



PICMICRO™ C COMPILER

Programming Guide

COPYRIGHT NOTICE

© Copyright 1998 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

C-SPY is a registered trademark of IAR Systems.

IAR Embedded Workbench, IAR XLINK Linker, and IAR XLIB are trademarks of IAR Systems. PICmicro is a trademark of Microchip Technology Inc. Windows is a trademark of Microsoft Corp.

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

First edition: October 1998

Part no: ICCPIC-1

WELCOME

Welcome to the PICmicro™ C Compiler Programming Guide.

This guide provides reference information about the IAR Systems C Compiler for the PICmicro™ family of microcontrollers, and applies to both the Embedded Workbench and command line versions of these tools.

Before reading this guide we recommend you refer to the *QuickStart Card*, or the chapter *Installation and documentation route map*, for information about installing the IAR Systems tools and an overview of the documentation.



If you are using the Embedded Workbench refer to the *PICmicro™ Embedded Workbench Interface Guide* for information about running the IAR Systems tools from the Embedded Workbench interface, and complete reference information about the Embedded Workbench commands and dialog boxes, and the Embedded Workbench editor.



If you are using the command line version refer to the *PICmicro™ Command Line Interface Guide* for general information about running the IAR Systems tools from the command line, and a simple tutorial to illustrate how to use them.

For information about programming with the PICmicro™ Assembler refer to the *PICmicro™ Assembler, Linker, and Librarian Programming Guide*.

If your product includes the optional PICmicro™ C-SPY debugger refer to the *PICmicro™ C-SPY User Guide* for information about debugging with C-SPY.

ABOUT THIS GUIDE

This guide consists of the following chapters:

Installation and documentation route map explains how to install and run the IAR Systems tools, and gives an overview of the documentation supplied with them.

The *Introduction to the PICmicro™ C Compiler* provides a brief summary of the PICmicro™ C Compiler's features and describes how the compiler represents each of the C data types. There are also recommendations for efficient coding, and a summary of the available language extensions.

The *Tutorials* illustrates how you might use the C compiler to develop a series of typical programs, and illustrates some of the compiler's most important features. It also describes a typical development cycle using the C compiler.

Configuration then describes how to configure the C compiler for different requirements.

C compiler options explains how to set the C compiler options, gives a summary of the options, and complete reference information about each C compiler option.

General C library definitions gives an introduction to the C library functions, summarizes them according to header file and describes how to build a target-specific library.

C library functions reference then gives reference information about each library function.

Extended keywords reference gives reference information about each of the extended keywords.

#pragma directives reference gives reference information about the #pragma keywords.

Predefined symbols reference gives reference information about the predefined symbols.

Intrinsic functions reference gives reference information about the intrinsic functions.

Assembly language interface describes the interface between C programs and assembly language routines.

Segment reference gives reference information about the C compiler's use of segments.

Implementation-defined behaviour describes how IAR C handles the implementation-defined areas of the C language.

IAR C extensions describes the IAR extensions to the ISO standard for the C programming language.

Diagnostics describes the diagnostic functions and lists the warning and error messages specific to the PICmicro™ C Compiler.

ASSUMPTIONS



This guide assumes that you already have a working knowledge of the following:

- ◆ The PICmicro™ microcontroller.
- ◆ The C programming language.
- ◆ Windows, MS-DOS, or UNIX, depending on your host system.

Note that the illustrations in this guide show the Embedded Workbench running in a Windows 95-style environment, and their appearance will be slightly different if you are using a different platform.

CONVENTIONS

This guide uses the following typographical conventions:

<i>Style</i>	<i>Used for</i>
computer	Text that you type in, or that appears on the screen.
<i>parameter</i>	A label representing the actual value you should type 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.
...	Multiple parameters can follow a command.
<i>reference</i>	A cross-reference to another part of this guide, or to another guide.
	Identifies instructions specific to the versions of the IAR Systems tools for the Embedded Workbench interface.
	Identifies instructions specific to the command line versions of IAR Systems tools.

CONTENTS

INSTALLATION AND DOCUMENTATION	1
Included in this package	1
Installing the Embedded Workbench with C-SPY	1
Installing the command line tools	3
Installed files	4
File types	6
Documentation	7
 INTRODUCTION TO THE PICMICRO™ C COMPILER.....	 9
Key features	9
Data representation	10
Programming hints	16
Language extensions	17
 TUTORIALS.....	 21
Typical development cycle	23
Getting started	23
Tutorial 1	25
Tutorial 2	42
Tutorial 3	46
 CONFIGURATION	 51
Introduction	51
Environment variables	52
Processor group	53
Memory model	53
XLINK command file	54
Run-time library	54
Target-specific header files	55
Stack size	55
Input and output	56
Register I/O	59
Heap size	59
Initialization	60

C COMPILER OPTIONS	63
Setting C compiler options	63
Options summary	65
Language	67
Code	69
Output	72
List	74
Preprocessor	76
Diagnostics	79
Target	82
Output directories	84
Miscellaneous	85
 GENERAL C LIBRARY DEFINITIONS	 87
Introduction	87
Building target-specific libraries	88
Library definitions summary	89
 C LIBRARY FUNCTIONS REFERENCE.....	 97
 EXTENDED KEYWORDS REFERENCE.....	 163
Storage	164
Functions	168
 #PRAGMA DIRECTIVES REFERENCE	 171
Type attribute	171
dataseg	172
constseg	172
Location	173
Function	173
Vector	173
Diagnostics	174
language	174
 PREDEFINED SYMBOLS REFERENCE.....	 175
 INTRINSIC FUNCTIONS REFERENCE	 179

ASSEMBLY LANGUAGE INTERFACE	183
Creating a shell	183
Calling convention	184
Calling assembly routines from C	191
SEGMENT REFERENCE.....	193
IMPLEMENTATION-DEFINED BEHAVIOR	201
Translation	201
Environment	202
Identifiers	202
Characters	202
Integers	204
Floating point	204
Arrays and pointers	205
Registers	205
Structures, unions, enumerations, and bit-fields	205
Qualifiers	206
Declarators	206
Statements	207
Preprocessing directives	207
Library functions	208
IAR C EXTENSIONS	213
DIAGNOSTICS.....	217
Severity levels	217
Messages	218
INDEX	223

INSTALLATION AND DOCUMENTATION

This chapter explains how to install and run the Embedded Workbench and command line versions of the IAR products, and gives an overview of the available documentation. It also describes the `iar` subdirectories and file types.

INCLUDED IN THIS PACKAGE

The PICmicro™ package contains the following items:

- ◆ CD-ROM or floppy disks.
- ◆ Product documentation:
 - PICmicro™ Embedded Workbench Interface Guide
 - PICmicro™ Command Line Interface Guide
 - PICmicro™ Assembler, Linker, and Librarian Programming Guide
 - PICmicro™ C Compiler Programming Guide
 - PICmicro™ C-SPY User Guide, if you have purchased the IAR C-SPY debugger.
- ◆ Licence agreement including the *Product Registration Form*, which we urge you to fill out and send us to ensure that you receive the latest release of the IAR development tools for the PICmicro™ family of microcontrollers.

INSTALLING THE EMBEDDED WORKBENCH WITH C-SPY

This section explains how to install and run the Embedded Workbench with C-SPY.

WHAT YOU NEED

- ◆ Windows 95/98, or Windows NT 3.51 or later.
- ◆ At least 40 Mbytes of free disk space for the Embedded Workbench.
- ◆ 32 Mbytes of RAM recommended for the Embedded Workbench and the IAR C-SPY Debugger.

