

8051 IAR C Compiler
Reference Guide

for the
8051 Family of Microcontrollers

COPYRIGHT NOTICE

© Copyright 2001 IAR Systems. All rights reserved.

No part of this document may be reproduced without the prior written consent of IAR Systems. The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such a license.

DISCLAIMER

The information in this document is subject to change without notice and does not represent a commitment on any part of IAR Systems. While the information contained herein is assumed to be accurate, IAR Systems assumes no responsibility for any errors or omissions.

In no event shall IAR Systems, its employees, its contractors, or the authors of this document be liable for special, direct, indirect, or consequential damage, losses, costs, charges, claims, demands, claim for lost profits, fees, or expenses of any nature or kind.

TRADEMARKS

IAR and C-SPY are registered trademarks of IAR Systems. IAR Embedded Workbench, IAR XLINK Linker, and IAR XLIB Librarian are trademarks of IAR Systems. Intel is a registered trademark of Intel Corporation. Microsoft is a registered trademark, and Windows is a trademark of Microsoft Corporation.

All other product names are trademarks or registered trademarks of their respective owners.

EDITION NOTICE

First edition: January 2001

Part number: C8051-1

Contents

Tables	ix
Preface	xi
Who should read this guide	xi
How to use this guide	xi
What this guide contains	xii
Document conventions	xiii
Part I. Using the compiler	1
Efficient coding techniques	3
Efficient coding	3
Using language extensions	3
Extended keywords	4
#pragma directives	5
Predefined symbols	5
Intrinsic functions	5
Configuration	7
Introduction	7
Run-time library	8
Memory models	9
Specifying the memory model	9
Memory location	10
Memory areas	11
Non-volatile RAM	20
Banking	20
Banked memory	21
How to use the banked memory model	24

Linker command file	27
Stack size	28
Estimating the required stack size	28
Heap size	28
Input and output	29
Puchar and getchar	29
Printf and sprintf	30
Scanf and sscanf	31
Initialization	32
Variable and I/O initialization	32
Modifying CSTARTUP	32
CSTARTUP.S03	34
Optimization	37
Customizing the run-time library	37
Multi-module linking	37
Target-specific support	38
80751 Support	38
80517 Support	39
80320 Support	40
Assembly language interface	43
Calling convention	43
Register usage	44
Parameters and local variables	44
Limitations	45
Reentrant parameters	46
Creating skeleton code	48
Assembler support directives	49
\$DEFFN	49
\$REFFN	50
\$IFREF	50
\$LOCBD, \$LOCBI, \$LOCBB, and \$LOCBX	50
\$PRMBD, \$PRMBI, \$PRMBB, and \$PRMBX	51
\$BYTE3	51

Example	51
Reentrant functions	54
Interrupt functions	54
Defining interrupt vectors	55
Part 2: Compiler reference	57
Data representation	59
Data types	59
Enum type	59
Bitfields	60
Char type	60
Floating point	60
Bit variables	60
Special Function Register variables	61
Pointers	61
Code pointers	61
Data pointers	61
Segments	67
Memory maps	67
Descriptions of segments	71
Compiler options	83
Setting compiler options	83
Specifying options using environment variables	83
Summary of compiler options	84
Descriptions of compiler options	85
Extended keywords	107
Using extended keywords	107
Address control	107
I/O access	108
Bit variables	108
Non-volatile RAM	108

Interrupt routines	108
Descriptions of extended keywords	108
#pragma directives	121
#pragma directives summary	121
Bitfield orientation	121
Extension control	121
Function attribute	121
Memory usage	121
Warning message control	122
Descriptions of #pragma directives	122
Predefined symbols	133
Descriptions of predefined symbols	133
Intrinsic functions	135
Descriptions of intrinsic functions	135
K&R and ANSI C language definitions	139
Introduction	139
Definitions	139
entry keyword	139
const keyword	139
volatile keyword	140
signed keyword	140
void keyword	140
enum keyword	140
Data types	141
Function definition parameters	141
Function declarations	141
Hexadecimal string constants	142
Structure and union assignments	142
Shared variable objects	143

Using C with PL/M	145
Using the object file converter	145
Linking the converter files	146
Compiling PL/M functions	147
Tiny-51	151
Introduction	151
General characteristics	151
Terminology	151
Principles of operation	153
Restrictions on TINY-51	155
Installing TINY-51	155
Configuring TINY-51	156
Task timer	156
Register bank 3	156
Task-switching time	156
Timeout task	156
Building a TINY-51 application	156
Using preemptive multitasking	157
Using non-preemptive multitasking	161
Descriptions of TINY-51 functions	164
Diagnostics	173
Severity levels	173
Command line error messages	173
Compilation error messages	173
Compilation warning messages	173
Compilation fatal error messages	173
Compilation internal error messages	173
Compilation memory overflow message	174
Compilation error messages	174
8051-specific error messages	187
80751-specific error messages	189
Compilation warning messages	190
8051-specific warning messages	196

Part 3. Library functions	197
General C library definitions	199
Introduction	199
Library object files	199
Header files	199
Library definitions summary	200
Character handling – ctype.h	200
Low-level routines – icclbutl.h	200
Mathematics – math.h	201
Non-local jumps – setjmp.h	202
Variable arguments – stdarg.h	202
Input/Output – stdio.h	202
General utilities – stdlib.h	202
String handling – string.h	204
Assertions – assert.h	205
Miscellaneous header files	205
Index	207

Tables

1: Typographic conventions used in this guide	xiii
2: Features handled by configurable elements	8
3: Run-time library files	8
4: Memory model characteristics	9
5: Command file names	27
6: Parameters, types, and locations	43
7: Return registers	44
8: Compiler options	46
9: Compiling skeleton code in IAR Embedded Workbench	49
10: Functions used in \$DEFFN	50
11: Data types	59
12: Code pointers	61
13: Memory area	62
14: Environment variables	84
15: Command line options	84
16: Reentrant functions	88
17: Options for the -h compiler option	96
18: Specifying memory model (-m)	98
19: Generating debug information (-r)	101
20: Optimizing for speed (-s)	102
21: Specifying processor options	104
22: Disabling warning messages	104
23: Including cross-references in list file (-x)	105
24: Optimizing for size (-z)	106
25: Reserved keywords	107
26: Extended keywords general parameters	108
27: Bit variable types	109
28: Memory areas for each register bank	113
29: <code>_args\$</code> (intrinsic function)	135
30: <code>_argt\$</code> (intrinsic function)	136
31: Function differences between K&R and ANSI	141

32: K&R and ANSI function declarations	141
33: Shared variable objects	143
34: Matched static and global variables from C to PL/M	148
35: Task states in TINY-51	152
36: TINY-51 functions summary	164
37: Options that cause the compiler to use more memory	174
38: Suggestions for illegally used keywords	188
39: Miscellaneous header files	205

Preface

Welcome to the 8051 IAR C Compiler Reference Guide. The purpose of this guide is to provide you with detailed reference information that can help you to customize the 8051 IAR C Compiler to best suit your application requirements. This guide also gives you suggestions on coding techniques so that you can develop applications with maximum efficiency. The 8051 IAR C Compiler supports C for the 8051 microcontrollers.

Who should read this guide

You should read this guide if you plan to develop an application using C for the 8051 microcontroller and need to get detailed, reference information on how to use the 8051 IAR C Compiler. In addition, you should have working knowledge of the following:

- The architecture and instruction set of the 8051 microcontroller; refer to the chip manufacturer's documentation for information about the 8051 architecture and instruction set
- The C programming language
- Windows 95/98/2000 or Windows NT, depending on your operating system.

For information about programming with the 8051 IAR Assembler, refer to the *8051 IAR Assembler Reference Guide*.

How to use this guide

When you first begin using 8051 IAR C Compiler, you should read *Part 1. Using the compiler* in this reference guide.

If you are an intermediate or advanced user and have already configured the compiler, you can focus more on *Part 2: Compiler reference*.

If you are new to using the IAR toolkit, we recommend that you first read the initial chapters of the *8051 IAR Embedded Workbench™ User Guide*. They include comprehensive information about the installation of all IAR tools and give product overviews, as well as tutorials that can help you get started.

What this guide contains

Below is a brief outline and summary of the chapters in this guide.

Part 1. Using the compiler

- *What's new in this product* highlights and summarizes the new features in this product.
- *Efficient coding techniques* provides programming hints and information about how to fine-tune your application using the IAR toolkit, and how to get the most use out of 8051 IAR C Compiler's features.
- *Configuration* describes how to configure the compiler using the IAR toolkit to suit the requirements of your application, for example, setting up project options and customizing the linker command file.
- *Assembly language interface* describes the interface between a C main program and the assembly language routines.

Part 2: Compiler reference

- *Data representation* describes the available data types, pointers, and structure types.
- *Segments* lists the segments available, describes the naming convention, and gives reference information about each segment.
- *Compiler options* explains how to set compiler options from the command line and gives detailed reference information about each option.
- *Extended keywords* describes the non-standard keywords that support specific features of the 8051 microcontroller for data storage, function execution, function calling convention, and function storage.
- *#pragma directives* describes the syntax and provides a description of the #pragma directives of the 8051 IAR C Compiler.
- *Predefined symbols* describes the syntax and provides a description of the predefined preprocessor symbols that are supported in the 8051 IAR C Compiler.
- *Intrinsic functions* lists and describes the intrinsic functions provided in the 8051 IAR C Compiler.
- *Diagnostics* contains information about the severity levels and lists the 8051-specific warning and error messages.

Part 3. Library functions

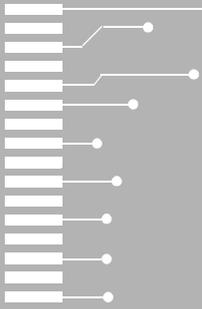
- *General C library definitions* gives an introduction to the C library functions and summarizes them according to the header file.

Document conventions

This guide uses the following typographic conventions:

Style	Used for
<code>computer</code>	Text that you enter or that appears on the screen.
<code>parameter</code>	A label representing the actual value you should enter as part of a command.
[option]	An optional part of a command.
{a b c}	Alternatives in a command.
bold	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.
<i>reference</i>	A cross-reference within or to another part of this guide.
	Identifies instructions specific to the versions of the IAR Systems tools for the IAR Embedded Workbench interface.
	Identifies instructions specific to the command line versions of IAR Systems development tools.

Table 1: Typographic conventions used in this guide



Part I. Using the compiler

This part of the 8051 IAR C Compiler Reference Guide contains the following chapters:

- Efficient coding techniques
- Configuration
- Assembly language interface.

Efficient coding techniques

This chapter provides programming hints that help you write more efficient code and gives information about how to fine-tune your application using the IAR toolkit so that you can fully take advantage of the 8051 IAR C Compiler's features.

Efficient coding

It is important to appreciate the limitations of the 8051 architecture in order to avoid the use of inefficient language constructs. The following is a list of recommendations on how best to use the 8051 IAR C Compiler.

- Use 16-bit data types and `char` whenever possible. `long` integers have no direct support in the 8051 architecture. Also note that, according to the ANSI C standard, all data types that are shorter than `int` should undergo integral promotion, that is, implicit type conversions, when used in arithmetic expressions.
- Use `unsigned` data types whenever possible. The 8051 IAR C Compiler generally performs `unsigned` operations more efficiently than the `signed` counterparts. Especially this applies to type conversions, comparison, array indexing and some arithmetic operations, such as `>>` and `/`.
- Use ANSI prototypes. Function calls to ANSI functions are performed more efficiently than K&R-style functions; see *K&R and ANSI C language definitions*, page 139.
- Use the smallest memory model possible.
- Sensible use of the memory attributes can enhance both speed and code size in critical applications; see *Extended keywords*, page 107, for detailed information.
- Bitfield types should be used only to conserve data memory space as they execute slowly on the 8051 microcontroller. Use a bit mask on `unsigned char` or `unsigned int` instead of bitfields. If you must use bitfields, use `unsigned` for efficiency.

Using language extensions

The IAR C Compiler provides a number of powerful extensions that support specific features of the 8051 family of microcontrollers.

The 8051-specific extensions are provided as extended keywords, `#pragma` directives, predefined symbols, and intrinsic functions.

EXTENDED KEYWORDS

By default, the compiler conforms to the ANSI specifications and 8051-specific extensions are not available. The compiler option `-e` makes the extended keywords available, and hence reserves them so that they cannot be used as variable names. See *-e*, page 89.

The extended keywords provide the following facilities:

Addressing control

By default the address range in which the compiler places a variable is determined by the selected memory configuration. The program may achieve additional efficiency for special cases by overriding the default storage by using the `data`, `idata`, `bdata`, `pdata`, `xdata`, `code` extended keywords.

I/O access

The program may access the I/O system of the 8051 microcontroller using the `sfr` data types.

Bit variables

The program may take advantage of the 8051 bit-addressing modes by using the `bit` data type.

Non-volatile RAM

Variables may be placed in non-volatile RAM by using the `no_init` data type modifier.

Calling mechanisms

By default the mechanism by which the compiler calls a function is determined by the memory model chosen. The program may achieve additional efficiency for special cases by overriding the default using one of the function modifiers:

```
reentrant
idata_reentrant
non_banked
plm
interrupt
monitor
```

For complete information about the available extended keywords, see the chapter *Extended keywords*.

#PRAGMA DIRECTIVES

The `#pragma` directives control how the compiler allocates memory, whether the compiler allows extended keywords, and whether the compiler outputs warning messages.

`#pragma` directives provide control of extension features while remaining within the standard language syntax.

Notice that `#pragma` directives are available regardless of the `-e` option.

For complete information about the `#pragma` directives, see the chapter *#pragma directives*.

PREDEFINED SYMBOLS

With the predefined preprocessor symbols, you can inspect your compile-time environment, for example the time and date of compilation.

For detailed descriptions of the predefined symbols, see the chapter *Predefined symbols*.

INTRINSIC FUNCTIONS

Intrinsic functions allow low-level control of the 8051 microcontroller. The intrinsic functions compile into in-line code, either as a single instruction or as a short sequence of instructions.

To use the intrinsic functions in a C application, include the header file `special.h`.

For complete information about the available intrinsic functions, see the chapter *Intrinsic functions*.

For details concerning the effects of the intrinsic functions, see the documentation of the 8051 microcontroller.

Configuration

This chapter describes how to configure the compiler using the IAR toolkit to suit the requirements of your application. This includes setting up project options used for specifying the 8051 derivative and memory model for your project, memory location, linker command file, run-time libraries, initialization of variables and I/O, input and output operations, module consistency, and optimization.

Introduction

The IAR toolkit for the 8051 microcontroller contains a number of components that you can modify according to the requirements of your application, as well as your preferences regarding the run-time situation. There can be considerable variations in the configuration of an 8051 microcontroller system's use of internal and external ROM and RAM. There are also differing requirements for stack size, the need for reentrancy, large libraries, or time-critical functions. This chapter provides information about the configuration and use of:

- Run-time libraries.
- The options used for specifying the memory model for your application. This requires selecting memory models and link options to match the ROM (non-bankable functions, bankable functions, constants, and initial values) and RAM (directly addressable internal memory, indirectly addressable internal memory, external memory, and external non-volatile memory).
- Banking.
- The linker command file which is used for specifying segments, address ranges, stack size, and heap size. The compiler and linker identify different types of memory by giving them different segment names (RCODE, CODE, CSTR, D_IDATA, and NO_INIT, for example).
- I/O operations.
- Initialization procedure.
- Optimization.
- Customizing the C library.

Note: Some of the configuration procedures involve editing the standard files, and we recommend that you make copies of the originals before beginning.

The chapter *Efficient coding techniques* gives some hints about how to write efficient code.

For information about how to configure the hardware or peripheral devices, refer to the hardware documentation.

The size and location of the segments and the function characteristics are determined by the command line options or control files used with the 8051 IAR C Compiler and XLINK. Each feature of the environment or usage is handled by one or more of the following configurable elements:

Feature	Configurable element
Memory model	Compiler option, linker option
Memory location	Linker command file
Non-volatile RAM	Compiler keyword, linker command file.
Stack size	Linker command file.
putchar and getchar functions	Run-time library module
printf/scanf facilities	Linker command file
Heap size	Heap library module
Initialization of hardware and memory	CSTARTUP module

Table 2: Features handled by configurable elements

Run-time library

Library files are provided for the different memory models, processor types and reentrant code. The library file to use is selected according to the following table:

Memory model	Default	a (80751)	Reentrant
Tiny	cl8051t.r03	cl8051ta.r03	cl8051tr.r03
Small	cl8051s.r03	cl8051sa.r03	cl8051sr.r03
Compact	cl8051c.r03		cl8051cr.r03
Medium	cl8051m.r03		cl8051mr.r03
Large	cl8051l.r03		cl8051lr.r03
Banked	cl8051b.r03		cl8051br.r03

Table 3: Run-time library files

By default, the library files are provided in the directory path:

```
\ew23\8051\lib\
```

For information about how to modify the run-time library, see *Customizing the run-time library*, page 37.



Specifying the run-time library in the IAR Embedded Workbench

In the IAR Embedded Workbench, the library file is selected automatically depending on the linker command file in use. You can, however, override the default library and specify your choice of run-time library in the **Library** page in the **XLINK** category when setting project options. Note that the system library filename should not be included in the linker command file.



Specifying the run-time library using the command line

For the command line version, you can specify the system library filename in the linker command file or on the command line.

Memory models

The 8051 IAR C Compiler supports six memory models. These offer a choice of default placement for local and global variables within the ROM (CODE) and RAM (DATA) memory.

The following table gives details of the different memory models:

Memory model	Command line option	Global data	Local data	External RAM	Code size	Typical chip	Run-time library	Run-time library 80751	Linker command file
Tiny	-mt	DATA	DATA	no	64K	8051	cl8051t	cl805ta	lnk8051.xcl, lnk8051a.xcl
Small	-ms	IDATA	IDATA	no	64K	8052	cl8051s	cl8051sa	lnk8051.xcl, lnk8051a.xcl (with the -vl option)
Compact	-mc	XDATA	DATA	yes	64K	8031	cl8051c	-	lnk8051.xcl
Medium	-mm	XDATA	IDATA	yes	64K	8032	cl8051m	-	lnk8051.xcl
Large	-ml	XDATA	XDATA	yes	64K	8032	cl8051l	-	lnk8051.xcl
Banked	-mb	XDATA	XDATA	yes	>64K	8032	cl8051b	-	lnk8051b.xc

Table 4: Memory model characteristics

SPECIFYING THE MEMORY MODEL

Your project may use only one memory model at a time, and the same model must be used by all user modules and all library modules.



Specifying the memory model using the IAR Embedded Workbench

The memory model is specified in the **Options** dialog box under the **General** category in the **Target** tab; see the *8051 IAR Embedded Workbench™ User Guide*.



Specifying the memory model using the command line

When using the command line you specify the memory model to the 8051 IAR C Compiler using the `-m` command line option, as shown in the above table.

Note: If the `-v1` (80751) option is selected then only `-mt` and `-ms` are available.

For example, to compile `myprog` with optimization in the medium memory model, use the command:

```
icc8051 myprog -mm -z9
```

If no memory model options are included, the compiler uses the tiny memory model.

To specify the memory model to the linker, select an appropriate normal or reentrant version of the library file and change it in the corresponding linker command file.

For example, to link the module `myprog` (previously compiled for the medium memory model) for the medium memory model, use the following command:

```
xlink myprog -f lnk8051
```

The `-f` option specifies a command filename (the extension `.xcl` is assumed).

In addition to these six standard library files, there are alternatives that provide support for reentrant code (`cl8051*r.r03`) and 80751 (`cl8051*a.r03`). This means that the precise filename to use will depend not just on the memory model, but also on which of these other options chosen. See *Run-time library*, page 8, for a complete list of library filenames.

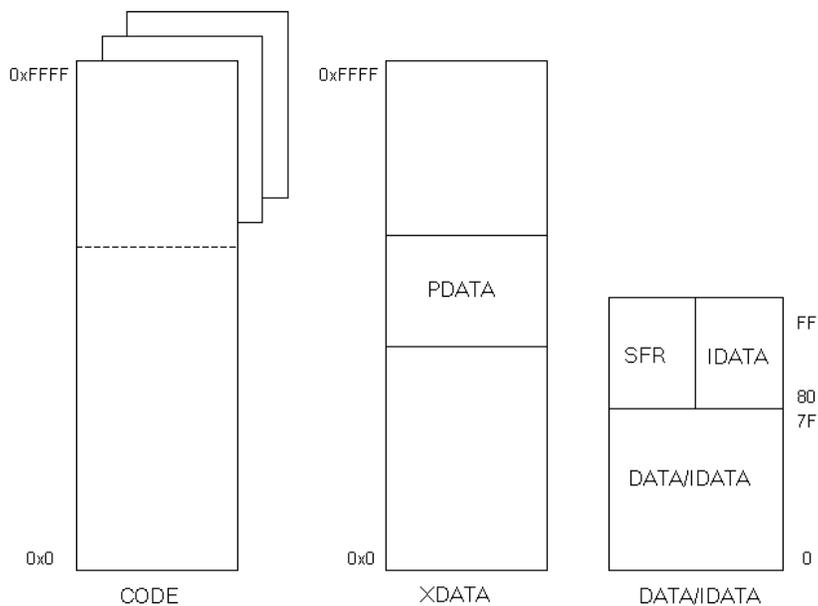
MEMORY LOCATION

You need to specify to XLINK your hardware environment's address ranges for ROM and RAM. Do this in your copy of the linker command file template.

For details of specifying the memory address ranges, see the contents of the linker command file template and the IAR XLINK Linker in the *IAR XLINK Linker™ and IAR XLIB Librarian™ Reference Guide*.

MEMORY AREAS

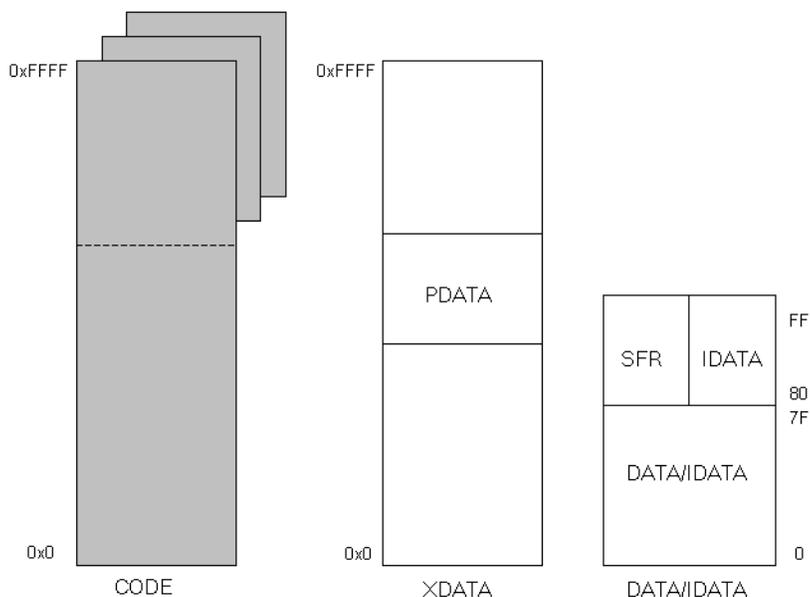
This section describes each of the seven memory areas individually—CODE, DATA, IDATA, XDATA, BDATA, PDATA, and SFR—and the relationship between the 8051 hardware and the 8051 IAR C Compiler memory models.



CODE memory

The CODE memory address area extends from address 0h to 0FFFFh. Depending on the particular type of chip, anywhere from none to 16 Kbytes or more may be on-chip. Off-chip ROM begins at the address following the end of internal ROM and is accessed using the same addressing mode as the internal ROM. If only a part of the full 64 Kbytes CODE address space is used, that part may be located anywhere within the 64 Kbyte area. Usually CODE memory will begin at 0000, because that is where the first instruction is fetched by the processor after a power-on or reset.

The CODE memory area contains the IVT (Interrupt Vector Table), initialization code, compiler run-time library routines, variable initializers, constant data, and user code.



The banked memory model allows up to 256 banks of ROM to be accessed by the compiler. In bank mode, a function is addressed by a 16-bit address plus a bank number. The bank number is sent to the hardware via one of the user ports. In bank mode, all of the C runtime library code, ISRs (Interrupt Service Routines), constant data, and variable initial values must remain in a root bank because they are always accessed non-banked functions. See *Banking*, page 20.

IDATA memory

The 256 bytes of the IDATA section of the 8051's internal RAM memory begin at address 00 and continue up to FF hex. Certain 8051-family chips, such as the original 8051, have only 128 bytes of internal RAM, and so IDATA user RAM is available only up to 7Fh, and between 80h and FFh is SFR space only.

Data objects in the program which are defined to reside in IDATA memory are placed in this memory area and are always accessed by the compiler using the indirect addressing mode of the 8051 MOV instruction.

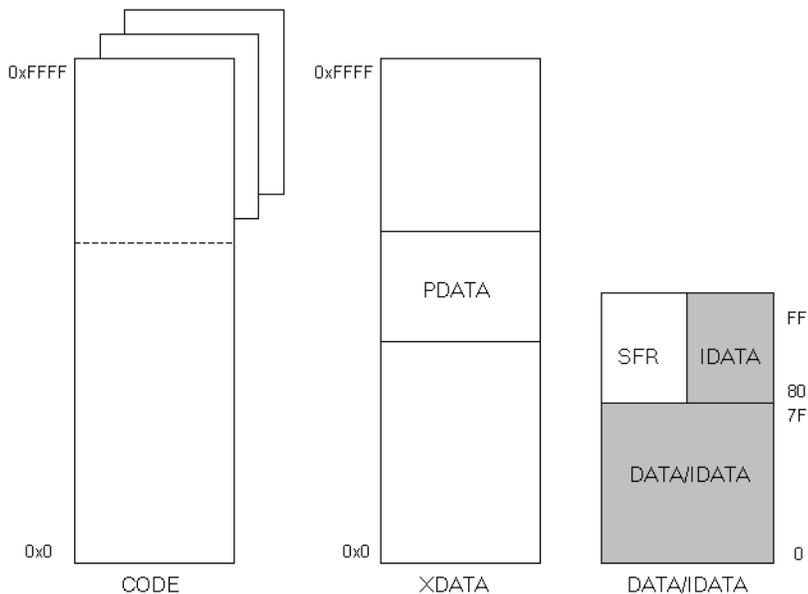
The direct form of the MOV instruction looks like this:

```
MOV 35, #AA
```

This will move the value AA into address 35, whereas the indirect MOV to accomplish the same thing in a two step process:

```
MOV R0, #35
```

```
MOV @R0, #AA.
```



DATA memory

The DATA memory area of internal RAM, residing between the addresses 00h and 7Fh, is an area of IDATA memory which can be accessed using either the indirect or the faster direct addressing mode of the 8051 MOV instruction. (For chips such as the original 8051 that have only 128 bytes of internal RAM, this is all the internal user RAM there is.)

Data objects which are defined to reside in the DATA memory are placed in this memory area, and are always accessed by the compiler using the direct addressing mode of the 8051 MOV instruction.

From the C source level, variables which require the fastest available access time can be assigned to reside in DATA space by using the extended language keyword `data`.

If the program attempts to place more objects in DATA memory than there is physical room, the linker will issue an 'out of range' error for the objects which would require addresses above 7Fh. Those objects would then require reassigning.

The segments listed in the DATA memory area are used by the compiler to contain variables defined by your C program to reside in the DATA memory area.

The segment `D_IDATA` is shown in the diagram in *Memory areas*, page 11, as the last segment in the DATA memory area. If there is still memory available in the DATA area after the contents of the five DATA segments have been linked, the linker will by default continue assigning the remaining DATA addresses to the contents of the IDATA segments beginning with the `C_ARGI` segment.

IDATA objects located in the DATA memory area will still be accessed by the compiler using the indirect addressing mode.

Once all addresses for the contents of the IDATA segments have been assigned, the next available address is used as the start of the stack.

As noted in the chart, DATA memory contains the four general purpose register banks and a bit-addressable area. These areas are discussed in the following paragraphs.

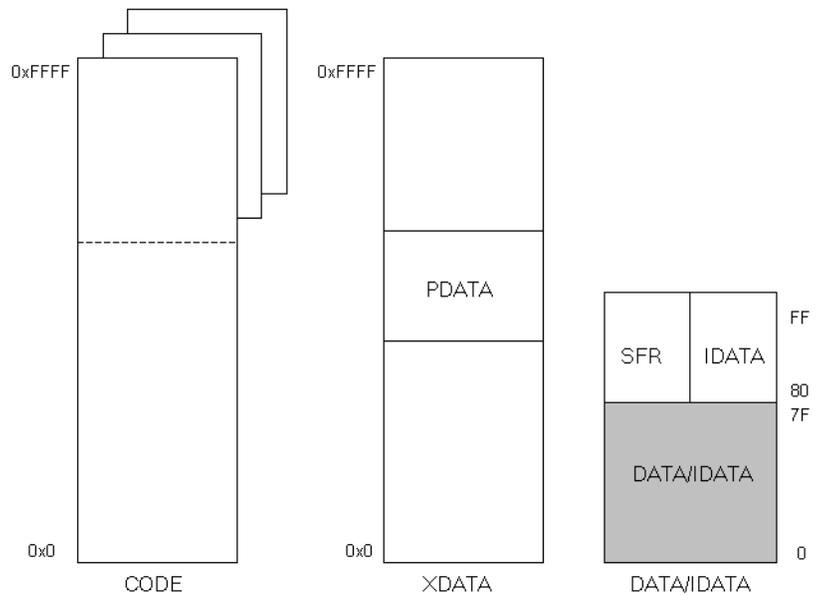
The four general purpose memory mapped register banks are located between address 00h and 1Fh. One of these register banks is used as the default register bank by ICC8051. This bank is specified in the linker command file via the symbol `_R`.

An interrupt service routine written in C may specify which register bank it will use. See the `using` keyword in *Extended keywords*, page 107, for details of specifying a register bank for an interrupt service routine.

The memory area occupied by unused register banks may be added to the pool of free memory used by the compiler. See the *IAR XLINK Linker™ and IAR XLIB Librarian™ Reference Guide*.

From the C source level, the extended language keyword `bit` is used to define a bit variable in the bit addressable memory. The compiler will then use the 8051's fast bit instructions to operate on bit variables.

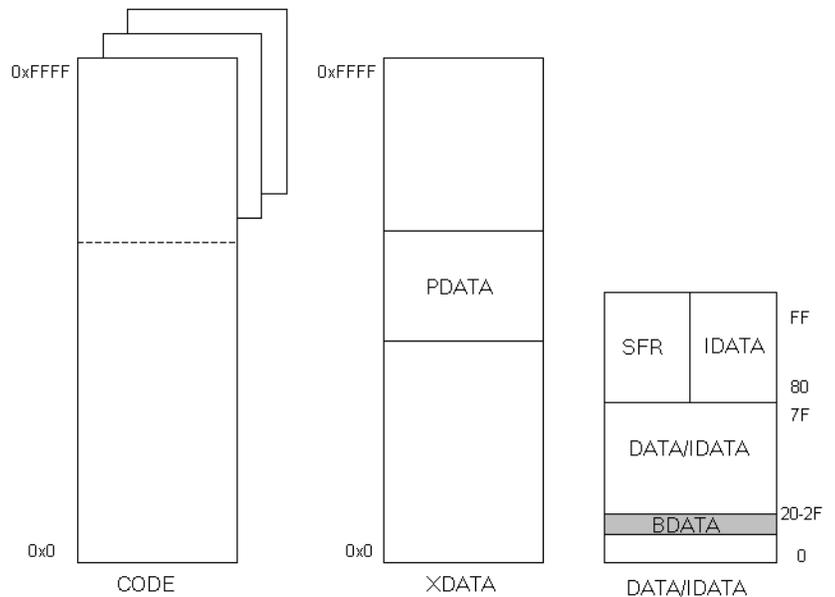
The C bit-field data type (defined using the ANSI C `struct` keyword) allocates a minimum of a full byte of storage. Bit-fields do not allocate memory in bit addressable area and do not generate code using the bit instructions. Thus when speed is important, use the bit data type rather than bit-fields. More information on the difference between bit-fields and bit variables is presented in *Data representation*, page 59.



BDATA memory

Data memory also contains 16 bytes of bit addressable memory, located between address 20h and 2Fh. 'Bit addressable' means that the 8051 instruction set includes instructions specifically designed to do fast set, clear, and other Boolean operations on individual bits without the need for byte access and then masking to isolate a particular bit.

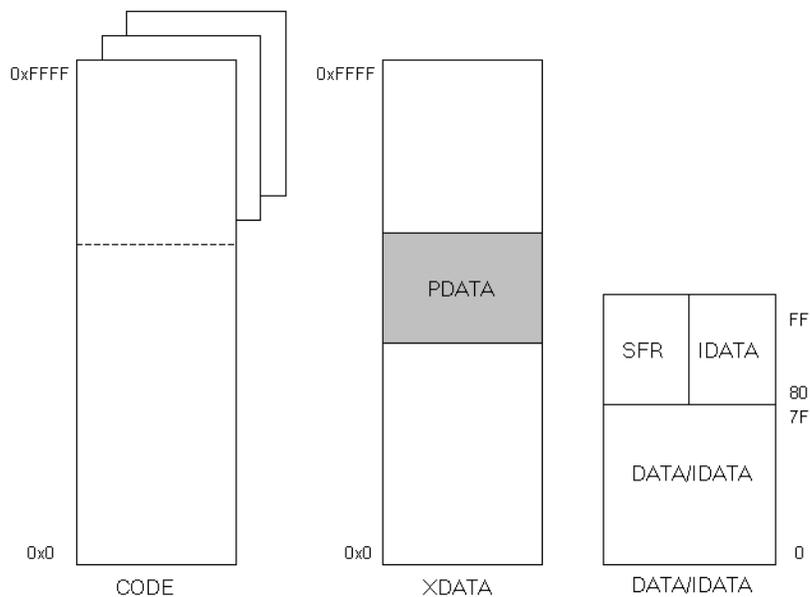
Note: BDATA cannot use the static overlay model when declared inside a function.



PDATA memory

The PDATA memory provides direct access to a defined page (256 bytes) of the external memory. This means that moving data to/from the PDATA segment can be done more efficiently. When PDATA is in use, CSTARTUP sets up port 2 (P2) to the base-page of the PDATA memory. See the diagram in *Memory areas*, page 11.

Note: PDATA cannot use the static overlay model when declared inside a function.



