218,237
 - atof517 237
 - atoi 218,238
 - atol 218,239
 - AUTOEXEC.BAT 17
 - AX51, defined 375
- ## B
-
- Basic I/O Functions 154
 - batch file, defined 375
 - bdata 89
 - bdata, tips for 372
 - big endian 367
 - Binary-Coded Decimal (BCD), defined 375
 - bit
 - As first parameter in function call 118
 - Bit shifting functions
 - _crol_ 219
 - _cror_ 219
 - _irol_ 219
 - _iror_ 219
 - _lrol_ 219
 - _lror_ 219
 - Bit Types 96
 - Bit-addressable objects 97
 - BL51, defined 375
 - block, defined 375
 - bold capital text, use of 5
 - bold type, use of 5

Books	
About the C Language	16
BR	28
braces, use of	5
BROWSE	28
Buffer manipulation routines	
memcpy	216,278
memchr	216,279
memcmp	216,280
memcpy	216,281
memmove	216,282
memset	216,283
Buffer Manipulation Routines	216
C	
C51 command	17
C51, defined	375
C517 CPU	48
C51C.LIB	208
C51FPC.LIB	208
C51FPL.LIB	208
C51FPS.LIB	208
C51INC	17
C51L.LIB	208
C51LIB	17
C51S.LIB	208
cabs	218,240
calloc	221,241
CALLOC.C	154
Case/Switch Optimizing	156
Categories of Cx51 directives	20
CBYTE	182,210
CD	29
ceil	219,242
Character Classification	
Routines	217
isalnum	217
isalpha	217
isctrl	217
isdigit	217
isgraph	217
islower	217
isprint	217
ispunct	217
isspace	217
isupper	217
isxdigit	217
Character Conversion and	
Classification Routines	217
Character Conversion Routines	217
_toupper	217
_toupper	217
toascii	217
toint	217
tolower	217
toupper	217
Choices, notational conventions	5
CO	31
code	88
CODE	29
Code generation options	156
Code Optimization	58
Common Block Subroutines	155
compact	119
COMPACT	30
Compact memory model	30
Compact Model	93
Compatibility	
differences from standard C	349
Differences to previous	
versions	353
Differences to Version 2	359
Differences to Version 3.0	358
Differences to Version 3.2	357
Differences to Version 3.4	356
Differences to Version 4	355
Differences to Version 5	354
Differences to Version 6.0	353
standard C library differences	
.....	349
Compiling	17
COND	31
Conditional compilation	31
const far	91
constant expression, defined	376
Constant Folding	155
Control directives	20
control, defined	376
cos	219,244
cos517	244
cosh	219,245
courier typeface, use of	5
CP	30
CTYPE.H	227

Customization Files	148
CWORD	182,210
Cx51	
Control directives	20
Errorlevel	19
Extensions	87
Output files.....	19
CX51 command.....	17
CX51, defined.....	376

D

Dallas 5240	52
Dallas 80C320/520 or variants	53
Dallas 80C390	52
Dallas 80C400	52
Dallas Semiconductor	
5240	141
80C320.....	140
80C390.....	141
80C400.....	141
80C420.....	140
80C520.....	140
80C530.....	140
data	89
Data Conversion Routines	218
abs	218
atof	218
atoi	218
atol	218
cabs	218
labs.....	218
strtod	218
strtol	218
strtoul	218
Data memory.....	89
Data Overlaying.....	156
data pointers.....	138,139,140,143,146
Data sizes	95
Data Storage Formats.....	174
Data type ranges.....	95
Data Types	95
DB.....	33
DBYTE.....	182,211
Dead Code Elimination.....	155
DEBUG	33
Debug information.....	33,59
Debugging.....	185
declaration, defined	376
define.....	133
DEFINE	34
Defining macros on the	
command line	34
definition, defined	376
Derivatives	15
DF	34
Differences from Standard C.....	349
Differences to Previous Versions	353
Directive categories.....	20
Directive reference	23
directive, defined.....	376
DISABLE.....	35
Disabling interrupts.....	35
Displayed text, notational	
conventions	5
Document conventions	5
double brackets, use of.....	5
DS80C390.....	150
DS80C400.....	150
DWORD.....	182,211