If you are using C-SPY you should install the Workbench before C-SPY.

INSTALLING FROM WINDOWS 95/98 OR NT 4.0

- 1** Insert the installation CD-ROM or the first installation disk.

If you install from a CD-ROM, follow the instructions on the screen.
If you install from a floppy disk, follow the instructions below:

- 2** Click the **Start** button in the taskbar, then click **Settings** and **Control Panel**.
- 3** Double-click the **Add/Remove Programs** icon in the **Control Panel** folder.
- 4** Click **Install**, then follow the instructions on the screen.

RUNNING FROM WINDOWS 95/98 OR NT 4.0

- 1** Click the **Start** button in the taskbar, then click **Programs** and **IAR Embedded Workbench**.
- 2** Click the **IAR Embedded Workbench** program icon.

INSTALLING FROM WINDOWS NT 3.51

- 1** Insert the first installation disk or the installation CD-ROM.

If you install from a CD-ROM, follow the instructions on the screen.
If you install from a floppy disk, follow the instructions below:

- 2** Double-click the **File Manager** icon in the **Main** program group.
- 3** Click the floppy disk icon in the **File Manager** toolbar.
- 4** Double-click the **setup.exe** icon, then follow the instructions on the screen.

RUNNING FROM WINDOWS NT 3.51

Go to the Program Manager and double-click the **IAR Embedded Workbench** icon.

RUNNING C-SPY

Either:

Start C-SPY in the same way as you start the Embedded Workbench (see above).

Or:

Choose **Debugger** from the Embedded Workbench **Project** menu.

INSTALLING THE COMMAND LINE TOOLS

This section describes how to install and run the command line versions of the IAR Systems tools. You should be familiar with your operating system.

WHAT YOU NEED

- ◆ Windows 95/98, or Windows NT 3.51 or later.
- ◆ At least 35 Mbytes of free disk space.
- ◆ 32 Mbytes of RAM recommended for the IAR applications.

INSTALLATION

- 1 Insert the first installation disk.
- 2 At the command line prompt type `a:\install` and press Enter.
- 3 Follow the instructions on the screen.

When the installation is complete:

- 4 Add the path to the IAR Systems command line executable files to the PATH variable. For a default installation you would add `c:\iar\exe`.

Define the environment variables `APIC_INC`, `C_INCLUDE`, `QPICINFO` and `XLINK_DFLTDIR` specifying the paths to the `inc` and `lib` directories; for example:

```
set APIC_INC=c:\iar\inc\  
set C_INCLUDE=c:\iar\inc\  
set QPICINFO=c:\iar\setup\  
set XLINK_DFLTDIR=c:\iar\lib\
```

RUNNING THE TOOLS

Type the appropriate command at the command line prompt.

For more information refer to the *PICmicro™ C Compiler Programming Guide*, and the *PICmicro™ Assembler, Linker, and Librarian Programming Guide*.

INSTALLED FILES

The installation procedure creates several directories to contain the different types of files used with the IAR Systems development tools. The following sections give a description of the files contained by default in each directory. During installation you have the option to specify other directories than the ones created by default.

DOCUMENTATION FILES

Your installation may include a number of ASCII-format text files (*.txt) containing recent additional information. It is recommended that you read all of these files before proceeding.

ASSEMBLER FILES

The `apic` subdirectory holds the document files and assembler include files for the PICmicro™ Assembler.



The `iar\ewnn\picmicro\apic\tutor` directory contains the files used for the PICmicro™ Assembler tutorials.

MISCELLANEOUS FILES

The `etc` subdirectory holds the XLINK-related files.

EXECUTABLE FILES



The `iar` directory contains the `ewnn.exe` and `cwnn.exe` files. Other executable files are located in the `iar\ewnn\picmicro\bin` directory.



The `bin` subdirectory holds the executable program files.

The installation procedure also includes an addition to the `autoexec.bat` PATH statement, directing having the `bin` subdirectory searched for command files. This allows you to issue a command from any directory.

C COMPILER FILES

The `iccpic` subdirectory holds various source files for basic I/O library routines.



The `iar\ewnn\picmicro\iccpic\tutor` directory contains the files used for the PICmicro™ C Compiler tutorials.

INCLUDE FILES

The `inc` subdirectory holds include files, such as the header files for the standard C library, as well as a specific header file defining SFRs (special function registers) for each supported PICmicro™ derivative. These files are used by the C compiler.

The C compiler searches for include files in the directory specified by the `C_INCLUDE` environment variable. If you set this environment variable to the path of the `inc` subdirectory, as suggested in the installation procedure, you can refer to `inc` header files simply by their base names.

The assembler has an equivalent environment variable, `APIC_INC`.

LIBRARY FILES

The `lib` subdirectory holds library modules used by the C compiler.

XLINK searches for library files in the directory specified by the `XLINK_DFLTDIR` environment variable. If you set this environment variable to the path of the `lib` subdirectory, you can refer to `lib` library modules simply by their basenames.

No library modules are installed; instead the modules should be built for the appropriate microcontroller configuration using the included `Buildlib` utility, see the chapter *General C library definitions*.

Pre-built library modules with standard configuration are supplied in the `\lib` directory on the CD. A library is supplied for each supported microcontroller and is named `c\xxxxx.r39`, where `xxxxx` corresponds to the microcontroller name.

LINKER COMMAND FILES

The `iccpic` subdirectory holds an example linker command file for each supported microcontroller.

FILE TYPES

The PICmicro™ versions of the IAR Systems development tools use the following default file extensions to identify the IAR-specific types of file:

<i>Ext.</i>	<i>Type of file</i>	<i>Output from</i>	<i>Input to</i>
.a39	Target program	XLINK	EPROM, C-SPY, etc
.c	C program source	Text editor	C compiler
.d39	Target program with debug XLINK information		C-SPY, etc
.h	C header source	Text editor	C compiler #include
.inc	Assembler header	Text editor	Assembler #include file
.i39	Compiler/debugger	Processor description file	–
.lst	List	C compiler and assembler	–
.mac	C-SPY macro definition	Text editor	C-SPY
.map	XLINK map	XLINK	–
.mem	Target memory layout	Text editor	C-SPY
.prj	Embedded Workbench project	Embedded Workbench	Embedded Workbench
.r39	Object module	C compiler and assembler	XLINK and XLIB
.s39	Assembler program source	Text editor	Assembler
.xcl	Extended command line	Text editor	XLINK and C compiler



The default extension may be overridden by simply including an explicit extension when specifying a filename.



Note that, by default, XLINK listings (maps) will have the .lst extension, and this may overwrite the listing file generated by the compiler. It is recommended that you explicitly name XLINK map files, for example demo1.map.



Files with the extensions `.ini` and `.cfg` are created dynamically when you install and run the tools. These files contain information about your configuration and other settings.

DOCUMENTATION

THE OTHER GUIDES

The other guides provided with the Embedded Workbench are as follows:

PICmicro™ Command Line Interface Guide

This guide explains how to configure and run the IAR Systems development tools from the command line. It also includes reference information about the command line environment variables.

PICmicro™ Embedded Workbench Interface Guide

This guide explains how to configure and run the IAR Systems development tools from the Embedded Workbench interface. It also includes complete reference information about the Embedded Workbench commands and dialog boxes, and the Workbench editor.

PICmicro™ Assembler, Linker, and Librarian Programming Guide

This guide provides reference information about the PICmicro™ Assembler, XLINK Linker, and XLIB Librarian for use with the Embedded Workbench.