SFR space

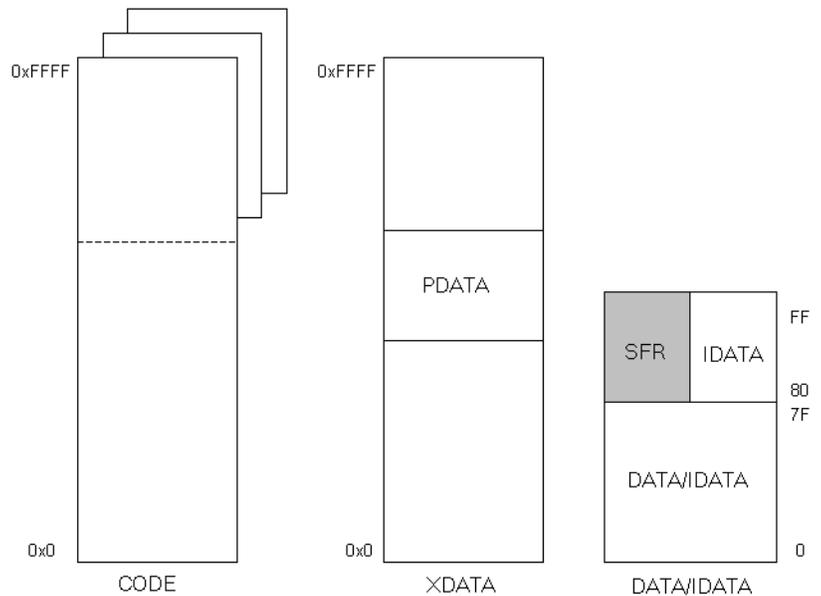
SFR (Special Function Register) space contains the DPTR, PC, and other memory mapped ports and registers which control the function of the microcontroller.

SFR's can be accessed from the C source level by their symbolic names once they have been declared using the `sfr` extended language keyword. The IAR Systems C development kit includes a set of C language header files that define the SFR's for a number of common 8051 family proliferation microcontrollers. These files are called `IO*.H`, where the `*` stands for the last digits of the microcontroller number. For example the file `IO51.H` includes the definitions for a basic 8051 microcontroller.

Some of the SFR's are bit-addressable. From both assembly and C, the 8051 IAR Assembler's `SFR.bit` notation may be used to access a particular bit of an SFR. For example,

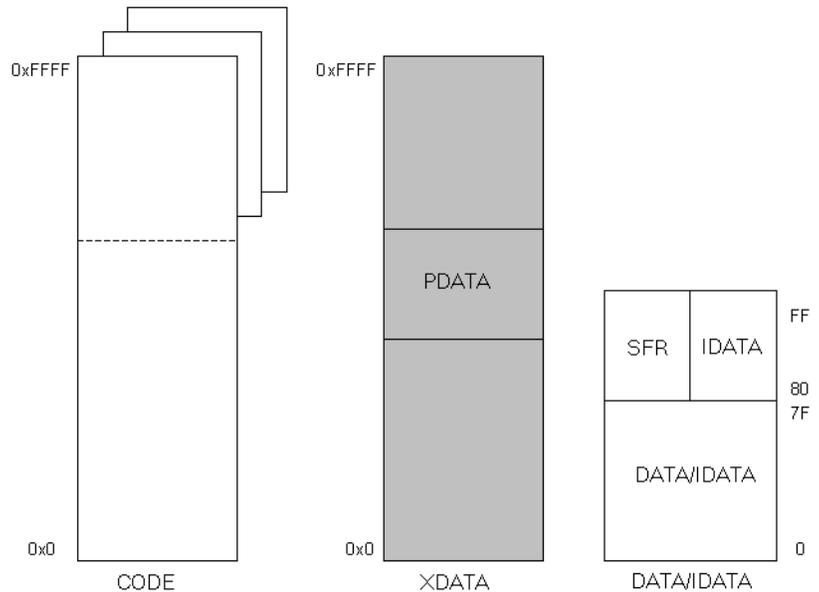
```
P1.7 = 0
```

assigns the value 0 to the most significant bit of port 1 (P1).



XDATA memory

External DATA memory is optional, off-chip RAM, addressable anywhere within the 0000h to 0FFFFh range. This may include a section which is non-volatile (battery backed) if desired, and memory mapped I/O devices.



Segments and memory

The compiler places the program instructions and data into the appropriate areas of physical memory by using segments.

At compile time, program instructions from each of the program modules are placed in the segment named `CODE`, uninitialized data objects defined to reside in the `DATA` memory area are placed in the `D_UDATA` segment, data objects declared using `const` are placed in the `CONST` segment, and so on.

At link time, the physical addresses for each of these segments are assigned by the linker command file.

The exact addresses where a segment is to be mapped can be specified by the `-z` (define segment) lines in the linker command file. Details of locating segments is available in the *IAR XLINK Linker™ and IAR XLIB Librarian™ Reference Guide*.

Note that there is a distinction between segment types and segment names. The segment type refers to which memory area the segment will be linked to and is used to provide this information to emulators and other debuggers via a symbolic linker output file such as AOMF8051.

For example, in the following `-Z` line, the segments `INTVEC` and `RCODE` are assigned to begin at `CODE` memory address `0h`:

```
-Z (CODE) INTVEC, RCODE . . . =0
```

`INTVEC` and `RCODE` are segment names, and `CODE` is the segment type. The ellipses (...) indicate the other `CODE` segments that are found in the complete `-Z (CODE)` line.

A functional description of the various segments is provided in *Segments*, page 67.

For general information about segments, including the creation of your own segments, see the *IAR XLINK Linker™ and IAR XLIB Librarian™ Reference Guide*.

NON-VOLATILE RAM

The compiler supports the declaration of variables that are to reside in non-volatile RAM through the `no_init` type modifier and the `#pragma memory` directive. The compiler places such variables in the separate segment `NO_INIT`, which you should assign to the address range of the non-volatile RAM of the hardware environment. The run-time system does not initialize these variables.

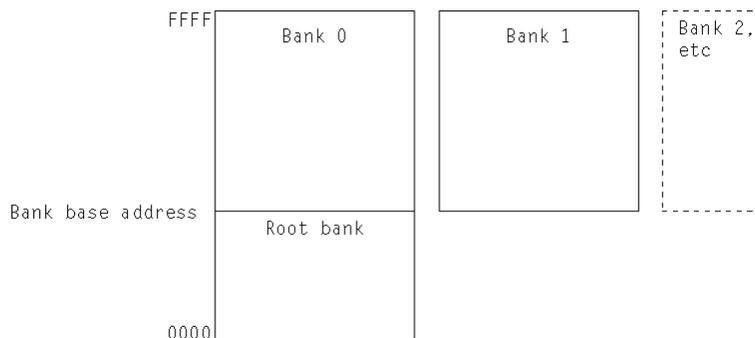
To assign the `NO_INIT` segment to the address of the non-volatile RAM, you need to modify the linker command file. For details of assigning a segment to a given address, see *XLINK Linker* in the *IAR XLINK Linker™ and IAR XLIB Librarian™ Reference Guide*.

Banking

Banking is a technique for extending the amount of memory that can be addressed by the processor beyond the limit set by the size of the natural addressing scheme of the processor.

BANKED MEMORY

The following memory map shows 8051's CODE memory area.



The upper section of the CODE address space is duplicated for the first 3 banks (numbered 0 to 2) of a banked system.

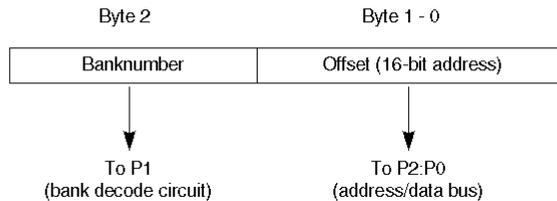
To access code residing in one of these banks, the compiler/linker generates banked addresses. A banked address has the form '16-bit-address-plus-bank-number'. The '16-bit-address' part of the banked address is presented onto the address bus, and the 'bank number' part of the banked address is presented via user port P1 (or another specified port). The hardware then decodes that bank number to select the appropriate ROM bank.

In the diagram above, the lower part of CODE space (labeled 'root bank') is not banked. This is because the compiler/linker only generates banked addresses for function calls, not for other code space objects such as `const` objects. Thus every object in ROM memory except for executable function code must reside in the root bank. Stated another way, all code space objects except for executable function code must reside at the same 16-bit address no matter which bank the currently executing function is in.

The root bank area thus contains all non-bankable code required by the program. Non-bankable code includes low-level run time library modules called in by the compiler, all `const`-type data objects, all variable initializers, interrupt service routine code, and the `CSTARTUP` code. The compiler always accesses all of these types of objects with non-banked addresses or function calls.

The size of the banked address range (and thus the bank size) is limited to the address area between the top of the root bank and address `FFFFh`. The root bank typically takes between 16 Kbytes and 48 Kbytes, allowing the bank size to be between 16 Kbytes and 48 Kbytes long.

The compiler keeps track of the bank number of a banked function by maintaining a three-byte pointer to it. The high byte of the pointer is the bank number, and the low bytes are the 16-bit-address. The format of a banked function pointer is shown in the following illustration:



Writing source code for banked mode

Writing code to be used in a banked mode system is not much different from writing code for the large memory model, but there are a few issues to be aware of. These primarily concern partitioning the code into source modules so that they can be placed into banks by the linker in the most efficient way and the distinction between banked and non-banked function calls.

Bank size and code segment size

Each C source module compiled contains a segment named `CODE` which contains the executable instructions defined in the source code of that module. The code contained in that segment is an indivisible unit. That is, the linker can not break up a module's code segment and place part of it in one bank and part of it in another. Thus, the size of each module's code segment must always be smaller than the bank size.

This means that the source code must be broken down into source files that will produce code segments small enough to fit into the banks, since each C source file will generate exactly one code segment.

For more specifics of assigning segments to banks, see the entry for the `-b` (banked segment definition) `XLINK` option in the *IAR XLINK Linker™ and IAR XLIB Librarian™ Reference Guide* and the use of the `-b` option in the examples below.

Banked versus non-banked function calls

The following discussion defines banked versus non-banked function calls, lists types of function declaration results in non-banked function calls, and gives an example of the definition and declaration of a function using the `non_banked` extended keyword.

In any memory model except for banked mode, the return address and new execution address are 16-bit (2 byte) values. A function call sequence using only 2-byte addresses is referred to as a ‘non-banked’ or ‘local’ function call. The local function call sequence generated by the compiler in bank mode is identical to the function call sequence used by the compiler in the large memory model.

A local function call can be used in bank mode when it is known by the compiler that both the calling and the called function reside in the same physical bank, so that the bank number does not need to be changed for that function call.

If the calling and the called function do not reside in the same bank, saving and restoring an execution address also requires specifying a bank number via a third byte. A function call sequence requiring saving all three bytes of the currently executing function, and then placing all three bytes of the new function execution address onto the ports is referred to as a ‘banked’ function call.

Differentiating between a ‘banked’ versus ‘non-banked’ function call is important because non-banked function calls are faster and take up less code space than banked function calls. This is because it takes a longer sequence of machine language instructions to make a banked call than a non-banked call. Also, the time required to make a function call will be inconsistent if banked and non-banked calls are arbitrarily interspersed. This can be undesirable in cases where the difference might be noticeable such as in timing applications.

In bank mode the compiler generates banked function calls whenever it does not know for certain that both the calling and the called functions reside in the same physical bank.

The compiler can be forced to generate a non-banked call to a function by declaring it non-banked. It is up to the programmer to place the non-banked function either in the same module as the callers or on the root-area. To place it in the root-area, the name of the code segment must be changed to preferably `RCODE`, by `-RRCODE`.

For example, `f1()` is to call `f2()`. They are each defined in separate source modules but will ultimately be linked to reside in the same physical bank. Then the definition of `f2()` would be of the form:

```
non_banked void f2(void) /* simple void function example */
{
    /* code here... */
}
```

The function prototype for `f2()` in the module where `f1()` will call `f2()` would be:

```
extern non_banked void f2(void);
```

Then the actual call to `f2()` from within `f1()` would be exactly as any other function, for example:

```
void f1(void)
{
    f2();
}
```

Calls to interrupt handlers in banked mode

Calls to interrupt handlers are always non-banked. Thus in the banked memory model interrupt handler code must be linked to reside in the root bank. To do this, we recommend placing the interrupt handlers in a separate file and then compiling this file with the `-R` compiler option, renaming the code segment to `RCODE`. The contents of the `RCODE` segment are always linked to reside in the root bank.

For example, the following compiler command line could be used to compile the file `isr.c`:

```
ICC8051 -mb -RRCODE isr
```

HOW TO USE THE BANKED MEMORY MODEL

This section describes how to implement bank mode, including the associated compiler, linker, and file configuration items.

Compiler options for banked mode

To compile your modules using the banked memory model, use the `-mb` (memory model banked) compiler option to compile each module.

If any of the code segments location needs to be specified individually (this issue is discussed further below), the code segment for each module may be renamed and referred to individually.

To rename the code segment of each source module, use the `-R` (Rename code segment) compiler option. For example if the first module is called `BANK1.C`, its code segment could be renamed by including the option `-Rbank1` on the command line.

The new code segment name used in the `-R` option is case sensitive.

If `-RCODE9` is entered on the compiler command line, the segments must be listed as `CoDe9` in the `-b` line of your linker command file (described below).

Linker options for banked mode

In linking a bank mode project, the central concern is placing the code segments into banks corresponding to the available physical banks in the hardware. The physical bank size and location, however, are dependent upon the size of the root bank which in turn is dependent on the source code.

As a result, it may be necessary to make a few trial passes through the linking process to determine the optimum hardware configuration.

For example, a “dummy” link can be used to determine the required size of the root bank. To do this, make a first pass through the link process as described below, taking a guess at the `-b` option parameters (described below). Then examine the segment dump table in the linker list file to determine the combined size of the root segments. The difference between the combined total size of the root bank segments and the available 64 Kbytes address range is the available bank size.

Thus if the root segments (details below) use 21 Kbytes, the available bank size is 64 Kbytes - 21 Kbytes = 43 Kbytes. Then the hardware can be set up and the `-b` arguments can be recalculated to reflect this organization.

Once the hardware configuration is fixed, the segments can be assigned to banks. First, link the root segments into the root bank, typically starting at 0 as with any other memory model. The root segments are all of the segments listed in the `-Z (CODE)` line of the example linker command file except for the segment named `CODE`. These segments are linked using the `-Z (Define segment) XLINK` option as follows:

```
-Z (CODE) INTVEC,RCODE,D_CDATA,I_CDATA,X_CDATA,CONST=0
```

To assign banked addresses to the code segments, use one or more instances of the `-b` (define banked segment) linker option.

The full syntax of the `-b` option is presented in the Linker chapter of the *IAR XLINK Linker™ and IAR XLIB Librarian™ Reference Guide*. Please refer to those pages as you proceed through the next example.

The `-b` option is a banked version of the `-Z` linker option, and involves the same type of segment type and segment name list. However, in the `-b` option, the ‘=ADDR’ (address range) part present in the `-Z` option line is replaced by the following three parameters: *bank start address*, *bank length*, and *bank increment*.

The *bank start address* consists of a two-byte value containing the bank number to start assigning addresses in, followed by a second two byte value containing the 16-bit address to start the banked code at. The *bank length* is the size of the bank.

The *bank increment* consists of a two-byte value containing the value by which the bank number will be incremented after the previous bank is filled. This value is followed by another 16-bit value containing an offset from the local address which can be used to accommodate asymmetrical bank arrangements (otherwise this value must be zero).

For example, if you have 12 modules with approximate code sizes of 500 (hex) bytes each, and you have determined that your root bank takes about 5500 (hex) bytes and for some reason the upper 32 Kbytes of code address space is not available to the C system. If you then decide to try a bank size of 2000 (hex) bytes starting at 16-bit-address 6000 (hex), you would have about 4 modules worth of code segments per bank ($2000/500 = 4$), for a total of three banks ($12/4 = 3$).

This would be implemented as the following `-b` line:

```
-b (CODE) CODE=00006000,2000,00010000
```

This `-b` line states that the code segments should be placed into as many 2000-(hex)-byte-long banks (second parameter is 2000) as are required. The first bank should be assigned bank number 0000 (first part of first parameter is 0000) and 16-bit-address 6000 (second part of first parameter is 6000 hex). Subsequent banks should be numbered with a bank number increment of 0001 so that banks are numbered 0, 1, 2, 3... (first part of third parameter is 0001). In addition, there should be an increment of 0000 (i.e. none) on the 16-bit-address so that all banks start at 16-bit-address 6000 (second half of third parameter is 0000).

The size of an individual module's code segment must not exceed the bank size, or the linker will flag you with an Error [21] 'Segment does not fit bank'.

As many `-b` lines as required can be created, even one per segment, each with its own set of parameters if required.

This is very useful if particular code segments need to be assigned to particular banks, so that non-banked function calls can be used as discussed in the section above. This also allows assignment of code to asymmetrical physical banks.

If more code segments need to be named than conveniently fit on a single `-b` line, a second `-b (CODE)` without the set of three ending parameters can be used.

For example, to add three more code segments to the first line shown below, a second line is added so that the whole list is as follows:

```
-b (CODE) code0,code1,code2,code3=00006000,2000,00010000
```

```
-b (CODE) code4,code5,code6
```

Alternatively, the linker command file will accept a backslash (\) followed by a carriage return as a line continuation character, as in:

```
-b (CODE) code0, code1, code2, code3, \
code4, code5, code6=00006000, 2000, 00010000
```

The banked hardware may involve several EPROM chips, in which case several passes through the link process might be needed to generate one EPROM-full of executable code with each pass. Doing this involves using the `-E` (Empty load input file) linker option.

Modifying the default bank port assignment

The default port used to present the bank number to the hardware is P1, but any port may be specified. To specify a port, the supplied assembly language source file `L18.S03` must be modified. The file `L18.S03` contains the bank-switching routines used by the compiler. The file is commented where modifications should be made.

If only a small number of banks are used, only certain bits of the port need to be used for bank-switching and the rest of the port might be used for something else. This particular refinement is not explicitly supported by IAR Systems, but may be implemented by further modifying the `L18.S03` file.

After any modification to `L18.S03`, reassemble it and replace the object module in the `CL8051B.R03` library using the `XLIB` librarian replace-module command; this is described in the *IAR XLINK Linker™ and IAR XLIB Librarian™ Reference Guide*.

Linker command file

The linker command file controls many of the configurable features of the system. In particular, it specifies the different memory segments.

Since the processor option states the addressing capabilities of the target processor, it is natural to provide individual linker command files for each option. However note that the supplied files are examples, which should be modified to fit the actual hardware used.

To create an linker command file for a particular project, you should first copy the appropriate template, as shown in the following table:

Linker command file	Description
<code>lnk8051.xcl</code>	Non-banked memory
<code>lnk8051a.xcl</code>	80751
<code>lnk8051b.xcl</code>	Bank-switched memory

Table 5: Command file names

You should then modify these files—as described within the files—to specify the details of the target system’s memory map and the stack size.



In the IAR Embedded Workbench, a linker command file is selected automatically depending on which memory model you specify. You can, however, override the default setting and specify your choice of linker command file on the **Include** page in the **XLINK** category when setting project options.

STACK SIZE

The compiler uses a stack for a variety of user program operations, and the required stack size depends heavily on the details of these operations. If the given stack size is too small, the stack will normally be allowed to overwrite variable storage resulting in likely program failure. If the given stack size is too large, RAM will be wasted.

There can be two software stacks in addition to the internal hardware stack.

- A stack maintained for the variables and parameters of re-entrant functions.
- A stack for variables and parameters of recursive functions.

Local variables for non-reentrant functions are not placed on the stack, but use an area of RAM dedicated to local variables of functions.

Note: If you use `reentrant_idata` functions, they will use the internal stack to store their parameters and return values.

The memory for several variables can overlap if the functions are not active at the same time. The compiler and linker will control the allocation and access to the local variable memory. This is called static overlay.

ESTIMATING THE REQUIRED STACK SIZE

The stack is used for the following:

- Storing temporary results in library routines.
- Saving the return address of function calls (not needed if the compiler stack expansion `-u` option is used).
- Saving the processor state during interrupts.

The total required stack size is the worst case total of the required sizes for each of the above.

HEAP SIZE

If the library functions `malloc` or `calloc` are used in the program, the compiler creates a heap of memory in external RAM from which their allocations are made. The default heap size is 2000 bytes.

The procedure for changing the heap size is described in the `heap.c` file which you can find in the `\ew23\8051\src\lib\` directory.

Input and output

PUTCHAR AND GETCHAR

The functions `putchar` and `getchar` are the fundamental functions through which C performs all character-based I/O. For any character-based I/O to be available, you must provide definitions for these two functions using whatever facilities the hardware environment provides.

The starting-point for creating new I/O routines is the files `putchar.c` and `getchar.c`.

Note: Be sure to save your original `c18051*.r03` file before you overwrite the `putchar` module.

Customizing putchar

The procedure for creating a customized version of `putchar` is as follows:

- 1 Make the required additions to the source `putchar.c`, and save it under the same name (or create your own routine using `putchar.c` as a model). The code below uses memory-mapped I/O to write to an LCD display.

```
#include <stdio.h>
int putchar(int outchar)
{
    unsigned char *LCD_10;
    LCD_IO= (unsigned char *) 0x8000;
    *LCD_IO=outchar;
    return(outchar);
}
```

- 2 Compile the modified `putchar` using the library module `-b` option.

For example, if your program uses the small memory model, compile `putchar.c` for the small memory model.

```
icc8051 putchar -ms -b -z9
```

This will create an optimized replacement object module file named `putchar.r03`.

- 3 Add the new `putchar` module to the appropriate run-time library module, replacing the original.

For example, to add the new `putchar` module to the standard library, use the command:

```
xlib
def-cpu sh
rep-mod putchar c18051
exit
```

The library module `c18051` will now have the modified `putchar` instead of the original one.

XLINK allows you to test the modified module before installing it in the library by using the `-A` option; see the *IAR XLINK Linker™ and IAR XLIB Librarian™ Reference Guide* for information about the XLINK options.

Place the following lines into your linker command file:

```
-A putchar
c18051
```

This causes your version of `putchar.r03` to load instead of the one in the `c18051` library.

Notice that `putchar` serves as the low-level part of the `printf` function.

Customizing `getchar`

The low-level I/O function `getchar` is supplied in the C file `getchar.c`.

The same procedure can be used as for customizing `putchar`.

PRINTF AND SPRINTF

The `printf` and `sprintf` functions use a common formatter called `_formatted_write`. The ANSI standard version of `_formatted_write` is very large, and provides facilities not required in many applications. To reduce the memory consumption the following two alternative smaller versions are also provided in the standard C library:

`_medium_write`

As for `_formatted_write`, except that floating-point numbers are not supported. Any attempt to use a `%f`, `%g`, `%G`, `%e`, and `%E` specifier will produce the error:

```
FLOATS? wrong formatter installed!
```

`_medium_write` is considerably smaller than `_formatted_write`.

`_small_write`

As for `_medium_write`, except that it supports only the `%%`, `%d`, `%o`, `%c`, `%s`, and `%x` specifiers for integer objects, and does not support field width and precision arguments. The size of `_small_write` is 10–15% of the size of `_formatted_write`.

The default version is `_small_write`.

Selecting the write formatter version

The selection of a write formatter is made in the XLINK control file. The default selection, `_small_write`, is made by the line:

```
-e_small_write=_formatted_write
```

To select the full ANSI version, remove this line.

To select `_medium_write`, replace this line with:

```
-e_medium_write=_formatted_write
```

Reduced printf

For many applications `sprintf` is not required, and even `printf` with `_small_write` provides more facilities than are justified by the memory consumed. Alternatively, a custom output routine may be required to support particular formatting needs and/or non-standard output devices.

For such applications, a highly reduced version of the entire `printf` function (without `sprintf`) is supplied in source form in the file `intwri.c`. This file can be modified to your requirements and the compiled module inserted into the library in place of the original using the procedure described for `putchar` above.

SCANF AND SSCANF

In a similar way to the `printf` and `sprintf` functions, `scanf` and `sscanf` use a common formatter called `_formatted_read`. The ANSI standard version of `_formatted_read` is very large, and provides facilities that are not required in many applications. To reduce the memory consumption, an alternative smaller version is also provided in the standard C library.

`_medium_read`

As for `_formatted_read`, except that no floating-point numbers are supported. `_medium_read` is considerably smaller than `_formatted_read`.

The default version is `_medium_read`.

Selecting read formatter version

The selection of a read formatter is made in the XLINK control file. The default selection, `_medium_read`, is made by the line:

```
-e _medium_read=_formatted_read
```

To select the full ANSI version, remove this line.

Initialization

On processor reset, execution passes to a run-time system routine called `CSTARTUP`, which normally performs the following:

- Initializes the stack pointers for data and program.
- Initializes C file-level and static variables.
- Calls the user program function `main`.

`CSTARTUP` is also responsible for receiving and retaining control if the user program exits, whether through `exit` or `abort`.

VARIABLE AND I/O INITIALIZATION

In some applications you may want to initialize I/O registers, or omit the default initialization of data segments performed by `CSTARTUP`.

You can do this by providing a customized version of the routine `__low_level_init`, which is called from `CSTARTUP` before the data segments are initialized.

The value returned by `__low_level_init` determines whether data segments are initialized. The run-time library includes a dummy version of `__low_level_init` that simply returns 1, which causes `CSTARTUP` to initialize data segments.

The source of `__low_level_init` is provided in the file `lowinit.c`, by default located in the `src\lib` directory. To perform your own I/O initializations, create a version of this routine containing the necessary code to do the initializations.

MODIFYING CSTARTUP

If you want to modify `CSTARTUP` itself you will need to reassemble `CSTARTUP` with options which match your selected compiler options.

The overall procedure for assembling an appropriate copy of `CSTARTUP` is as follows:

- I Make any required modifications to the assembler include file `defmodel.inc` and its include files which describe the memory model used.

- 2 Assemble the modified CSTARTUP using the appropriate memory model. For example, if the user program uses the small memory model, reassemble CSTARTUP for the small memory model. This will create a replacement object module file named `cstartup.r03`.
- 3 Add the new CSTARTUP module to the appropriate run-time library module, replacing the original.

For example, to add the new CSTARTUP module to the simplest small memory model library, use the XLIB commands:

```
xlib
def-cpu 8051
rep-mod cstartup c18051s
exit
```

The library module `c18051s` will now have the modified CSTARTUP instead of the original.

Note: You can test the modified `cstartup` before installing it in the library by using the XLINK `-C` option; see the *IAR XLINK Linker™ and IAR XLIB Librarian™ Reference Guide* for details.

Example

For example, the startup code could change the I/O register. It might be desirable to include initialization routines in CSTARTUP rather than in the main code if your implementations always use a standard environment.

When a C program is compiled, it is executed after an initialization routine, but before an exit routine. Normally the exit routine is never called as the code is used in a dedicated controller which loops continuously. If you want to include special startup or shutdown code, edit the `cstartup` file, using the `cstartup.s03` assembly source module as a starting point for modification. The code is commented to indicate what action is taking place during the execution:

- Stack pointer initialized.
- Data memory initialized.
- C main called.
- Jump to exit routine.

Add code at the appropriate point to initialize your hardware, and assemble the modified source file. After successful assembly, use XLIB to place the new module in the library file for the processor and memory type (for example, `c18051s.r03`).

When using the IAR Embedded Workbench, you can simply include the `cstartup.s03` to your project files and in the **Library** page under the XLINK category. To make XLINK use your new CSTARTUP instead of the one in the base library, set the Load library module libraries in the **Library** page under the XLINK category. To enable this option, check the **Override Default Library** box. This makes the CSTARTUP module in the base library (`c18051s.r03`) a library module instead of a program module.

The following section describes several important sections of the startup file.

CSTARTUP.S03

The `cstartup.s03` module contains the entire code executed before the C `main` function is called. The code can be tailored to suit special hardware needs and is designed to run on any processor based on the 8051 architecture.

```

NAME      CSTARTUP
PUBLIC   init_C

$defmodel.inc                ; Defines memory model

EXTERN   ?C_EXIT              ; Where to go when program ends
EXTERN   _R                   ; Register bank (0, 8, 16 or 24)
EXTERN   main                  ; First C function usually
EXTERN   _low_level_init      ; Setup low level things
IF       banked_mode
EXTERN?X_CALL_L18
ENDIF

```

The C stack segment should be mapped into internal data RAM. The C stack is used for LCALL instructions and temporary storage of code generator help routines such as math routines. The stack will be located after all other internal RAM variables if the standard linking procedure is followed. Note that C interrupt routines can double the stack size demands.

```

RSEG     CSTACK
stack_begin:
DS       30                      ; Increase if needed

```

The stack size can be set here, but it is simpler to use the following declaration in the linker command file:

```
-Z(DATA) CSTACK+stack_size
```

(Using the linker command file avoids having to re-assemble CSTARTUP).

```
COMMON  INTVEC                  ; Should be at location zero
```

INTVEC is used by the compiler to set up an interrupt vector table.

C interrupt routines with defined [vectors] will reserve space in this area. So will handlers written in assembler if they follow the recommended format.

```
startup:
    IF lcall_mode
        LJMP init_C
    ELSE
        AJMP init_C
    ENDIF

    RSEGRCODE                ; Should be loaded after INTVEC
init_C:
    MOVSP,#stack_begin - 1    ; From low to high addresses
    MOVDPTR,#SFB (P_IDATA)    ; Initialize high byte of PDATA
    MOVDP2,DPH                ;
```

If there is no demand that global or static C variables should have a defined value at startup (required by ANSI), the following section can be removed to conserve code memory size. Note that this part calls functions at the end of this file, which also can be removed if initialized values are not needed.

Systems controlled by a watch-dog may require additional code insertions as the initialization can take several milliseconds (if there are many variables) to complete. These parts are marked with `*** WDG ***`.

Zero out sections containing variables without explicit initializers such as:

```
int i;
xdata double d[10];
```

Copy initializers into the proper memory segments for declarations such as:

```
int i = 7;
idata char *cp = "STRING";
```

If hardware must be initiated from assembly code or if interrupts should be on when reaching main, this is the place to insert such code.

The initialization of the timer as a baud clock for the serial I/O port could be done at this location. For example, add the assembly code below to start the timer:

```
MOV SCON, #52H    ; set timer mode
MOV TMOD, #20H    ; auto reload
MOV TCON, #69H    ; start timer 1
MOV TH1, #0F3H    ; reload value gives 1200 baud at 12Mhz
```

If you have any other code you want executed before main starts, insert that here as well.

```
IF                banked_mode
```

```

MOV      A,#$BYTE3 main
MOV      DPTR,#main
LCALL   ?X_CALL_L18          ; main()

ELSE

IF      lcall_mode
LCALL   main                ; main()
ELSE
ACALL   main                ; main()
ENDIF
ENDIF

```

Now when we are ready with our C program (usually 8051 C programs are continuous) we must perform a system-dependent action. In this simple case we just stop.

Do not change the next line of `cstartup` if you want to run your software with the aid of the IAR C-SPY Debugger. It may, however, be removed if your program is continuous (no exit).

If it is removed the `EXTERN ?C_EXIT` line should also be removed to avoid linking of the `exit` module.

```

IF      lcall_mode
LJMP   ?C_EXIT
ELSE
AJMP   ?C_EXIT
ENDIF

```

When C-SPY is used this code will automatically be replaced by a debug version of `exit()`.

```

MODULE   exit

PUBLIC   exit
$DEFFN   exit(0,0,0,0,0,0,0,0)
PUBLIC   ?C_EXIT

RSEG     RCODE

?C_EXIT:
exit:

```

The next line could be replaced by user defined code.

```

SJMP   $          ;Forever

ENDMOD

```

If you want your system to take some action when the C program exits, place your routines in place of the exit code.

Optimization

The 8051 IAR C Compiler allows you to generate code that is optimized either for size or for speed, at a selectable optimization level. The chapter *Compiler options* contains reference information about the `-s [0-9]` and `-z [0-9]` command line options. Refer to the *8051 IAR Embedded Workbench™ User Guide* for information about the compiler options available in the IAR Embedded Workbench.

The purpose of optimization is either to reduce the code size or to improve the execution speed. In the 8051 IAR C Compiler, however, most speed optimization alternatives also reduce the code size.

A high level of optimization will result in increased compile time and may also make debugging more difficult, since it will be less clear how the generated code relates to the source code. We therefore recommend that you use a low optimization level during the development and test phases of your project, and a high optimization level for the release version.

Customizing the run-time library

Included with this product are ready-made object libraries which can be modified or customized. It is highly recommended that you make a copy of the original before modifying these object libraries.

MULTI-MODULE LINKING

You may want to split your program into several C or assembly modules to simplify maintenance. The entry to your modules should be declared in the C main block:

```
extern int mymod1 (int myint1, int myint2);
int total;
void main(void)
{
    total=mymod1 (3, 4);
}
```

A separate file will contain the code for mymod1, for example:

```
int mymod1(int para1, int para2)
{
    return (para1+para2)
}
```

Compile the two modules in the regular way with the same memory model. Place the objects resulting from the compilations into a library by using XLIB, as follows:

```
xlib
def-cpu 8051
fetch-mod main mylib
fetch-mod mymod1 mylib
list-mod mylib mylib.lst
exit
```

Modify a copy of the linker command file to include access to the library and add your output file specifications. Name the new linker command file `mymod.xcl`.

```
-! link the library module which contains our object modules
-!
mylib
-! specify the Intel AOMF8051 symbolic output format -!
-FAOMF8051
-! specify the output file name -!
-o mulmod.a03
```

The linker command file can now use the modules in the library and produce an executable program. Use the linker with the new linker command file file:

```
xlink -f mymod
```

Target-specific support

The 8051 IAR C Compiler supports several Siemens and Dallas microcontrollers as described in the following sections.

80751 SUPPORT

The 8051 IAR C Compiler includes support for the Siemens 80751/752 chip.



Using the 80751 with the IAR Embedded Workbench

Create a new project by choosing **New Project...** from the **Project** menu, and select **80751** in the **Project Options** dialog box.



Using the 80751 with the command line

Specify the `-v1` compiler option. The `-v1` switch should only be used with either the tiny memory model `-mt` or the small memory model `-ms`.

The libraries supplied for the 80751 are `c18051ta` and `c18051sa`. The linker command file is `lnk8051a.xcl`, and the include files are `io751.h` and `io752.h`.

80751 limitations

When using the 80751 the following limitations are imposed:

- No recursive functions are allowed.
- No calls to `longjmp` or `setjmp` are allowed.
- No reentrant functions are allowed.
- No floats are allowed.
- No `xdata` memory or `xdata` pointer attributes are allowed.
- No memory model other than `tiny` or `small` is allowed for the 80751.
- No function call can be made to an address outside the 2 Kbytes allowed area.

For details of the error messages produced in these cases see *80751-specific error messages*, page 189.

Library routines

The following library routines are not included for the 80751 chip:

Floating-point routines

`acos`, `asin`, `atan`, `atan2`, `atof`, `ceil`, `cos`, `exp`, `exp10`, `fabs`, `floor`, `fmod`, `frexp`, `ldexp`, `log`, `log10`, `modf`, `pow`, `sin`, `sqrt`, `strtod`, `tan`, `cosh`, `sinh`, `tanh` (ie no float routines are supplied).

printf

`frmwri`, `mediumwr`, `smallwri` (therefore no `printf` routines are supplied).

scanf

`frmrd`, `mediumrd`, `scanf`, `sscanf` (therefore no `scanf` routines are supplied).

malloc

`calloc`, `free`, `heap`, `malloc`, `realloc` (therefore no heap routines are supplied).