E

EJ	37
EJECT	37
elif	133
ellipses, use of.....	5
ellipses, vertical, use of.....	5
else	133
ENDASM.....	26
endian	367
endif	133
Environment Variables.....	17
EOF	229
error.....	133
ERRORLEVEL.....	19
escape sequence, defined.....	376
exp.....	219,248
exp517.....	248
exponent	177
expression, defined.....	376
Extended Memory	91
Extensions for Cx51	87
Extensions to C	87
External Data Memory	90

F

fabs	219,249
far	91
far memory	83
FARRAY	182,212
Fatal Error Messages	187
FCARRAY	182,212
FCVAR	182,213
FF	38
Filename, notational conventions	5
Files generated by Cx51	19
float	
exponent.....	177
mantissa	177
float numbers	177
FLOATFUZZY	38
Floating-point compare.....	38
Floating-Point Errors	180
+INF	180
-INF	180
Nan	180
Floating-point numbers.....	177
floor	219,250
fmod	219,251
Form feeds.....	37
formal parameters, defined	376
free.....	221,252
FREE.C	154
function body, defined	376
function call, defined	376
function declaration, defined	376
Function Declarations.....	116
function definition, defined	377
Function extensions	116,117
Function Parameters	161
Function Pointers, tips for	373
function prototype, defined.....	377
Function return values	118
Function Return Values	163
function, defined.....	376
Functions	116
Interrupt	123
Memory Models.....	119
Parameters in Registers.....	118
Recursive	127
Reentrant.....	127
Register Bank.....	120

Stack & Parameters	117
FVAR.....	182,213

G

General Optimizations	155
getchar.....	222,253
GETKEY.C.....	154
gets.....	222,255
Global Common Subexpression Elimination.....	155
Global register optimization.....	69
Glossary	375

H

High-Speed Arithmetic	144
-----------------------------	-----

I

I/O Functions	154
IBPSTACK	149
IBPSTACKTOP	149
ICE, defined.....	377
idata	89
IDATALEN	149
IEEE-754 standard.....	177
if.....	133
ifdef.....	133
ifndef.....	133
INCDIR.....	39
in-circuit emulator, defined.....	377
include.....	133
Include file listing	46
include file, defined	377
Include Files.....	39,226
80C517.H	226
ABSACC.H.....	227
ASSERT.H.....	227
CTYPE.H.....	227
INTRINS.H.....	227
MATH.H.....	228
REGxxx.H.....	226
SETJMP.H	228
STDARG.H.....	228
STDDEF.H.....	228
STDIO.H.....	229
STDLIB.H.....	229

STRING.H 229
 -INF
 described 180
 Infineon
 C517, 80C537, C509 143
 Infineon C517 48
 INIT.A51 151
 INIT_MEM.C 154
 init_mempool 221,256
 INIT_TNY.A51 151
 Initializing memory 149
 Initializing the stream I/O
 routines 222
 In-line assembly 26
 Integer promotion 41
 Interfacing C Programs to A51 161
 Interfacing C Programs to
 PL/M-51 173
 Internal Data Memory 89
 interrupt 121,124
 Interrupt
 Addresses 123
 Description 123
 Function rules 126
 Functions 123
 Numbers 123
 Interrupt vector 43
 Interrupt vector interval 40
 Interrupt vector offset 43
 INTERVAL 40
 INTPROMOTE 41
 INTRINS.H 227
 Intrinsic Routines 207
 crol 207
 cror 207
 irol 207
 ior 207
 lrol 207
 lror 207
 nop 207
 testbit 207
 INTVECTOR 43
 IP 41
 isalnum 217,259
 isalpha 217,260
 iscntrl 217,261
 isdigit 217,262

isgraph 217,263
 islower 217,264
 isprint 217,265
 ispunct 217,266
 isspace 217,267
 isupper 217,268
 isxdigit 217,269
 italicized text, use of 5
 IV 43