The assembler programming sections include details of the assembler source format, and reference information about the assembler operators, directives, and mnemonics.

The XLINK Linker programming reference sections provide information about the XLINK Linker commands and output formats.

The XLIB Librarian programming sections provide information about the XLIB Librarian commands.

Finally, the guide includes a list of diagnostic messages for each of these tools.

PICmicro™ C-SPY User Guide

This optional guide describes how to use C-SPY Debugger for the PICmicro™ series of microcontrollers, and provides reference information about the features of C-SPY.

In addition to the information contained in the PICmicro™ guides, also online information is available.

ONLINE HELP

From the **Help** menu in the Embedded Workbench and the IAR C-SPY Debugger, you can access the PICmicro™ online information. It contains complete reference information for the PICmicro™ Embedded Workbench, C-SPY, C compiler, assembler, XLINK Linker, and XLIB Librarian.

READ-ME FILES

We recommend that you read the following Read-Me files for recent information that is not included in the guides:

apic.txt
cwpic.txt
ewpic.txt
iccpic.txt
xlink.txt

IAR ON THE WEB

The latest news from IAR Systems is available at the web site **www.iar.com**. You can access the IAR site directly from the Embedded Workbench **Help** menu and receive information about:

- ◆ Product announcements.
- ◆ Special offerings.
- ◆ Evaluation copies of the IAR products.
- ◆ Technical Support including FAQs (frequently asked questions).
- ◆ Links to chip manufacturers and other interesting sites.
- ◆ Distributor information.

INTRODUCTION TO THE PICMICRO™ C COMPILER

The IAR Systems PICmicro™ C Compiler is available in two versions: a command line version, and a Windows version integrated with the IAR Systems Embedded Workbench development environment.

In this chapter you will find information about the compiler's key features and its data representation. There is also a section containing hints on how to write programs efficiently for the PICmicro™ compiler, and information about the language extensions available.

KEY FEATURES

The IAR Systems C Compiler for the PICmicro™ family of microcontrollers offers the standard features of the C language, plus many extensions designed to take advantage of the PICmicro™-specific facilities. The compiler is supplied with the IAR Systems Assembler for the PICmicro™, with which it shares linker and librarian manager tools.

It provides the following features:

LANGUAGE FACILITIES

- ◆ Conformance to the ANSI specification for free-standing C in accordance to what the controller design allows.
- ◆ Standard library of functions applicable to embedded systems, with source available.
- ◆ IEEE-compatible floating-point arithmetic.
- ◆ Powerful extensions for PICmicro™-specific features, including efficient I/O.
- ◆ LINT-like checking of program source.
- ◆ Linkage of user code with assembly routines.
- ◆ Long identifiers – up to 255 significant characters.
- ◆ Up to 32000 external symbols.
- ◆ Maximum compatibility with other IAR Systems C compilers.

PERFORMANCE

- ◆ Fast compilation.
- ◆ Memory-based design which avoids temporary files or overlays.
- ◆ Rigorous type checking at compile time.
- ◆ Rigorous module interface type checking at link time.

CODE GENERATION

- ◆ Selectable optimization for code speed or size.
- ◆ Comprehensive output options, including relocatable binary, ASM, ASM + C, cross-references, etc.
- ◆ Easy-to-understand error and warning messages.
- ◆ Compatibility with the C-SPY high-level debugger.

TARGET SUPPORT

- ◆ Flexible variable allocation.
- ◆ Interrupt functions.
- ◆ `#pragma` directives to maintain portability while using processor-specific extensions.

**DATA
REPRESENTATION**

This section describes how the PICmicro™ C Compiler represents each of the C data types.

The PICmicro™ C Compiler supports all ISO/ANSI C basic elements. Variables are stored with the least significant part located at low memory address.

INTEGER TYPES

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

<i>Data type</i>	<i>Bytes</i>	<i>Range</i>	<i>Alignment</i>
signed char	1	-128 to 127	1
unsigned char	1	0 to 255	1
short, int	2	-2^{15} to $2^{15}-1$	2

<i>Data type</i>	<i>Bytes</i>	<i>Range</i>	<i>Alignment</i>
unsigned short, unsigned int	2	0 to $2^{16}-1$	2
long	4	-2^{31} to $2^{31}-1$	2
unsigned long	4	0 to $2^{32}-1$	2

Enum type

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

Char type

The char type is, by default, unsigned in the compiler, but the ‘**Char**’ is ‘**signed char**’ (- - char_is_signed) option allows you to make it signed. Note, however, that the library is compiled with char types as unsigned.

Bitfields

Char, short, and long bitfields are extensions to the 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.

FLOATING POINT TYPES

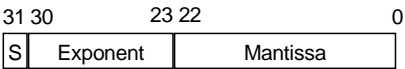
Floating-point values are represented by 4-byte numbers in standard IEEE format; float and double values have the same representation. Floating-point values below the smallest limit will be regarded as zero, and overflow gives undefined results.

The ranges and sizes for the different floating-point types are:

<i>Type</i>	<i>Range</i>	<i>Decimal</i>	<i>Byte</i>	<i>Exponent</i>	<i>Mantissa</i>
float	1.8E-38 to 3.39E + 38	7	4	8	23
double	1.8E-38 to 3.39E + 38	7	4	8	23
long double	1.8E-38 to 3.39E + 38	7	4	8	23

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 4 bytes of zeros.

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

POINTERS

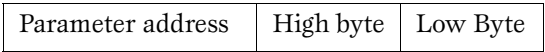
This section describes the PICmicro™ C Compiler’s use of code pointers, data pointers, and constant pointers.

Code pointers

There are four code pointers:

<i>Keyword</i>	<i>Storage in bytes</i>	<i>Restrictions</i>
__bank0_func	3	No restrictions.
__bank1_func	3	No restrictions.
__bank2_func	3	No restrictions.
__bank3_func	3	No restrictions.

The pointers inform the compiler which bank to use for parameters and auto variables. All pointers are three bytes in size. The first byte contains the start address of the parameter area, and the following two bytes contain the actual function address.



The default code pointer is __bank0_func.

Data pointers

<i>Keyword</i>	<i>Storage in bytes</i>	<i>Restrictions</i>
<code>__bank0 – __bank7</code>	1	No restrictions.
<code>__nonbanked</code>	1	No restrictions.
<code>__eeprom</code>	1	Only for microcontrollers with EEPROM (16F84).
<code>__ptable</code>	2	Only for high-end chips.
<code>__rtable</code>	2	Only for high-end chips.
<code>__constptr</code>	2	No restrictions.
<code>__bank</code>	2	No restrictions.
<code>__dptr</code>	3	(Default) No restrictions.

The PICmicro™ C Compiler supports up to eight GPR banks of RAM. These can be accessed through the keywords `__bank0` to `__bank7`. Each of these pointers is one byte in size and points to the area from address 0x20 and up in each bank. To access the SFR area located below 0x20 in each bank you must place the variable using the `@` symbol.

Example:

```
__no_init volatile unsigned char __bank0 STATUS @ 0x03;
```

The `__nonbanked` keyword tells the compiler that bank switching is not necessary to access the variable.

For the microcontrollers with EEPROM (16F84), the `__eeprom` keyword is supplied. All access of the EEPROM is made through the assembly libraries since the access is dependent on the hardware. The libraries must be modified to conform to the hardware before the eeprom access can work correctly. The assembly library is located in the file L07.S39. The pointer is one byte in size.