80517 SUPPORT

The 8051 IAR C Compiler development kit includes library files `c1517.r03` and `c1517i.r03` to support the Siemens 80517 and 80537 microcontrollers. The library files are provided to support the extended math unit (MDU) on these microcontrollers.

The `c1517i.r03` calls the MDU with interrupt protection, while the `c1517.r03` calls the external ALU without interrupt protection. These files must be inserted into the compile library for your memory model.

To insert the `c1517` file into the library for the tiny memory model, use XLIB as follows:

```
xlib
def-cpu 8051
repl-mod c1517 c18051t
quit
```

Using the 80517 library files will improve long and integer operations that use the intrinsic routines (for example, multiplication and division). Improvements of 10 to 30% are possible when using the interrupt protected library or 20 to 40% when using the non-protected library. A small improvement will also occur for floating-point math operations.

The development kit also includes the `c1517str.r03` library file to support the multiple data pointers (DPTR) in the 8XC517 microcontroller. This means that some of the library functions such as `strcmp` and `memcpy` can take advantage of the multiple DPTRs and produce more efficient code.

The implementation is that the library uses a pair of DPTRs—that is, it flips the low bit in `sfr DPSEL (0x92)` to get hold of the other DPTR. At reset of the microcontroller the lower 3 bits of `DPSEL` is set to 0, and this will force the libraries to use DPTR 0 and 1. You can change this by setting the `sfr DPSEL` to another DPTR pair.

Using multiple DPTRs with interrupt functions

Interrupt functions may destroy the contents of the multiple DPTRs because the multiple DPTRs are not automatically saved by the interrupt functions. When using multiple DPTRs together with interrupt functions you have to make sure that the contents of the DPTRs are not destroyed. This may be done by simply saving the content of the DPTRs if the interrupt function may use it. In the case of 8XC517, you can shift to an other pair of DPTRs to avoid saving the contents of the DPTRs.

80320 SUPPORT

The 8051 IAR C Compiler development kit includes the library file `CL320STR.r03` to support the dual DPTRs in the Dallas Semiconductor 80C320 microcontroller. The library takes advantage of the multiple DPTRs for library functions such as `strcmp` and `memcpy` and will produce more efficient code. The `CL320STR.r03` file contains library functions for string operations that use 8X0517 multiple DPTRs.

Using multiple DPTRs with interrupt functions

Interrupt functions may destroy the contents of the multiple DPTRs because the multiple DPTRs are not automatically saved by the interrupt functions. When using multiple DPTRs together with interrupt functions you have to make sure that the contents of the DPTRs are not destroyed. This may be done by simply saving the content of the DPTRs if the interrupt function may use it.

Assembly language interface

This chapter describes the interface between a C main program and the assembly language routines.

The 8051 IAR C Compiler allows assembly language modules to be combined with compiled C modules. This is particularly useful for small, time-critical routines that need to be written in assembly language and then called from a C main program.

Calling convention

The C compiler uses two areas of memory for administrating function calls:

- The parameter blocks in memory.
- The hardware stack (addressed by the SP register), to hold return addresses.

The interface between the compiler and the assembler selects the parameters that can be placed in the registers. This table shows the type and location of each parameter:

Parameter 1	Parameter 2	Parameter 3	Parameter 4
Bit			
C (treat as a byte)			
Byte R4	Byte R5	Byte R6	Byte R7
Byte R4	Byte R5	Word R6, R/	
Byte R4	Word R6, R/		
Byte R4	General pointer R6, R/		
Word R4, R5	Byte R6	Byte R7	
Word R4, R5	Word R6, R7		
General pointer R5, R6, R7			
Long R4, R5, R6, R7			
Float R4, R5, R6, R7			

Table 6: Parameters, types, and locations

Even if the parameters are passed in registers, the compiler will always allocate space in the different memories.

If a routine in assembler is created directly, it will need to use the calling conventions. It is much simpler to create a C skeleton with the correct number and type of parameters and then modify the assembler output.

The return value from a function will be placed into the registers as shown below:

Type	Register
bit	C
char	R4
word	R4 and R5
pointer	R5, R6, and R7
long or float	R4, R5, R6, and R7

Table 7: Return registers

Note that in a type larger than a byte the most significant byte is in the highest register number, for example a word in R4 or R5 has the MSB in R5 and LSB in R4.

REGISTER USAGE

The following rules apply to how the registers should be handled at function calls and in called functions:

- 1 The called function *must* always preserve all used registers except:
 - Registers used for return values.
 - Registers used for passing parameters to that function.
- 2 The calling function must preserve all registers used as:
 - Return values.
 - Parameters.

Parameters and local variables

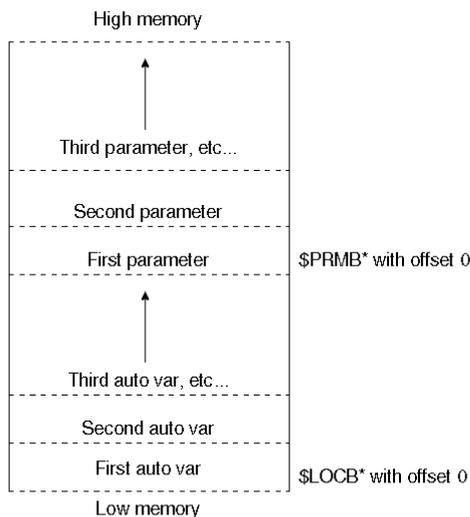
Local variables in other compilers are often placed onto the hardware stack. Because of limitations of the 8051 stack, the 8051 IAR C Compiler places local variables into a dedicated memory area.

The local variable area is defined for each function, but the areas for several functions can overlap if the local variables will not be active at the same time.

When possible, parameters are passed in registers, but a parameter block is always created for the function.

There can be up to eight memory areas associated with each function. Four are used for each of the local variable types, DATA, IDATA, XDATA, and BIT and four for each of the parameter types.

The memory organization for DATA parameters and local variables in memory is shown in the figure below.



The lines below from a linker map file indicate the size and location of a local block.

```
do_foreground_process  00B8           Not referred to
    data = 000A ( 0002 , 0000 )
```

For `do_foreground_process` located at `0x0B8` in CODE memory, there is a local data block at `0x000A` in DATA memory with two bytes for local variables and no bytes for parameters.

See *Assembly language interface*, page 43, for more information on local variable storage in memory.

LIMITATIONS

Because the 8051 IAR C Compiler uses by default an overlay technique instead of a stack, there are some restrictions. For example:

- The same function can not be activated in two function trees at the same time.

- Recursive functions do not work.

If the program contains functions that apply to one or more of the above groups, these functions should be small or large reentrant functions. Refer to the next section. See also chapter *Assembly language interface*, page 43.

REENTRANT PARAMETERS

A reentrant function is one that may, directly or indirectly, call itself. It may also be called by an interrupt routine. This could occur while the function itself is currently executing.

Because parameters and local variables are normally stored in a fixed area of memory, a software stack must be used with reentrant functions. To illustrate, a non-reentrant function which calls itself will fail at run-time because the second pass through the function code will overwrite the local variable area of the first pass.

The reentrant code `-E` option forces all functions and function pointers to be reentrant; alternatively, the `reentrant` keyword allows you to define selected functions as reentrant.

Example

The following program `recursive` is a simple example of reentrant functions:

```
void recursive (int value)
{
    value-=1;
    if (value>10) recursive(value);
}
void main(void)
{
    recursive(50);
}
```



Compile this with the following **ICC8051** options in the **Options** dialog box:

Page	Option
List	List file Insert mnemonics in list file
Code Generation	Reentrant code

Table 8: Compiler options



Use the command:

```
icc8051 recur -r -q -L -E
```



Then examine the list file. Some of the lines from the file are listed below:

```
\ 0000          NAME    ex02(16)
\ 0000          RSEG    CODE(0)
\ 0000          PUBLIC  main
\ 0000          PUBLIC  recursive
\ 0000          EXTERN  ?LD_AR5_ST_A_L17
\ 0000          EXTERN  ?ST_AR5_ST_DPTR_L17
\ 0000          EXTERN  ?STACK_ENTER_4_L17
\ 0000          EXTERN  ?STACK_RET_4_L17
\ 0000          EXTERN  ?CL8051T_5_20_L17
\ 0000          RSEG    CODE
```

Several special routines have been included in the listing. These maintain a software stack to hold parameters and local variables for the reentrant functions.

```
\ 0000          recursive:
1          void recursive(int value)
2          {
\ 0000 120000          LCALL  ?STACK_ENTER_4_L17
```

Set up the stack-block for the function. It holds parameters, auto variables, and the return address.

```
3          value-=1;
\ 0003 EC          MOV    A,R4
\ 0004 24FF        ADD    A,#255
\ 0006 FC          MOV    R4,A
\ 0007 ED          MOV    A,R5
\ 0008 34FF        ADDC   A,#255
4          if (value>10) recursive(value);
\ 000A FD          MOV    R5,A
\ 000B EC          MOV    A,R4
\ 000C 900002      MOV    DPTR,#2
\ 000F 120000      LCALL  ?ST_AR5_ST_DPTR_L17
```

Calculate the new value.

```
\ 0015 D3          SETB   C
\ 0016 FC          MOV    R4,A
\ 0017 EC          MOV    A,R4
\ 0018 940A        SUBB   A,#10
\ 001A ED          MOV    A,R5
\ 001B 6480        XRL   A,#128
\ 001D 9480        SUBB   A,#128
\ 001F 4009        JC     ?0001
```

Should we recurse one more time?

```

\ 001C          ?0000:          ; [IF_TRUE]      2:1
\ 001C 7402          MOV      A,#2
\ 001E 120000       LCALL   ?LD_AR5_ST_A_L17
\ 0021 FC          MOV      R4,A
\ 0022 120000       LCALL   recursive

```

Load argument to register and call.

```

\ 0025          ?0001:          ; [IF_FALSE]    3:1
5          }
\ 0025 020000       LJMP    ?STACK_RET_4_L17
\ 0028          main:
6          void main(void)
7          {

```

Set up the stack block for main.

```

8          recursive(50);
\ 0028 E4          CLR      A
\ 0029 FD          MOV      R5,A
\ 002A 7C32        MOV      R4,#50
\ 002C 120000       LCALL   recursive
9          }
\ 002F 22          RET
\ 0030          END

```

Deallocate the stack block for the function and return.

See *Segments*, page 67, for more information on memory segments.

Creating skeleton code

The recommended method of creating an assembly language routine with the correct interface is to start with an assembly language source created by the compiler. To this skeleton you can easily add the functional body of the routine.

The skeleton code needs only to declare the variables required and perform simple accesses to them, for example:

```

int k;
int foo(int i, int j)
{
    char c;
    i++;    /* Access to i */
    j++;    /* Access to j */
    c++;    /* Access to c */
    k++;    /* Access to k */
}

```

```

}
void f(void)
{
    foo(4,5); /* Call to foo */
}

```



Compiling the program using the IAR Embedded Workbench

The program should be compiled with the following **ICC8051** options selected in the **Options** dialog box:

Category	Option
List	Assembly output file
List	List file
	Insert mnemonics

Table 9: Compiling skeleton code in IAR Embedded Workbench



Compiling the program using the command line

This program should be compiled as follows:

```
icc8051 shell -A -q -L
```

The `-A` option creates an assembly language output, the `-q` option includes the C source lines as assembler comments, and the `-L` option creates a listing.

The result is the assembler source `shell.s03` containing the declarations, function call, function return, variable accesses, and a listing file `shell.lst`.

Assembler support directives

There are six types of assembler directive that are used to create addresses for functions.

\$DEFFN

`$DEFFN` is the define function directive and takes three parameters:

- The function name.
- The eight size specifiers specify the number of bytes of the data area used:
 - local `DATA`
 - local `IDATA`
 - local `XDATA`
 - local `bit` variables
 - `DATA` parameters
 - `IDATA` parameters

XDATA parameters
bit parameters

- The functions which call the defined function.

The high byte of local DATA contains a set of flags to identify how the function is used:

Bit	Function use
0	Makes indirect calls
1	Interrupt
2	Not able to overlay
3-7	Not used

Table 10: Functions used in \$DEFFN

If bit 15 of the DATA parameter is set, the function list contains functions called by the function defined. For external functions, the local sizes are omitted.

\$REFFN

\$REFFN is the operator which references the function's address for expressions using a sixteen-bit address: LCALL, LJMP, or DW.

\$IFREF

\$IFREF is the operator which references indirect (pointer) function addresses: MOV, DPTR, or DW.

The #LOW and #HIGH assembler operators can be used with \$IFREF to provide the low or high byte of an address. The example below moves the fourth byte of the local block of func into A.

```
MOV A, #HIGH($IFREF func +3)
```

To move the address of the function to DPTR, use:

```
MOV DPTR, $IFREF func
```

\$LOCBD, \$LOCBI, \$LOCBB, AND \$LOCBX

\$LOCBD, \$LOCBI, \$LOCBB, and \$LOCBX are the operators which give the local variables (autos and parameters) in the DATA, IDATA, BIT, and XDATA memory area blocks.

The data types must match the expected values: \$LOCBX gives a sixteen-bit XDATA address for DW or MOV DPTR; \$LOCBD, \$LOCBI, and \$LOCBB produce an eight-bit address.

The #LOW and #HIGH assembler operators can be used with the \$LOCB directives to provide the low or high byte of an address.

\$PRMBD, \$PRMBI, \$PRMBB, AND \$PRMBX

\$PRMBD, \$PRMBI, \$PRMBB, and \$PRMBX are the operators which give the start address of the parameter memory area blocks in the DATA, IDATA, BIT, and XDATA memory area blocks.

The data types must match the expected values: \$PRMBX gives a sixteen-bit XDATA address for DW or MOV DPTR; \$PRMBD, \$PRMBI, and \$PRMBB produce an eight-bit address.

The #LOW and #HIGH assembler operators can be used with the \$PRMB directives to provide the low or high byte of an address.

\$BYTE3

\$BYTE3 is the operator which references the third byte of a function pointer (for banked memory).

EXAMPLE

The shell program produces the following assembler source as shell.s03. The file demonstrates several of the methods of parameter passing.

Assembler head

```
NAME      shell (16)
RSEG      CODE (0)
PUBLIC    assem
$DEFFN    assem(4,0,0,0,32772,0,2,0)
PUBLIC    main
$DEFFN    main(2,0,0,0,32768,0,0,0), assem
```

The \$DEFFN assembler directive is used to declare the two functions in this module. The three arguments to the directive are:

- The function name:
In this case `assem` and `main`.
- The eight size specifiers specify the number of bytes of data area used:
Local DATA (two int for `assem`, one int for `main`).
DATA parameters (one int and two char for `assem`).
XDATA parameters (one int for `assem`).
- The functions which are called from the defined function (If bit 15 of the DATA parameter is set, the function list contains functions called by the function defined. For external functions, the local sizes are omitted).

In this case `assem` is called by `main`.

Body

```
EXTERN ?CL8051C_5_20_L17
```

This declaration indicates the compiler (8051 version 5.20, compact memory model) and run-time source (`117.s03`).

```
RSEG    CODE
; 1.  char assem (char pc1, char pc2, xdata int pi1,
      int pi2)
; 2.  {
assem:
; 3.      int my_a;
; 4.      int my_b;
; 5.      my_a=pc1;
MOV     $LOCBD assem+4,R4
MOV     A,R4
MOV     R3,#0
```

The two character parameters are passed in R4 and R5.

```
MOV     $LOCBD assem+1,A
MOV     $LOCBD assem,R3
; 6.      my_b=pc2;
MOV     $LOCBD assem+5,R5
MOV     A,R5
MOV     R5,#0
MOV     $LOCBD assem+3,A
MOV     $LOCBD assem+2,R5
```

The assembler directive `$LOCBD` is used to create the address of the local variable. Note that the compiler and linker have produced the correct offsets for each of the parameters.

These data tables seem to refer to static variables, but the compiler and linker will keep track of which local function variables need to be maintained.

If a function's local variable is not needed (the function has not been entered or has already exited), the variable area will be reused by another function for its local variables.

Assembler return

```

; 7.      return(pi1+pi2);
MOV      A,R6
MOV      R4,A
MOV      A,R6
MOV      DPTR,#$LOCBX assem
XCH      A,R7
MOVX     @DPTR,A
INC      DPTR
XCH      A,R7
MOVX     @DPTR,A
MOV      A,R4

```

When there are too many parameters to pass in the registers, the parameter data area is used to hold the additional parameters. In this case the XDATA storage was specified explicitly.

```

ADD      A,$LOCBD assem+7
MOV      R4,A
; 8.    }
RET

```

Register R4 is used to hold the return char value. If more bytes were needed R4 to R7 would be used (R4 as the low byte).

```

; 9.    void main(void)
; 10.   {
main:
; 11.       int main_x=255;
CLR      A
MOV      $LOCBD main,A
DEC      A
MOV      $LOCBD main+1,A

```

Routine main has its own local data area.

```

; 12.   assem('x', 'y', main_x, 2);
CLR      A
MOV      $PRMBD assem+2,A
MOV      $PRMBD assem+3,#2

```

Two bytes for the constant value 2 are placed into the parameter area since R4, R5, R6, and R7 will be used for other values.

```

MOV      R6,$LOCBD main+1
MOV      R7,$LOCBD main

```

The local variable `main_x` is accessed and copied to R6 and R7.

```
MOV    R5, #121
MOV    R4, #120
```

Constant values are loaded into R4 and R5.

```
LDCALL $REFFN assem
```

The `REFFN` directive supplies the address of the `assem` function.

```
; 13. }
RET
END
```

Reentrant functions

Large reentrant functions use the reentrant stack in external RAM for their parameters. Small reentrant functions use the reentrant stack in internal RAM for their parameters. To call the function, push the parameters onto the stack and call the function address. If you need space for local variables, you must allocate them yourself. The compiler cannot create multiple local data blocks for reentrant functions.

Compile the example above with the C compiler reentrant code `-E` option to get a reentrant skeleton.

There are two styles of reentrant functions: K&R and ANSI.

- For K&R functions, when the function returns, pop the parameters off the stack (or restore the stack in some other way to deallocate the variables).
- For ANSI functions, deallocate the parameters before exiting the function.

Interrupt functions

The calling convention cannot be used for assembler interrupt functions since the interrupt may occur during the calling of a foreground function. Therefore, the requirements for interrupt function routine are different from those of a normal function routine, as follows:

- The routine cannot accept or return values.
- The routine must preserve all registers. The 8051 automatically saves PC on the hardware stack.
- The routine must exit using `RETI`. This automatically restores PC from the hardware stack
- The routine must treat all flags as undefined.

DEFINING INTERRUPT VECTORS

See the microcontroller include files for the interrupt templates. The simplest way to create an assembler interrupt is to use the `#pragma function=interrupt` statement from within the C compiler to create a skeleton interrupt program which can then be extended in assembler.

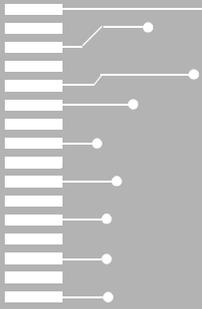
If you are writing an interrupt handler from assembler, use the `RCODE` and `INTVEC` segments to control the location for the code and interrupt vector table.

The `RCODE` segment is not banked (interrupt functions cannot be in banked memory).

The code for the interrupt handler can be in any module, but all interrupt handler addresses must be placed into the common `INTVEC` area.

The data registers present when the interrupt occurs must be preserved. You can either push all registers onto the stack or change the register bank to one dedicated to handling the interrupt (The `SFR` Processor Status Word contains bits which select the register bank). This can be done with the `using` keyword for the 8051 IAR C Compiler. For more information, see *using*, page 119.

Interrupts must restore the registers before returning with a `RETI` instruction.



Part 2: Compiler reference

This part of the 8051 IAR C Compiler Reference Guide contains the following chapters:

- Data representation
- Segments
- Compiler options
- Extended keywords
- #pragma directives
- Predefined symbols
- Intrinsic functions
- K&R and ANSI C language definitions
- Using C with PL/M.
- Tiny-51
- Diagnostics

Data representation

This chapter describes the C data types and pointers supported in the 8051 IAR C Compiler and shows how data is being represented. See the *Efficient coding techniques* chapter for information about which data types and pointers provide the most efficient code.

Data types

The 8051 IAR C Compiler supports all ANSI C basic elements. Variables are stored with the least significant part located at high memory address.

The following table gives the size and range of each C data type:

Data type	Bytes	Range	Notes
bit	1 bit	0 or 1	Single bit
sfr	1	0 to 255	Equivalent to unsigned char
char (by default)	1	0 to 255	Equivalent to unsigned char
char (using -c option)	1	-128 to 127	Equivalent to signed char
signed char	1	-128 to 127	
unsigned char	1	0 to 255	
short, int	2	-2^{15} to $2^{15}-1$	-32768 to 32767
unsigned short, unsigned int	2	0 to $2^{16}-1$	0 to 65535
long	4	-2^{31} to $2^{31}-1$	-2147483648 to 2147483647
unsigned long	4	0 to $2^{32}-1$	0 to 4294967295
pointer	1, 2 or 3		See the chapter <i>Extended keywords</i>
float	4	$\pm 1.18\text{E}-38$ to $\pm 3.39\text{E}+38$	
double, long double	4	$\pm 1.18\text{E}-38$ to $\pm 3.39\text{E}+38$	

Table 11: Data types

ENUM TYPE

The enum keyword creates each object with the shortest integer type (char, short, int, or long) required to contain its value.

BITFIELDS

Bitfield unions and structures are extensions to ANSI C integer bitfields.

Bitfields in expressions will have the same data type as the base type (signed or unsigned char, short, int, or long).

Bitfield variables are packed in elements of the specified type starting at the LSB position. For reversed packed bitfields, see `#pragma bitfields=reversed` directive page 122.

CHAR TYPE

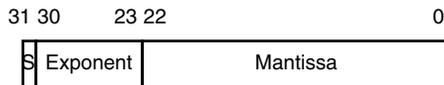
The `char` type is, by default, unsigned in the compiler, but the `-c` option allows you to make it signed. **Note:** The library is compiled with `char` types as unsigned.

FLOATING POINT

Floating-point values are represented by 4-byte numbers in standard IEEE format; float and double values have the same representation. Floating-point values below the smallest limit will be regarded as zero, and overflow gives undefined results.

4-byte floating-point format

The memory layout of 4-byte floating-point numbers is:



The value of the number is:

$$(-1)^S * 2^{(\text{Exponent}-127)} * 1.\text{Mantissa}$$

Zero is represented by the two most significant bytes which are zero. The two least significant bytes are then ignored.

The precision of the float operators (+, -, *, and /) is approximately 7 decimal digits.

BIT VARIABLES

Bit variables are stored as a single bit in the bit addressable area of the 8051 direct internal memory (0x20 to 0x2F). Bit variables are treated as 1-bit unsigned `chars` in expressions. The `BITVAR` segment is used for static bit variables and must be linked to a location in the bit-addressable memory (typically beginning at bit address 0). Bit address 0 is the least significant bit of internal RAM location 0x20.

Bit variables should not be confused with the C `bitfields` type, which does not have to be placed into the bit-addressable memory.

SPECIAL FUNCTION REGISTER VARIABLES

Special Function Register (`sfr`) variables are located in direct internal RAM locations `0x80` to `0xFF`. The `sfr` type allows a symbolic name to be associated with a byte in this range. The register at that address can be addressed symbolically, but no memory space is allocated. To define and use an `sfr` variable, see *sfr*, page 118.

Pointers

This section describes the 8051 IAR C Compiler's use of code pointers, data pointers, and constant pointers.

CODE POINTERS

The following code pointers are available:

Keyword	Storage	Restrictions
<code>non-banked</code>	2 bytes	May only point into the non-banked code area.
<code>banked</code>	3 bytes	No restrictions.

Table 12: Code pointers

The `non-banked` pointer is used to reference only functions that are in the non-banked code area. Non-banked code gives more efficient access than bank-switched code.

The `banked` pointer can reference any function and is less efficient.

DATA POINTERS

A generic pointer can point to any of the 8051 memory areas. A specific pointer points to one memory area.

General pointers are stored in three bytes of memory. The first byte of a data pointer identifies the memory area (`DATA`, `XDATA`, etc.), while the second and third byte specify the offset from the start of the memory area:

MEMTYPE	MSB of offset	LSB of offset
---------	---------------	---------------

The memory area is identified by a value from 0–3.

Area	MEMTYPE	Description
DATA, IDATA	0	Internal RAM
XDATA	1	External RAM
CODE	2	ROM
PDATA	3	Used with MOVX @R0

Table 13: Memory area

A specific pointer is declared as:

```
memory_segment data_type {idata|xdata|code} * pointer_name;
```

or

```
data_type {idata|xdata|code} * memory_segment pointer_name;
```

Examples

To create a pointer to `idata` located in the `XDATA` segment, use:

```
xdata int idata *px;
```

To declare a pointer to `code` located in the `IDATA` segment, use:

```
long code * idata pc;
```

A constant specific pointer uses a `define` with:

```
#define pointer_name (* (data_type segment *) constant_value
```

For example,

```
#define CODE_AREA (* (char code *) 0x0)
#define XDATA_IO (* (char xdata *) 0x4444)
#define MY_IDATA (* (char idata *) 0x10)
```

The code below shows how to create a three-byte pointer for an address in memory by specifying all three bytes.

```
#define CODE_AREA (* (char *) 0x020000)
#define XDATA_IO (* (char *) 0x014444)
#define PDATA_AREA (* (char *) 0x030000)
#define MY_IDATA (* (char *) 0x000010)
```

For example, `CODE_AREA` `MEMTYPE` is 02, `MSB` is 00, and `LSB` is 00.

The C program below contains several examples of pointer use:

```
#pragma language=extended
char mychar; /* global variable to */
```

```

/* hold char data from */
/* pointers */
char *gp; /* global generic */
/* pointer to character */
idata char xdata * pcix1; /* pointer to xdata */
/* character allocated */
/* in idata */
xdata char * pcxx1, * pcxx2; /* global generic */
/* pointer allocated in */
/* xdata */
#define my_xdata (* (char xdata *) 0xD000)
/* constant pointer to */
/* D000 in xdata */
data char * pcd; /* global generic */
/* pointer to char */
/* allocated in data*/

data char text[10];
void main(void)
{
    char *lp; /* local pointer to char */
    pcd="abcd"; /* pointer to string in */
/* ROM */

    pcxx1=(char *) 0x01DEEE;
    mychar=my_xdata;
    pcxx2=(char xdata *) 0xDEED;
/* constant pointer to */
/* char in xdata */

    text[0]= * pcxx1;
    putchar(text[0]);
    pcix1=(char xdata *) pcxx1; /* generic ptr is cast */
/* to specific pointer */

    mychar=* pcxx2;
    pcxx1=(char idata *) text;
    mychar=* pcxx2;
    text[1]=* lp;
    putchar(* (text+1));
    * pcxx2= mychar;
    * pcix1='a';
    * gp= 'a';
}

```

Notice how the code for the pointers differs between the types by examining the list file. The exact listing depends on the compile options used; the file below has been simplified to emphasize the code related to each C statement. Some lines of the list are reproduced below:

```

23          pcd="abcd";
\ 0000 750600          MOV     pcd+2, #LOW(?0000)

```

```

\ 0003 750500      MOV    pcd+1, #HIGH(?0000)
\ 0006 750402      MOV    pcd, #2

```

The pointer `pcd` is given the address of the constant string. The `?0000` will be replaced by the actual address by the XLINK Linker.

```

25      pcxx1=(char *) 0x01DEEE;
\ 0009 900000      MOV    DPTR, #pcxx1
\ 000C 7401        MOV    A, #1
\ 000E F0          MOVX   @DPTR, A
\ 000F A3          INC    DPTR
\ 0010 74DE        MOV    A, #222
\ 0012 F0          MOVX   @DPTR, A
\ 0013 A3          INC    DPTR
\ 0014 74EE        MOV    A, #238
\ 0016 F0          MOVX   @DPTR, A
26      mychar=my_xdata;
\ 0017 90D000      MOV    DPTR, #53248
\ 001A E0          MOVX   A, @DPTR
27      pcxx2=(char xdata *) 0xDEED;
\ 001B FC          MOV    R4, A
\ 001C 900003      MOV    DPTR, #pcxx2
\ 001F 7401        MOV    A, #1
\ 0021 F0          MOVX   @DPTR, A
\ 0022 A3          INC    DPTR
\ 0023 74DE        MOV    A, #222
\ 0025 F0          MOVX   @DPTR, A
\ 0026 A3          INC    DPTR
\ 0027 74ED        MOV    A, #237
\ 0029 F0          MOVX   @DPTR, A

```

The pointers `pcxx1` and `pcxx2` are given constant values (note that the memory specifier is 1 for `xdata`). The constant specific pointer `my_xdata` is used to access a char from memory location `0xD000`.

```

29      text[0]= * pcxx1;
\ 002A 900000      MOV    DPTR, #pcxx1
\ 002D E0          MOVX   A, @DPTR
\ 002E FF          MOV    R7, A
\ 002F A3          INC    DPTR
\ 0030 E0          MOVX   A, @DPTR
\ 0031 FE          MOV    R6, A
\ 0032 A3          INC    DPTR
\ 0033 E0          MOVX   A, @DPTR
\ 0034 FD          MOV    R5, A
\ 0035 120000      LCALL ?LD_A_R567_L17
\ 0038 F507        MOV    text, A

```

The character at pointer `pcxx1` is copied to the first position of the text string `text`.

```

31          pcix1=(char xdata *) pcxx1;
\ 0044 900001      MOV    DPTR,#pcxx1+1
\ 0047 E0         MOVX   A,@DPTR
\ 0048 FD         MOV    R5,A
\ 0049 A3         INC    DPTR
\ 004A E0         MOVX   A,@DPTR

```

The content of `pcxx1` is converted to an `xdata` pointer and left in registers A and R5 in preparation for storing it at location `pcix1`.

```

34          pcxx1=(char idata *) text;
\ 005A 7907      MOV    R1,#text
\ 005C 7A00      MOV    R2,#0
\ 005E 7B00      MOV    R3,#0
\ 0060 FE       MOV    R6,A
\ 0061 900000    MOV    DPTR,#pcxx1
\ 0064 EB       MOV    A,R3
\ 0065 F0       MOVX   @DPTR,A
\ 0066 A3       INC    DPTR
\ 0067 EA       MOV    A,R2
\ 0068 F0       MOVX   @DPTR,A
\ 0069 A3       INC    DPTR
\ 006A E9       MOV    A,R1
\ 006B F0       MOVX   @DPTR,A

```

The ‘pointer’ `text` is cast to be a pointer to an area in IDATA. The generic pointer, `pcxx1`, is assigned the value of `text`. Since IDATA is in page zero (0x0000 to 0x00FF) byte 1 is set to zero. Byte 2 is set to zero to identify the type of memory.

```

36          text[1]=* lp;
\ 007C A900      MOV    R1,$LOCBD main+2
\ 007E AA00      MOV    R2,$LOCBD main+1
\ 0080 AB00      MOV    R3,$LOCBD main
\ 0082 FE       MOV    R6,A
\ 0083 120000    LCALL  ?LD_A_R123_L17
\ 0086 F508      MOV    text+1,A

```

The contents of the indirection offset pointer are then copied to the second position of the `text` array.

```

38          * pcxx2= mychar;
\ 0099 E500      MOV    A,mychar
\ 009B FC       MOV    R4,A
\ 009C 900003    MOV    DPTR,#pcxx2
\ 009F E0       MOVX   A,@DPTR
\ 00A0 FF       MOV    R7,A
\ 00A1 A3       INC    DPTR
\ 00A2 E0       MOVX   A,@DPTR
\ 00A3 FE       MOV    R6,A

```

```

\ 00A4 A3          INC    DPTR
\ 00A5 E0          MOVX   A,@DPTR
\ 00A6 FD          MOV    R5,A
\ 00A7 EC          MOV    A,R4
\ 00A8 120000      LCALL  ?ST_A_R567_L17

```

The contents of the location pointed to by `pcxx2` is set to the contents of the variable `mychar`.

```

\ 0000          RSEG   CSTR
\ 0000          ?0000:   ; [UNKNOWN]
\ 0000 61626364  DB     'abcd',0
\ 0004 00
\ 0000          RSEG   D_UDATA
\ 0000          mychar:
\ 0001          DS     1
\ 0001          gp:
\ 0004          DS     3
\ 0004          ped:
\ 0007          DS     3
\ 0007          text:
\ 0011          DS     10
\ 0000          RSEG   I_UDATA
\ 0000          pcix1:
\ 0002          DS     2
\ 0000          RSEG   X_UDATA
\ 0000          pcxx1:
\ 0003          DS     3
\ 0003          pcxx2:
\ 0006          DS     3
\ 0006          END

```

Space is reserved in the data segments for the variables and pointers.

Note: The amount of memory reserved matches the data type. In addition, a generic pointer takes more time and size to use than a specific pointer.

Segments

The 8051 IAR C Compiler places code and data into named segments which are referred to by XLINK. Details of the segments are required for programming assembly language modules and are also useful when interpreting the assembly language output of the compiler.

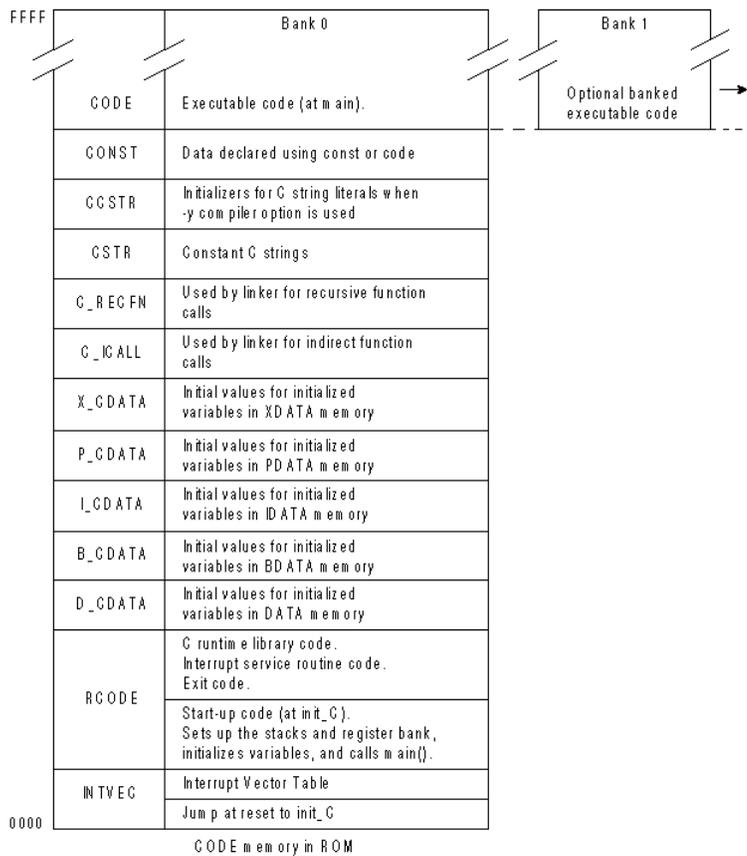
This chapter covers information about the segments used in this product and includes detailed reference information. Many of the extended keywords are also mentioned in this chapter. For additional information, see *Memory areas*, page 11.