J

jmp_buf 209
 Jump Optimizing 155

K

Key names, notational
 conventions 5
 keyword, defined 377
 Keywords 87

L

L51, defined 377
 LA 45
 labs 218,270
 Language elements, notational
 conventions 5
 Language Extensions 87
 large 119
 LARGE 45
 Large memory model 45
 Large Model 93
 LC 46
 LIB51, defined 377
 Library Files 208
 80C751.LIB 208
 C51C.LIB 208
 C51FPC.LIB 208
 C51FPL.LIB 208
 C51FPS.LIB 208
 C51L.LIB 208
 C51S.LIB 208
 Library Reference 207
 Library Routines
 ANSI, excluded from Cx51 350
 ANSI, included in Cx51 349

non-ANSI.....	351
Library Routines by Category.....	216
library, defined	377
LIBX51, defined.....	377
line.....	133
Linker Location Controls.....	183
LISTINCLUDE.....	46
Listing file generation.....	68
Listing file page length.....	65
Listing file page width.....	66
Listing include files	46
little endian.....	367
log.....	219,271
log10.....	219,272
log10517.....	272
log517.....	271
longjmp.....	225,273
LSB, defined.....	377
LX51, defined.....	377

M

macro, defined.....	377
malloc.....	221,277
MALLOC.C.....	154
manifest constant, defined	377
mantissa.....	177
Manual organization.....	4
Math Routines	219
chkfloat.....	219
crol.....	219
cror.....	219
irol.....	219
iror.....	219
lrol.....	219
lrer.....	219
acos.....	219
asin.....	219
atan.....	219
atan2.....	219
ceil.....	219
cos.....	219
cosh.....	219
exp.....	219
fabs.....	219
floor.....	219
fmod.....	219
log.....	219
log10.....	219
modf.....	219
pow.....	219
rand.....	219
sin.....	219
sinh.....	219
sqrt.....	219
srand.....	219
tan.....	219
tanh.....	219
MATH.H.....	228
MAXARGS.....	47
Maximum arguments in variable-length argument lists.....	47
MCS [®] 51, defined.....	377
memcpy.....	216,278
memchr.....	216,279
memcmp.....	216,280
memcpy.....	216,281
memmove.....	216,282
Memory Allocation.....	154
Memory Allocation Routines.....	221
calloc.....	221
free.....	221
init_mempool.....	221
malloc.....	221
realloc.....	221
Memory areas.....	88
external data.....	90
internal data.....	89
program.....	88
special function register.....	91
Memory class names.....	81
Memory Model.....	92
Compact.....	93
Function.....	119
Large.....	93
Small.....	92
memory model, defined.....	378
Memory Type.....	93
bdata.....	89,94
code.....	88,94
const far.....	91
data.....	89
far.....	83,91,94
idata.....	89,94
pdata.....	90,94

xdata..... 90,94
 Memory Typedata..... 94
 memset..... 216,283
 MicroConverter..... 150
 MicroConverter B2 Series 138
 MicroConverter B2 Series 51
 Miscellaneous Routines 225
 nop..... 225
 testbit..... 225
 longjmp..... 225
 setjmp..... 225
 mnemonic, defined..... 378
 MOD517..... 48,143
 MODA2..... 50,139
 MODAB2..... 51,138
 MODDA2..... 52
 MODDP2..... 53,140
 modf..... 219,284
 MODP2..... 54,146
 monitor51, defined..... 378
 MSB, defined..... 378

N

NaN..... 244,287,302,303,332
 described..... 180
 newline character, defined 378
 NOAMAKE..... 55
 NOAREGS 24
 NOAU..... 49
 NOCO..... 31
 NOCOND 31
 NODP8 49
 NOEXTEND 56
 NOINTPROMOTE..... 41
 NOINTVECTOR..... 43
 NOIP..... 41
 NOIV 43
 NOMOD517 48
 NOMODA2..... 50,139
 NOMODAB2..... 51,138
 NOMODDA2..... 52
 NOMODDP2..... 53,140
 NOMODP2..... 54,146
 NOOBJECT..... 57
 NOOJ..... 57
 NOPR 68
 NOPRINT..... 68

NOREGPARDS 71
 NULL..... 229
 null character, defined 378
 null pointer, defined 378