For high-end chips the two keywords `__ptable` and `__rtable` are supplied. These keywords point to internal memory (`__ptable`) and external memory (`__rtable`). All access is made through the assembly library routines since access is dependent on the actual hardware used. The libraries must be modified to conform to the hardware to before TABLE access can work correctly. The assembly libraries are located in the file L06.S39. The pointers are each two bytes in size and word-addressed with the high byte first and the low byte second.

`__constptr` is used to access variables declared as `const` and is stored in memory as a series of RETLW instructions. `__constptr` is a two-byte word-addressed pointer.

`__bank` is a generic two-byte pointer that can access all eight banks. The first byte tells which bank should be addressed and the second is the address in the bank.

`__dptr` is the generic 3-byte pointer that is able to point to all of the data pointers. `__dptr` is the default pointer for the PICmicro™ C Compiler. The first byte tells which of `__bank`, `__constptr`, `__eeprom`, `__rtable` or `__ptable` it points to.

Casting

Casting an integer to a smaller pointer type will be done by truncation.

Casting a pointer type to larger type will first be done to the largest possible pointer (`__bank` or `__dptr`) that fits in the integer and then if necessary zero extended.

Because of the Harvard architecture of the microcontroller, you can only cast up to `__bank` and `__dptr`. Since all the other pointers point to different memories there is no meaningful way of converting the pointers. Downcasting from `__dptr` and `__bank` generates an error by default, but this error may be overridden to a warning using the **Treat these as warnings** (`-diag_warning`) option. This must only be done if you are really sure casting is feasible in this case.

Function pointers cannot be cast to another type of function pointer. Interrupt routines may not be accessed through function pointers.

size_t

`size_t` is the unsigned integer type required to hold the maximum size of an object. The type used for `size_t` in IAR C is unsigned short.

ptrdiff_t

`ptrdiff_t` is the type of integer required to hold the difference between two pointers to elements of the same array, The `ptrdiff_t` integer type is a signed short in IAR C.

STRUCTURES

Structure members are stored sequentially in the order in which they are declared: the first member has the lowest memory address.

Anonymous structures and unions

An anonymous structure or union is a structure or union object that is declared without a name. Its members are promoted to the surrounding scope. An anonymous structure or union may not have a tag. Example:

```
struct                                /* Anonymous */
{
    char tag;
    union                            /* Anonymous */
    {
        long l;
        float f;
    }
};
void f()
{
    l=5;
}
```

The member names must be unique in the surrounding scope. Having anonymous structures and unions at file scope, as a `global`, `external`, or `static` is also allowed. This is for instance used to declare SFRs, as in the following example:

```
union
{
    char IOPORT;
    struct
    {
        char way: 1;
        char out: 1;
    }
} @ 100;
```

This declares an SFR byte IOPORT at address 100. The SFR has 2 bits declared, `way` and `out`.

PROGRAMMING HINTS

It is important to appreciate the limitations of the PICmicro™ architecture in order to avoid the use of inefficient language constructs. The following is a list of recommendations on how to write efficient code for the PICmicro™ C Compiler.

- ◆ Avoid mixing banks in a function since it increases the number of bank switches.
- ◆ If possible, avoid the use of pointers since access may be slower than with direct access.
- ◆ The PICmicro™ has a limited call depth of 8 or 16 levels. Avoid using long call chains since library calls may use up to 2 levels internally and interrupts adding one level for each interrupt in progress.
- ◆ Bitfields with sizes other than 1 bit should be avoided since they would result in inefficient code compared to bit operations.
- ◆ Avoid use of global variables since they occupy RAM during the whole program. Use autos since they can be overlayed and thus making better use of the limited memory of the PICmicro.
- ◆ Avoid passing objects larger than 3-4 bytes as parameters, pass them as pointers to save RAM. It has the side effect of generating slightly larger and slower code.
- ◆ ANSI prototypes must be used for the overlay model to work.
- ◆ For chips with RAM located below address 0x20 in a bank, the way to access it is to declare a variable with the wanted location. For example:

```
__bank0 char count @0x1E;
```
- ◆ Use small integral types to save memory and time.
- ◆ C functions are not re-entrant or recursive because of the overlay model used.
- ◆ Sensible use of the memory attributes (see the chapter *Extended keyword reference*) can enhance both speed and code size in critical applications.

LANGUAGE EXTENSIONS

This section summarizes the extensions provided in the PICmicro™ C Compiler to support specific features of the PICmicro™ microcontroller.

The extensions are provided in three ways:

- ◆ Extended keywords.

For a complete description of the extended keywords, see the chapter *Extended keywords reference*.

- ◆ #pragma keywords. These provide #pragma directives which control how the compiler allocates memory, whether the compiler allows extended keywords, and whether the compiler outputs warning messages.

For a complete description of the #pragma directives, see the chapter *#pragma directives reference*.

- ◆ Intrinsic functions. These provide direct access to very low-level processor details.

For a complete description of the intrinsic functions, see the chapter *Intrinsic functions reference*.

EXTENDED KEYWORDS

The extended keywords provide the following facilities:

Storage

By default the address range in which the compiler places a variable is in bank0. The program may achieve additional efficiency for special cases by overriding the default by using one of the storage modifiers:

`__bankN`, `__nonbanked`, `const`, `__rtable`, `__ptable`, `__eeprom`

Data pointers are `__bank` and `__dptr`. Function pointers are `__bank0_func`. These defaults can be overridden by use of the following modifiers:

`__bankN`, `__bank`, `__rtable`, `__ptable`, `__eeprom`,
`__constptr`, `__dptr`

Non-volatile RAM

Variables may be placed in non-volatile RAM by using the following data type modifier:

`__no_init`

Absolute variable location

It is possible to specify the location of a variable (its absolute address) by using the `@` operator followed by a constant-expression. See *Absolute location*, page 167, for more information.

The `#pragma location` directive is, however, recommended for specifying an absolute variable location. See *Location*, page 173, for more information.

Functions

By default the compiler passes variables to a function in bank0. To override the default in special cases, use one of the following function modifiers:

◆ `__bankN_func`

Specifies memory area. Used for auto variables and parameters.

◆ `__interrupt`

Specifies interrupt functions. See *Interrupt and trap functions*, page 168.

◆ `__monitor`

Specifies a monitor function.

INTRINSIC FUNCTIONS

Intrinsic functions allow very low-level control of the PICmicro™ microcontroller. To use them in a C application, include the header file `inpic.h`. The intrinsic functions compile into in-line code, either a single instruction or a short sequence of instructions.

For details concerning the effects of the intrinsic functions, see the manufacturer's documentation of the PICmicro™ microcontroller.

<i>Generic intrinsic functions</i>	<i>Description</i>
<code>__no_operation(void)</code>	Generates a no operation, NOP instruction.
<code>__enable_interrupt(void)</code>	Enables (global) interrupts.
<code>__disable_interrupts(void)</code>	Disables (global) interrupts.
<code>__sleep(void)</code>	Generates a SLEEP instruction.
<code>__clear_watchdog_timer(void)</code>	Generates a CLRWDWT instruction.

<i>Generic intrinsic functions</i>	<i>Description</i>
<code>__option(void)</code>	Generates an OPTION instruction. Only for mid-range micro-controllers.
<code>__tris(unsigned char)</code>	Generates a TRIS instruction. Only for mid-range micro-controllers.
<code>__set_configuration_word (unsigned short)</code>	Sets the configuration word for the controller.
<code>asm(const unsigned char *)</code>	Inserts an assembler record.