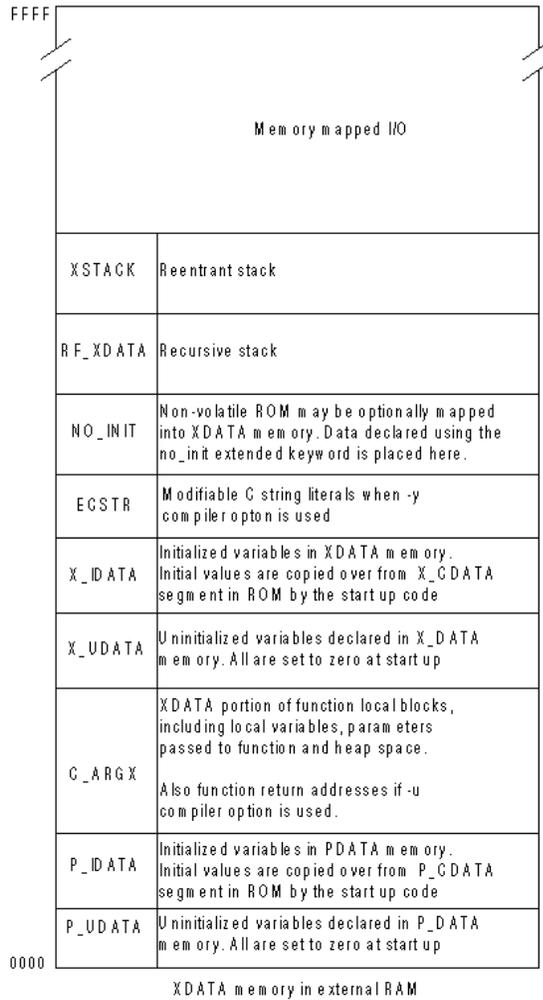
Memory maps

The diagrams on the following pages show the 8051 memory map, and the allocation of segments within each memory area.

IDATA memory		SRF Space Special function registers
FF		FF
	CSTACK The stack. Uses IDATA memory and the rest of the DATA memory	
	I_IDATA Initialized variables in IDATA memory. Initial values are copied over from I_CDATABA at startup	SBUF 99
	I_UDATA Uninitialized variables declared in IDATA memory. All set to Zero at startup	
	C_ARGI IDATA portion of function local blocks local vars and parameters	DPL 82
	D_IDATA Initialized variables declared in DATA memory. Initial values are copied over from D_CDATABA in ROM at startup	SP 81
	D_UDATA Uninitialized variables declared in DATA memory. All set to 0 at startup	PO 80
	C_ARGD DATA portion of function local blocks. Local variables and parameters	
	B_UDATA Uninitialized variables declared in BITVAR memory. All set to 0 at startup	
	B_IDATA Initialized variables declared in BITVAR memory. Initial values are copied over from B_CDATABA in ROM at startup	
	BITVAR Global bit variables. Must be linked between 20 and 2F	
	C_ARGB Bit addressable portion of function local blocks, including local variables and parameters. Must be linked between 20-2F	
18-1F	Register bank 3	
10-17	Register bank 2	
8-F	Register bank 1	
0-7	Register bank 0	

DATA, IDATA and SRF space in internal RAM





Descriptions of segments

This section provides an alphabetical list of the segments. For each segment, it shows

- The name of the segment.
- A brief description of the contents.
- Whether the segment is read/write or read-only.
- A fuller description of the segment contents and use.
- Whether the segment is 'Assembly-accessible'. This means that the user can add contents to any segment if the user follows the rules for the applicable segment(s). Certain segments come in pairs, data-places and constant initializers. For instance, others come alone or with constant data. Note that to use any segment that is not 'assembly-accessible' will probably induce a crash at the startup of the program.

`BITVAR` Bit variables with addresses between 20h and 2Fh (32 and 47) in DATA memory.

Type

Read/write.

Description

Assembly-accessible.

Holds `bit` variables and can also hold user-written relocatable bit-variables.

`B_CDATA` Initialized constants in CODE memory.

Type

Read-only.

Description

Assembly-accessible.

`CSTARTUP` copies initialized values from segments to the `B_IDATA` segment.

`B_IDATA` Initialized static data.

Type

Assembly-accessible.

Read/write.

Description

Initialized variables in BDATA memory. Initialized values are copied over B_CDATA segments in ROM by the startup code. Must be linked between 20h and 2Fh.

B_UDATA Uninitialized static data.

Type

Read/write.

Description

Assembly-accessible.

Holds static variables in BDATA memory that are not explicitly initialized; these are implicitly initialized all to zero, which is performed by CSTARTUP.

C_ARGB Local bit variables or parameters with addresses between 20h and 2Fh (32 and 47) in DATA memory.

Type

Read/write.

Description

Assembly-accessible.

Holds dynamically allocated `bit` variables such as local variables and parameters.

C_ARGD Local variables and parameters.

Type

Read/write.

Description

Assembly-accessible.

Holds dynamically allocated data such as local variables and parameters. These will be placed in DATA memory.

`C_ARGI` Local variables and parameters.

Type

Read/write.

Description

Assembly-accessible.

Holds dynamically allocated data such as local variables and parameters. These will be placed in `IDATA` memory.

`C_ARGX` Local variables and parameters.

Type

Read/write.

Description

Assembly-accessible.

Holds dynamically allocated data such as local variables and parameters. These will be placed in `XDATA` memory. If the **Stack expansion** `-u` option is selected, then function return addresses will also be stored in this segment.

`CCSTR` String literals.

Type

Read-only.

Description

Holds C string literal initializers when the `-y` C compiler option is active. For additional information, refer to the C compiler **Writable strings** (`-y`) option; see `-y`, page 105. See also `CSTR`, and `ECSTR`.

`C_ICALL` Indirect function call memory (`CODE`).

Type

Read-only.

Description

Assembly-accessible.

Used by the compiler when functions are called indirectly. The final address assignment is made by the linker.

CODE Code.

Type

Read-only.

Description

Holds user program code and various library routines that can run in alternative banks, and code from assembly language modules.

Notice that any assembly language routines included in the `CODE` segment must meet the calling convention of the memory model in use. For more information see *Assembler support directives*, page 49.

CONST Constants.

Type

Read-only.

Description

Used for storing `const` and `code` objects when the **Writable strings** (`-y`) option is not active. Can be used in assembly language routines for declaring constant data.

C_RECFN Recursive function memory (`CODE`).

Type

Read-only.

Description

Used by the compiler and linker to support recursive functions.

CSTACK Data stack.

Type

Read/write.

Description

Assembly-accessible.

Holds the internal stack in DATA/IDATA memory.

CSTR String literals.

Type

Read-only.

Description

Holds C string literals when the C compiler **Writable strings** (`-y`) option is not active, which is the default. For additional information see `-y`, page 105. See also *CCSTR*, and *ECSTR*.

D_CDATA Initialization constants in CODE memory.

Type

Read-only.

Description

Assembly-accessible.

CSTARTUP copies initialization values from this segment to the D_IDATA segment.

D_IDATA Initialized static data.

Type

Read/write.

Description

Assembly-accessible.

Holds static variables in DATA memory that are automatically initialized. See also D_CDATA.

D_UDATA Uninitialized static data.

Type

Read/write.

Description

Assembly-accessible.

Holds static variables in DATA memory that are not explicitly initialized; these are implicitly initialized to all zero, which is performed by CSTARTUP.

ECSTR Writable copies of string literals.

Type

Read/write.

Description

Holds writable copies of C string literals when the `-y` C compiler option is active. For more information, refer to the C compiler **Writable strings** (`-y`) option; see `-y`, page 105. See also *CCSTR* and *CSTR*.

I_CDATA Initialization constants placed in CODE memory.

Type

Read-only.

Description

Assembly-accessible.

CSTARTUP copies initialization values from this segment to the I_IDATA segment.

I_IDATA Initialized static data.

Type

Read/write.

Description

Assembly-accessible.

Holds static variables in indirect internal data (IDATA) memory that are automatically initialized. See also C_CDATAB.

I_UDATA Uninitialized static data.

Type

Read/write.

Description

Assembly-accessible.

Holds static variables in IDATA memory that are not explicitly initialized; these are implicitly initialized to all zero, which is performed by CSTARTUP.

INTVEC Interrupt vectors.

Type

Read-only.

Description

Assembly-accessible.

Holds the interrupt vector table generated by the use of the `interrupt` extended keyword (which can also be used for user-written interrupt vector table entries). The start of this segment should have address zero so that it may contain the reset and power-on vectors.

`NO_INIT` Non-volatile variables in external (XDATA) memory.

Type

Read/write.

Description

Assembly-accessible.

Holds variables to be placed in non-volatile memory. These will have been allocated by the compiler, declared `no_init` or created `no_init` by use of the memory `#pragma`, or created manually from assembly language source.

`P_CDATA` Initialized constants in CODE memory.

Type

Read-only.

Description

Assembly-accessible.

CSTARTUP copies initialized values from segments to the `P_IDATA` segment.

`P_IDATA` Initialized static data.

Type

Read/write.

Description

Assembly-accessible.

Holds static variables in `PDATA` memory that are automatically initialized. See also `P_CDATA`.

`P_UDATA` Uninitialized static data.

Type

Read/write.

Description

Assembly-accessible.

Holds static variables in PDATA memory that are not explicitly initialized; these are implicitly initialized all to zero, which is performed by CSTARTUP.

RCODE Vector or library handling code.

Type

Read-only.

Description

Assembly-accessible.

Used for start-up code, libraries, and interrupt handlers that must reside in non-banked code memory.

RF_XDATA External recursion stack.

Type

Read/write.

Description

Assembly-accessible.

Used by the linker to allocate a recursion stack in external data memory.

X_CDATA Initialization constants in CODE memory.

Type

Read-only.

Description

Assembly-accessible.

CSTARTUP copies initialization values from this segment to the X_IDATA segment.

X_CONST XDATA constants.

Type

Read-only.

Description

Assembly-accessible.

Holds constant data that should be stored in XDATA PROMs.

X_CSTR Constant strings in XDATA memory.

Type

Read-only.

Description

Assembly-accessible.

Holds constant strings that should be stored in XDATA PROMs.

X_IDATA Initialized static data.

Type

Read/write.

Description

Assembly-accessible.

Holds static variables in XDATA memory that are automatically initialized. See also X_CDATA.

X_UDATA Uninitialized static data.

Type

Read/write.

Description

Assembly-accessible.

Holds static variables in XDATA memory that are not explicitly initialized; these are implicitly initialized to all zero, which is performed by CSTARTUP.

XSTACK External stack.

Type

Read/write.

Description

Assembly-accessible.

Holds a simulated stack, including the stack-pointer, in XDATA memory. This stack is used for reentrant functions.

Compiler options

This chapter explains how to set the compiler options from the command line and gives reference information about each option so that you can run the compiler according to the application's requirements.



Refer to the *8051 IAR Embedded Workbench™ User Guide* for information about the compiler options available in the IAR Embedded Workbench and how to set them.

Setting compiler options

To set compiler options from the command line, include them on the command line after the `icc8051` command, either before or after the source filename. For example, when you compile the source `prog` to generate a listing to the default listing filename (`prog.lst`), it is entered as

```
icc8051 prog -L
```

Some options accept a filename, included after the option letter with a separating space. For example, to generate a listing to the file `list.lst`, enter it as

```
icc8051 prog -l list.lst
```

Some other options accept a string that is not a filename. This is included after the option letter, but without a space. For example, to generate a listing to the default filename but in the subdirectory `list`, it is entered as

```
icc8051 prog -Llist\
```

Generally, the order of options on the command line, both relative to each other and to the source filename, is not significant. The exception to that is the order in which two or more `-I` options are used. In that case, it is significant.

SPECIFYING OPTIONS USING ENVIRONMENT VARIABLES

Options can also be specified in the `QCC8051` environment variable. The compiler automatically appends the value of this variable to every command line, so it provides a convenient method of specifying options that are required for every compilation.

The following environment variables can be used by the 8051 IAR C Compiler:

Environment variable	Description
C_INCLUDE	Specifies directories to search for include files; for example: C_INCLUDE=\iar systems\ew23\8051\inc
QCC8051	Specifies command line options; for example: QCC8051=-q -L -z9

Table 14: Environment variables

Summary of compiler options

The following table summarizes the command line compiler options.

Option	Description
-A [<i>prefix</i>]	Assembly output to prefixed filename
-a <i>filename</i>	Assembly output to named file
-b	Make a LIBRARY module
-C	Nested comments
-c	char is signed char
-Dsymb	Defined symbols
-e	Enable language extensions
-E [L S]	Reentrant code generation
-F	Form feed after function
-f <i>filename</i>	Extend the command line
-G	Open standard input as source
-g	Global strict type checking
-gA	Flag old-style functions
-g0	No type info in object code
-Hname	Set object module name
-H[0 1 2 3]	Generate register dependent code.
-i	Add #include file text
-I [<i>prefix</i>]	Include paths
-K	// comments
-l <i>filename</i>	List to named file
-L [<i>prefix</i>]	List to prefixed source name

Table 15: Command line options

Option	Description
-m[<i>tsmc1b</i>]	Memory model
-n <i>filename</i>	Preprocessor to named file
-N[<i>prefix</i>]	Preprocessor to prefixed filename
-o <i>filename</i>	Set object filename
-O <i>prefix</i>	Set object filename prefix
-P	Generate PROMable code
-pnn	Lines/page
-q	Insert mnemonics
-r[012][i][n][r][e]	Generate debug information
-R <i>name</i>	Set code segment name
-s[0-9]	Optimize for speed
-S	Set silent operation
-T	Active lines only
-tn	Tab spacing
-U <i>symb</i>	Undefine symbol
-u	Stack expansion
-v[0 1]	Processor variant
-w[<i>s</i>]	Warnings
-X	Explain C declarations
-x[DFT2]	Cross reference
-y	Writable strings
-z[0-9]	Optimize for size

Table 15: Command line options (continued)

Descriptions of compiler options

The following sections give reference information about each compiler option.

-A -A*prefix*

Use this option to generate assembler source to the file *prefix*.s03.

By default the compiler does not generate an assembler source. To send assembler source to the file with the same name as the source leafname but with the extension s03, use -A without an argument.

For example:

```
icc8051 prog -A
```

generates an assembly source to the file `prog.s03`.

To send assembler source to the same filename but in a different directory, use the `-A` option with the directory as the argument. For example:

```
icc8051 prog -Aasm\
```

generates an assembly source in the file `asm\prog.s03`.

The assembler source may be assembled by the appropriate IAR assembler.

If the `-l` or `-L` and `-q` options are also used, the C source lines are included in the assembly source file as comments.

The `-A` option may not be used at the same time as the `-a` option.

`-a` *-a filename*

Use this option to generate assembler source to the file `filename.s03`.

By default the compiler does not generate an assembler source. This option generates an assembler source to the named file.

The filename consists of a leafname optionally preceded by a pathname and optionally followed by an extension. If no extension is given, the target-specific assembler source extension is used.

The assembler source may be assembled by the appropriate IAR Assembler.

If the `-l` or `-L` and `-q` options are also specified, the C source lines are included in the assembly source file as comments.

The `-a` option may not be used at the same time as `-A`.

`-b` *-b*

By default the compiler produces a program module ready for linking with the IAR XLINK Linker.

Use this option if you instead want a library module for inclusion in a library with the IAR XLIB Librarian.

`-C -C`

By default the compiler treats nested comments as a fault and issues a warning when it encounters one, resulting for example from a failure to close a comment. If you want to use nested comments, for example to comment-out sections of code that include comments, use the `-C` option to disable this warning.

`-c -c`

By default the compiler interprets the `char` type as `unsigned char`. To make the compiler interpret the `char` type as `signed char` instead, for example for compatibility with a different compiler, use this option.

Note: The run-time library is compiled without the `-c` option, so if you use this option for your program and enable type checking with the `-g` or `-r` options, you may get type mismatch warnings from the linker.

`-D -Dsymbol[=value]`

This option defines a symbol with the name `symbol` and the value `value`. If no value is specified, `1` is used.

The `-D` option has the same effect as a `#define` statement at the top of the source file.

`-Dsymbol`

is equivalent to:

```
#define symbol
```

The option `-D` is useful for specifying a value or choice that would otherwise be specified in the source file more conveniently on the command line.

There is no limit to the number of `-D` options that can be used on a single command line.

Command lines can become very long when using the `-D` option, in which case it may be useful to use a command file; see `-f`, page 89.

Example

For example, you could arrange your source to produce either the test or production version of your program depending on whether the symbol `testver` was defined. To do this you would use include sections such as:

```
#ifdef testver
... ; additional code lines for test version only
#endif
```

Then, you would select the version required in the command line as follows:

Production version: `icc8051 prog`

Test version: `icc8051 prog -Dtestver`

To include spaces in the expression, surround the entire option with double quotes. For example:

```
"-DEXPR=F + g"
```

is equivalent to:

```
#define EXPR F + g
```

To include double quote characters, use a backslash immediately in front of each double quote character. For example:

```
-DSTRING=\"microproc\"
```

is equivalent to:

```
#define STRING "microproc"
```

`-E[L|S]` `-E[L|S]`

This option enables generation of reentrant code.

Use the `-E` option when functions must be able to be called by interrupt functions (reentrant) or when a function calls itself (recursion).

The `-E` option will force the compiler to store parameters and local variables on a stack.

Without this option, simple recursive functions will work correctly but mutual recursion may not function as expected because speed local variables may be stored in fixed locations rather than on a stack.

Syntax	Description
<code>-E, -EL</code>	Large reentrant. Forces the compiler to use a simulated stack in XDATA for parameters and locals.
<code>-ES</code>	Small reentrant. Forces the compiler to use the hardware stack located in IDATA memory.

Table 16: Reentrant functions

`-e -e`

Enables target-dependent extensions to the C language.

By default language extensions are disabled in order to preserve portability. If you are using language extensions in the source, you must enable them by including this option.

For additional information, see *Using language extensions, page 3*.



Notice that in the IAR Embedded Workbench, this option is enabled by default.

`-F -F`

Generates a form-feed after each listed function in the listing.

By default the listing simply starts each function on the next line. To cause each function to appear at the top of a new page, you would include this option.

Form-feeds are never generated for functions that are not listed, for example, as in `#include` files.

`-f -f filename`

Extends the command line by reading command line options from the named file, with the default extension `xcl`.

By default the compiler accepts command parameters only from the command line itself and the `QCC8051` environment variable. To make long command lines more manageable, and to avoid any operating system command line length limit, you use the `-f` option to specify a command file, from which the compiler reads command line items as if they had been entered at the position of the option.

Note: Make sure that there is a space between the `-f` and filename.

In the command file, you format the items exactly as if they were on the command line itself, except that you may use multiple lines since the newline character acts just as a space or tab character.

Example

For example, you could replace the command line:

```
icc8051 prog -re -L -Dtestver "-Dusername=John Smith"
-Duserid=463760
```

with

```
icc8051 prog -re -L -Dtestver -f userinfo
```

and the file `userinfo.xcl` containing:

```
"-Dusername=John Smith"
-Duserid=463760
```

-G -G

Opens the standard input as source, instead of reading source from a file.

By default the compiler reads source from the file named on the command line. If you wish it to read source instead from the standard input (normally the keyboard), you use the `-G` option and omit the source filename.

The source filename is set to `stdin.c`.

-g -g [A] [O]

Enables checking of type information throughout the source.

There may be conditions in the source that indicate possible programming faults but which for compatibility the compiler and linker normally ignore. To cause the compiler and linker to issue a warning each time they encounter such a condition, use the `-g` option.

The conditions checked by the `-g` option are:

- Calls to undeclared functions.
- Undeclared K&R formal parameters.
- Missing return values in non-void functions.
- Unreferenced local or formal parameters.
- Unreferenced `goto` labels.
- Unreachable code.
- Unmatching or varying parameters to K&R functions.
- `#undef` on unknown symbols.
- Valid but ambiguous initializers.
- Constant array indexing out of range.

Flag old-style functions

Syntax: `-gA`

By default the `-g` option does not warn of old-style K&R functions. To enable such warnings, use the `-gA` option.

No type info in object code**Syntax:** `-g0`

By default the `-g` option includes type information in the object module, increasing its size and link time, allowing the linker to issue type check warnings. To exclude this information, avoiding this increase in size and link time but inhibiting linker type check warnings, use the `-g0` option.

When linking multiple modules, notice that objects in a module compiled without type information, i.e. without any `-g` option or with a `-g` option without a modifier, are considered typeless. Hence there will never be any warning of a type mismatch from a declaration from a module compiled without type information, even if the module with a corresponding declaration has been compiled with type information.

Examples

The following examples illustrate each type of error.

Calls to undeclared functions

Program:

```
void my_fun(void) { }
int main(void)
{
    my_func(); /* mis-spelt my_fun gives undeclared
               return 0;function warning */
}
```

Error:

```
my_func(); /* mis-spelt my_fun gives undeclared
-----^ function warning */

"undecfn.c",5 Warning[23]: Undeclared function 'my_func';
assumed "extern" "int"
```

Undeclared K&R formal parameters

Program:

```
int my_fun(parameter) /* type of parameter not declared */
{
    return parameter+1;
}
```

Error:

```
int my_fun(parameter) /* type of parameter not declared */
-----^
```

```
"undecfp.c",1 Warning[9]: Undeclared function parameter
'parameter'; assumed "int"
```

Missing return values in non-void functions

Program:

```
int my_fun(void)
{
    /* ... function body ... */
}
```

Error:

```
}
^
```

```
"noreturn.c",4 Warning[22]: Non-void function: explicit
"return" <expression>; expected
```

Unreferenced local or formal parameters

Program:

```
void my_fun(int parameter)/* unreferenced formal
                           parameter */
{
    int localvar;/* unreferenced local variable */
    /* exit without reference to either variable */
}
```

Error:

```
}
^
```

```
"unrefpar.c",6 Warning[33]: Local or formal 'localvar' was
never referenced
```

```
"unrefpar.c",6 Warning[33]: Local or formal 'parameter' was
never referenced
```

Unreferenced goto labels

Program:

```
int main(void)
{
    /* ... function body ... */
    exit: /* unreferenced label */
    return 0;
}
```

Error:

```
}
^
```

```
"unreflab.c",7 Warning[13]: Unreferenced label 'exit'
```

Unreachable code

Program:

```
#include <stdio.h>
int main(void)
{
    goto exit;
    puts("This code is unreachable");
    exit:
    return 0;
}
```

Error:

```
    puts("This code is unreachable");
-----^
```

```
"unreach.c",7 Warning[20]: Unreachable statement(s)
```

Unmatching or varying parameters to K&R functions

Program:

```
int my_fun(len, str)
int len;
char *str;
{
    str[0]='a' ;
    return len;
}
char buffer[99] ;
int main(void)
{
    my_fun(buffer, 99) ;/* wrong order of parameters */
    my_fun(99) ; /* missing parameter */
    return 0 ;
}
```

Error:

```
my_fun(buffer, 99) ;/* wrong order of parameters */
-----^
```

```
"varyparm.c",14 Warning[26]: Inconsistent use of K&R function
- changing type of parameter
```

```
my_fun(buffer,99) ;/* wrong order of parameters */
-----^
```

```
"varyparm.c",14 Warning[26]: Inconsistent use of K&R function
- changing type of parameter
```

```
my_fun(99) ;      /* missing parameter */
-----^
```

```
"varyparm.c",15 Warning[25]: Inconsistent use of K&R function
- varying number of parameters
```

#undef on unknown symbols

Program:

```
#define my_macro 99
/* Misspelt name gives a warning that the symbol is unknown */
#undef my_macor
int main(void)
{
    return 0;
}
```

Error:

```
#undef my_macor
-----^
```

```
"hundef.c",4 Warning[2]: Macro 'my_macor' is already #undef
```

Valid but ambiguous initializers

Program:

```
typedef struct t1 {int f1; int f2;} type1;
typedef struct t2 {int f3; type1 f4; type1 f5;} type2;
typedef struct t3 {int f6; type2 f7; int f8;} type3;
type3 example = {99, {42,1,2}, 37};
```

Error:

```
type3 example = {99, {42,1,2}, 37} ;
-----^
```

```
"ambigini.c",4 Warning[12]: Incompletely bracketed
initializer
```

Constant array indexing out of range

Program:

```
char buffer[99] ;
int main(void)
{
    buffer[500] = 'a' ;/* Constant index out of range */
    return 0;
}
```

Error:

```
buffer[500] = 'a' ;/* Constant index out of range */
-----^
```

```
"arrindex.c",5 Warning[28]: Constant [index] outside array
bounds
```

-H **-Hname**

Use this option to set the object module name.

By default the internal name of the object module is the name of the source file, without directory name or extension. To set the object module name explicitly, you use the **-H** option, for example:

```
icc8051 prog -Hmain
```

This is particularly useful when several modules have the same filename, since normally the resulting duplicate module name would cause a linker error, for instance when the source file is a temporary file generated by the preprocessor.

Example

The following example—in which `%1` is an operating system variable containing the name of the source file—will give duplicate name errors from the linker:

```
preproc %1.c temp.c; preprocess source, generating
                    temp.c
icc8051 temp.c    ; module name is always 'temp'
```

To avoid this, use **-H** to retain the original name:

```
preproc %1.c temp.c; preprocess source, generating
                    temp.c
icc8051 temp.c -H%1; use original source name as module
                    name
```

`-h` `-h[0|1|2|3]`

This option allows generation of register-bank-dependent code. The parameter specifies the register bank 0 to 3; the default is 0.

The options are as follows:

Syntax	Description
<code>-h0</code> (default)	Register bank 0
<code>-h1</code>	Register bank 1
<code>-h2</code>	Register bank 2
<code>-h3</code>	Register bank 3

Table 17: Options for the `-h` compiler option

`-I` `-Iprefix`

Adds a prefix to the list of `#include` file prefixes.

By default the compiler searches for include files only in the source directory (if the filename is enclosed in quotes as opposed to angle brackets), the `C_INCLUDE` paths, and finally the current directory. If you have placed `#include` files in some other directory, you must use the `-I` option to inform the compiler of that directory.

For example:

```
icc8051 prog -I\mylib\
```

Notice that the compiler simply adds the `-I` prefix onto the start of the include filename, so it is important to include the final backslash if necessary.

There is no limit to the number of `-I` options allowed on a single command line. When many `-I` options are used, to avoid the command line exceeding the operating system's limit, you would use a command file; see the `-f` option, page 89.

Note: The full description of the compiler's `#include` file search procedure is as follows:

When the compiler encounters an `#include` file name in angle brackets such as:

```
#include <stdio.h>
```

it performs the following search sequence:

- 1 The filename prefixed by each successive `-I` prefix.
- 2 The filename prefixed by each successive path in the `C_INCLUDE` environment variable if any.
- 3 The filename alone.

When the compiler encounters an `#include` file name in double quotes such as:

```
#include "vars.h"
```

it searches the filename prefixed by the source file path, and then performs the sequence as for angle-bracketed filenames.

-i `-i`

Use this option to make the compiler include `#include` files in the list file.

Normally the listing does not include `#include` files, since they usually contain only header information that would waste space in the listing. To include `#include` files, for example because they include function definitions or preprocessed lines, you include the `-i` option.

-K `-K`

Enables comments in C++ style, i.e. comments introduced by `/**` and extending to the end of the line.

For compatibility reasons, the compiler normally does not accept C++ style comments. If your source includes C++ style comments, you must use the `-K` option for them to be accepted.

-L `-L[prefix]`

Generates a listing for the file with the same name as the source but with extension `lst`, prefixed by the argument, if any.

By default the compiler does not generate a listing. To simply generate a listing, you use the `-L` option without a prefix.

`-L` may not be used at the same time as `-l`.

Example

To generate a listing in the file `prog.lst`, you use:

```
icc8051 prog -L
```

To generate a listing to a different directory, you use the `-L` option followed by the directory name. For example, to generate a listing on the corresponding filename in the directory `\list`:

```
icc8051 prog -Llist\
```

This sends the file to `list\prog.lst` rather than the default `prog.lst`.

`-l filename`

Generates a listing to the named file with the default extension `lst`.

By default the compiler does not generate a listing. To generate a listing to a named file, you use the `-l` option.

More often you do not need to specify a particular filename, in which case you can use the `-L` option instead.

This option may not be used at the same time as the `-L` option.

Example

To generate a listing to the file `list.lst`, use:

```
icc8051 prog -l list
```

`-m [t|s|c|m|l|b]`

The memory model determines the maximum size of code and maximum size of data normally available.

Use this option to specify the memory model for which the code is to be generated, as follows:

Option	Memory model	Default C library	a (80751)	Reentrant
<code>-mt</code>	Tiny	<code>c18051t.r03</code>	<code>c18051ta.r03</code>	<code>c18051tr.r03</code>
<code>-ms</code>	Small	<code>c18051s.r03</code>	<code>c18051sa.r03</code>	<code>c18051sr.r03</code>
<code>-mc</code>	Compact	<code>c18051c.r03</code>	—	<code>c18051cr.r03</code>
<code>-mm</code>	Medium	<code>c18051m.r03</code>	—	<code>c18051mr.r03</code>
<code>-ml</code>	Large	<code>c18051l.r03</code>	—	<code>c18051lr.r03</code>
<code>-mb</code>	Banked	<code>c18051b.r03</code>	—	<code>c18051br.r03</code>

Table 18: Specifying memory model (`-m`)

Note: The memory model option can only be set as a global option. All modules of a program must use the same memory model and must be linked with a library file for that model.

`-N Nprefix`

Use this option to generate preprocessor output to the file `prefix source.i`.

By default the compiler does not generate preprocessor output. To send preprocessor output to the file with the same name as the source leafname but with the extension `i`, use the `-N` without an argument.

The `-N` option may not be used at the same time as the `-n` option.

Example

To generate preprocessor output to the file `prog.i`, use the command:

```
icc8051 prog -N
```

To send preprocessor output to the same filename but in a different directory, use the `-N` option with the directory as the argument:

```
icc8051 prog -Npreproc\
```

generates a preprocessed source in the file `preproc\prog.i`.

`-n filename`

Generates preprocessor output to `filename.i`.

By default the compiler does not generate preprocessor output. This option generates preprocessor output to the named file.

The filename consists of a leafname optionally preceded by a pathname and optionally followed by an extension. If no extension is given, the extension `i` is used.

This option may not be used at the same time as `-N`.

`-O prefix`

Sets the `prefix` to be used on the filename of the object.

By default the object is stored with the filename corresponding to the source filename, but with the extension `r03`. To store the object in a different directory, you use the `-O` option.

The `-O` option may not be used at the same time as the `-o` option.

Example

To store the object in the `\obj` directory, use:

```
icc8051 prog -O\obj\
```

`-o filename`

Sets the *filename* in which the object module will be stored. The filename consists of an optional pathname, obligatory leafname, and optional extension (default `r03`).

By default the compiler stores the object code in a file whose name is

- The prefix specified by `-o`, plus
- The leafname of the source, plus
- The extension `r03`.

To store the object in a different filename, you use the `-o` option. For example, to store it in the file `obj.03`, you would use:

```
icc8051 prog -o obj
```

If instead you want to store the object with the corresponding filename but in a different directory, use the `-O` option.

The `-o` option may not be used at the same time as the `-O` option.

`-P -P`

This option causes the compiler to generate code suitable for running in read-only memory (PROM).

This option is included for compatibility with other IAR compilers; in the 8051 IAR C Compiler, it is always active.

`-p pnn`

This option causes the listing to be formatted into pages, and specifies the number of lines per page in the range 10 to 150.

By default the listing is not formatted into pages. To format it into pages with a form feed at every page, you use the `-p` option.

Example

To print a listing with 50 lines per page:

```
icc8051 prog -p50
```

`-q -q`

Includes generated assembly lines in the listing.

By default the compiler does not include the generated assembly lines in the listing. If you want these to be included, for example to be able to check the efficiency of code generated by a particular statement, you use the `-q` option.

Note: This option is only available if `-l` or `-L` is specified. For additional information, see also the options `-A`, page 85, `-a`, page 86, `-L`, page 97, and `-l`, page 98.

`-R` `-Rname`

Sets the name of the code segment.

By default the compiler places executable code in the segment named `CODE` which, by default, the linker places at a variable address. If you want to be able to specify an explicit address for the code, you use the `-R` option to specify a special code segment name which you can then assign to a fixed address in the linker command file.

`-r` `-r[012][i][n][r][e]`

Causes the compiler to include additional information required by C-SPY and other symbolic debuggers in the object modules.

By default, the compiler does not include debugging information, for code efficiency. To make code debuggable with C-SPY, you simply include the option with no modifiers. This gives source file references in object code. Using the `e` modifier includes the full source file into object code.

To make code debuggable with other debuggers, you select one or more options, as follows:

Command line modifier	Description
<code>e</code>	Embed C source into object file
<code>i</code>	Embed with include files
<code>n</code>	Embed but suppress source in object file
<code>0, 1, 2</code>	Source statement trace
<code>r</code>	Suppress temporary register variables

Table 19: Generating debug information (-r)

The `-r` option alone adds C source file references, symbol debug information, and other debug information to the object file. This makes it possible for the debugger to show source code in C source files as well as include files, track variables etc.

The modifiers *e*, *i*, and *n* are there for compatibility reasons, i.e. to be able to generate the UBROF 5 object file format from the linker at a later stage. This format is sometimes demanded by other debuggers. The modifier *e* is automatically chosen if *i* or *n* is chosen.

The `-re` option will copy the C source file into the object file using source references into the copied source.

The option `-ri` (same as `-rei`) will insert include files into the copied source as well, giving the possibility to debug code statements inside include files. A side effect is that the source line number is the global (=total) line count so far in the copied source. The option `-rn` (same as `-ren`) will give the same line count as the `-re` option but will not embed the source files into the object file.

By default the compiler tries to put locals as register variables. However, some debuggers cannot handle register variables; to suppress the use of register variables use the `-rr` option.

The source statement trace (`-r0`, `-r1`) will for `-r0` make sure that every C statement is at least one byte of code, by adding a `NOF`. For the `-r1` it will add a `NOF` in front of every C statement. Only use one of these options if your debugging tools specifically require you to do so.

The `-r2` option is provided for backward compatibility. It is normally not used.

`-S -S`

Sets silent operation by causing the compiler to operate without sending unnecessary messages to standard output (normally the screen).