O

O2..... 61
 OA..... 58
 OB..... 60
 OBJECT..... 57
 Object file generation..... 57
 object file, defined..... 378
 object, defined..... 378
 OBJECTADVANCED..... 58
 OBJECTEXTEND..... 59
 OE..... 59
 offsetof..... 286
 OH51, defined..... 378
 OHS51..... 356
 OHX51, defined..... 378
 OJ..... 57
 OMF2..... 61
 Omitted text, notational
 conventions 5
 ONEREBANK 60
 opcode, defined 378
 operand, defined 378
 operator, defined 379
 OPTIMIZE..... 62
 Optimizer 155
 Optimizing programs..... 62
 Optimum Code
 Local Variables..... 364
 Memory Model..... 361
 Other Sources 364
 Variable Location 363
 Variable Size 363
 Variable Types 364
 Optional items, notational
 conventions 5
 Options for Code Generation 156
 OR..... 64
 ORDER..... 64
 Order of variables..... 64
 OT..... 62
 Output files..... 19
 Overlapping Segments..... 166

P

Page length in listing file	65
Page width in listing file	66
PAGELNGTH	65
PAGEWIDTH	66
Parameter Passing in Fixed Memory Locations.....	163
Parameter Passing in Registers	162
Parameter Passing Via Registers	155
parameter, defined	379
Passing arguments in registers	71
Passing Parameters in Registers	118
PATH	17
PBPSTACK.....	150
PBPSTACKTOP	150
PBYTE	182,214
pdata	90
PDATALEN	149
PDATASTART	149
Peephole Optimization	156
Philips	
80C51MX	146
8xC750	145
8xC751	145
8xC752	145
dual DPTR support	146
Philips 80C51MX	150
Philips 80C75x	150
Philips dual DPTR.....	54
Philips LPC.....	150
PL	65
PL/M-51	130
Defined	379
Interfacing	173
Pointer Conversions.....	109
Pointers.....	104
Generic.....	104
Memory-specific	107
pointers, defined	379
pow	219,287
PP	67
PPAGE	150
PPAGEENABLE.....	150
PR.....	68
pragma	133
pragma, defined	379
Predefined Macro Constants.....	136
__C51__	136
__CX51__	136
__DATE__	136
__DATE2__	136
__FILE__	136
__LINE__	136
__MODEL__	136
__STDC__	136
__TIME__	136
Preface	3
PREPRINT	67
Preprocessor.....	133
Preprocessor directives	
define.....	133
elif	133
else	133
endif	133
error.....	133
if	133
ifdef.....	133
ifndef	133
include.....	133
line.....	133
pragma.....	133
undef.....	133
Preprocessor output file	
generation.....	67
preprocessor, defined	379
PRINT	68
Printed text, notational conventions	5
printf	222,288
printf, tips for	370
printf517	288
Program Memory	88
Program memory size	75
putchar	222,293
PUTCHAR.C	154
puts.....	222,294
PW	66
PWORD.....	182,214
R	
R0-R7.....	24
rand	219,295
Range for data types.....	95
RB	70

realloc 221,296
 REALLOC.C 154
 Real-Time Function Tasks 131
 Recursive Code, tips for 369
 Recursive Functions 127
 reentrant 127
 Reentrant Functions 127
 REGFILE 69
 Register bank 24,70
 Register Bank 120,122
 Register banks 24
 Register Usage 166
 Register Variables 155
 REGISTERBANK 70
 Registers used for parameters 71
 Registers used for return values 118
 REGPARMS 71
 relocatable, defined 379
 Rename memory classes 81
 RESTORE 76
 RET_PSTK 73
 RET_XSTK 73
 Return values 118
 Reuse of Common Entry Code 155
 RF 69
 ROM 75,141
 Routines by Category 216
 RTX51 Full, defined 379
 RTX51 Tiny, defined 379
 Rules for interrupt functions 126