Inline assembler

The `asm` function assembles and inserts the supplied assembler statement in-line. The statement can include instruction mnemonics, register mnemonics, constants, and/or a reference to a global variable.

Using the `asm` intrinsic function will disable most of the optimization for that function.

Example:

```
asm("MOVF A0,0");
```

TUTORIALS

This chapter illustrates how you might use the PICmicro™ C Compiler to develop a series of typical programs, and illustrates some of the C Compiler's most important features.

Before reading this chapter you should:

- ◆ Have installed the C compiler software; see the *QuickStart Card* or the chapter *Installation and documentation*.
- ◆ Be familiar with the architecture and instruction set of the PICmicro™ microcontroller. For more information, see the manufacturer's data book.

It is also recommended that you complete the introductory tutorial in the *PICmicro™ Embedded Workbench Interface Guide* to familiarize yourself with the interface you are using.

The tutorials in this chapter demonstrate:

- ◆ A typical development cycle.
- ◆ Creating a new project.
- ◆ Compiling and linking a simple program.
- ◆ Interpreting the extended command line file (.xcl) and list files (.lst and .map).
- ◆ Running a program using C-SPY.
- ◆ Using macros and breakpoints.
- ◆ Using the following PICmicro™-specific features: #pragma directives, header files, I/O variables, bit variables, and interrupt functions.

Summary of the tutorials

The following table summarizes the tutorials in this chapter:

<i>Tutorial</i>	<i>What it demonstrates</i>
Tutorial 1	In this tutorial we show how to create and set up your project. We compile, link and finally run a simple program using the C-SPY debugger.

<i>Tutorial</i>	<i>What it demonstrates</i>
Tutorial 2	This tutorial demonstrates how to utilize PICmicro™ peripherals with the IAR C Compiler features. The <code>#pragma</code> directive allows us to use the PICmicro™ specific language extensions. Our program will be extended to handle polled I/O.
Tutorial 3	In this tutorial we modify our initial tutorial program by adding an interrupt handler. The system is extended to handle the real-time interrupt using the PICmicro™ C Compiler intrinsics and <code>__interrupt</code> keyword.

LOCATING THE TUTORIAL FILES

If you follow the default installation, all files used in the tutorials can be found in the following directories:



`\iar\iccpic\tutor`



`\iar\ewnn\picmicro\iccpic\tutor`

We recommend that you create a specific directory where you can store all your project files, for example a `projects` directory on the `c:` drive, `c:\projects\`.

In this tutorial we assume that the following files needed for the project are created and stored in the directory `c:\projects`.

RUNNING THE EXAMPLE PROGRAMS

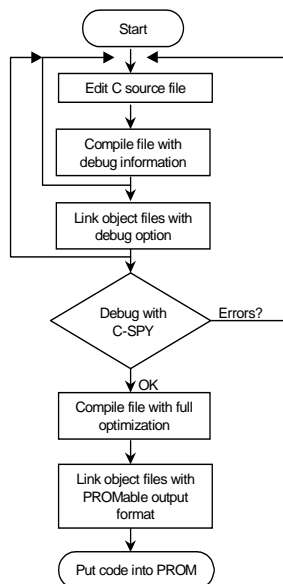
These tutorials show how to run the example programs using the optional C-SPY Simulator from the Embedded Workbench or the command line.

You can also run the examples on a target system with an EPROM emulator and debugger. In this case you will first need to configure the I/O routines.

Alternatively, you may still follow this tutorial by examining the list files created. The `.lst` and `.map` files show which areas of memory to monitor.

TYPICAL DEVELOPMENT CYCLE

Development will normally follow the cycle illustrated below:



The following tutorial follows this cycle.

GETTING STARTED

The first step in developing a project using the C compiler is to decide on an appropriate configuration to suit your target system.

CONFIGURING TO SUIT THE TARGET SYSTEM

During the tutorials, we will use the 17C43 processor set-up file.



To select the processor setup file, open the **Target** options page and select the appropriate file from the list box, see *Target*, page 82.



To select the processor setup file, use the `-v17c43` command line option, see *Processor setup file*, page 83.

Note: Before you start you must set the QPICINFO environment variable to point to the processor set-up files. See *Environment variables*, page 52.

Choosing the linker command file

Each project needs an XLINK command file containing details of the target system's memory map.

A suitable linker command file, `l17c43.xcl`, is provided in the `\iccpic` subdirectory.

Examine `l17c43.xcl` using a suitable text editor, such as the Embedded Workbench editor or the MS-DOS `edit` editor, or print a copy of the file, and verify the entries to see that they match the requirements.

The file first contains the following XLINK command to define the CPU type as PIC:

```
-cpic
```

It then contains a series of `-Z` commands to define the segments used by the compiler. When using the 17C43 processor, the key segments are as follows:

<i>Segment type</i>	<i>Segment names</i>	<i>Address range</i>
CODE	INTVEC	0000-004F
CODE	SCODE	50-1FF
CODE	RCODE (Split into 200H pages.)	[50-1FFF]/200
CODE	HCODE	50-1FFF
CODE	LCODE (Split into 4000H pages.)	[50-1FFF]/4000
CODE	CODE (Split into 200H pages.)	[50-1FFF]/200
CODE	BANK0_ID, BANK1_ID, PTABLE_ID, RTABLE_ID, PTABLE_Z, PTABLE_I, PTABLE_N, CONST	0-1FFF
CODE	CALLSTACK	100000-10001F
DATA	OVERLAY0, INTSAVE0, WRKSEG, BANK0_I, BANK0_Z, BANK0_N	20-0FF
DATA	OVERLAY1, INTSAVE1, BANK1_I, BANK1_Z, BANK1_N	120-1FF

The file then defines the routines to be used for `printf` and `scanf`.

Finally it contains the following line to load the appropriate C library:

```
c117C43.r39
```

See *Run-time library*, page 54, for details of how to produce libraries.

Note that these definitions are not permanent; they can be altered later on to suit your project if the original choice proves to be incorrect, or less than optimal.

For detailed information on configuring to suit the target memory, see *Memory location*, page 53. For detailed information on choosing stack size, see *Stack size*, page 55.

TUTORIAL 1

In this tutorial we show how to create and set up your project. We compile, link and finally run a simple program using the C-SPY debugger.

CREATING A NEW PROJECT

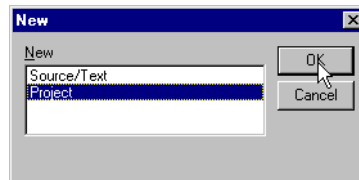
The first step is to create a new project for the tutorial programs.



Creating a new project using the Embedded Workbench

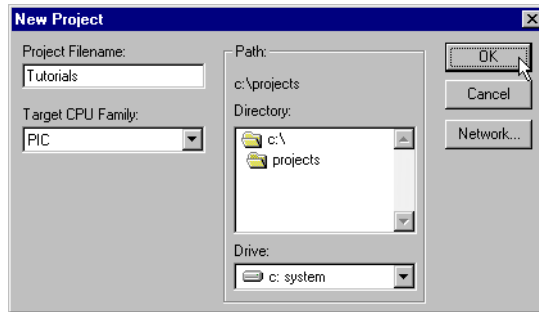
First, run the Embedded Workbench, and create a project for the tutorial as follows.

Choose **New** from the **File** menu to display the following dialog box:



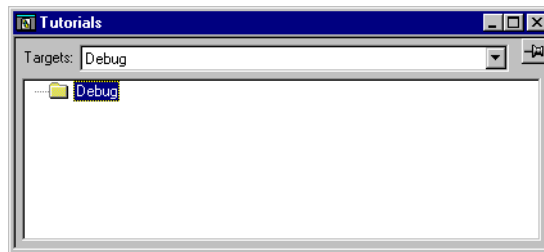
Select **Project** and choose **OK** to display the **New Project** dialog box.