By default, the compiler issues introductory messages and a final statistics report. To inhibit this output, you use the `-S` option. This does not affect the display of error and warning messages.

`-s -s [0-9]`

Causes the compiler to optimize the code for maximum execution speed.

By default the compiler optimizes for maximum execution speed at level 3 (see below). You can change the level of optimization for maximum execution speed using the `-s` option as follows:

Modifier	Level
0	No optimization.
1-3	Fully debuggable.

Table 20: Optimizing for speed (-s)

Modifier	Level
4–6	Some constructs not debuggable.
7–9	Full optimization.

Table 20: Optimizing for speed (-s) (continued)

Notice that the `-z` and `-s` options cannot be used at the same time.

`-T` `-T`

Causes the compiler to list only active source lines.

By default, the compiler lists all source lines. To save listing space by eliminating inactive lines, such as those in false `#if` structures, you use the `-T` option.

`-t` `-tn`

Sets tab spacing.

Set the number of character positions per tab stop to `n`, which must be in the range 2 to 9.

By default the listing is formatted with a tab spacing of 8 characters. If you want a different tab spacing, you set it with the `-t` option.

`-U` `-Usymb`

Removes the definition of the named symbol.

By default the compiler provides various predefined symbols. If you want to remove one of these, for example to avoid a conflict with a symbol of your own with the same name, you use the `-U` option.

For a list of the predefined symbols, see the chapter *Predefined symbols*.

Example

For example, to remove the symbol `__VER__`, use:

```
icc8051 prog -U__VER__
```

`-u` `-u`

Use this option to enable the function return stack expansion.

The `-u` option will force the compiler to store function return addresses in external (XDATA) memory.

-v -v[0|1]

Use this option to specify the processor variant:

Command line option	Description
-v0	Selects the 8051 processor option (default).
-v1	Selects the 80751 processor option. The 80751 is sufficiently similar to the 8051 when sharing a common compiler; however a few of the features of the 8051 are missing in the 80751. The -v1 option causes the compiler to avoid these incompatibilities.

Table 21: Specifying processor options

-w -ws

Disables warnings.

By default the compiler issues standard warning messages, and any additional warning messages enabled with the -g option.

To disable all warning messages, use the -w option. To make warnings give exit code 1, use the -ws option.

Exit codes	Description
0	No errors, warnings may appear.
1	Warnings
2	Errors

Table 22: Disabling warning messages

-X -X

Use this option to obtain descriptions in English of the C declarations, for example to aid the investigation of error messages.

Example

For example, the declaration:

```
void (* signal(int __sig, void (* func) ())) (int);
```

gives the description:

```
Identifier: signal
storage class: extern
  prototyped near_func function returning
    near - near_func code pointer to
```

```

        prototyped near_func function returning
        near - void
        and having following parameter(s):
        storage class: auto
        near - int
and having following parameter(s):
storage class: auto
near - int
storage class: auto
near - near_func code pointer to
near_func function returning
near - void

```

-x **-x [DFT2]**

Includes a cross-reference section in the listing.

By default the compiler does not include global symbols in the listing. To include at the end of the listing a list of all variable objects, and all functions, `#define` statements, `enum` statements, and `typedef` statements that are referenced, you use the `-x` option with no modifiers.

The following table describes the available modifiers:

Command line option	Description
<code>-xD</code>	Include unreferenced <code>#defines</code> .
<code>-xF</code>	Include unreferenced functions.
<code>-xT</code>	Include unreferenced <code>enum</code> and <code>typedefs</code> constants.
<code>-x2</code>	Dual line spacing.

Table 23: Including cross-references in list file (-x)

-Y **-Y**

Causes the compiler to compile string literals and other constants as initialized variables.

By default string literals and constants are compiled as read-only. If you want to be able to write to them, use the `-y` option, causing them to be compiled as writable variables.

Note: Arrays initialized with strings (i.e. `char c[] = string`) are always compiled as initialized variables, and are not affected by the `-y` option.

`-z` `-z [0-9]`

Causes the compiler to optimize the code for minimum size.

By default the compiler optimizes for minimum size at level 3 (see below). You can change the level of optimization for minimum size using the `-z` option as follows:

Modifier	Level
0	No optimization.
1-3	Fully debuggable.
4-6	Some constructs not debuggable.
7-9	Full optimization.

Table 24: Optimizing for size (-z)

Notice that `-z` and `-s` cannot be used at the same time.

Extended keywords

This chapter describes the non-standard keywords that support specific features of the 8051 microcontroller for data storage, function execution, function calling convention and function storage.

Using extended keywords

You can use keywords and the `sfr` keyword only if language extensions are enabled in the 8051 IAR C Compiler. Use the `-e` compiler option to enable language extensions. See `-e`, page 89, for additional information.



In the IAR Embedded Workbench, language extensions are enabled by default.

The extended keywords provide the following facilities:

ADDRESS CONTROL

By default the address range in which the compiler places a variable or function is determined by the memory model chosen. The program may achieve additional efficiency for special cases by overriding the default by using the `#pragma memory=data keyword area extended language statement`:

Data keyword	Description
<code>bdata</code>	Bit addressable memory (address 20H-2FH)
<code>bit</code>	Bit variables
<code>code</code>	Constants in CODE memory
<code>data</code>	Data addressable directly (address 00H-7FH)
<code>default</code>	Resets compiler memory selector to use its default segments
<code>idata</code>	Indirectly accessed data (normally in internal RAM)
<code>no_init</code>	Data in non-volatile external RAM
<code>pdata</code>	Paged memory segment in external RAM
<code>xdata</code>	Data in external RAM
<code>xdataconst</code>	Constant in external RAM

Table 25: Reserved keywords

Initialized variables use two memory segments per memory area in the chip. The modifiable variable is in RAM and the value used to initialize the variable is in ROM.

For example, the `X_CDATAL` segment is the initialized data segment in ROM for data in the `X_IDATAL` segment in external RAM.

You can also create your own code segment by using the `RSEG` relocatable segment directive in assembler. Code segments may be renamed at compile time using the `-R` compiler option; for more information, see `-R`, page 101.

Use the `pragma memory=dataseg (segment name)` to force the compiler to use a particular data segment. For more information, see `memory=dataseg`, page 129.

You can also specify to the compiler how functions use memory. For example the function types `banked` and `non-banked` determine whether the function is placed into the bank switched area of ROM or the non-banked area.

The `pdata` memory provides direct addressing access to a defined page (256 bytes) of the external memory.

I/O ACCESS

The program may access the 8051 I/O system using the `sfr` data types or absolute bit addressing.

BIT VARIABLES

The program may take advantage of the 8051 bit-addressing modes by using the `bit` data type.

NON-VOLATILE RAM

Variables may be placed in external non-volatile RAM by using the `no_init` data type modifier.

INTERRUPT ROUTINES

Interrupt routines may be written in C using the following keywords:

<code>interrupt</code>	The function is an interrupt function
<code>monitor</code>	The function cannot be interrupted.

Descriptions of extended keywords

The following general parameters are used in several of the definitions:

Parameter	Description
<code>storage-class</code>	Denotes an optional keyword <code>extern</code> or <code>static</code>
<code>declarator</code>	Denotes a standard C variable or function declarator

Table 26: Extended keywords general parameters

```
bdata storage-class bdata function-declarator
      storage-class type decl-list
```

where *decl-list* is a list of: <bdata> name.

Description

Places a variable of type `char` in the bit-addressable memory 0020h to 002Fh allowing the variable to be bit-addressable.

Note: The type has to be `char`.

Examples

```
bdata char my_bit_data;
static char ch1, bdata ch2, ch3;
if( my_bit_data.2 == my_bit_data.3 )
    my_bit_data.2 = 0;
```

```
bit Relocatable address:
```

```
storage-class bit identifier
```

Fixed address:

```
bit identifier = constant-expression.bit-selector
```

SFR:

```
bit identifier = sfr-identifier.bit-selector
```

Description

Declares a bit variable.

The `bit` variable is a variable whose storage is a single bit. It may have values 0 and 1 only. Bit variables should not be confused with the standard C bitfields.

A bit variable can be one of the following kinds:

Bit variable type	Description
Relocatable address	The variable is one bit allocated in the bit memory of the 8051
Fixed address	The variable is one bit in either the bit memory or in a bit <code>sfr</code>

Table 27: Bit variable types

Bit variables can be used in all places where it is allowed to use other integral types, except:

- As operand to the unary & (address) operator
- As struct/union elements
- As a parameter in an indirectly-called function
- As a parameter in a recursive or reentrant function
- As a parameter in a K&R function type
- Bit arrays are not allowed
- Bit casts are not allowed.

Examples

```
bit bit_addr_27=0x23.7;
bit p1_1=P1.2;
```

`code` *storage-class* `code declarator`

Description

Places an object in code (ROM) memory.

The `code` memory type attribute is used to place an object in code (ROM) memory. It overrides the default memory area of the memory model currently in effect.

The code attribute cannot be used in the following situations:

- With parameters and auto variables
- As struct/union elements
- In cast expressions.

Examples

```
extern code char *myarray[10] ;
```

See also the `const` keyword, page 139.

`code (pointer)` *memory_storage pointer_to_type* `code * pointer_name`

Description

Defines a pointer as a pointer to code memory.

Parameters

<code>memory_storage</code>	Location of pointer.
<code>pointer_to_type</code>	Type of data pointed to.
<code>pointer_name</code>	Name of the pointer.

Examples

```
xdata char code *myptr;
extern xdata int code *mycptr;
```

See also the keywords *idata*, page 111, and *xdata*, page 119.

```
data storage-class data declarator
```

Description

Places an object in internal RAM.

The `data` memory type attribute is used to place an object in the directly-addressable internal RAM memory of the 8051 microcontroller (0x00 to 0x7F). It overrides the default memory area of the memory model currently in effect.

The `data` attribute can be used in any standard C variable declaration, except:

- In parameters of indirectly called functions.
- In cast expressions.
- As `struct/union` elements.

Examples

```
data char *myarray[10] ;
void f(data int myint);
```

See also the keywords *idata*, page 111, and *xdata*, page 119.

```
idata storage-class idata declarator
```

Description

Places an object in the indirectly-addressable internal RAM.

The `idata` memory type attribute is used to place an object in the indirectly-addressable internal RAM memory of the 8051 (0x00 to 0x7F or 0x00 to 0xFF, depending on the microcontroller version). It overrides the default memory area of the memory model currently in effect.

The `idata` attribute can be used in any standard C variable declaration, except:

- In parameters of indirectly called functions.
- In cast expressions.
- As struct/union elements.

Examples

```
idata char *myarray[10] ;
void f(idata int myint);
```

See also the keywords *data*, page 111, and *xdata*, page 119.

`idata (pointer)` *memory_storage pointer_to_type idata * pointer_name*

Description

Defines a pointer as a pointer to `idata` memory.

Parameters

<code>memory_storage</code>	Location of pointer.
<code>pointer_to_type</code>	Type of data pointed to.
<code>pointer_name</code>	Name of the pointer.

Examples

```
idata char idata *myptr;
extern xdata int idata *mycptr;
```

See also the keywords *code*, page 110, and *xdata*, page 119.

`interrupt` *storage-class interrupt function-declarator*
storage-class interrupt [vector] function-declarator
storage-class interrupt [vector] using [reg-bank]
function-declarator

Description

Declares an interrupt function.

The `interrupt` keyword declares a function that is called upon a processor interrupt. The function must be void and have no arguments.

If a vector is specified, the address of the function is inserted in that vector. If no vector is specified, an appropriate entry must be provided in the vector table (preferably placed in the `cstartup` module) for the interrupt function.

The run-time interrupt handler takes care of saving and restoring processor registers, and returning via the `RETI` instruction.

The compiler disallows calls to interrupt functions from the program itself. It does allow interrupt function addresses to be passed to function pointers which do not have the `interrupt` attribute. This is useful for installing interrupt handlers in conjunction with operating systems.

The `using` keyword defines which register bank to use for the default register bank. Switching the register banks allows faster interrupt processing and eliminates the need to stack the current register contents. When switching register banks, it must be ensured that the area of memory for the interrupt register bank is not used by the linker options for data area. If the `using` keyword is omitted, the default register bank will be used. The memory area for each register bank is listed below.

Bank	Internal RAM location
0	00H-07H
1	08H-0FH
2	10H-17H
3	18H-1FH

Table 28: Memory areas for each register bank

Parameters

<i>function-declarator</i>	A void function declarator having no arguments.
[vector]	A square-bracketed constant expression yielding the vector address.
[reg-bank]	The register bank to use for the interrupt.

```
monitor storage-class monitor function-declarator
       storage-class monitor [enable-point] function-declarator
```

Description

Makes function atomic.

The `monitor` keyword causes interrupts to be disabled during execution of the function. This allows atomic operations to be performed, such as operations on semaphores that control access to resources by multiple processes. After execution the `monitor` function will re-enable interrupts.

A function declared with `monitor` is equivalent to a normal function in all other respects.

Parameters

<i>function-declarator</i>	A function declarator
<i>enable-point</i>	A value in the range to 2, as follows: <ul style="list-style-type: none"> In default mode (no <code>enable-point</code> set), the <code>monitor</code> function saves the interrupt status and then disables interrupts by clearing bit 7 of register IE when entering the function and restoring status on exiting. 1 Disables interrupts and then passes the parameters to the <code>monitor</code> function. Interrupts are enabled on exiting. 2 Saves the interrupt status and then disables interrupts and passes the parameters to the <code>monitor</code> function. Interrupt status is restored on exiting.

Examples

The example below disables interrupts while the flag is modified.

```
char printer_free; /* printer-free */
                /* semaphore */

monitor int got_flag(char *flag) /* With no danger of */
    /* interruption ... */
{
    if (!*flag) /* test if available */
    {
```

```

    return (*flag = 1);/* yes - take */
}
return (0);/* no - do not take */
}
void f(void)
{
    if (got_flag(&printer_free))/* act only if */
        /* printer is free */
        .... action code ....
}

```

`non_banked` *storage-class non_banked declarator*

Description

Function or function pointer modifier.

By default, in the banked memory model, all functions are callable from any bank. The `non_banked` keyword indicates that the function is always in the same bank as the caller, and so can be called by the faster non-banked method.

Examples

Function `test` is local to one file, and is only called by functions within the same file. It is therefore always in the same bank as the caller:

```

static non_banked void test(void)
{
    ...
}
void testcaller(void)
{
    ...
    test();/* call test by faster */
        /* non-banked method */
}

```

`no_init` *storage-class no_init declarator*

Description

Type modifier for non-volatile variables.

By default, the compiler places variables in main, volatile RAM and initializes them on start-up. The `no_init` type modifier causes the compiler to place the variable in non-volatile RAM and not to initialize it on start-up.

`no_init` variables are assumed to reside in external RAM. `no_init` variable declarations may not include initializers.

If non-volatile variables are used, it is essential for the program to be linked to refer to the non-volatile RAM area. For details, see *Non-volatile RAM*, page 20.

Examples

The example below shows valid and invalid methods of using the `no_init` keyword.

```
no_init int settings[50];/* array of non-volatile */
    /* settings */
no_init int idata i ;/* conflicting type */
    /* modifiers - invalid */
no_init int i = 1 ;/* initializer included */
    /* - invalid */
```

`pdata` *storage-class pdata declarator*

storage-class type decl-list

where *decl-list* is a list of: `<pdata> name`

*memory_storage base_type pdata * pointer_name*

Description

Sets one PDATA segment (256 bytes) on the microcontroller. This means that moving data to/from this PDATA segment can be done more efficient. The `pdata` memory type attribute is used to place an object in an external memory area 0000h to FFFFh. It overrides the default memory area of the memory model currently in effect. All `pdata` objects are static. If you use the first syntax rule, all declarations in *declarator* are `pdata`. If you use the second rule, only the declarations prefixed by `pdata` are `pdata`.

When `pdata` is in use, `CSTARTUP` sets up the base-page (P2).

Examples

```
pdata char mychar[10];
int i, pdata j,k;
pdata char pdata * my_pointer;
```

As another example, if we want to load A-reg some data that is not located in a PDATA segment, then the compiler generates the following code:

```
\ 0000 900000          MOV  DPTR,#?0000
\ 0003 E0             MOVX  A,@DPTR
```

The corresponding code putting the data in the PDATA segment would be:

```
\ 0000 7800          MOV  R0,#LOW(?0000)
\ 0002 E2           MOVX  A,@R0
```

`plm` *storage-class plm function_declarator*

Description

The `plm` keyword enables functions to call `plm` functions or to be callable from `plm` functions (or functions that use the PL/M-51 interface). For more information, see the *Using C with PL/M* chapter.

Examples

```
extern plm void plm_F();
plm void myfun()
{
    /* C code here */
}
```

`reentrant` *storage-class reentrant function_declarator*

Description

Places function stack in XDATA memory.

Use the `reentrant` keyword to give the function a simulated stack in XDATA memory. This enables it to be called from `main` and from interrupt functions.

Examples

The example below declares `myfun` as `reentrant`:

```
extern reentrant void allagazam_F();
reentrant void myfun()
{
    /* C code here */
}
```

```
reentrant_idata storage-class reentrant_idata function_declarator
```

Description

Makes a small reentrant function that uses the internal stack for its local variables. This means that the software stack is placed in `idata` memory.

Examples

```
extern re_entrant_idata void my_func();
reentrant_idata void my_func()
{
    /* C code here */
}
```

```
sfr sfr identifier = constant-expression
```

Description

Declares an object of one-byte I/O data type.

`sfr` denotes an 8051 SFR register which:

- Is equivalent to unsigned char
- Can only be directly addressable
- Resides at a fixed location in the range 0x80 to 0xFF.

The value of an `sfr` variable is the contents of the SFR register at the address *constant-expression*. All operators that apply to integral types except the unary & (address) operator may be applied to `sfr` variables.

In expressions, `sfr` variables may also be appended by a period followed by a bit-selector.

Predefined `sfr` declarations for popular members of the 8051 family are supplied; see the directory `\ew23\8051\inc\io51.h`

Examples

The example below uses `sfr` to access the port at 0x80.

```
sfr P0 = 0x80; /* Defines P0 */
void func()
{
    P0 = 4; /* Set entire variable */
        /* P0 = 00000100 */

    P0.2 = 1; /* Only affects one bit */
}
```

```

        /* P0 = XXXXX1XX */

        if (P0 & 4) printf("ON"); /* Read entire P0 and */
        /* mask bit 2 */

        if (P0.2) printf("ON"); /* Same but does bit */
        /* access only */
    }

```

using The `using` keyword allows an interrupt service routine declaration to specify a register bank for that routine to use.

Interrupt Service Routines defined (or declared) with the `using` keyword are subject to more stringent reentrancy requirements than those which use the default register bank; not even C run-time library routines calls are permitted.

Because the `using` keyword is only used with the `interrupt` keyword, see *interrupt*, page 112 and *using*, page 119 for syntax and further details.

xdata *storage-class xdata declarator*

Description

Places an object in external memory.

The `xdata` memory type attribute is used to place an object in the external memory area (0x0000 to 0xFFFF). It overrides the default memory area of the memory model currently in effect.

The `xdata` attribute can be used in any standard C variable declaration, except:

- In parameters of indirectly called functions.
- In cast expressions.
- As struct/union elements.

Examples

```

xdata char *myarray[10] ;
void f(xdata int myint);

```

See also keywords “idata” on page 111 and “data” on page 111.

`xdata (pointer) memory_storage pointer_to_type xdata * pointer_name`

Description

Defines a pointer as a pointer to XDATA memory.

Parameters

<i>memory_storage</i>	Location of pointer
<i>pointer_to_type</i>	Type of data pointed to
<i>pointer_name</i>	Name of the pointer

Examples

```
idata char xdata *myptr;
extern xdata int xdata *mycptr;
```

See also the keywords *code*, page 110, and *idata*, page 111.

#pragma directives

#pragma directives control how the compiler allocates memory, whether the compiler allows extended keywords, and whether the compiler outputs warning messages.

The use of #pragma directives is always enabled in the 8051 IAR C Compiler. They are consistent with the ISO/ANSI C and are very useful when you want to make sure that the source code is portable.

This chapter describes the syntax and gives a description of the #pragma directives of the 8051 IAR C Compiler.

#pragma directives summary

#pragma directives provide control of extension features while remaining within the standard language syntax.

The following categories of #pragma functions are available:

BITFIELD ORIENTATION

```
#pragma bitfields=default  
#pragma bitfields=reversed
```

EXTENSION CONTROL

```
#pragma language=default  
#pragma language=extended
```

FUNCTION ATTRIBUTE

```
#pragma codeseg (seg_name)  
#pragma function=default  
#pragma function=interrupt  
#pragma maxargs number_of_bytes  
#pragma function=monitor  
#pragma function=non_banked  
#pragma function=plm  
#pragma function=reentrant  
#pragma function=reentrant_idata  
#pragma overlay=default  
#pragma overlay=off
```

MEMORY USAGE

```
#pragma memory=code  
#pragma memory=constseg (seg_name)  
#pragma memory=data
```

```
#pragma memory=dataseg (seg_name)
#pragma memory=default
#pragma memory=idata
#pragma memory=no_init
#pragma memory=pdata
#pragma stringalloc=default
#pragma stringalloc=xdata
#pragma memory=xdata
#pragma memory=xdataconst
```

WARNING MESSAGE CONTROL

```
#pragma warnings=default
#pragma warnings=off
#pragma warnings=on
```

Descriptions of #pragma directives

This section describes the #pragma directives in alphabetical order. The #pragma directives are available regardless of the -e option.

Note: The #pragma function=intrinsic, which can be seen in the IAR C library files, is for IAR internal use only. You should *not* use it in your code since this could result in unexpected behavior.

```
bitfields=default #pragma bitfields=default
```

Description

Restores default order of storage of bitfields.

This directive causes the compiler to allocate bitfields in its normal order. See bitfields=reversed.

```
bitfields=reversed #pragma bitfields=reversed
```

Description

Reverses order of storage of bitfields.

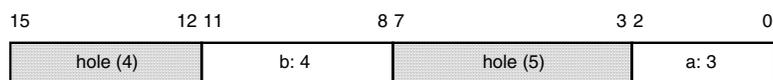
This directive causes the compiler to allocate bitfields starting at the most significant bit of the field, instead of at the least significant bit. The ANSI standard allows the storage order to be implementation-dependent; you may run into portability problems, which can be avoided by use of this keyword.

Example

The default layout of

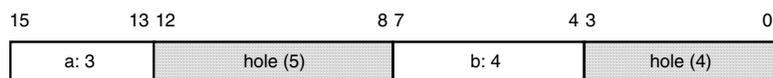
```
struct
{
  short a:3; /* a is 3 bits */
  short :5; /* this reserves a hole of 5 bits */
  short b:4; /* b is 4 bits */
} bits; /* bits is 16 bits */
```

in memory is:



```
#pragma bitfields=reversed
struct
{
  short a:3; /* a is 3 bits */
  short :5; /* this reserves a hole of 5 bits */
  short b:4; /* b is 4 bits */
} bits; /* bits is 16 bits */
```

has the following layout:



```
codeseg #pragma codeseg (seg_name)
```

where *seg_name* specifies the segment name, which must not conflict with data segments.

Description

Places subsequent code in the named segment and is equivalent to using the C compiler `-R` option; see `-R`, page 101. The `#pragma` directive can only be executed once by the compiler.

```
function=default #pragma function=default
```

Description

Restores function definitions to the default type.

Example

The example below shows how to insert a non-banked function and return to banked mode (it assumes that banked memory is available).

```
#pragma function=non_banked
extern void f1(); /* Identical to extern far void
                f1() */
#pragma function=default
extern int f3(); /* Default function type */
```

```
function=interrupt #pragma function=interrupt
```

Description

Makes subsequent function definitions interrupt.

This directive makes subsequent function definitions of interrupt type. It is an alternative to the function attribute interrupt. For more information, see *interrupt*, page 112.

Notice that #pragma function=interrupt does not offer a vector option.

Example

```
#pragma function=interrupt
void process_int() /* an interrupt function */
{
    ...
}
#pragma function=default
```

```
function=monitor #pragma function=monitor
```

Description

Makes function definitions atomic (non-interruptable).

Makes subsequent function definitions of monitor type. It is an alternative to the function attribute monitor.

Example

The function `f2` in the following example will execute with interrupts temporarily disabled.

```
#pragma function=monitor
void f2()/* Will make f2 a monitor function */
{
    ...
}
#pragma function=default
```

```
function=non_banked #pragma function=non-banked
```

Description

This directive makes subsequent function definitions of `non_banked` type. It is an alternative to the function attribute `non_banked`. For more information, see *non_banked*, page 115.

Example

The example below shows a non-banked function `f2`.

```
#pragma function=non_banked
void f2(void)
{
    ...
}
#pragma function=default
```

```
function=plm #pragma function=plm
```

Description

The `plm` `#pragma` directive enables functions to call PL/M-51 functions or to be callable from PL/M-51 functions (or functions that use the PL/M-51 interface). For more information, see the *Using C with PL/M* chapter.

Examples

```
#pragma function=plm
int plm_sum (int myi, int myj)
{
    return (myi+myj) ;
}
```

```
function=reentrant #pragma function=reentrant
```

Description

Make subsequent function definitions of small reentrant type. This means that a simulated XDATA stack will be used, and locals and parameters will be placed in XDATA memory.

Example

```
#pragma function=reentrant
void f2(void)
{
    ...
}
#pragma function=default
```

```
function=reentrant_idata #pragma function=reentrant_idata
```

Description

Makes subsequent function definitions of small reentrant type. This means that the hardware stack will be used, and locals and parameters will be placed in IDATA memory.

Example

```
#pragma function=reentrant_idata
void f2(void)
{
    ...
}
#pragma function=default
```

```
language=default #pragma language=default
```

Description

Returns extended keyword availability to the default set by the compiler -e option. See *language=extended*.

Example

See the example *language=extended* below.

```
language=extended #pragma language=extended
```

Description

Makes the extended keywords available regardless of the state of the compiler option `-e`; see page 88 for additional information.

Example

In the example below, the `shortad` extended language modifier is enabled for the definition of the function `func`. `mycount` is defined in the standard way.

```
#pragma language=extended
interrupt [0x0B] void func (void);
#pragma language=default
int mycount;
```

```
maxargs #pragma maxargs number_of_bytes
```

Parameters

<i>number_of_bytes</i>	The maximum number of bytes of function arguments that will be passed.
------------------------	--

Description

The `#pragma maxargs` informs the compiler that indirect functions may have variable arguments sizes. This is necessary when the indirectly called function is in one module and the calling function is in another.

The `#pragma` directive is only effective for the single function immediately following it.

Example

The example below shows a function which can accept one to four characters. Four bytes have been reserved for its parameters.

```
#pragma maxargs 4
void myfunc(char, . . . )
{
    /* code to access one to four characters here */
}
```

The module which calls the function will have the following lines:

```
void (* fp) (char first, . . . );
void main(void)
```

```
{
    fp=myfunc;
    fp('a');
    fp('a', 'b');
}
```

`memory=memory_segment` #pragma `memory=memory_segment`

Parameters

<code>memory_segment</code>	One of the standard memory segments: CODE, code (ROM) memory DATA, directly addressable internal RAM IDATA, indirectly addressable internal RAM PDATA, paged indirectly addressable external memory XDATA, external data memory XDATACONST, external data memory as constant
-----------------------------	--

Description

Directs variables to the memory area by default. The default may be overridden by the memory attributes.

Use `xdataconst` if you run out of CODE (PROM) space but still have some place in XDATA area where you can place a PROM to store constants. `xdataconst` places such constants in segment `X_CONST` to be located in the XDATA PROM area at link time.

See the *Extended keywords* chapter for examples.

`memory=constseg` #pragma `memory=constseg(seg_name)`

Description

Directs constants to the named segment by default. It is an alternative to the memory attribute keywords. The default may be overridden by the memory attributes.

Note: The `seg_name` must not be one of the compiler's reserved segment names.

Example

The example below places the constant array `arr` into the ROM segment `TABLE`.

```
#pragma memory=constseg(TABLE)
char arr[] = {6, 9, 2, -5, 0};
#pragma memory = default
```

If another module accesses the array it must use an equivalent declaration:

```
#pragma memory=constseg(TABLE)
extern char arr[];
```

```
memory=dataseg #pragma memory=dataseg(seg_name)
```

Description

Directs variables to the named XDATA segment by default and can only be used as global, not local, inside a function. The default may be overridden by the memory attributes.

Note: The `seg_name` must not be one of the compiler's reserved segment names.

No initial value may be supplied in the variable definitions. Up to 10 different alternative data segments can be defined in any given module. You can switch to any previously defined XDATA data segment name at any point in the program.

Example

The example below places three variables into the read/write area called `USART`.

```
#pragma memory = dataseg(USART)
char USART_data;      /* offset 0 */
char USART_control;  /* offset 1 */
int USART_rate;      /* offset 2, 3 */
#pragma memory = default
```

If another module needs to access these symbols, the equivalent extern declaration should be used:

```
#pragma memory = dataseg(USART)
extern char USART_data;
```

```
memory=default #pragma memory=default
```

Description

Restores memory allocation of objects to the default area, as specified by the memory model in use.

See *memory=no_init*, below, for an example.

```
memory=no_init #pragma memory=no_init
```

Description

Directs variables to the NO_INIT segment so that they will not be initialized and will reside in non-volatile RAM. It is an alternative to the memory attribute *no_init*. The default may be overridden by the memory attributes.

The NO_INIT segment must be linked to coincide with the physical address of non-volatile RAM; see the *Configuration* chapter for details.

Example

The example below places the variable `buffer` into non-initialized memory. Variables `i` and `j` are placed into the DATA area.

```
#pragma memory=no_init
char buffer [1000];          /* in uninitialized memory */
#pragma memory=default
int i,j;                    /* default memory type */
```

Notice that a non-default #pragma memory directive will generate error messages if function declarators are encountered. Local variables and parameters cannot reside in any other segment than their default segment, the stack.

```
overlay=default #pragma overlay=default
```

Description

Leaves the decision on whether to overlay function parameters and local variables to the IAR XLINK Linker.

```
overlay=off #pragma overlay=off
```

Description

Turns off the overlaying of function parameters and local variables.

```
stringalloc=default #pragma stringalloc=default
```

Description

Allocates constant string declarations to the default memory area for the memory model currently in effect.

```
stringalloc=xdata #pragma stringalloc=default
```

Description

Allocates constant string declarations to XDATA in ROM memory.

```
warnings=default #pragma warnings=default
```

Description

Restores the compiler warning output to the default set with the `-w` compiler option. See *warnings=on*, and *warnings=off*.

```
warnings=off #pragma warnings=off
```

Description

Disables output of compiler warning messages regardless of the state of the `-w` compiler option; see page 104 for additional information.

```
warnings=on #pragma warnings=on
```

Description

Enables output of compiler warning messages regardless of the state of the `-w` compiler option; see page 104 for additional information.

Predefined symbols

With the predefined preprocessor symbols, you can inspect your compile-time environment, for example, the time and date of compilation. This chapter describes the syntax and gives a description of the predefined preprocessor symbols that are supported in the 8051 IAR C Compiler.

Descriptions of predefined symbols

The following section describes the available predefined symbols.

`__DATE__` `__DATE__`

Expands to the date of compilation in the form `Mmm dd yyyy`.

`__FILE__` `__FILE__`

Expands to the name of the file currently being compiled.

`__IAR_SYSTEMS_ICC__` `__IAR_SYSTEMS_ICC__`

Expands to a number that identifies the IAR Compiler platform. The current identifier is 1. Notice that the number could be higher in a future version of the product.

This symbol can be tested with `#ifdef` to detect that the code was compiled by an IAR Compiler.

`__LINE__` `__LINE__`

Expands to the current line number of the file currently being compiled.

`__STDC__` `__STDC__`

Expands to the number 1. This symbol can be tested with `#ifdef` to detect that the compiler used adheres to ANSI C.

`__TID__` `__TID__`

Target identifier.

The target identifier contains a number unique for each IAR Systems compiler (that is, it is a number unique for each target), the intrinsic flag, the value of the `-v` option, and the value corresponding to the `-m` option.

For the 8051 microcontroller, the target identifier is 14.

Assuming that these four values are named *f*, *t*, *v*, and *m*, the `__TID__` value for the 8051 IAR C Compiler is constructed as:

```
(0x8000 | (t << 8) | (v << 4) | m)
```

You can extract the values as follows:

```
f = (__TID__) & 0x8000;
t = (__TID__ >> 8) & 0x7F;
v = (__TID__ >> 4) & 0x0F;
m = __TID__ & 0x0F;
```

Notice that there are two underscores at each end of the macro name.

To find the value of the target identifier for the current compiler, execute:

```
printf("%ld", (__TID__>>8)&0x7F)
```

For an example of the use of `__TID__`, see the file `stdarg.h`.

The highest bit `0x8000`, is set in the 8051 IAR C Compiler to indicate that the compiler recognizes intrinsic functions. This may affect how you write header files.

```
__TIME__ __TIME__
```

Expands to the time of compilation in the form `hh:mm:ss`.

```
__VER__ __VER__
```

Expands to the version number of the compiler as an integer.

Example

The example below prints a message for version 3.34.

```
#if __VER__ == 334
#message "Compiler version 3.34"
#endif
```

Intrinsic functions

Intrinsic functions provide direct access to low-level processor operations and can be very useful in, for example, time-critical routines. The intrinsic functions compile into in-line code, either as a single instruction or as a short sequence of instructions.

This chapter gives reference information about the intrinsic functions that can be used in the 8051 IAR C Compiler.

Note: Make sure that you include the header file `special.h` if you use intrinsic functions in your application.

Descriptions of intrinsic functions

The following section describes the available intrinsic functions.

`_args$` `_args$`

Returns an array of the parameters to a function.

`_args$` is a reserved word that returns a char array (`char *`) containing a list of descriptions of the formal parameters of the current function:

Offset	Contents
0	Parameter 1 type in <code>_argt\$</code> format.
1	Parameter 1 size in bytes.
2	Parameter 2 type in <code>_argt\$</code> format.
3	Parameter 2 size in bytes.
$2n-2$	Parameter n type in <code>_argt\$</code> format.
$2n-1$	Parameter n size in bytes.
$2n$	<code>\0</code>

Table 29: `_args$` (intrinsic function)

Sizes greater than 127 are reported as 127.

`_args$` may be used only inside function definitions. For an example where `_args$` is used, see the file `stdarg.h`.

If a variable length (`varargs`) parameter list was specified, then the parameter list will terminate at the final explicit parameter; you cannot easily determine the types or sizes of the optional parameters.

`_argt$` `_argt$(variable name)`

Returns the type of the parameter.

The returned values and their corresponding meanings are shown in the following table:

Value	Type
1	unsigned char
2	char
3	unsigned short
4	short
5	unsigned int
6	int
7	unsigned long
8	long
9	float
10	double
11	long double
12	pointer/address
13	union
14	struct

Table 30: `_argt$` (intrinsic function)

Example

The following example uses `_argt$` and tests for int or long parameter types:

```
switch (_argt$(i))
{
    case 6:
        printf("int %d\n", i);
        break;
    case 8:
        printf("long %ld\n", i);
        break;
    default:
        printf("int or long expected\n");
        break;
}
```

`_opc` `_opc(c)`

Inserts an opcode.