S

sans serif typeface, use of 5
 SAVE 76
 SB 80
 sbit 100
 scalar types, defined 379
 scanf 222,297
 scanf517 297
 scope, defined 379
 Segment Naming Conventions 157
 Serial Port, initializing for
 stream I/O 222
 setjmp 225,301
 SETJMP.H 228
 sfr 99
 sfr16 100
 SIECO-51 357
 sin 219,302
 sin517 302
 sinh 219,303
 Size of data types 95
 SM 77
 small 119
 SMALL 77
 Small memory model 77
 Small Model 92
 source file, defined 380
 Special Function Register (SFR),
 defined 379
 Special Function Register
 Memory 91
 Special Function Registers 99
 sprintf 222,304
 sprintf517 304
 sqrt 219,306
 sqrt517 306
 srand 219,307
 SRC 78
 sscanf 222,308
 sscanf517 308
 ST 79
 Stack 117
 Stack usage 73
 stack, defined 380
 Standard Types 209
 jmp_buf 209
 va_list 209
 START_AD.A51 150
 START390.A51 150
 START751.A51 150
 STARTLPC.A51 150
 STARTUP.A51 149,150
 static, defined 380
 STDARG.H 228
 STDDEF.H 228
 STDIO.H 229
 STDLIB.H 229
 Storage format
 bit 174
 char 175
 code pointer 175
 data pointer 175
 enum 175

far pointer	176	strrpos.....	224
float.....	177	strspn.....	224
generic pointer	176	strstr.....	224
idata pointer	175	string, defined	380
int.....	175	STRING.H.....	229
long.....	175	Stringize Operator.....	134
pdata pointer	175	strlen	224,315
short.....	175	strncat.....	224,316
xdata pointer	175	strncmp.....	224,317
Store return addresses.....	73	strncpy.....	224,318
strcat	224,310	strops.....	224
strchr.....	224,311	strpbrk.....	224,319
strcmp.....	224,312	strpos.....	320
strcpy.....	224,313	strrchr.....	224,321
strcspn.....	224,314	strrpbrk.....	224,322
stream functions, defined.....	380	strrpos	224,323
Stream I/O Routines	222	strspn.....	224,324
_getkey.....	222	strstr	224,325
getchar	222	strtod	218,326
gets.....	222	strtod517	326
Initializing.....	222	strtol	218,328
printf.....	222	strtoul	218,330
putchar	222	structure member, defined.....	380
puts	222	structure, defined.....	380
scanf.....	222	Symbol table generation.....	80
sprintf.....	222	SYMBOLS.....	80
sscanf	222	Syntax and Semantic Errors.....	191
ungetchar.....	222		
vprintf.....	222	T	
vsprintf.....	222	tan	219,332
Stream Input and Output.....	222	tan517	332
STRING	79	tanh	219,333
string literal, defined.....	380	TMP.....	17
String Manipulation Routines.....	224	toascii.....	217,335
strcat	224	toint.....	217,336
strchr.....	224	token, defined.....	380
strcmp.....	224	Token-pasting operator	135
strcpy.....	224	tolower	217,337
strcspn.....	224	toupper	217,339
strlen	224	two's complement, defined	380
strncat.....	224	type cast, defined	380
strncmp.....	224	type, defined.....	380
strncpy.....	224		
strncpy.....	224	U	
strpbrk.....	224	UCL	81
strpos.....	224	Uncalled Functions, tips for.....	371
strrchr.....	224		
strrpbrk.....	224		

undef 133
 ungetchar 222,341
 USERCLASS 81
 using 120,125
 Using Monitor-51, tips for 371

V

va_arg 225,342
 va_end 225,344
 va_list 209
 va_start 225,345
 VARBANKING 83
 Variable-length argument list
 routines 225
 Variable-Length Argument List
 Routines
 va_arg 225
 va_end 225
 va_start 225
 Variable-length argument lists 47
 Variables, notational
 conventions 5
 VB 83
 vertical bar, use of 5
 vprintf 222,346
 vsprintf 222,348

W

Warning detection 84
 WARNINGLEVEL 84
 Warnings 203
 WATCHDOG 151
 whitespace character, defined 381
 wild card, defined 381
 WL 84

X

XBANKING.C 152
 XBPSTACK 150
 XBPSTACKTOP 150
 XBYTE 182,215
 XC 85
 XCROM 85
 xdata 90
 XDATALEN 149
 XDATASTART 149
 XOFF 154
 XON 154
 XRAM 90
 XWORD 182,215