Enter **Tutorials** in the **Project Filename** box, and set the **Target CPU Family** to **PIC**: In the **Path** box you specify where you want to place your project files, for example **c:\projects**:



Then choose **OK** to create the new project.

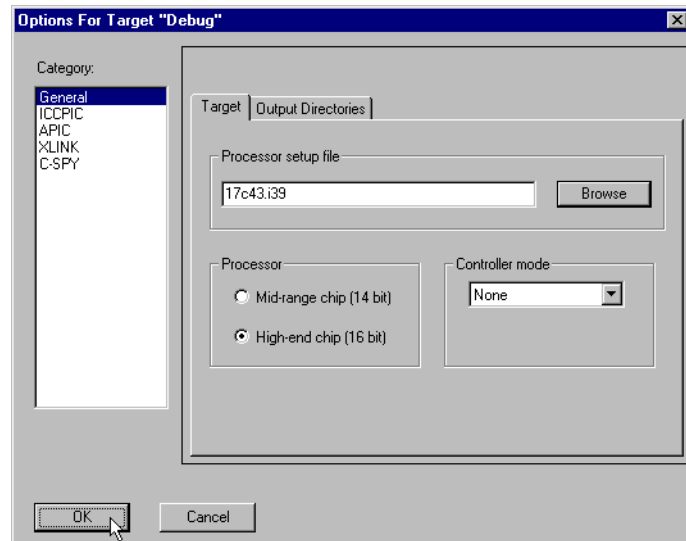
The Project window will be displayed. If necessary, select **Debug** from the **Targets** drop-down list box to display the **Debug** target:



Now set up the target options to suit the processor and memory model we have chosen.

Select the **Debug** folder icon in the Project window, choose **Options...** from the **Project** menu, and select **General** in the **Category** list to display the target options page.

Set the **Processor setup file** to **17C43** and select **High-end chip (16 bit)**:



Then choose **OK** to save the target options.



Creating a new project using the command line

It is a good idea to keep all the files for a particular project in one directory, separate from other projects and the system files.

ENTERING THE PROGRAM

The first program is a simple program using only standard C facilities. It repeatedly calls a function that prints a number series to the Terminal I/O window:

```
#include "tutor.h"

int call_count;

/*
   Increase the 'call_count' variable by one.
*/
void next_counter(void)
{
    call_count += 1;      /* from d_f_p */
}
```

```
}

/*
    Increase the 'call_count' variable.
    Get and print the associated Fibonacci number.
*/
void do_foreground_process(void)
{
    unsigned int fib;
    next_counter();
    fib = get_fib( call_count );
    put_fib( fib );
}

/*
    Main program for tutor1.
    Prints the Fibonacci numbers.
*/
void main(void)
{
    call_count=0;

    init_fib();

    while ( call_count < MAX_FIB )
        do_foreground_process();
}
```



Writing the program using the Embedded Workbench

Choose **New** from the **File** menu to display the New dialog box.

Select **Source/Text** and choose **OK** to open a new text document.

Enter the program given above and save it in a file `tutor.c`.

Alternatively, a copy of the program is provided in the tutorial files directory.



Writing the program using the command line

Enter the program using any standard text editor, such as the MS-DOS `edit` editor, and save it in a file called `tutor.c`. Alternatively, a copy is provided in the C compiler files directory.

You now have a source file which is ready to compile.

COMPILING THE PROGRAM

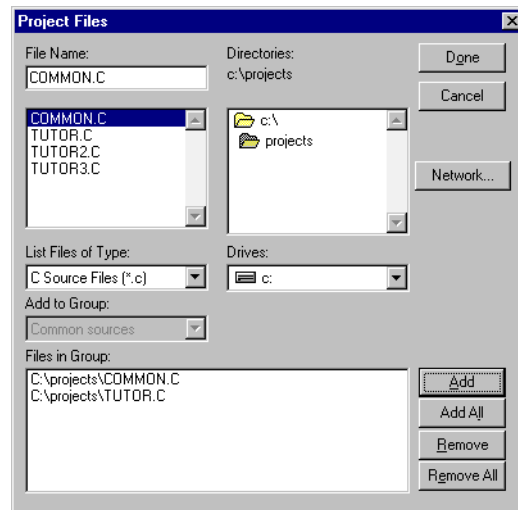


Compiling the program using the Embedded Workbench


To compile the program first add it to the **Tutorials** project as follows.

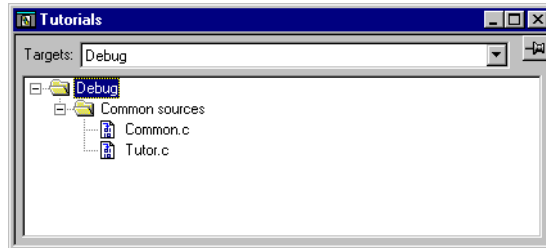
Choose **Files...** from the **Project** menu to display the **Project Files** dialog box. Locate the file `tutor.c` in the file selection list in the upper half of the dialog box, and choose **Add** to add it to the **Common Sources** group.

Locate the file `common.c` and add it to the group. The **Common Sources** group was created by the Embedded Workbench when you created the project. You can read more about groups in the *PICmicro™ Embedded Workbench Interface Guide* or in the on-line help, which can be found under the **Help** menu.



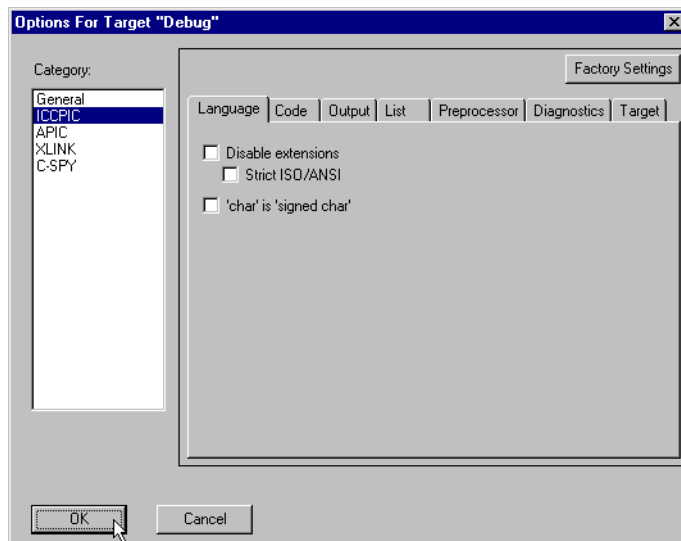
Then click **Done** to close the **Project Files** dialog box.

Click on the  symbol to display the file in the **Project** window tree display:



Then set up the compiler options for the project as follows:

Select the **Debug** folder icon in the **Project** window, choose **Options...** from the **Project** menu, and select **ICCPIC** in the **Category** list to display the C compiler options pages:

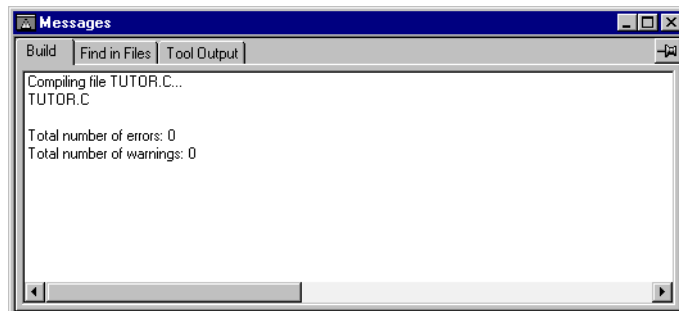


Make sure that the following options are selected on the appropriate pages of the **Options** dialog box:

<i>Page</i>	<i>Options</i>
Output	Generate debug info
List	List file Assembler mnemonics

When you have made these changes choose **OK** to set the options you have specified.