The `_opc()` macro takes a single constant character `c` as a parameter; this is emitted by the compiler in the form of a DB assembler command. The intention of the macro is to create assembler opcodes for instructions difficult to describe in C.

To use this macro, include the line:

```
#include <special.h>
```

and select the `-e` option either from the command line or with the corresponding `#pragma language=extended`. For more information, see the *#pragma directives* chapter.

Example

The example below clears A and jumps to address 0x2000.

```
void stop_and_return_to_OS()
{
    _opc(0xE4);          /* CLR A */
    _opc(0x02);          /* LJMP 2000H */
    _opc(0x20);
    _opc(0x00);
}
```

`_tbac` `_tbac(b)`

Test Bit And Clear (atomic read, modify, write).

The `_tbac()` macro takes a single bit variable `b` and uses the JBC assembler instruction to carry out an atomic read, modify, write instruction. The macro returns the original value of `b` (0 or 1) and resets `b` to zero. This may be used to create semaphores or similar mutual-exclusion functions.

To use this macro, you must include the line:

```
#include <special.h>
```

and select the `-e` option either from the command line or with the corresponding `#pragma language=extended`.

K&R and ANSI C language definitions

This chapter describes the differences between the K&R description of the C language and the ANSI standard. It also summarizes the differences between the standards, and is particularly useful to programmers who are familiar with K&R C but would like to use the new ANSI facilities.

Introduction

There are two major standard C language definitions:

- Kernighan & Ritchie, commonly abbreviated to K&R.
This is the original definition by the authors of the C language, and is described in their book *The C Programming Language*.
- ANSI.

The ANSI definition is a development of the original K&R definition. It adds facilities that enhance portability and parameter checking, and removes a small number of redundant keywords. The IAR Systems C Compiler follows the ANSI approved standard X3.159-1989.

Both standards are described in depth in the later editions of *The C Programming Language* by Kernighan & Ritchie.

Definitions

This sections describes the C language definitions.

ENTRY KEYWORD

In ANSI C the `entry` keyword is removed, so allowing `entry` to be a user-defined symbol.

CONST KEYWORD

ANSI C adds `const`, an attribute indicating that a declared object is unmodifiable and hence may be compiled into a read-only memory segment. For example:

```
const int i;      /* constant int */
const int *ip;   /* variable pointer to
                 constant int */
```

```
int *const ip;   /* constant pointer to
                 variable int */
```

```

typedef struct          /* define the struct
                        'cmd_entry' */
{
    char *command;
    void (*function) (void);
}
cmd_entry
const cmd_entry table[]={/* declare a constant object of
                        type 'cmd_entry' */
{
    "help", do_help,
    "reset", do_reset,
    "quit", do_quit
};

```

VOLATILE KEYWORD

ANSI C adds `volatile`, an attribute indicating that the object may be modified by hardware and hence any access should not be removed by optimization.

SIGNED KEYWORD

ANSI C adds `signed`, an attribute indicating that an integer type is signed. It is the counterpart of `unsigned` and can be used before any integer type-specifier.

VOID KEYWORD

ANSI C adds `void`, a type-specifier that can be used to declare function return values, function parameters, and generic pointers. For example:

```

void f();          /* a function without return
                  value */
type_spec f(void);/* a function with no parameters */
void *p;          /* a generic pointer which can be
                  /* cast to any other pointer and
                  is assignment-compatible with any
                  pointer type */

```

ENUM KEYWORD

ANSI C adds `enum`, a keyword that conveniently defines successive named integer constants with successive values. For example:

```

enum {zero, one, two, step=6, seven, eight};

```

DATA TYPES

In ANSI C the complete set of basic data types is:

```
{unsigned | signed} char
{unsigned | signed} int
{unsigned | signed} short
{unsigned | signed} long
float
double
long double
*          /* Pointer */
```

FUNCTION DEFINITION PARAMETERS

In K&R C, function parameters are declared by conventional declaration statements before the body of the function. In ANSI C, each parameter in the parameter list is preceded by its type identifiers. For example:

K&R	ANSI
long int g(s)	long int g (char * s)
char * s;	
{	{

Table 31: Function differences between K&R and ANSI

The arguments of ANSI-type functions are always type-checked. The IAR Systems C compiler checks the arguments of K&R-type functions only if the **Global strict type checking** (-g) option is used.

FUNCTION DECLARATIONS

In K&R C, function declarations do not include parameters. In ANSI C they do. For example:

Type	Example
K&R	extern int f();
ANSI (named form)	extern int(long int val);
ANSI (unnamed form)	extern int(long int);

Table 32: K&R and ANSI function declarations

In the K&R case, a call to the function via the declaration cannot have its parameter types checked, and if there is a parameter-type mismatch, the call will fail.

In the ANSI C case, the types of function arguments are checked against those of the parameters in the declaration. If necessary, a parameter of a function call is cast to the type of the parameter in the declaration, in the same way as an argument to an assignment operator might be. Parameter names are optional in the declaration.

ANSI also specifies that to denote a variable number of arguments, an ellipsis (three dots) is included as a final formal parameter.

If external or forward references to ANSI-type functions are used, a function declaration should appear before the call. It is unsafe to mix ANSI and K&R type declarations since they are not compatible for promoted parameters (`char` or `float`).

Notice that in the IAR Systems Compiler, the **Global strict type checking** (`-g`) option will find all compatibility problems among function calls and declarations, including between modules.

HEXADECIMAL STRING CONSTANTS

ANSI allows hexadecimal constants denoted by backslash followed by `x` and any number of hexadecimal digits. For example:

```
#define Escape_C "\x1b\x43" /* Escape 'C' \0 */
```

`\x43` represents ASCII C which, if included directly, would be interpreted as part of the hexadecimal constant.

STRUCTURE AND UNION ASSIGNMENTS

In K&R C, functions and the assignment operator may have arguments that are pointers to struct or union objects, but not struct or union objects themselves.

ANSI C allows functions and the assignment operator to have arguments that are struct or union objects, or pointers to them. Functions may also return structures or unions:

```
struct s a,b;          /* struct s declared earlier
                       */
struct s f(struct s parm); /* declare function accepting
                           and returning structs */
a = f(b);             /* call it */
```

To increase the usability of structures further, ANSI allows auto structures to be initialized.

SHARED VARIABLE OBJECTS

Various C compilers differ in their handling of variable objects shared among modules. The IAR Systems C compiler uses the scheme called *Strict REF/DEF*, recommended in the ANSI supplementary document *Rationale For C*. It requires that all modules except one use the keyword `extern` before the variable declaration. For example:

Module #1	Module #2	Module #3
<code>int i;</code>	<code>extern int i;</code>	<code>extern int i;</code>
<code>int j=4;</code>	<code>extern int j;</code>	<code>extern int j;</code>

Table 33: Shared variable objects

#elif

ANSI C's new `#elif` directive allows more compact nested else-if structures.

```
#elif expression
...
```

is equivalent to:

```
#else
#if expression
...
#endif
```

#error

The `#error` directive is provided for use in conjunction with conditional compilation. When the `#error` directive is found, the compiler issues an error message and terminates.

Using C with PL/M

This chapter describes how to convert files produced by the Intel PL/M compiler in OMF format to the UBROF format used by the IAR XLINK Linker.

The conversion is performed by the `omfconv` object file converter. This will convert everything in the OMF format with the exception of line numbers, function-block and block information with their information of function and block locals. In effect, this limits the level of debug to assembler with public symbols.

Using the object file converter

The usage is:

```
omfconv OMF-file UBROF-file [list-file]
```

The converter lists, either on the screen or in `list-file`, the modules it converts and its segments (relocatable as well as absolute ones).

For assembler object files the relocatable segment information will be in the format:

```
Segment name of type type alignment align
```

For PL/M-51 object files the relocatable segment information will be in the format:

```
Segment name of type type alignment align mapped to name2
```

In each case the parameters are as follows:

<i>name</i>	The OMF type of segment name.
<i>type</i>	One of: CODE, XDATA, DATA, IDATA, BIT, or unknown which is treated by XLINK as UNTYPED.
<i>align</i>	One of the following:
UNIT	Allocate anywhere
BITADDRESS	Allocate in bit memory at start of byte unit
INPAGE	Allocate in a 256 byte block
INBLOCK	Allocate in a 2048 byte block
PAGE	Allocate at the start of a 256 byte block
unknown	Will be of UNIT alignment to XLINK

<i>name2</i>	One of the following:
PLM_CODE	Code
PLM_CONST	Constants
PLM_XDATA	Auxiliary memory
PLM_BIT	BIT variables
PLM_DATA	Variables
PLM_IDATA	IDATA variables
PLM_BITDATA	Bit-addressable data memory
PLM_DATA_OV	Locals and parameters
PLM_IDATA_OV	IDATA locals and parameters
PLM_BIT_OV	BIT locals and parameters
PLM_BITDATA_OV	Locals in bit-addressable data memory

The converter will map PL/M-51 type segments to a *name2* segment.

Linking the converter files

The IAR XLINK Linker will overlay the PL/M-51 overlayable segments, but there is no way to tell the XLINK Linker that a module calls another module indirectly. To overcome this, do a dummy call from the calling module to the called module. When you convert a PL/M-51 or Intel assembler system, remember to convert the libraries and include them when linking with C. Make sure that there is only one start-up routine and one reset vector to that routine.

Use the `lnkplm.xcl` linker command file when linking.

For assembler type of segments, add the segment name to the segment list with the segment type in the `lnkplm.xcl` file.

For assembler and PL/M-51 type of segments, if the alignment is `INPAGE` or `INBLOCK`, make sure that the segment part is contained in a 256 or 2048 byte block of memory. Do it with the following command for `INPAGE` segments:

```
-Z (type) name, name, ...=0-FF,100-1FF,200-2FF,...
```

For `INBLOCK` segments, use

```
-Z (type) name, name, ...=0-7FF,800-FFF,1000-17FF,....
```

If the alignment is `PAGE` you must put the segment to start at a 256 byte block. Use the following command in the `LNKPLM.XCL` file:

```
-Z (type) name=256byteblock
```

Compiling PL/M functions

Note that the PL/M-51 system only uses upper-case characters in symbols, therefore define or declare all your C symbols that you want to use in or get from PL/M-51 in uppercase characters.

To be able to call PL/M-51 routines or let PLM call C functions there is a keyword `plm` in the 8051 C Compiler. You must either use the compiler `-e` option, or set:

```
#pragma language=extended
```

in your source code.

Example

```
extern plm int F1(data char PARM);/* A PLM routine */
plm void F2(data char PARM)/* A PLM callable function */
{
}
```

An option to define or declare is to use the `#pragma function` directive:

```
#pragma function=plm
def/DECL1
def/DECL2
...
def/DECLN
#pragma function=default
```

Note that PL/M-51 functions must be prototyped. The following examples are incorrect:

```
int plm F();/* ERROR void parameter missing */
int plm F(I);/* ERROR Old-style not allowed */
int I;
{
}
```

Memory attribute (storage) for PL/M-51 function arguments is `data` but can be overridden. Note that the PL/M-51 only stores parameters in `data` or `bit` storage.

The usable function return values and function parameters for `plm` functions are `void`, `char`, `short`, and `int` that will be mapped to PL/M-51 `none`, `byte`, and `word`.

Static and global variables with the following types will be matched:

Type	Matched to
char	byte
short	word
int	word
array of the above	array of the above
struct with the above	struct with the above

Table 34: Matched static and global variables from C to PL/M

These types will not automatically be matched from C to PL/M-51:

```
pointer
long
float
double
long double
```

To map C pointers to PL/M-51 pointers and vice-versa use the macros in the include file `PLM.H`.

`PLM_to_C_p(mem, adr)` will convert a PL/M-51 two or one byte pointer to the C three-byte pointer.

`C_to_PLM_byte_p(p)` will convert a C three-byte pointer into a PLM one-byte pointer.

`C_to_PLM_word_p(p)` will convert a C three-byte pointer into a PLM two-byte pointer.

`C_to_PLM_memory(p)` will get the memory type from a C three-byte pointer.

You can also map the C three-byte pointer to the following PL/M-51 structure:

```
declare pointer structure (memory_type byte, address word);
```

And to use the pointer do:

```
declare idata_p based pointer.address byte idata;
declare xdata_p based pointer.address byte auxiliary;
declare const_p based pointer.address byte constant;
do case pointer.memory_type;
byte = idata_p;
byte = xdata_p;
byte = constant;
end;
```

To try out the `omfconv` converter, you can use the following files that are provided with the product:

```
C00.C  
C01.C  
C02.C  
C03.C  
P01.P  
P02.P
```


Tiny-51

TINY-51 is a multi-tasking real-time kernel specially designed for embedded system applications built around processors from the 8051-family. When using TINY-51 in your system, you leave all the job switching and other functions to the kernel. This allows you to concentrate on your application and helps cut development time.

This chapter explains the use of TINY-51 together with the 8051 IAR C Compiler, the IAR XLINK Linker, and other tools from IAR Systems.

Introduction

TINY-51 is basically a library of functions that handle the execution and co-operation between parallel tasks. As the name implies, the set of functions, or possible system calls, is limited. However, TINY-51 can easily be extended and modified to meet with user demands.

GENERAL CHARACTERISTICS

TINY-51 is a small multitasking kernel for 80x51 single chip processors. It is also possible to use this kernel with a 8031 processor with only 128 Bytes user RAM and without external RAM (XDATA). But in such a configuration only a few tasks can be installed and special task functions can be used with restriction only. For an efficient execution of the system the processor should have 256 Bytes of internal RAM and XDATA should be used for stack relocation.

The task switch time can be configured by changing a constant in the include files `tiny51.h` and `tiny51.i` as described in the following chapter. The internal timer 0 is used for task switching but another timer can be used as well. Register bank 3 is used because a special function in the task structure to avoid memory restrictions.

TERMINOLOGY

This section briefly describes some of the concepts of TINY-51. The terminology might differ slightly from those of other operating systems or real-time kernels, but most of the principles are the same for all of them.

Task

A task is a program entity that can be logically executed in parallel with and independently of other tasks. Different tasks can communicate with each other and also synchronize their execution. In a single processor system the different tasks have to share the processor that is allocated to different tasks by a scheduler.

Idle-task

The idle-task is a special task that is executed when no other task is waiting for execution. The processor is said to be in idle state.

Task states

Every task can be in one of five different states:

State	Description
Running	Task is currently executing.
Ready	Task is ready to execute and is waiting in task list.
Waiting	Task is waiting for communication with other tasks.
Stopped	Task is not executing and is taken off the task list.
New	Task has just been added to task list.

Table 35: Task states in TINY-51

Dispatcher

The dispatcher is the part of a multitasking system that interrupts the executing task and starts the execution of the next waiting task. The task to activate is chosen by the scheduler.

Scheduler

The scheduler is the part of a multitasking system that decides in which order to run the tasks. There are many different scheduling algorithms and the performance of the system is very much depending on the choice of the scheduling algorithm.

Round-robin

Round-robin is a scheduling algorithm where every task in the ready list is executed in strict order as they are listed. The amount of time for each executing task is limited. When the time expires for the executing task, it is suspended and inserted at the end of the list of ready tasks.

Preemptive multitasking

In preemptive multitasking the dispatcher activates and terminates tasks. The dispatcher is responsible for distributing processor time to all the tasks.

Non-preemptive multitasking

With non-preemptive multitasking the executing task is responsible for its own execution and it must give up the processor when it has finished its work.

Signals

By use of signals tasks can easily synchronize with other tasks.

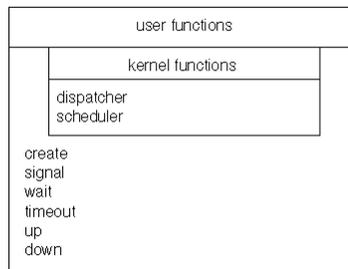
Semaphores

Semaphores are used to avoid multiple access to critical sections.

PRINCIPLES OF OPERATION

TINY-51 consists of two parts, the kernel functions and the user functions. The kernel functions are written in assembly language. There are two kernel functions, the dispatcher and the scheduler.

The user functions are written in C and they are therefore portable. These functions allow the user to do primitive multitasking functions like inserting a task in a task list and doing process communication.



Every task has a structure that describes the task to the kernel and to user functions. These structures are put in the task list.

The elements of the task structure are described as follows:

```

struct TASK {
    struct TASK *nextptr
    byte pid;
    byte wait_signals;
    byte rec_signals;
    byte timeout;
    byte state;
    byte *sp;
    void (*pushfunc) ();
    void (*popfunc) ();
};
  
```

nextptr

Points to the next task. If there is no other task in the task list this variable is set to zero.

pid

A unique number for every task. Value 0 is reserved for `TimeoutTask` and value 255 is used for internal purpose. The user tasks can be numbered from 1 to 254. User function `create` checks if `pid` number is unique.

wait_signals

A bit mask showing which signal the task is waiting for. The `task` can also wait for more than one signal but one signal is enough for waking up the task.

rec_signals

A bit mask to show which received signal is responsible for waking up this task.

timeout

The internal timeout counter for this task. This counter is decreased by `TimeoutTask` and set from user function `timeout`. If `timeout` reaches the value zero `TimeoutTask` sends a signal to this task.

state

A variable for internal use. The `state` is used by the scheduler to choose the next task from the list.

sp

Used to store the actual task stack pointer when the task is dispatched.

pushfunc

A special function for the 8051 kernel. This function is called by the dispatcher before dispatching task. It can be used for saving a functions-local memory. This is necessary because local variables in 8051 C-functions are stored statically.

popfunc

A special function for 8051 kernel. This function is called after suspending the old `task`. It is used together with the function `pushfunc`. Great attention has to be paid to these two functions because the complete multitasking environment could be inconsistent if they are not used properly.

RESTRICTIONS ON TINY-51

Timer 0 is used by the dispatcher. This can be modified in the source file `d_iar51.asm`.

Register bank 3 is used by the dispatcher. This register bank is only used to handle the special task functions `pushfunc` and `popfunc`. This can be modified in the source files `d_iar51.asm`.

The internal stack is limited to 10 bytes. This stack will be used by `pushfunc` and `popfunc` so only simple functions should be used for them. The stack size can be modified by changing the value `STACKSIZE` in the header file `tiny51.h`.

Installing TINY-51

TINY-51 is delivered in source format and can easily be modified by the user. It consists of the following files:

```
tiny51.c
tiny51.h
tiny51.i
d_iar51.asm
maketiny.bat
tiny51.xcl
exmpl1.c to exmpl6.c
```

In `tiny51.c` you can find the user function of TINY-51. `tiny51.h` contains all declarations of global constants used in `tiny51.c`. All functions in `tiny51.c` are described with comments.

Kernel functions for 8051 are described in `d_iar51.asm`. Comments describe the dispatcher and the scheduler.

All files should be copied into a work directory. After modifying any source file you have to start the batch file `maketiny.bat` to make a new kernel library. This library can be added to your application with the IAR XLINK Linker.

Note: The `expm1.c` and `expm2.c` files are the only examples that run in the C-SPY simulator. The other examples need the timer function which is not supported in the C-SPY simulator.

Configuring TINY-51

Many parameters of TINY-51 can be changed for in order to optimize the kernel for special applications.

TASK TIMER

Timer 0 of 80C31 is used for dispatching tasks. This timer can be changed to any other timer. The initial part of `tiny51.c` and the interrupt vector in `d_iar51.asm` have to be changed for use of a different timer.

REGISTER BANK 3

The dispatcher uses this register bank to avoid conflicts with the special functions `pushfunc` and `popfunc`. Those two functions can call library functions which use of the default register bank. To attain faster-task switching the dispatcher changes to register bank 3 instead of doing 8 times `PUSH` and 8 times `POP`. There are two possibilities to avoid using register bank 3:

- 1 Saving register bank 1 in reserved area in dispatcher.
- 2 Avoiding the use of special functions and deleting register switching in dispatcher source.

TASK-SWITCHING TIME

The task-switching time can be controlled by the constant `DISPATCH_DELAY` in `tiny51.h` and `tiny51.i`. If this value is increased, the dispatcher overhead is decreased.

TIMEOUT TASK

If the user function `timeout` is not used in an application this task can be removed by changing function `StartDispatcher` in `tiny51.c`. No time is then wasted for calling an unused task.

After modifying any source file it is necessary to recompile all the source files.

Building a TINY-51 application

This section describes a step-by-step introduction to building an application for TINY-51.

USING PREEMPTIVE MULTITASKING

In this example you will build a simple task environment with two functions running in parallel. This first example runs in preemptive mode, with `printf` controlled by a semaphore.

This example is supplied in source code on the distribution disk as the file `exmpl6.c`. This file can be compiled with `ICC8051` and linked with `XLINK`.

```

/* preemptive example for TINY51 */

#include <stdio.h>

#include "tiny51.h"

#define TASK_STACKSIZE 30
typedef struct stack
{
    byte vect[TASK_STACKSIZE];
};

void task1a(void);
void task2a(void);

void PushFunction1(void);
void PopFunction1(void);
void PushFunction2(void);
void PopFunction2(void);

#pragma memory=idata
struct stack taskstack;

#pragma memory=xdata
struct stack taskstack1;
struct stack taskstack2;
byte regstack1[8];
byte regstack2[8];

#pragma memory=idata
int counter1, counter2;

static semaphore s1 = 1;

#pragma memory=default

/* main function is idle task !!! */
void main(void);

```

```

void main()
{
    int ch;
    #pragma memory=xdata
        struct TASK task1, task2;
    #pragma memory=default

    /* initialize task structure */
    task1.pid = 1;
    task1.sp = &taskstack.vect[0];
    task1.pushfunc = PushFunction1;
    task1.popfunc = PopFunction1;
    create(&task1, task1a);

    task2.pid = 2;
    task2.sp = &taskstack.vect[0];
    task2.pushfunc = PushFunction2;
    task2.popfunc = PopFunction2;
    create(&task2, task2a);

    counter1 = 0;
    counter2 = 0;

    StartDispatcher(TINY51_PREEMPTIVE);

    do
    {
        down(&s1);
        printf("counter1: %d  counter2:
%d\n", counter1, counter2);
        ch = FALSE;
        up(&s1);
    }
    while (!ch);

    StopDispatcher(TINY51_PREEMPTIVE);
}

void task1a()
{
    for (;;)
    {
        down(&s1);
        printf("Task1\n");
        counter1++;
        up(&s1);
    }
}

```

```

}

void task2a()
{
    for (;;)
    {
        down(&s1);
        printf("Task2\n");
        counter2++;
        up(&s1);
    }
}

void PushFunction1()
{
    StoreRegs(regstack1);
    taskstack1=taskstack;
}

void PopFunction1()
{
    RestoreRegs(regstack1);
    taskstack=taskstack1;
}

void PushFunction2()
{
    StoreRegs(regstack2);
    taskstack2=taskstack;
}

void PopFunction2()
{
    RestoreRegs(regstack2);
    taskstack=taskstack2;
}

```

This example shows how two tasks can use the function `printf` without collision. As the library function `printf` is not reentrant the access to this function is controlled by a semaphore. The semaphore variable is declared static and is global so all the tasks in the file can access it.

In the example one global stack declared in internal memory is used for both tasks. The idle task, the main function, uses the stack defined in the startup file `cstartup.s03`. This method saves memory in the small internal RAM. This common stack has to be swapped to external RAM on every task switch to avoid data collision on stack.

The stack switch is activated by a special function pointer in the task structure called `pushfunc` and `popfunc`. In this case these two functions used are `PushFunction1` and `PopFunction1` for `task1` and `PushFunction2` and `PopFunction2` for `task2`. These functions are similar, except for the swap area. Therefore, only `PushFunction1` and `PopFunction1` are explained.

Push function

`PushFunction1` stores the internal register bank 1 in `regstack1`. If storing register bank 1 is not sufficient more register banks can be stored by modifying function `StoreRegs` in `d_iar51.asm`. After storing the registers the function saves the internal stack to an external memory area. `PushFunction1` is called before dispatching the task. `PushFunction1` has to be declared in the task structure before starting the multitasking kernel.

Pop function

`PopFunction1` does the reverse operation of `PushFunction1`. The function register bank and local stack are restored from memory. This function is called after dispatching the new task and so gets a new stack environment. It also has to be declared in the task structure.

In the main function the task structure variables are put in external memory to save internal RAM space. TINY-51 can handle task structures either in internal or in external memory and in a mix of the two as well. There are no restrictions in the use of memory models.

Creating tasks

Before starting the multitasking kernel all tasks have to be created by the user function `create`. The function `create` is called with two parameters. The first parameter is a predefined task structure variable and the second parameter is the task function itself. The task structure variable has to be defined with a few parameters. Field `pid` has to be assigned an unique byte number. Field `sp` is loaded with the address of the top of stack.

As the 8051 uses stack in a different way than many other processors, the stack pointer does not point to the end of the stack area. For extended use of the multitasking environment in the 8051 the special functions `pushfunc` and `popfunc` have to be defined with their function address. All other fields are filled by the function `create`. The initial stack environment is built by this function, too.

Start dispatcher

After defining all the tasks, the function `StartDispatcher` is called. It needs one parameter only, which defines the mode of execution. TINY-51 can be started either in `NON-PREEMPTIVE` or in `PREEMPTIVE` mode. The differences between these two modes are explained in *Introduction*, page 151.

When running the system, four functions are dispatched every task timer tick. These four functions are the main loop, `task1`, `task2`, and `TimeoutTask`.

Note: It is very important that the task functions do not make a return because there is no return address stored on the stack.

Function `main` runs in an endless loop, printing out the values of two counter variables. `Task1` and `task2` also run forever printing their messages and increasing their counter variables.

USING NON-PREEMPTIVE MULTITASKING

The following example has the same functionality as the first example, but instead of using preemptive multitasking it uses non-preemptive multitasking. Using this method every task has to give up CPU by itself.

This example is supplied in source code on the distribution media as the file `exmpl1.c`. There are several other examples in the files `exmpl2.c` to `exmpl5.c`.

```
/* non-preemptive example for TINY51 */

#include <stdio.h>
#include "tiny51.h"

#define TASK_STACKSIZE20
typedef struct stack
{
    byte vect[TASK_STACKSIZE];
};

void task1a(void);
void task2a(void);

void PushFunction1(void);
void PopFunction1(void);
void PushFunction2(void);
void PopFunction2(void);

#pragma memory=idata
struct stack taskstack;
```

```
#pragma memory=xdata
struct stack taskstack1;
struct stack taskstack2;
byte regstack1[8];
byte regstack2[8];

#pragma memory=idata
int counter1, counter2;

static semaphore s1 = 1;

#pragma memory=default

/* main function is idle task !!! */

void main(void);

void main(void)
{
    int ch;
    #pragma memory=xdata
    struct TASK task1, task2;
    #pragma memory=default

    /* initialize task structure */

    task1.pid = 1;

    task1.sp = &taskstack.vect[0];
    task1.pushfunc = PushFunction1;
    task1.popfunc = PopFunction1;

    create(&task1, task1a);

    task2.pid = 2;

    task2.sp = &taskstack.vect[0];
    task2.pushfunc = PushFunction2;
    task2.popfunc = PopFunction2;

    create(&task2, task2a);

    counter1 = 0;
    counter2 = 0;

    StartDispatcher(TINY51_NONPREEMPTIVE);
```

```

do
{
    printf("%d %d\n", counter1, counter2);
    ch = FALSE;
    wait(SIG_CPU);
}
while (!ch);

StopDispatcher(TINY51_NONPREEMPTIVE);
}

void task1a(void)
{
    for (;;)
    {
        printf("Task1\n");
        counter1++;
        wait(SIG_CPU);
    }
}

void task2a(void)
{
    for (;;)
    {
        printf("Task2\n");
        counter2++;
        wait(SIG_CPU);
    }
}

void PushFunction1(void)
{
    StoreRegs(regstack1);
    taskstack1=taskstack;
}

void PopFunction1(void)
{
    RestoreRegs(regstack1);
    taskstack=taskstack1;
}

void PushFunction2(void)
{
    StoreRegs(regstack2);
    taskstack2=taskstack;
}

```

```

    }

void PopFunction2(void)
{
    RestoreRegs (regstack2);
    taskstack=taskstack2;
}
    
```

In this example every task, after printing a message, calls the function `wait`, waiting for signal `SIG_CPU`. This means that the next task is scheduled. Normally if a task calls `wait` it waits for the signal of another task. It is waiting for the next scheduling. Using this kind of scheduling a control method with semaphores is not necessary, because the task is not interrupted by a dispatcher and can therefore complete its critical section without disturbance. In this case the resource `printf` does not have to be controlled by the semaphore control functions `up` and `down`.

The `NON-PREEMPTIVE` mode is initiated by calling `StartDispatcher` with the parameter `NONPREEMPTIVE`.

Descriptions of TINY-51 functions

The following table summarizes the TINY-51 functions:

Function	Summary
<code>create</code>	Inserts a task in the task list.
<code>down</code>	Semaphore operation to enter critical section.
<code>signal</code>	Sends a signal to a task.
<code>StartDispatcher</code>	Initializes dispatcher environment.
<code>StopDispatcher</code>	Resets dispatcher environment.
<code>timeout</code>	Special task, generates timeout signals.
<code>up</code>	Semaphore operation to leave critical section.
<code>wait</code>	Waits for a signal.

Table 36: TINY-51 functions summary

```
create tiny51.h
```

Declaration

```
int create (struct *task, void (*taskfunc))
```

Parameters

<i>task</i>	A pointer to a TASK structure with the following format: <pre> struct TASK { struct TASK *nextptr; /* pointer to next task */ byte pid; /* task number */ byte wait_signals; /* compare for waiting signal */ byte rec_signals; /* received signals */ byte timeout; /* Timeout counter */ byte state; /* task state */ byte *sp; /* task stack */ void (pushfunc)(); /*store function for additional memory */ void (popfunc)(); /* restore function for additional memory */ </pre>
<i>taskfunc</i>	A pointer to the task function.

Description

The `create` function inserts a task in the ready-list and builds a stack environment for this task. Before calling the function `create`, the task structure has to be initialized.

Return value

Function returns -1 if task cannot be inserted in task list.

Example

The following example creates a single process:

```

#include <tiny51.h>
struct TASK task1;
void task1_func(void);
byte taskstack[10];
int counter;
main()
{
    task1.pid = 1;
    task1.sp = (stacktype *)taskstack;
    task1.pushfunc = NULL;
    task1.popfunc = NULL;
    if (create(&task1, task1_func))
        printf("can't create task");
}

```

```

}
void task1_func()
{
    for (;;) {
        counter++;
    }
}

```

The following example creates a single process with pushfunc and popfunc:

```

#include <tiny51.h>
struct TASK task1;
void task1_func(void);
byte taskstack[10];
int counter, counter1;
main()
{
    task1.pid = 1;
    task1.sp = (stacktype *)taskstack;
    task1.pushfunc = pushregs;
    task1.popfunc = popregs;
    if (create(&task1, task1_func))
        printf("can't create task");
}
void task1_func()
{
    for (;;) {
        counter++;
    }
}
void pushregs()
{
    counter1 = counter;
}
void popregs()
{
    counter = counter1;
}

```

down tiny51.h

Declaration

```
void down(semaphore s)
```

Parameters

s Boolean semaphore variable.

Description

The function down-decrements a semaphore if possible. This function can be used for sharing resources. If the semaphore is set to 1 a critical section can be entered otherwise the task will wait for a semaphore.

Note: The function pair up/down can handle only one semaphore variable.

Return value

This function returns no value.

Example

The following example shows how to use a semaphore:

```
#include <tiny51.h>
semaphore s1;
void task3()
{
    down(&s1);
    printf("hello, I'm task 3");
    up(&s1);
}
```

signal tiny51.h

Declaration

```
int signal (byte tasknr, word signalnr)
```

Parameters

tasknr The task number signal to send.

signalnr The signal number; one of the following;

- SIG_TIMEOUT*, timeout signal.
- SIG_SEMAPHORE*, semaphore signal.
- SIG_KEYBOARD*, keyboard signal.
- SIG_CPU*, give up CPU.

Description

The `signal` function sends a signal to the task with `tasknr`. If the receiver task is not waiting for a signal the function returns an error and the signal is lost. The function can also send combined signals.

Every signal can be user defined. `SIG_TIMEOUT` and `SIG_SEMAPHORE` are predefined and should not be redefined. `SIG_CPU` can only be sent by the scheduler. It is not allowed to use `SIG_CPU` in function `signal`.

`PID_NEXT` is a special task number that directs a signal to the next task that is waiting for a the specified signal.

Return value

This function returns -1 if the receiver task is not waiting for a signal.

Example

The following example shows how to send a signal to another process:

```
#include <tiny51.h>
#define SIG_NEXTONE 0x10
task2 ()
{
    if (signal(1,SIG_NEXTONE))
        printf("receiver task didn't accept signal");
}
```

StartDispatcher tiny51.h

Declaration

```
void StartDispatcher(int mode)
```

Parameters

mode An integer defining the dispatcher preemptive mode.

Description

The `StartDispatcher` function initializes the environment of the dispatcher. It initializes interrupts and timer hardware of the processor.

This function should be called after initializing all tasks with function `create`. The parameter `mode` defines the dispatcher operating modes. There are two modes running the dispatcher.

Mode `TINY51_PREEMPTIVE`

In this mode the function prepares the interrupt and timer hardware of the processor to switch tasks asynchronously. This way every task will get CPU time (Round Robin Scheduling).

Mode `TINY51_NONPREEMPTIVE`

In this mode the function only prepares the software environment for the dispatcher. One task can only get CPU time when another task gives the CPU up. The task gives up CPU by waiting for signal `SIG_CPU`.

Return value

This function returns no value.

Example

The following example shows how to start the dispatcher:

```
#include <tiny51.h>
main()
{
    StartDispatcher(TINY_NONPREEMPTIVE);
}
```

StopDispatcher tiny51.h

Declaration

```
void StopDispatcher()
```

Description

The `StopDispatcher` function resets the environment of the dispatcher and stops the dispatcher.

Return value

This function returns no value.

Example

The following example shows how to stop the dispatcher:

```
#include <tiny51.h>
main()
{
```

```
    StopDispatcher();
}
```

timeout tiny51.h

Declaration

```
void timeout(int timeoutval)
```

Parameters

timeoutval The timeout value.

Description

The `timeout` function generates a special signal. This will be sent when the time-out counter reaches the *timeoutval*.

Return value

This function returns no value.

Example

The following example shows how to use the `timeout` function:

```
#include <tiny51.h>
void task3()
{
    timeout(100);
    if (wait(SIG_USER | SIG_TIMEOUT) & SIG_TIMEOUT)
        /* Timeout occurred */
    else
        /* SIG_USER arrived */
}
```

up tiny51.h

Declaration

```
void up(semaphore s)
```

Parameters

s Boolean semaphore variable.

Description

The function `up` is called when leaving a critical section. This function increments the semaphore variable and sends a signal to a task waiting for the semaphore variable.

Return value

This function returns no value.

Example

The following example shows how to use a semaphore:

```
#include <tiny51.h>
semaphore s1;
void task3()
{
    down(&s1);
    printf("hello, I'm task3");
    up(&s1);
}

void task2()
{
    down(&s1);
    printf("hello, I'm task2");
    up(&s1);
}
```

wait tiny51.h

Declaration

```
int wait(word signalnr)
```

Parameters

signalnr Signal waiting for.

Description

The `wait` function transfers the task into the wait state. This task can return to ready state only when a signal is received from another task. It is also possible to wait for more than one signal. The first signal arrived turns the task into ready-state. The `wait` function returns the arrived signal.

SIG_CPU is a special signal for giving up CPU. This signal is important for a non-preemptive multitasking system but can also be used in a preemptive environment.

Return value

This function returns the signal that reactivated the task.

Example

The following example shows how to wait for a signal:

```
#include <tiny51.h>
void task1()
{
    word signals;
    timeout(5);
    signals = wait(SIG_TIMEOUT * SIG_NEXTONE);
    if (signals & SIG_TIMEOUT)
        printf("Error : Timeout");
}
```

Diagnostics

When the 8051 IAR C Compiler performs a diagnostic check, it may detect errors in your application and give a remark, warning, or error message. This chapter explains the different levels of severity for the diagnostic messages and gives a brief explanation of the 8051-specific warning and error messages.

Severity levels

The diagnostic error and warning messages fall into these categories:

COMMAND LINE ERROR MESSAGES

Command line errors occur when the compiler finds a fault in the parameters given on the command line. In this case, the compiler issues a self-explanatory message.

COMPILATION ERROR MESSAGES

Compilation error messages are produced when the compiler has found a construct which clearly violates the C language rules, such that code cannot be produced.

The IAR C compiler is more strict on compatibility issues than many other C compilers. In particular, pointers and integers are considered incompatible when not explicitly cast.

COMPILATION WARNING MESSAGES

Compilation warning messages are produced when the compiler finds a programming error or omission which is of concern, but is not so severe as to prevent the completion of compilation.

COMPILATION FATAL ERROR MESSAGES

Compilation fatal error messages are produced when the compiler has found a condition that not only prevents code generation, but which makes further processing of the source not meaningful. After the message has been issued, compilation terminates. Compilation fatal error messages are described in *Compilation error messages*, page 174, and are marked as fatal.

COMPILATION INTERNAL ERROR MESSAGES

An internal error is a diagnostic message that indicates a serious and unexpected failure due to a fault in the compiler. It is displayed as:

```
Internal error: message
```

where *message* is an explanatory message.

Internal errors should not occur and should be reported to your software distributor or IAR Technical Support. Please include detailed information to reproduce the problem. This would typically include:

- The exact internal error message text
- The source file of the program that generated the internal error
- A list of the options that were used when the internal error occurred.

COMPILATION MEMORY OVERFLOW MESSAGE

When the compiler runs out of memory, it issues the special message:

```
* * * C O M P I L E R   O U T   O F   M E M O R Y * * *
      Dynamic memory used: nnnnnn bytes
```

If this error occurs, add system memory or split source files into smaller modules. Also note that the following options cause the compiler to use more memory:

Option	Command line
insert mnemonics	-q
Cross-reference	-x
Assembly output to prefixed filename	-A
Generate PROMable code	-P
Generate debug information	-r (except -rn)

Table 37: Options that cause the compiler to use more memory

See the *Compiler options* chapter for more information.

Compilation error messages

The following list describes the compilation error messages:

- 0 Invalid syntax**
The compiler could not decode the statement or declaration.
- 1 Too deep #include nesting (max is 10)**
Fatal. The compiler limit for nesting of #include files was exceeded. One possible cause is an inadvertently recursive #include file.
- 2 Failed to open #include file name**
Fatal. The compiler could not open an #include file. Possible causes are that the file does not exist in the specified directories (possibly due to an incorrect -I option or C_INCLUDE path) or is disabled for reading.

- 3 Invalid #include filename**
Fatal. The `#include` filename was invalid. Notice that the `#include` filename must be written `<file>` or `"file"`.
- 4 Unexpected end of file encountered**
Fatal. The end of file was encountered within a declaration, function definition, or during macro expansion. The probable cause is bad `()` or `{ }` nesting.
- 5 Too long source line (max is 512 chars);truncated**
The source line length exceeds the compiler limit.
- 6 Hexadecimal constant without digits**
The prefix `0x` or `0X` of hexadecimal constant was found without following hexadecimal digits.
- 7 Character constant larger than long**
A character constant contained too many characters to fit in the space of a `long` integer.
- 8 Invalid character encountered: '\xhh';ignored**
A character not included in the C character set was found.
- 9 Invalid floating point constant**
A floating-point constant was found to be too large or have invalid syntax. See the ANSI standard for legal forms.
- 10 Invalid digits in octal constant**
The compiler found a non-octal digit in an octal constant. Valid octal digits are: 0–7.
- 11 Missing delimiter in literal or character constant**
No closing delimiter `'` or `"` was found in character or literal constant.
- 12 String too long (max is 509)**
The limit for the length of a single or concatenated string was exceeded.
- 13 Argument to #define too long (max is 512)**
Lines terminated by `\` resulted in a `#define` line that was too long.
- 14 Too many formal parameters for #define (max is 127)**
Fatal. Too many formal parameters were found in a macro definition (`#define` directive).
- 15 ', ' or ')' expected**
The compiler found an invalid syntax of a function definition header or macro definition.

- 16 Identifier expected**
An identifier was missing from a declarator, goto statement, or pre-processor line.
- 17 Space or tab expected**
Pre-processor arguments must be separated from the directive with tab or space characters.
- 18 Macro parameter name redefined**
The formal parameter of a symbol in a #define statement was repeated.
- 19 Unmatched #else, #endif or #elif**
Fatal. A #if, #ifdef, or #ifndef was missing.
- 20 No such pre-processor command: name**
was followed by an unknown identifier.
- 21 Unexpected token found in pre-processor line**
A preprocessor line was not empty when the argument part was read.
- 22 Too many nested parameterized macros (max is 50)**
Fatal. The pre-processor limit was exceeded.
- 23 Too many active macro parameters (max is 256)**
Fatal. The pre-processor limit was exceeded.
- 24 Too deep macro nesting (max is 100)**
Fatal. The pre-processor limit was exceeded.
- 25 Macro name called with too many parameters**
Fatal. A parameterized #define macro was called with more arguments than declared.
- 26 Actual macro parameter too long (max is 512)**
A single macro argument may not exceed the length of a source line.
- 27 Macro name called with too few parameters**
A parameterized #define macro was called with fewer arguments than declared.
- 28 Missing #endif**
Fatal. The end of file was encountered during skipping of text after a false condition.
- 29 Type specifier expected**
A type description was missing. This could happen in struct, union, prototyped function definitions/declarations, or in K&R function formal parameter declarations.

- 30 Identifier unexpected**
There was an invalid identifier. This could be an identifier in a type name definition like: `sizeof(int*ident);` or two consecutive identifiers.
- 31 Identifier name redeclared**
There was a redeclaration of a declarator identifier.
- 32 Invalid declaration syntax**
There was an undecodable declarator.
- 33 Unbalanced '(' or ')' in declarator**
There was a parenthesis error in a declarator.
- 34 C statement or func-def in #include file, add i to the -r switch**
To get proper C source line stepping for `#include` code when the C-SPY debugger is used, the `-ri` option must be specified.

Other source code debuggers (that do not use the UBROF output format) may not work with code in `#include` files.
- 35 Invalid declaration of struct, union or enum type**
A `struct`, `union`, or `enum` was followed by an invalid token(s).
- 36 Tag identifier name redeclared**
A `struct`, `union`, or `enum` tag is already defined in the current scope.
- 37 Function name declared within struct or union**
A function was declared as a member of `struct` or `union`.
- 38 Invalid width of field (max is nn)**
The declared width of field exceeds the size of an integer (`nn` is 16 or 32 depending on the target processor).
- 39 ';' or ';' expected**
There was a missing `,` or `;` at the end of the declarator.
- 40 Array dimension outside of unsigned int bounds**
Array dimension is negative or too large to be represented in an unsigned integer.
- 41 Member name of struct or union redeclared**
A member of `struct` or `union` was redeclared.
- 42 Empty struct or union**
There was a declaration of `struct` or `union` containing no members.
- 43 Object cannot be initialized**
There was an attempted initialization of `typedef` declarator or `struct` or `union` member.

- 44 ;' expected**
A statement or declaration needs a terminating semicolon.
- 45 ']' expected**
There was a bad array declaration or array expression.
- 46 ':' expected**
There was a missing colon after default, case label, or in ?-operator.
- 47 '(' expected**
The probable cause is a misformed for, if, or while statement.
- 48 ')' expected**
The probable cause is a misformed for, if, or while statement or expression.
- 49 ;' expected**
There was an invalid declaration.
- 50 '{' expected**
There was an invalid declaration or initializer.
- 51 '}' expected**
There was an invalid declaration or initializer.
- 52 Too many local variables and formal parameters (max is 1024)**
Fatal. The compiler limit was exceeded.
- 53 Declarator too complex (max is 128 '(' and/or '*')**
The declarator contained too many (,), or *.
- 54 Invalid storage class**
An invalid storage-class for the object was specified.
- 55 Too deep block nesting (max is 50)**
Fatal. The { } nesting in a function definition was too deep.
- 56 Array of functions**
An attempt was made to declare an array of functions.
The valid form is array of pointers to functions:

```
int array [ 5 ] ();          /* Invalid */
int (*array [ 5 ]) ();     /* Valid */
```
- 57 Missing array dimension specifier**
There was a multi-dimensional array declarator with a missing specified dimension. Only the first dimension can be excluded (in declarations of extern arrays and function formal parameters).

- 58 Identifier name redefined**
There was a redefinition of a declarator identifier.
- 59 Function returning array**
Functions cannot return arrays.
- 60 Function definition expected**
A K&R function header was found without a following function definition, for example:

```
int f(i); /* Invalid */
```
- 61 Missing identifier in declaration**
A declarator lacked an identifier.
- 62 Simple variable or array of a void type**
Only pointers, functions, and formal parameters can be of `void` type.
- 63 Function returning function**
A function cannot return a function, as in:

```
int f()(); /* Invalid */
```
- 64 Unknown size of variable object name**
The defined object has unknown size. This could be an external array with no dimension given or an object of an only partially (forward) declared struct or union.
- 65 Too many errors encountered (>100)**
Fatal. The compiler aborts after a certain number of diagnostic messages.
- 66 Function name redefined**
Multiple definitions of a function were encountered.
- 67 Tag name undefined**
There was a definition of a variable of `enum` type with type undefined or a reference to undefined `struct` or `union` type in a function prototype or as a `sizeof` argument.
- 68 case outside switch**
There was a `case` without any active `switch` statement.
- 69 interrupt function may not be referred or called**
An `interrupt` function call was included in the program. Interrupt functions can be called by the run-time system only.
- 70 Duplicated case label: nn**
The same constant value was used more than once as a case label.

- 71 default outside switch**
 There was a `default` without any active `switch` statement.
- 72 Multiple default within switch**
 More than one `default` in one `switch` statement.
- 73 Missing while in do - while statement**
 Probable cause is missing `{ }` around multiple statements.
- 74 Label name redefined**
 A label was defined more than once in the same function.
- 75 continue outside iteration statement**
 There was a `continue` outside any active `while`, `do ... while`, or `for` statement.
- 76 break outside switch or iteration statement**
 There was a `break` outside any active `switch`, `while`, `do ... while`, or `for` statement.
- 77 Undefined label name**
 There is a `goto` label with no `label:` definition within the function body.
- 78 Pointer to a field not allowed**
 There is a pointer to a field member of `struct` or `union`:
- ```
struct
{
 int *f:6; /* Invalid */
}
```
- 79 Argument of binary operator missing**  
 The first or second argument of a binary operator is missing.
- 80 Statement expected**  
 One of `?` `:` `,` `]` or `}` was found where statement was expected.
- 81 Declaration after statement**  
 A declaration was found after a statement.  
 This could be due to an unwanted `;` for example:
- ```
int i;;
char c; /* Invalid */
```
- Since the second `;` is a statement it causes a declaration after a statement.
- 82 else without preceding if**
 The probable cause is bad `{ }` nesting.

- 83 enum constant(s) outside int or unsigned int range**
An enumeration constant was created too small or too large.
- 84 Function name not allowed in this context**
An attempt was made to use a function name as an indirect address.
- 85 Empty struct, union or enum**
There is a definition of `struct` or `union` that contains no members or a definition of `enum` that contains no enumeration constants.
- 86 Invalid formal parameter**
There is a fault with the formal parameter in a function declaration.
Possible causes are:
- ```
int f(); /* valid K&R declaration */
int f(i); /* invalid K&R declaration */
int f(int i); /* valid ANSI declaration */
int f(i); /* invalid ANSI declaration */
```
- 87 Redeclared formal parameter: name**  
A formal parameter in a K&R function definition was declared more than once.
- 88 Contradictory function declaration**  
`void` appears in a function parameter type list together with other type of specifiers.
- 89 "... without previous parameter(s)**  
... cannot be the only parameter description specified.  
For example:
- ```
int f( ... );           /* Invalid */
int f( int, ... );     /* Valid */
```
- 90 Formal parameter identifier missing**
An identifier of a parameter was missing in the header of a prototyped function definition.
For example:
- ```
int f(int *p, char, float ff) /* Invalid - second
 parameter has
 no name */
{
 /* function body */
}
```

- 91 Redeclared number of formal parameters**  
A prototyped function was declared with a different number of parameters than the first declaration.
- For example:
- ```
int f(int, char);      /* first declaration -valid */
int f(int);           /* fewer parameters -invalid */
int f(int, char, float); /* more parameters -invalid */
```
- 92 Prototype appeared after reference**
A prototyped declaration of a function appeared after it was defined or referenced as a K&R function.
- 93 Initializer to field of width nn (bits) out of range**
A bit-field was initialized with a constant too large to fit in the field space.
- 94 Fields of width 0 must not be named**
Zero length fields are only used to align fields to the next `int` boundary and cannot be accessed via an identifier.
- 95 Second operand for division or modulo is zero**
An attempt was made to divide by zero.
- 96 Unknown size of object pointed to**
An incomplete pointer type is used within an expression where size must be known.
- 97 Undefined static function name**
A function was declared with static storage class but never defined.
- 98 Primary expression expected**
An expression was missing.
- 99 Extended keyword not allowed in this context**
An extended processor-specific keyword occurred in an illegal context; eg `interrupt int i`.
- 100 Undeclared identifier: name**
There was a reference to an identifier that had not been declared.
- 101 First argument of '.' operator must be of struct or union type**
The dot operator `.` was applied to an argument that was not `struct` or `union`.
- 102 First argument of '->' was not pointer to struct or union**
The arrow operator `->` was applied to an argument that was not a pointer to a `struct` or `union`.

103 Invalid argument of sizeof operator

The `sizeof` operator was applied to a bit-field, function, or extern array of unknown size.

104 Initializer string exceeds array dimension

An array of `char` with explicit dimension was initialized with a string exceeding array size.

For example:

```
char array [ 4 ] = "abcde"; /* invalid */
```

105 Language feature not implemented: language feature

The compiler does not currently support the language feature used. For a list of the different target-specific messages that can appear under this error message number, see *8051-specific error messages*, page 187.

106 Too many function parameters (max is 127)

Fatal. There were too many parameters in function declaration/definition.

107 Function parameter *name* already declared

A formal parameter in a function definition header was declared more than once.

For example:

```
/* K&R function */ int myfunc( i, i ) /* invalid */
int i;
{
}
/* Prototyped function */
int myfunc( int i, int i ) /* invalid */
{
}
```

108 Function parameter name declared but not found in header

In a K&R function definition, the parameter was declared but not specified in the function header.

For example:

```
int myfunc( i )
int i, j /* invalid - j is not specified in the
         function header */
{
}
```

109 ';' unexpected

An unexpected delimiter was encountered.

- 110 **)' unexpected**
An unexpected delimiter was encountered.
- 111 **{' unexpected**
An unexpected delimiter was encountered.
- 112 **;' unexpected**
An unexpected delimiter was encountered.
- 113 **?' unexpected**
An unexpected delimiter was encountered.
- 114 **[' unexpected**
An unexpected delimiter was encountered.
- 115 **(' unexpected**
An unexpected delimiter was encountered.
- 116 **Integral expression required**
The evaluated expression yielded a result of the wrong type.
- 117 **Floating point expression required**
The evaluated expression yielded a result of the wrong type.
- 118 **Scalar expression required**
The evaluated expression yielded a result of the wrong type.
- 119 **Pointer expression required**
The evaluated expression yielded a result of the wrong type.
- 120 **Arithmetic expression required**
The evaluated expression yielded a result of the wrong type.
- 121 **Lvalue required**
The expression result was not a memory address.
- 122 **Modifiable lvalue required**
The expression result was not a variable object or a const.
- 123 **Prototyped function argument number mismatch**
A prototyped function was called with a number of arguments different from the number declared.
- 124 **Unknown struct or union member: name**
An attempt was made to reference a non-existent member of a struct or union.
- 125 **Attempt to take address of field**
The & operator may not be used on bitfields.

126 Attempt to take address of register variable

The & operator may not be used on objects with register storage class.

127 Incompatible pointers

There must be full compatibility of objects that pointers point to.

In particular, if pointers point (directly or indirectly) to prototyped functions, the code performs a compatibility test on return values and also on the number of parameters and their types. This means that incompatibility can be hidden quite deeply, for example:

```
char ((*p1)[8])(int);
char ((*p2)[8])(float);
/* p1 and p2 are incompatible - the function
parameters have incompatible types */
```

The compatibility test also includes the checking of array dimensions if they appear in the description of the objects being pointed to, for example:

```
int (*p1)[8];
int (*p2)[9];
/* p1 and p2 are incompatible - array dimensions
differ */
```

128 Function argument incompatible with its declaration

A function argument is incompatible to the argument in the declaration.

129 Incompatible operands of binary operator

The type of one or more operands to a binary operator was incompatible with the operator.

130 Incompatible operands of '=' operator

The type of one or more operands to = was incompatible with =.

131 Incompatible return expression

The result of the expression is incompatible with the return value declaration.

132 Incompatible initializer

The result of the initializer expression is incompatible with the object to be initialized.

133 Constant value required

The expression in a case label, #if, #elif, bitfield declarator, array declarator, or static initializer was not constant.

134 Unmatching struct or union arguments to '?' operator

The second and third argument of the ? operator are different.

135 pointer + pointer operation

Pointers may not be added.

- 136 Redeclaration error**
The current declaration is inconsistent with earlier declarations of the same object.
- 137 Reference to member of undefined struct or union**
The only allowed reference to undefined `struct` or `union` declarators is a pointer.
- 138 - pointer expression**
The `-` operator may be used on pointers only if both operators are pointers, that is, `pointer - pointer`. This error means that an expression of the form `non-pointer - pointer` was found.
- 139 Too many extern symbols declared (max is 32767)**
Fatal. The compiler limit was exceeded.
- 140 void pointer not allowed in this context**
A pointer expression such as an indexing expression involved a `void pointer` (element size unknown).
- 141 #error any message**
Fatal. The preprocessor directive `#error` was found, notifying that something must be defined on the command line in order to compile this module.
- 142 interrupt function can only be void and have no arguments**
An interrupt function declaration had a non-void result and/or arguments, neither of which are allowed.
- 143 Too large, negative or overlapping interrupt [value] in name**
Check the vector [*value*] of the declared interrupt functions.
- 144 Bad context for storage modifier (storage-class or function)**
The `no_init` keyword can only be used to declare variables with `static storage-class`. That is, `no_init` cannot be used in `typedef` statements or applied to auto variables of functions. An active `#pragma memory=no_init` can cause such errors when function declarations are found.
- 145 Bad context for function call modifier**
The keywords `interrupt` or `monitor` can be applied only to function declarations.
- 146 Unknown #pragma identifier**
An unknown `#pragma` identifier was found. This error will terminate object code generation only if the `-g` option is in use.

- 147 Extension keyword name**
Upon executing:
`#pragma language=extended`
the compiler found that the named identifier has the same name as an extension keyword. This error is only issued when compiler is executing in ANSI mode.
- 148 '=' expected**
An `sfr`-declared identifier must be followed by `=value`.
- 149 Attempt to take address of sfr or bit variable**
The `&` operator may not be applied to variables declared as `bit` or as `sfr`.
- 150 Illegal range for sfr or bit address**
The address expression is not a valid `bit` or `sfr` address.
- 151 Too many functions defined in a single module.**
There may not be more than 256 functions in use in a module. Note that there are no limits to the number of declared functions.
- 152 '.' expected**
The `.` was missing from a `bit` declaration.
- 153 Illegal context for extended specifier**

8051-SPECIFIC ERROR MESSAGES

The following list shows the 8051-specific compilation error messages.

- 105 Language feature not implemented:**
- 105 Returning bit is not allowed in reentrant function.**
Use another return value (eg `char`).
- 105 Only xdata allowed as local memory attribute in reentrant function**
Do not override the default storage class in reentrant function.
- 105 Only idata allowed as local memory attribute in reentrant_idata function**
- 105 Do not override the default storage class in reentrant function.**
- 105 Reentrant_idata function called with structure with over 255 bytes**
- 105 Returning bit is not allowed in reentrant function**
- 105 Only xdata allowed as local memory attribute in reentrant function**

- 105 **Only idata allowed as local memory attribute in reentrant_idata function**
- 105 **Interrupt function with "using" attribute calls other funtions**
- 105 **Internal stack has been overflowed by function**
- 105 **8075I not available for Baseline**
- 105 **Floats are not available for 805I Baseline**
- 105 **Reentrant code not available for Baseline**
- 105 **Banked/large memory model not available for Baseline**
- 105 **Floats are not available for Baseline**
- 105 **"Intrinsic" missing operand**
Intrinsic can be any intrinsic function.
- 105 **"Intrinsic" missing first operand**
Intrinsic can be any intrinsic function.
- 105 **"Intrinsic" missing second operand**
Intrinsic can be any intrinsic function.
- 105 **"Intrinsic" missing third operand**
Intrinsic can be any intrinsic function.
- 105 **"Intrinsic" too many operands**
Intrinsic can be any intrinsic function.
- 105 **"_tbac" operand not bit variable**
- 105 **"_opc" operand not char constant**
- 153 **Illegal context for [bit, sfr, xdata, idata, code, no_init] specifier**
This error is displayed because of the illegal use of a keyword:

Illegally used keyword	Suggestion
<code>xdata int my_f();</code>	The only extension allowed is bit.
<code>bit my_array[10];</code>	bit and sfr cannot be array.
<code>void my_f(code int my_var)</code>	Function parameters cannot be in code memory.
<code>void my_f(sfr my_port)</code>	sfr cannot define a variable.
<code>void(*my_fptr)(bit my_var)</code>	Function pointer arguments cannot contain extended keywords.

Table 38: Suggestions for illegally used keywords

Illegally used keyword	Suggestion
void my_f(int my_var) xdata int my_var	Incorrect style for function argument.
void my_f(int my_var) { no_init int my_var;	no_init not allowed with auto variables.

Table 38: Suggestions for illegally used keywords

154 Writing to CODE memory

80751-SPECIFIC ERROR MESSAGES

The following errors can be generated when using the `-v1` switch to produce 80751 code:

- 18 Range error in module 'module' ('file'), segment 'segment' at address 'address'. Value 'value', in tag t_ref_fn5 I, is out of bounds ACALL not in page.**
If a function call is made that is to an address outside of the 2 Kbytes allowed area, the linker will produce the error. The only way to delete this error is to reduce the amount of code.
- 73 Label ?ARG_MOVE not found (recursive functions need it)**
No recursive functions are allowed. The linker will generate the error if there is recursion in the system.
- 46 Undefined external setjmp/longjmp referred in module (file)**
No calls to `longjmp/setjmp` are allowed. The linker will generate the error if `longjmp/setjmp` is used.
- 105 Reentrant code not available for 80751.**
No reentrant functions are allowed. The compiler will generate the error if it is used. This is also generated if the compiler switch `-E` is used.
- 105 Floats are not available for 80751.**
No floats are allowed. The compiler will generate the error if they are used.
- 105 80751 does not support xdata.**
No `XDATA` memory or `XDATA` pointer attributes are allowed. The compiler will generate the error if it is used. This error is also issued if the compiler switch `-y` is used.
- 105 Only memory model tiny and small available for 80751.**
If any other memory model than `tiny` or `small` is used for the 80751, the compiler will generate the error.

Compilation warning messages

The following table lists the compilation warning messages:

- | | |
|-----------|--|
| 0 | Macro 'name' redefined
A symbol defined with <code>#define</code> was redeclared with a different argument or formal list. |
| 1 | Macro formal parameter 'name' is never referenced
A <code>#define</code> formal parameter never appeared in the argument string. |
| 2 | Macro 'name' is already #undef
<code>#undef</code> was applied to a symbol that was not a macro. |
| 3 | Macro 'name' called with empty parameter(s)
A parameterized macro defined in a <code>#define</code> statement was called with a zero-length argument. |
| 4 | Macro 'name' is called recursively; not expanded
A recursive macro call makes the pre-processor stop further expansion of that macro. |
| 5 | Undefined symbol 'name' in #if or #elif; assumed zero
It is considered as bad programming practice to assume that non-macro symbols should be treated as zeros in <code>#if</code> and <code>#elif</code> expressions. Use either: <code>#ifdef</code> symbol or <code>#if defined (symbol)</code> . |
| 6 | Unknown escape sequence ('\c'); assumed 'c'
A backslash (<code>\</code>) found in a character constant or string literal was followed by an unknown escape character. |
| 7 | Nested comment found without using the -C option
The character sequence <code>/*</code> was found within a comment, and ignored. |
| 8 | Invalid type-specifier for field; assumed int
In this implementation, bit-fields may be specified only as <code>int</code> or <code>unsigned int</code> . |
| 9 | Undeclared function parameter 'name'; assumed int
An undeclared identifier in the header of a K&R function definition is by default given the type <code>int</code> . |
| 10 | Dimension of array ignored; array assumed pointer
An array with an explicit dimension was specified as a formal parameter, and the compiler treated it as a pointer to object. |
| 11 | Storage class static ignored; name declared extern
An object or function was first declared as <code>extern</code> (explicitly or by default) and later declared as <code>static</code> . The <code>static</code> declaration is ignored. |

- 12 Incompletely bracketed initializer**
To avoid ambiguity, initializers should either use only one level of { } brackets or be completely surrounded by { } brackets.
- 13 Unreferenced label 'name'**
Label was defined but never referenced.
- 14 Type specifier missing; assumed int**
No type specifier given in declaration – assumed to be int.
- 15 Wrong usage of string operator ('#' or '##'); ignored**
This implementation restricts usage of # and ## operators to the token-field of parameterized macros.

In addition the # operator must precede a formal parameter:
- ```
#define mac(p1)#p1 /* Becomes "p1" */
#define mac(p1,p2)p1+p2##add_this
/* Merged p2 */
```
- 16 Non-void function: return with 'expression'; expected**  
A non-void function definition should exit with a defined return value in all places.
- 17 Invalid storage class for function; assumed to be extern**  
Invalid storage class for function– ignored. Valid classes are `extern`, `static`, or `typedef`.
- 18 Redeclared parameter's storage class**  
Storage class of a function formal parameter was changed from `register` to `auto` or vice versa in a subsequent declaration/definition.
- 19 Storage class extern ignored; 'name' was first declared as static**  
An identifier declared as `static` was later explicitly or implicitly declared as `extern`. The `extern` declaration is ignored.
- 20 Unreachable statement(s)**  
One or more statements were preceded by an unconditional jump or return such that the statement or statements would never be executed.  
  
For example:
- ```
break;
i = 2; /* Never executed */
```
- 21 Unreachable statement(s) at unreferenced label 'name'**
One or more labeled statements were preceded by an unconditional jump or return but the label was never referenced, so the statement or statements would never be executed.

For example:

```
break;
here:
i = 2; /* Never executed */
```

- 22 Non-void function: explicit return 'expression'; expected**
A non-void function generated an implicit return. This could be the result of an unexpected exit from a loop or switch. Note that a switch without default is always considered by the compiler to be 'exitable' regardless of any case constructs.
- 23 Undeclared function 'name'; assumed extern int**
A reference to an undeclared function causes a default declaration to be used. The function is assumed to be of K&R type, have extern storage class, and return int.
- 24 Static memory option converts local auto or register to static**
A command line option for static memory allocation caused auto and register declarations to be treated as static.
- 25 Inconsistent use of K&R function - varying number of parameters**
A K&R function was called with a varying number of parameters.
- 26 Inconsistent use of K&R function - changing type of parameter**
A K&R function was called with changing types of parameters.

For example:

```
myfunc ( 34 ); /* int argument */
myfunc( 34.6 ); /* float argument */
```

- 27 Size of extern object 'name' is unknown**
extern arrays should be declared with size.
- 28 Constant [index] outside array bounds**
There was a constant index outside the declared array bounds.
- 29 Hexadecimal escape sequence larger than char**
The escape sequence is truncated to fit into char.
- 30 Attribute ignored**
Since const or volatile are attributes of objects they are ignored when given with a structure, union, or enumeration tag definition that has no objects declared at the same time. Also, functions are considered as being unable to return const or volatile.

For example:

```
const struct s
```

```

{
    ...
}; /* no object declared, const ignored - warning */
const int myfunc(void);
    /* function returning const int - warning */
const int (*fp)(void);
    /* pointer to function returning const int -
    warning */
int (*const fp)(void);
    /* const pointer to function returning int - OK,
    no warning */

```

31 **Incompatible parameters of K&R functions**

Pointers (possibly indirect) to functions or K&R function declarators have incompatible parameter types.

The pointer was used in one of following contexts:

```

pointer - pointer,
expression ? ptr : ptr,
pointer relational_op pointer
pointer equality_op pointer
pointer = pointer
formal parameter vs actual parameter

```

32 **Incompatible numbers of parameters of K&R functions**

Pointers (possibly indirect) to functions or K&R function declarators have a different number of parameters.

The pointer is directly used in one of following contexts:

```

pointer - pointer
expression ? ptr : ptr
pointer relational_op
pointerpointer equality_op pointer
pointer = pointer
formal parameter vs actual parameter

```

33 **Local or formal 'name' was never referenced**

A formal parameter or local variable object is unused in the function definition.

34 **Non-printable character \xhh found in literal or character constant**

It is considered as bad programming practice to use non-printable characters in string literals or character constants. Use \0xhhh to get the same result.

- 35 Old-style (K&R) type of function declarator**
An old style K&R function declarator was found. This warning is issued only if the `-gA` option is in use.
- 36 Floating point constant out of range**
A floating-point value is too large or too small to be represented by the floating-point system of the target.
- 37 Illegal float operation: division by zero not allowed**
During constant arithmetic a zero divide was found.
- 38 Tag identifier 'name' was never defined**
- 39 Dummy statement. Optimized away!**
Redundant code found. This usually indicates a typing mistake in the user code or it might also be generated when using macros which are a little bit too generic (which is not a fault).
- For example:
- ```
a+b;
```
- 40 Possible bug! If statement terminated**  
This usually indicates a typing mistake in the user code.
- For example:
- ```
if (a==b);
{
  <if body>
}
```
- 41 Possible bug! Uninitialized variable**
A variable is used before initialization (the variable has a random value).
- For example:
- ```
void func (p1)
{
 short a;
 p1+=a;
}
```
- This message does not exist.
- 42 Possible bug! Integer promotion may cause problems. Use cast to avoid it**  
The rule of integer promotion says that all integer operations must generate a result as if they were of `int` type if they have a small precision than `int` and this can sometimes lead to unexpected results.

For example:

```
short tst(unsigned char a)
{
 if (-a)
 return (1);
 else
 return (-1);
}
```

This example will always return the value 1 even with the value 0xff. The reason is that the integer promotion casts the variable a to 0x00ff first and then performs a bit not.

Integer promotion is ignored by many other C compilers, so this warning may be generated when recompiling an existing program with the IAR Systems compiler.

#### 43 **Possible bug! Single '=' instead of '==' used in if statement**

This usually indicates a typing mistake in the user code.

For example:

```
if (a=1)
{
 <if body>
}
```

#### 44 **Redundant expression. Example: Multiply with 1, add with 0**

This might indicate a typing mistake in the user code, but it can also be a result of stupid code generated by a case tool.

#### 45 **Possible bug! Strange or faulty expression. Example: Division by zero**

This usually indicates a bug in the user code.

#### 46 **Unreachable code deleted by the global optimizer**

Redundant code block in the user code. It might be a result of a bug but is usually only a sign of incomplete code.

#### 47 **Unreachable returns. The function will never return**

The function will never be able to return to the calling function.

This might be a result of a bug, but is usually generated when you have never ending loops in a RTOS system.

**48      Unsigned compare always true/false**

This indicates a bug in the user code! A common reason is a missing `-c` compiler switch.

For example:

```
for (uc=10; uc>=0; uc--)
{
 <loop body>
}
```

This is a never ending loop because an unsigned value is always larger than or equal to zero.

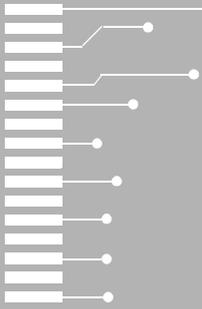
This message does not exist.

**49      Signed compare always true/false**

This indicates a bug in the user code!

**8051-SPECIFIC WARNING MESSAGES**

There are no 8051-specific warning messages.



## Part 3. Library functions

This part of the 8051 IAR C Compiler Reference Guide contains the following chapter:

- General C library definitions



# General C library definitions

This chapter introduces C library functions provided in the 8051 IAR C Compiler. It also lists and explains the usage of header files that are used for accessing library definitions.

For reference information about all the C library functions, see the `iarclib.pdf` file, which is provided with the product.

---

## Introduction

The 8051 IAR C Compiler package provides most of the important C library definitions that apply to PROM-based embedded systems. These are of three types:

- Standard C library definitions available for user programs. These are documented in this chapter.
- `CSTARTUP`, the single program module containing the start-up code. This is described in the *Configuration* chapter, page 32.
- Intrinsic functions, allowing low-level use of 8051 features. See the chapter *Intrinsic functions* for more information.

### LIBRARY OBJECT FILES

The IAR XLINK Linker includes only those routines that are required (directly or indirectly) by the user's program.

Most of the library definitions can be used without modification, that is, directly from the library object files that are supplied with the product. There are some I/O-oriented routines (such as `putchar` and `getchar`) that you may need to customize for your target application.

The library object files are supplied having been compiled with the **Flag old-style functions** `-gA` option.