To compile the file select it in the **Project** window and choose **Compile** from the **Project** menu. The progress will be displayed in the **Messages** window:



The Embedded Workbench has now created new directories in your project directory. Since you have chosen the **Debug** target, a Debug directory has been created containing the new directories `List`, `Obj`, and `Exe`. In the `List` directory, your list files from the Debug target will be placed. In the `Obj` directory the object files from the compiler and the assembler will be placed, and in the `Exe` directory you will find the executable file.

The listing is created in a file `tutor.lst`. Open this by choosing **Open...** from the **File** menu, and choosing `tutor.lst` from the `debug\list` directory.



Compiling the program from the command line

To compile the program enter the command:

```
iccpic -v17c43 -r -lC . -e tutor -I\iar\inc\ ↵
```

There are several compile options used here:

<i>Option</i>	<i>Description</i>
-v17c43	Selects PICmicro™ 17C43 processor.
-r	Allows the code to be debugged.
-lC .	Creates a list file with assembler code.
-e	Enables extended commands (not used in this tutorial).
-I	Specifies the pathname for include files.

This creates an object module called `tutor.r39` and a list file called `tutor.lst`.

Compile `common.c` with the same options:

```
iccpic -v17c43 -r -lC . -e common -I\iar\inc\ ↵
```



Viewing the listing

Examine the list file produced and see how the variables are assigned to different segments:

```
#####
#
# IAR PICmicro C-Compiler Vx.xx                dd/Mmm/yyyy  hh:mm:ss #
# (c) Copyright IAR Systems 1998                #
#
# Target          = 17C43                        #
# Source file     = C:\IAR\EWnn\PICmicro\TUTORIAL\tutor.c      #
# Command line    = -I C:\IAR\EWnn\PICMICRO\INC\ -lCN          #
#                  c:\projects\Debug\List\ -o c:\projects\Debug\Obj\ -e #
#                  -z3 --no_cse --no_unroll --no_inline --no_code_motion #
#                  --debug -vC:\IAR\EWnn\PICmicro\SETUP\17c43.i39 #
#                  C:\IAR\EWnn\PICmicro\TUTORIAL\tutor.c      #
# List file       = c:\projects\Debug\List\tutor.lst          #
# Object file     = c:\projects\Debug\Obj\tutor.r39           #
#
#
#####
```

```

\                                     RSEG BANK0_Z:NEARDATA:SORT:NOROOT(0)

1      #include "tutor.h"
2
3      int call_count;
\      call_count:
\ 000000                                DS 2
\ 000002                                REQUIRE __INIT_BANK0_Z

\                                     RSEG CODE:CODE:NOROOT(1)

4
5      /*
6          Increase the 'call_count' variable by one.
7      */
8      void next_counter(void)
\      next_counter:
\ 000000                                REQUIRE ?CLPIC17C43_1_00_L00
9      {
10         call_count += 1;      /* from d_f_p */
\ 000000 B001        MOVLW 1                                ;
\ 000002 BA00        MOVLW 0                                ;
\ 000004 0F..        ADDWF call_count+1,1
\                                     ; (call_count+1,1)
\ 000006 98..        BTFSC _A_ALUSTA,0
\                                     ; (ALUSTA,0)
\ 000008 15..        INCF call_count,1
\                                     ; (call_count,1)
\ 00000A ....        MOVFP call_count,WREG
\                                     ; (call_count,WREG)
\ 00000C ....        MOVFP call_count+1,WREG
\                                     ; (call_count+1,WREG)
11     }
\ 00000E                                ; Overlay area size: 0
\ 00000E 0002        RETURN                                ;
\                                     RSEG CODE:CODE:NOROOT(1)

12
13     /*
14         Increase the 'call_count' variable.
15         Get and print the associated Fibonacci number.
16     */
17     void do_foreground_process(void)
\     do_foreground_process:
\ 000000                                REQUIRE ?CLPIC17C43_1_00_L00
18     {

```

```

19          unsigned int fib;
20          next_counter();
\ 000000          ; Setup parameters for call to function next_counter
\ 000000          ; Total param area: 0 bytes
\ 000000 BA00      MOVLR 0
\ 000002 ....      CALL  A:next_counter
21          fib = get_fib( call_count );
\ 000004          ; Setup parameters for call to function get_fib
\ 000004 BA00      MOVLR 0
\ 000006 ....      MOVFP  call_count,WREG
                      ; (call_count,WREG)
\ 000008 01..      MOVWF  PRM(get_fib,OVERLAY0,0)
                      ; (Param:1 Offset:0)
\ 00000A ....      MOVFP  call_count+1,WREG
                      ; (call_count+1,WREG)
\ 00000C 01..      MOVWF  PRM(get_fib,OVERLAY0,1)
                      ; (Param:1 Offset:1)
\ 00000E          ; Total param area: 2 bytes
\ 00000E ....      CALL  A:get_fib
\ 000010 BA00      MOVLR 0
\ 000012 ....      MOVFP  ?A1,WREG
                      ; (?A1,WREG)
\ 000014 01..      MOVWF  LOC(do_foreground_process,OVERLAY0,2)
                      ; (CTemp_0)
\ 000016 ....      MOVFP  ?A0,WREG
                      ; (?A0,WREG)
\ 000018 01..      MOVWF  LOC(do_foreground_process,OVERLAY0,3)
                      ; (CTemp_1)
\ 00001A ....      MOVFP  LOC(do_foreground_process,OVERLAY0,2),WREG
                      ; (CTemp_0,WREG)
\ 00001C 01..      MOVWF  LOC(do_foreground_process,OVERLAY0,0)
                      ; (fib)
\ 00001E ....      MOVFP  LOC(do_foreground_process,OVERLAY0,3),WREG
                      ; (CTemp_1,WREG)
\ 000020 01..      MOVWF  LOC(do_foreground_process,OVERLAY0,1)
                      ; (fib+1)
22          put_fib( fib );
\ 000022          ; Setup parameters for call to function put_fib
\ 000022 ....      MOVFP  LOC(do_foreground_process,OVERLAY0,0),WREG
                      ; (fib,WREG)
\ 000024 01..      MOVWF  PRM(put_fib,OVERLAY0,0)
                      ; (Param:1 Offset:0)
\ 000026 ....      MOVFP  LOC(do_foreground_process,OVERLAY0,1),WREG
                      ; (fib+1,WREG)