### HEADER FILES

The user program gains access to library definitions through header files, which it incorporates using the `#include` directive. To avoid wasting time at compilation, the definitions are divided into a number of different header files. Each of these files covers a particular functional area, letting you include only those that are required.

It is essential to include the appropriate header file before making any reference to its definitions. Failure to do this can cause the call to fail during execution or generate error or warning messages at compile time or link time.

---

## Library definitions summary

This section lists the header files and summarizes the functions included in each. Header files may additionally contain target-specific definitions.

### CHARACTER HANDLING – ctype.h

|          |                             |                                         |
|----------|-----------------------------|-----------------------------------------|
| isalnum  | int isalnum(int <i>c</i> )  | Letter or digit equality.               |
| isalpha  | int isalpha(int <i>c</i> )  | Letter equality.                        |
| iscntrl  | int iscntrl(int <i>c</i> )  | Control code equality.                  |
| isdigit  | int isdigit(int <i>c</i> )  | Digit equality.                         |
| isgraph  | int isgraph(int <i>c</i> )  | Printable non-space character equality. |
| islower  | int islower(int <i>c</i> )  | Lower case equality.                    |
| isprint  | int isprint(int <i>c</i> )  | Printable character equality.           |
| ispunct  | int ispunct(int <i>c</i> )  | Punctuation character equality.         |
| isspace  | int isspace(int <i>c</i> )  | White-space character equality.         |
| isupper  | int isupper(int <i>c</i> )  | Upper case equality.                    |
| isxdigit | int isxdigit(int <i>c</i> ) | Hex digit equality.                     |
| tolower  | int tolower(int <i>c</i> )  | Converts to lower case.                 |
| toupper  | int toupper(int <i>c</i> )  | Converts to upper case.                 |

### LOW-LEVEL ROUTINES – iclbutl.h

|                               |                                                                                                                                           |                                                         |
|-------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------|
| <code>_formatted_read</code>  | int <code>_formatted_read</code> (const char ** <i>line</i> , const char ** <i>format</i> , va_list <i>ap</i> )                           | Reads formatted data.                                   |
| <code>_formatted_write</code> | int <code>_formatted_write</code> (const char* <i>format</i> , void <i>outputf</i> (char, void *), void * <i>sp</i> , va_list <i>ap</i> ) | Formats and writes data.                                |
| <code>_medium_read</code>     | int <code>_formatted_read</code> (const char ** <i>line</i> , const char ** <i>format</i> , va_list <i>ap</i> )                           | Reads formatted data excluding floating-point numbers.  |
| <code>_medium_write</code>    | int <code>_formatted_write</code> (const char* <i>format</i> , void <i>outputf</i> (char, void *), void * <i>sp</i> , va_list <i>ap</i> ) | Writes formatted data excluding floating-point numbers. |

|                           |                                                                                                                                       |                                     |
|---------------------------|---------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------|
| <code>_small_write</code> | <code>int _formatted_write (const char* <i>format</i>, void <i>outputf</i> (char, void *), void *<i>sp</i>, va_list <i>ap</i>)</code> | Small formatted data write routine. |
|---------------------------|---------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------|

### MATHEMATICS – math.h

|                    |                                                                    |                                                        |
|--------------------|--------------------------------------------------------------------|--------------------------------------------------------|
| <code>acos</code>  | <code>double acos(double <i>arg</i>)</code>                        | Arc cosine.                                            |
| <code>asin</code>  | <code>double asin(double <i>arg</i>)</code>                        | Arc sine.                                              |
| <code>atan</code>  | <code>double atan(double <i>arg</i>)</code>                        | Arc tangent.                                           |
| <code>atan2</code> | <code>double atan2(double <i>arg1</i>, double <i>arg2</i>)</code>  | Arc tangent with quadrant.                             |
| <code>ceil</code>  | <code>double ceil(double <i>arg</i>)</code>                        | Smallest integer greater than or equal to <i>arg</i> . |
| <code>cos</code>   | <code>double cos(double <i>arg</i>)</code>                         | Cosine.                                                |
| <code>cosh</code>  | <code>double cosh(double <i>arg</i>)</code>                        | Hyperbolic cosine.                                     |
| <code>exp</code>   | <code>double exp(double <i>arg</i>)</code>                         | Exponential.                                           |
| <code>fabs</code>  | <code>double fabs(double <i>arg</i>)</code>                        | Double-precision floating-point absolute.              |
| <code>floor</code> | <code>double floor(double <i>arg</i>)</code>                       | Largest integer less than or equal.                    |
| <code>fmod</code>  | <code>double fmod(double <i>arg1</i>, double <i>arg2</i>)</code>   | Floating-point remainder.                              |
| <code>frexp</code> | <code>double frexp(double <i>arg1</i>, int *<i>arg2</i>)</code>    | Splits a floating-point number into two parts.         |
| <code>ldexp</code> | <code>double ldexp(double <i>arg1</i>, int <i>arg2</i>)</code>     | Multiply by power of two.                              |
| <code>log</code>   | <code>double log(double <i>arg</i>)</code>                         | Natural logarithm.                                     |
| <code>log10</code> | <code>double log10(double <i>arg</i>)</code>                       | Base-10 logarithm.                                     |
| <code>modf</code>  | <code>double modf(double <i>value</i>, double *<i>iptr</i>)</code> | Fractional and integer parts.                          |
| <code>pow</code>   | <code>double pow(double <i>arg1</i>, double <i>arg2</i>)</code>    | Raises to the power.                                   |
| <code>sin</code>   | <code>double sin(double <i>arg</i>)</code>                         | Sine.                                                  |
| <code>sinh</code>  | <code>double sinh(double <i>arg</i>)</code>                        | Hyperbolic sine.                                       |
| <code>sqrt</code>  | <code>double sqrt(double <i>arg</i>)</code>                        | Square root.                                           |
| <code>tan</code>   | <code>double tan(double <i>x</i>)</code>                           | Tangent.                                               |

|      |                         |                     |
|------|-------------------------|---------------------|
| tanh | double tanh(double arg) | Hyperbolic tangent. |
|------|-------------------------|---------------------|

### NON-LOCAL JUMPS – setjmp.h

|         |                                       |                              |
|---------|---------------------------------------|------------------------------|
| longjmp | void longjmp(jmp_buf env,<br>int val) | Long jump.                   |
| setjmp  | int setjmp(jmp_buf env)               | Sets up a jump return point. |

### VARIABLE ARGUMENTS – stdarg.h

|          |                                  |                                        |
|----------|----------------------------------|----------------------------------------|
| va_arg   | type va_arg(va_list ap, mode)    | Next argument in function call.        |
| va_end   | void va_end(va_list ap)          | Ends reading function call arguments.  |
| va_list  | char *va_list[1]                 | Argument list type.                    |
| va_start | void va_start(va_list ap, parmN) | Starts reading function call arguments |

### INPUT/OUTPUT – stdio.h

|         |                                                       |                                     |
|---------|-------------------------------------------------------|-------------------------------------|
| getchar | int getchar(void)                                     | Gets character.                     |
| gets    | char *gets(char *s)                                   | Gets string.                        |
| printf  | int printf(const char<br>*format, ...)                | Writes formatted data.              |
| putchar | int putchar(int value)                                | Puts character.                     |
| puts    | int puts(const char *s)                               | Puts string.                        |
| scanf   | int scanf(const char<br>*format, ...)                 | Reads formatted data.               |
| sprintf | int sprintf(char *s, const<br>char *format, ...)      | Writes formatted data to a string.  |
| sscanf  | int sscanf(const char *s,<br>const char *format, ...) | Reads formatted data from a string. |

### GENERAL UTILITIES – stdlib.h

|       |                               |                                    |
|-------|-------------------------------|------------------------------------|
| abort | void abort(void)              | Terminates the program abnormally. |
| abs   | int abs(int j)                | Absolute value.                    |
| atof  | double atof(const char *nptr) | Converts ASCII to double.          |

|         |                                                                                                                                   |                                                |
|---------|-----------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------|
| atoi    | int atoi(const char *nptr)                                                                                                        | Converts ASCII to int.                         |
| atol    | long atol(const char *nptr)                                                                                                       | Converts ASCII to long int.                    |
| bsearch | void *bsearch(const void *key, const void *base, size_t nmemb, size_t size, int (*compare)(const void *_key, const void *_base)); | Makes a generic search in an array.            |
| calloc  | void *calloc(size_t nelem, size_t elsize)                                                                                         | Allocates memory for an array of objects.      |
| div     | div_t div(int numer, int denom)                                                                                                   | Divide.                                        |
| exit    | void exit(int status)                                                                                                             | Terminates the program.                        |
| free    | void free(void *ptr)                                                                                                              | Frees memory.                                  |
| labs    | long int labs(long int j)                                                                                                         | Long absolute.                                 |
| ldiv    | ldiv_t ldiv(long int numer, long int denom)                                                                                       | Long division.                                 |
| malloc  | void *malloc(size_t size)                                                                                                         | Allocates memory.                              |
| qsort   | void qsort(const void *base, size_t nmemb, size_t size, int (*compare)(const void *_key, const void *_base));                     | Makes a generic sort of an array.              |
| rand    | int rand(void)                                                                                                                    | Random number.                                 |
| realloc | void *realloc(void *ptr, size_t size)                                                                                             | Reallocates memory.                            |
| srand   | void srand(unsigned int seed)                                                                                                     | Sets random number sequence.                   |
| strtod  | double strtod(const char *nptr, char **endptr)                                                                                    | Converts a string to double.                   |
| strtol  | long int strtol(const char *nptr, char **endptr, int base)                                                                        | Converts a string to a long integer.           |
| strtoul | unsigned long int strtoul(const char *nptr, char **endptr, base int)                                                              | Converts a string to an unsigned long integer. |

**STRING HANDLING – string.h**

|          |                                                       |                                                              |
|----------|-------------------------------------------------------|--------------------------------------------------------------|
| memchr   | void *memchr(const void *s, int c, size_t n)          | Searches for a character in memory.                          |
| memcmp   | int memcmp(const void *s1, const void *s2, size_t n)  | Compares memory.                                             |
| memcpy   | void *memcpy(void *s1, void *s2, size_t n)            | Copies memory.                                               |
| memmove  | void *memmove(void *s1, void *s2, size_t n)           | Moves memory.                                                |
| memset   | void *memset(void *s, int c, size_t n)                | Sets memory.                                                 |
| strcat   | char *strcat(char *s1, const char *s2)                | Concatenates strings.                                        |
| strchr   | char *strchr(const char *s, int c)                    | Searches for a character in a string.                        |
| strcmp   | int strcmp(const char *s1, const char *s2)            | Compares two strings.                                        |
| strcoll  | int strcoll(const char *s1, const char *s2)           | Compares strings.                                            |
| strcpy   | char *strcpy(char *s1, const char *s2)                | Copies string.                                               |
| strcspn  | size_t strcspn(const char *s1, const char *s2)        | Spans excluded characters in string.                         |
| strerror | char *strerror(int errnum)                            | Gives an error message string.                               |
| strlen   | size_t strlen(const char *s)                          | String length.                                               |
| strncat  | char *strncat(char *s1, const char *s2, size_t n)     | Concatenates a specified number of characters with a string. |
| strncmp  | int strncmp(const char *s1, const char *s2, size_t n) | Compares a specified number of characters with a string.     |
| strncpy  | char *strncpy(char *s1, const char *s2, size_t n)     | Copies a specified number of characters from a string.       |
| strpbrk  | char *strpbrk(const char *s1, const char *s2)         | Finds any one of specified characters in a string.           |
| strrchr  | char *strrchr(const char *s, int c)                   | Finds character from right of string.                        |

|                      |                                                                     |                                             |
|----------------------|---------------------------------------------------------------------|---------------------------------------------|
| <code>strspn</code>  | <code>size_t strspn(const char *s1,<br/>const char *s2)</code>      | Spans characters in a string.               |
| <code>strstr</code>  | <code>char *strstr(const char *s1, const<br/>char *s2)</code>       | Searches for a substring.                   |
| <code>strtok</code>  | <code>char *strtok(char *s1, const char<br/>*s2)</code>             | Breaks a string into tokens.                |
| <code>strxfrm</code> | <code>size_t strxfrm(char *s1,<br/>const char *s2, size_t n)</code> | Transforms a string and returns the length. |

### ASSERTIONS – `assert.h`

|                     |                                          |                       |
|---------------------|------------------------------------------|-----------------------|
| <code>assert</code> | <code>void assert(int expression)</code> | Checks an expression. |
|---------------------|------------------------------------------|-----------------------|

### MISCELLANEOUS HEADER FILES

The following table shows header files that do not contain any functions, but specify various definitions and data types:

| Header file           | Description                                                                                                                 |
|-----------------------|-----------------------------------------------------------------------------------------------------------------------------|
| <code>stddef.h</code> | Common definitions including <code>size_t</code> , <code>NULL</code> , <code>ptrdiff_t</code> , and <code>offsetof</code> . |
| <code>limits.h</code> | Limits and sizes of integral types.                                                                                         |
| <code>float.h</code>  | Limits and sizes of floating-point types.                                                                                   |
| <code>errno.h</code>  | Error return values.                                                                                                        |

*Table 39: Miscellaneous header files*



**A**

|                                 |     |
|---------------------------------|-----|
| active lines only, listing      | 103 |
| address creation for functions  | 49  |
| addressing control              | 4   |
| ANSI definition                 | 139 |
| data types                      | 141 |
| function declarations           | 141 |
| function definition parameter   | 141 |
| hexadecimal string constant     | 142 |
| ANSI prototypes                 | 3   |
| ANSI standard X3.159-1989       | 139 |
| assembler interface             | 43  |
| creating skeleton code          | 48  |
| assembler mnemonics, in listing | 100 |
| assembler support directives    | 49  |
| \$BYTE3                         | 51  |
| \$DEFFN                         | 49  |
| \$IFREF                         | 50  |
| \$LOCBB                         | 50  |
| \$LOCBD                         | 50  |
| \$LOCBI                         | 50  |
| \$LOCBX                         | 50  |
| \$PRMBB                         | 51  |
| \$PRMBD                         | 51  |
| \$PRMBI                         | 51  |
| \$PRMBX                         | 51  |
| \$REFFN                         | 50  |
| assert.h (header file)          | 205 |

**B**

|                                        |        |
|----------------------------------------|--------|
| banked code pointer                    | 61     |
| bdata (extended keyword)               | 4, 109 |
| bit variables                          | 60     |
| bit (data type)                        | 4, 59  |
| bit (extended keyword)                 | 109    |
| bitfields                              | 60     |
| bitfields=default (#pragma directive)  | 122    |
| bitfields=reversed (#pragma directive) | 122    |

|                   |    |
|-------------------|----|
| BITVAR (segment)  | 71 |
| B_CDATA (segment) | 71 |
| B_IDATA (segment) | 71 |
| B_UDATA (segment) | 72 |

**C****C compiler options**

|                                          |            |
|------------------------------------------|------------|
| disabling warnings                       | 104        |
| enabling function return stack expansion | 103        |
| explaining C declarations                | 104        |
| form feed after function                 | 89         |
| global strict type checking              | 90         |
| lines per page                           | 100        |
| listing active lines only                | 103        |
| making a library module                  | 29         |
| running in PROM                          | 100        |
| selecting processor options              | 104        |
| setting                                  | 83         |
| summary                                  | 84         |
| tab spacing                              | 103        |
| -A                                       | 49, 85     |
| -a                                       | 86         |
| -b                                       | 29, 86     |
| -C                                       | 87         |
| -c                                       | 60, 87     |
| -D                                       | 87         |
| -E                                       | 88         |
| -e                                       | 4, 89, 126 |
| -F                                       | 89         |
| -f                                       | 89         |
| -G                                       | 90         |
| -g                                       | 90         |
| -H                                       | 95         |
| -I                                       | 96         |
| -i                                       | 97         |
| -K                                       | 97         |
| -L                                       | 49, 97     |
| -l                                       | 98         |
| -m                                       | 98         |

|                                       |             |                                   |         |
|---------------------------------------|-------------|-----------------------------------|---------|
| -N                                    | 98          | compiler version number           | 134     |
| -n                                    | 99          | const (keyword)                   | 139     |
| -O                                    | 99          | CONST (segment)                   | 74      |
| -o                                    | 95, 100     | cross-reference                   | 105     |
| -P                                    | 100         | CSTACK (segment)                  | 75      |
| -p                                    | 100         | CSTARTUP                          | 32      |
| -q                                    | 49, 100     | cstartup.s03                      | 34      |
| -R                                    | 101         | modifying                         | 32–33   |
| -r                                    | 101         | CSTR (segment)                    | 75      |
| -S                                    | 102         | ctype.h (header file)             | 200     |
| -s                                    | 102         | C_ARGB (segment)                  | 72      |
| -T                                    | 103         | C_ARGD (segment)                  | 72      |
| -t                                    | 103         | C_ARGI (segment)                  | 73      |
| -U                                    | 103         | C_ARGX (segment)                  | 73      |
| -u                                    | 103         | C_ICALL (segment)                 | 73      |
| -v                                    | 104         | C_INCLUDE (environment variable)  | 84      |
| -w                                    | 104, 131    | C_RECFN (segment)                 | 74      |
| -X                                    | 104         |                                   |         |
| -x                                    | 105         |                                   |         |
| -y                                    | 73, 76, 105 |                                   |         |
| -z                                    | 106         |                                   |         |
| C data types                          | 59          | <b>D</b>                          |         |
| C declarations, explaining in listing | 104         | data pointers                     | 61      |
| C library, customizing                | 37          | data types                        | 59, 141 |
| calling convention                    | 43          | bit                               | 4       |
| calling mechanisms                    | 4           | sfr                               | 4       |
| CCSTR (segment)                       | 73          | unsigned                          | 3       |
| CDATA (segment)                       | 74          | data (extended keyword)           | 4, 111  |
| char (data type)                      | 59–60       | data_reentrant (extended keyword) | 4       |
| character-based I/O                   | 29          | debug information, generating     | 101     |
| cl8051*.r03 (library module)          | 29          | diagnostics                       | 173     |
| code pointers                         | 61          | error messages                    | 174     |
| code segment                          | 101         | internal error                    | 173     |
| code (extended keyword)               | 4, 110      | warning messages                  | 196     |
| code (pointer) (extended keyword)     | 110         | double (data type)                | 59      |
| CODE (segment)                        | 74          | D_CDATA (segment)                 | 75      |
| codeseg (#pragma directive)           | 123         | D_IDATA (segment)                 | 75      |
| coding, efficient                     | 3           | D_UDATA (segment)                 | 76      |
| comments, nested                      | 87          | <b>E</b>                          |         |
|                                       |             | ECSTR (segment)                   | 76      |

efficient coding . . . . . 3  
 entry (keyword) . . . . . 139  
 enum . . . . . 59  
 enum (keyword) . . . . . 140  
 environment variables . . . . . 84  
   C\_INCLUDE . . . . . 84  
   QCC8051 . . . . . 83  
 errno.h (header file) . . . . . 205  
 error messages . . . . . 173–174, 196  
 extended keywords . . . . . 4, 107  
   bdata . . . . . 4, 109  
   bit . . . . . 109  
   code . . . . . 4, 110  
   code (pointer) . . . . . 110  
   data . . . . . 4, 111  
   data\_reentrant . . . . . 4  
   idata . . . . . 4, 111  
   idata (pointer) . . . . . 112  
   interrupt . . . . . 4, 112  
   monitor . . . . . 4, 114  
   non\_banked . . . . . 4, 115  
   no\_init . . . . . 20, 115  
   pdata . . . . . 4  
   plm . . . . . 4, 117  
   reentrant . . . . . 4, 117  
   reentrant\_idata . . . . . 118  
   sfr . . . . . 118  
   using . . . . . 119  
   xdata . . . . . 4, 119  
   xdata (pointer) . . . . . 120  
 extensions, language . . . . . 89

## F

file types  
   .i . . . . . 99  
   .xcl . . . . . 27  
 float (data type) . . . . . 59  
 floating-point format . . . . . 60  
 float.h (header file) . . . . . 205

form feed after function (C compiler option) . . . . . 89  
 function calls . . . . . 44  
 function modifiers . . . . . 4  
   function=default (#pragma directive) . . . . . 124  
   function=interrupt (#pragma directive) . . . . . 124  
   function=monitor (#pragma directive) . . . . . 124  
   function=version\_2 (#pragma directive) . . . . . 126

## G

getchar (library function) . . . . . 29  
 getchar.c . . . . . 29  
 global strict type checking (C compiler option) . . . . . 90

## H

header files . . . . . 199  
   assert.h . . . . . 205  
   ctype.h . . . . . 200  
   errno.h . . . . . 205  
   float.h . . . . . 205  
   icclbutl.h . . . . . 200  
   limits.h . . . . . 205  
   math.h . . . . . 201  
   setjmp.h . . . . . 202  
   stdarg.h . . . . . 202  
   stddef.h . . . . . 205  
   stdio.h . . . . . 202  
   stdlib.h . . . . . 202  
   string.h . . . . . 204  
 hexadecimal string constants . . . . . 142

## I

icclbutl.h (header file) . . . . . 200  
 idata (extended keyword) . . . . . 4, 111  
 idata (pointer) (extended keyword) . . . . . 112  
 input . . . . . 29  
 internal error . . . . . 173  
 interrupt functions

|                                  |        |
|----------------------------------|--------|
| calling functions with -E option | 88     |
| for the assembler                | 54     |
| interrupt vectors, defining      | 55     |
| interrupt (extended keyword)     | 4, 112 |
| intrinsic functions              | 5      |
| _argc\$                          | 135    |
| _argt\$                          | 136    |
| _opc                             | 137    |
| _tbac                            | 137    |
| INTVEC (segment)                 | 77     |
| I/O access                       | 4      |
| I/O initialization               | 32     |
| I/O, character based             | 29     |
| I_CDATA (segment)                | 76     |
| I_IDATA (segment)                | 77     |
| I_UDATA (segment)                | 77     |

## K

|                                |     |
|--------------------------------|-----|
| Kernighan & Richie definitions | 139 |
| keywords                       |     |
| const                          | 139 |
| entry                          | 139 |
| enum                           | 140 |
| signed                         | 140 |
| struct                         | 142 |
| union                          | 142 |
| void                           | 140 |
| volatile                       | 140 |

## L

|                                       |     |
|---------------------------------------|-----|
| language extensions                   | 3   |
| enabling                              | 89  |
| language=default (#pragma directive)  | 126 |
| language=extended (#pragma directive) | 127 |
| library functions                     |     |
| getchar                               | 29  |
| printf                                | 30  |
| putchar                               | 29  |

|                                       |         |
|---------------------------------------|---------|
| scanf                                 | 31      |
| sprintf                               | 30      |
| sscanf                                | 31      |
| summary                               | 200     |
| _formatted_read                       | 31      |
| _formatted_write                      | 30      |
| _medium_read                          | 31      |
| _medium_write                         | 30      |
| _small_write                          | 31      |
| library module                        | 86      |
| making                                | 29      |
| limits.h (header file)                | 205     |
| lines per page (C compiler option)    | 100     |
| linker command file                   | 27      |
| listings, formatting                  | 89, 100 |
| Load as PROGRAM module (XLINK option) | 30      |
| local variables                       | 44–45   |
| long double (data type)               | 59      |
| long (data type)                      | 59      |
| lowinit.c                             | 32      |
| low-level control                     | 5       |

## M

|                                     |         |
|-------------------------------------|---------|
| math.h (header file)                | 201     |
| memory models                       | 9       |
| specifying                          | 98      |
| memory=constseg (#pragma directive) | 128     |
| memory=dataseg (#pragma directive)  | 129–130 |
| memory=default (#pragma directive)  | 130     |
| memory=no_init (#pragma directive)  | 130     |
| mnemonics, assembler                | 100     |
| monitor (extended keyword)          | 4, 114  |

## N

|                               |        |
|-------------------------------|--------|
| nested comments               | 87     |
| non-volatile RAM              | 4, 20  |
| non_banked (extended keyword) | 4, 115 |
| as a code pointer             | 61     |

no\_init (extended keyword) . . . . . 4, 20, 115  
 NO\_INIT (segment) . . . . . 20, 78  
 NULL . . . . . 205

## O

object file converter . . . . . 145  
 object filename . . . . . 100  
 offsetof . . . . . 205  
 omfconv . . . . . 145  
 optimization . . . . . 37, 102, 106  
 options summary . . . . . 84  
 output . . . . . 29

## P

parameters . . . . . 44  
   organization in memory . . . . . 45  
   reentrant . . . . . 46  
 pdata (extended keyword) . . . . . 4  
 plm (extended keyword) . . . . . 4, 117  
 PL/M . . . . . 145  
   compiling functions . . . . . 147  
   linking . . . . . 146  
   mapping C pointers . . . . . 148  
   using object file converter (omfconv) . . . . . 145  
 pointer (data type) . . . . . 59  
 predefined symbols . . . . . 5  
   \_\_DATE\_\_ . . . . . 133  
   \_\_FILE\_\_ . . . . . 133  
   \_\_IAR\_SYSTEMS\_ICC . . . . . 133  
   \_\_LINE\_\_ . . . . . 133  
   \_\_STDC\_\_ . . . . . 133  
   \_\_TID\_\_ . . . . . 133  
   \_\_TIME\_\_ . . . . . 134  
 printf (library function) . . . . . 30  
 programming hints . . . . . 3  
 PROM (C compiler option) . . . . . 100  
 ptrdiff\_t . . . . . 205  
 putchar (library function) . . . . . 29

putchar.c . . . . . 29  
 P\_CDATA (segment) . . . . . 78  
 P\_IDATA (segment) . . . . . 78  
 P\_UDATA (segment) . . . . . 78

## Q

QCCH8300 (environment variable) . . . . . 84  
 QCC8051 (environment variable) . . . . . 83–84

## R

RCODE (segment) . . . . . 79  
 recommendations, programming . . . . . 3  
 recursive functions . . . . . 88  
 reentrant code . . . . . 54, 88  
 reentrant functions . . . . . 54  
 reentrant parameters . . . . . 46  
 reentrant (extended keyword) . . . . . 4, 117  
 reentrant\_idata (extended keyword) . . . . . 118  
 register usage . . . . . 44  
 RF\_XDATA (segment) . . . . . 79  
 run-time library . . . . . 8  
 run-time library, customizing . . . . . 37

## S

scanf (library function) . . . . . 31  
 segments . . . . . 67  
   BITVAR . . . . . 71  
   B\_CDATA . . . . . 71  
   B\_IDATA . . . . . 71  
   B\_UDATA . . . . . 72  
   CCSTR . . . . . 73  
   CDATA . . . . . 74  
   CODE . . . . . 74  
   code . . . . . 101  
   CONST . . . . . 74  
   CSTACK . . . . . 75  
   CSTR . . . . . 75

|                                         |        |
|-----------------------------------------|--------|
| C_ARGB                                  | 72     |
| C_ARGD                                  | 72     |
| C_ARGI                                  | 73     |
| C_ARGX                                  | 73     |
| C_ICALL                                 | 73     |
| C_RECFN                                 | 74     |
| D_CDATA                                 | 75     |
| D_IDATA                                 | 75     |
| D_UDATA                                 | 76     |
| ECSTR                                   | 76     |
| INTVEC                                  | 77     |
| I_CDATA                                 | 76     |
| I_IDATA                                 | 77     |
| I_UDATA                                 | 77     |
| NO_INIT                                 | 20, 78 |
| P_CDATA                                 | 78     |
| P_IDATA                                 | 78     |
| P_UDATA                                 | 78     |
| RCODE                                   | 79     |
| RF_XDATA                                | 79     |
| XSTACK                                  | 81     |
| X_CDATA                                 | 79     |
| X_CONST                                 | 80     |
| X_CSTR                                  | 80     |
| X_IDATA                                 | 80     |
| X_UDATA                                 | 80     |
| setjmp.h (header file)                  | 202    |
| sfr variables                           | 61     |
| sfr (data type)                         | 4, 59  |
| sfr (extended keyword)                  | 118    |
| shared variable objects                 | 143    |
| short, int (data type)                  | 59     |
| signed char (data type)                 | 59     |
| signed (keyword)                        | 140    |
| silent operation                        | 102    |
| size_t                                  | 205    |
| skeleton code, interfacing to assembler | 48     |
| special function register variables     | 61     |
| special.h (header file)                 | 5      |

|                                 |     |
|---------------------------------|-----|
| speed optimization              | 102 |
| sprintf (library function)      | 30  |
| sscanf (library function)       | 31  |
| stack size                      | 28  |
| expansion                       | 103 |
| stack, reentrant functions      | 54  |
| stdarg.h (header file)          | 202 |
| stddef.h (header file)          | 205 |
| stdio.h (header file)           | 202 |
| stdlib.h (header file)          | 202 |
| storage of variables, modifying | 4   |
| string.h (header file)          | 204 |
| struct (keyword)                | 142 |
| symbols, predefined             | 5   |

## T

|                             |     |
|-----------------------------|-----|
| tab spacing                 | 103 |
| target identifier           | 133 |
| Tiny-51                     | 151 |
| configuration               | 156 |
| dispatcher                  | 152 |
| functions                   | 164 |
| idle-tasks                  | 152 |
| installation                | 155 |
| non-preemptive multitasking | 152 |
| preemptive multitasking     | 152 |
| principles of operation     | 153 |
| register bank 3             | 156 |
| restrictions                | 155 |
| round-robin                 | 152 |
| scheduler                   | 152 |
| semaphores                  | 153 |
| signals                     | 153 |
| task states                 | 152 |
| task timer                  | 156 |
| tasks                       | 151 |
| task-switching time         | 156 |
| terminology                 | 151 |
| timeout task                | 156 |

|                   |     |
|-------------------|-----|
| tutorial          | 156 |
| Tiny-51 functions |     |
| create            | 164 |
| down              | 166 |
| signal            | 167 |
| StartDispatcher   | 168 |
| StopDispatcher    | 169 |
| timeout           | 170 |
| up                | 170 |
| wait              | 171 |
| type checking     | 90  |

## U

|                            |     |
|----------------------------|-----|
| union (keyword)            | 142 |
| unsigned char (data type)  | 59  |
| unsigned int (data type)   | 59  |
| unsigned long (data type)  | 59  |
| unsigned short (data type) | 59  |
| using (extended keyword)   | 119 |

## V

|                         |     |
|-------------------------|-----|
| variable initialization | 32  |
| variables, local        | 45  |
| version number          | 134 |
| void (keyword)          | 140 |
| volatile (keyword)      | 140 |

## W

|                                      |          |
|--------------------------------------|----------|
| warning messages                     | 173, 190 |
| warnings, disabling                  | 104      |
| warnings=default (#pragma directive) | 131      |
| warnings=off (#pragma directive)     | 131      |
| warnings=on (#pragma directive)      | 131      |
| write formatter, selecting           | 31       |

## X

|                                    |        |
|------------------------------------|--------|
| xdata (extended keyword)           | 4, 119 |
| xdata (pointer) (extended keyword) | 120    |
| XLINK options                      |        |
| Load as PROGRAM module             | 30     |
| -A                                 | 30     |
| XSTACK (segment)                   | 81     |
| X_CDATA (segment)                  | 79     |
| X_CONST (segment)                  | 80     |
| X_CSTR (segment)                   | 80     |
| X_IDATA (segment)                  | 80     |
| X_UDATA (segment)                  | 80     |
| X3.159-1989 ANSI standard          | 139    |

## Symbols

|                        |         |
|------------------------|---------|
| #pragma directives     | 5, 121  |
| bitfields=default      | 122     |
| bitfields=reversed     | 122     |
| codeseg                | 123     |
| function=default       | 124     |
| function=interrupt     | 124     |
| function=monitor       | 124     |
| function=version_2     | 126     |
| language=default       | 126     |
| language=extended      | 127     |
| memory                 | 20      |
| memory=constseg        | 128     |
| memory=dataseg         | 129–130 |
| memory=default         | 130     |
| memory=no_init         | 130     |
| warnings=default       | 131     |
| warnings=off           | 131     |
| warnings=on            | 131     |
| -A (C compiler option) | 49, 85  |
| -a (C compiler option) | 86      |
| -A (XLINK option)      | 30      |
| -b (C compiler option) | 29, 86  |
| -C (C compiler option) | 87      |

|                              |             |
|------------------------------|-------------|
| -c (C compiler option)       | 60, 87      |
| -D (C compiler option)       | 87          |
| -E (C compiler option)       | 88          |
| -e (C compiler option)       | 4, 89, 126  |
| -F (C compiler option)       | 89          |
| -f (C compiler option)       | 89          |
| -G (C compiler option)       | 90          |
| -g (C compiler option)       | 90          |
| -H (C compiler option)       | 95          |
| -I (C compiler option)       | 96          |
| -i (C compiler option)       | 97          |
| -K (C compiler option)       | 97          |
| -L (C compiler option)       | 49, 97      |
| -l (C compiler option)       | 98          |
| -m (C compiler option)       | 98          |
| -N (C compiler option)       | 98          |
| -n (C compiler option)       | 99          |
| -O (C compiler option)       | 99          |
| -o (C compiler option)       | 95, 100     |
| -P (C compiler option)       | 100         |
| -p (C compiler option)       | 100         |
| -q (C compiler option)       | 49, 100     |
| -R (C compiler option)       | 101         |
| -r (C compiler option)       | 101         |
| -S (C compiler option)       | 102         |
| -s (C compiler option)       | 102         |
| -T (C compiler option)       | 103         |
| -t (C compiler option)       | 103         |
| -U (C compiler option)       | 103         |
| -u (C compiler option)       | 103         |
| -v (C compiler option)       | 104         |
| -w (C compiler option)       | 104, 131    |
| -X (C compiler option)       | 104         |
| -x (C compiler option)       | 105         |
| -y (C compiler option)       | 73, 76, 105 |
| -z (C compiler option)       | 106         |
| .i (file extension)          | 99          |
| .xcl file                    | 27          |
| _args\$ (intrinsic function) | 135         |

|                                         |     |
|-----------------------------------------|-----|
| _argt\$ (intrinsic function)            | 136 |
| _formatted_read (library function)      | 31  |
| _formatted_write (library function)     | 30  |
| _medium_read (library function)         | 31  |
| _medium_write (library function)        | 30  |
| _opc (intrinsic function)               | 137 |
| _small_write (library function)         | 31  |
| _tbac (intrinsic function)              | 137 |
| __DATE__ (predefined symbol)            | 133 |
| __FILE__ (predefined symbol)            | 133 |
| __IAR_SYSTEMS_ICC__ (predefined symbol) | 133 |
| __LINE__ (predefined symbol)            | 133 |
| __STDC__ (predefined symbol)            | 133 |
| __TID__ (predefined symbol)             | 133 |
| __TIME__ (predefined symbol)            | 134 |

## Numerics

|                              |     |
|------------------------------|-----|
| 4-byte floating-point format | 60  |
| 8051 processor option        | 104 |
| 80751 processor option       | 104 |