```

```

\ 000028 01..      MOVWF   PRM(put_fib,OVERLAY0,1)
                        ; (Param:1 Offset:1)
\ 00002A          ; Total param area: 2 bytes
\ 00002A ....      CALL    A:put_fib                      ;
23              }
\ 00002C          ; Overlay area size: 4
\ 00002C 0002      RETURN                                ;
\
                        RSEG CODE:CODE:NOROOT(1)
24
25
26      /*
27      Main program for tutor1.
28      Prints the Fibonacci numbers.
29      */
30      void main(void)
\
                main:
\ 000000          REQUIRE ?CLPIC17C43_1_00_L00
31      {
32      call_count=0;
\ 000000 BA00      MOVLRL  0                                ;
\ 000002 29..      CLRF    call_count,1
                        ; (call_count,1)
\ 000004 29..      CLRF    call_count+1,1
                        ; (call_count+1,1)
33
34      init_fib();
\ 000006          ; Setup parameters for call to function init_fib
\ 000006          ; Total param area: 0 bytes
\ 000006 ....      CALL    A:init_fib                      ;
\ 000008 ....      GOTO    A:??main_0                      ;
35
36      while ( call_count < MAX_FIB )
37      do_foreground_process();
\
                ??main_1:
\ 00000A          ; Setup parameters for call to function
do_foreground_process
\ 00000A          ; Total param area: 0 bytes
\ 00000A ....      CALL    A:do_foreground_process        ;
\
                ??main_0:
\ 00000C BA00      MOVLRL  0                                ;
\ 00000E ....      MOVFP   call_count,WREG
                        ; (call_count,WREG)
\ 000010 01..      MOVWF   ?A1
                        ; (?A1)
\ 000012 ....      MOVFP   call_count+1,WREG

```

```

; (call_count+1,WREG)
\ 000014 01..      MOVWF  ?A0
; (?A0)
\ 000016 B080      MOVLW  128
\ 000018 3F..      BTG     ?A1,7
; (?A1,7)
\ 00001A 04..      SUBWF  ?A1,0
; (?A1,0)
\ 00001C 92..      BTFSS  _A_ALUSTA,2
; (ALUSTA,2)
\ 00001E ....      GOTO   A:??main_2
\ 000020 B00A      MOVLW  10
\ 000022 04..      SUBWF  ?A0,0
; (?A0,0)

\      ??main_2:
\ 000024      ; Cond:LTS
\ 000024 9A..      BTFSC  _A_ALUSTA,2
; (ALUSTA,2)
\ 000026 ....      GOTO   A:??main_4
\ 000028 98..      BTFSC  _A_ALUSTA,0
; (ALUSTA,0)
\ 00002A ....      GOTO   A:??main_4
\ 00002C ....      GOTO   A:??main_1
38      }

\      ??main_4:
\ 00002E      ; Overlay area size: 0
\ 00002E ....      GOTO   A:?C_EXIT
\      RSEGA BANK0_A:NEARDATA:NOROOT,04H
\      _A_ALUSTA:
\ 000000      DS 1
\      RSEGA BANK0_A:NEARDATA:NOROOT,0aH
\      WREG:
\ 000000      DS 1

```

```

2 bytes in segment BANK0_A
2 bytes in segment BANK0_Z
110 bytes in segment CODE

```

```

110 bytes of CODE memory
2 bytes of NEARDATA memory (+ 2 bytes shared)

```

```

Errors: none
Warnings: none

```

LINKING THE PROGRAM



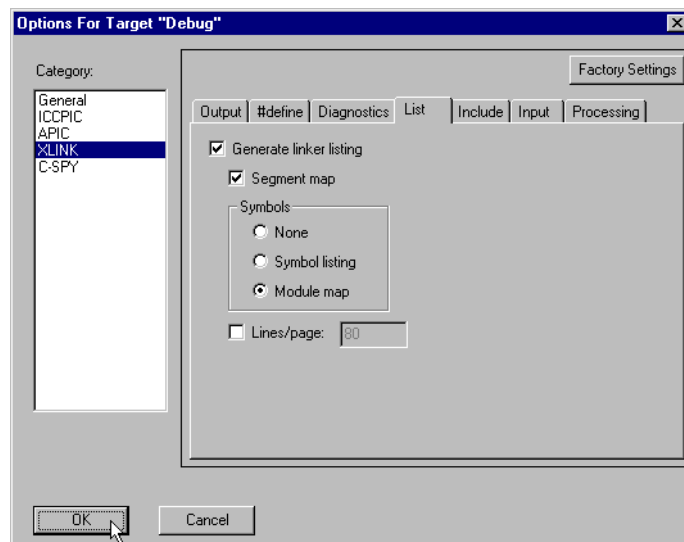
Linking the program using the Embedded Workbench

First set up the options for the XLINK Linker. Select the **Debug** folder icon in the **Project** window, choose **Options...** from the **Project** menu, and select **XLINK** in the **Category** list to display the XLINK options pages.

Click **Output** to make sure that the **Debug with terminal I/O** is chosen.

Then click **List** to display the page of list options.

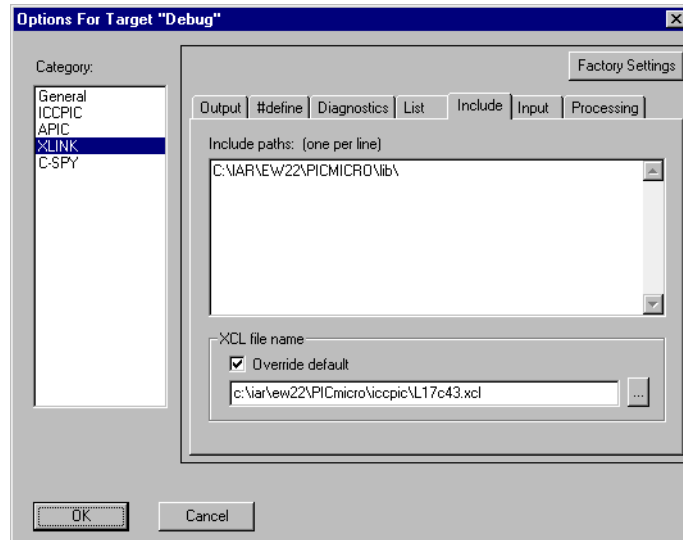
Select **Generate linker listing**, **Segment map**, and **Module map** to generate a map file to `tutorials.map`.



Select the **Include** tab. In the XCL file name options, select **Override default**, and set the XCL file name to:

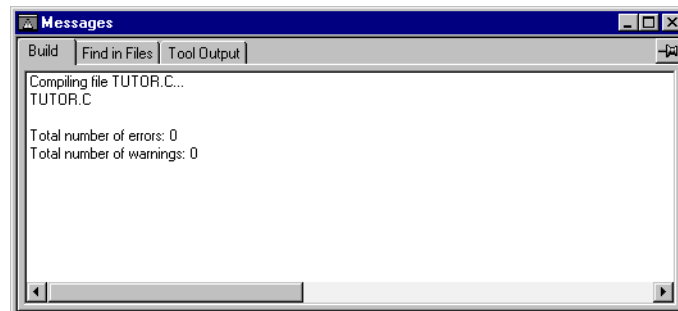
```
c:\iar\ewnn\picmicro\iccpic\l17c43.xcl
```

(or whatever is the appropriate path for your installation). This specifies the linker command file `l17c43.xcl`, designed for this tutorial.



Then choose **OK** to save the XLINK options.

To link the object file to generate code that can be debugged choose **Link** from the **Project** menu. The progress will be displayed in the **Messages** window:



The result of linking is a code file `tutorials.d39` and a map file `tutorials.map`.



Linking the program from the command line

To link the object file with the appropriate library module to produce code that can be executed by the C-SPY debugger, enter the command:


```
xlink tutor common -f l17c43 -rt -xms -l tutorials.map -o
tutorials ↵
```

The `-f` option specifies your XLINK command file `l17c43`, and the `-rt` option allows the code to be debugged with terminal I/O.

The `-x` creates a map file and the `-l filename` gives the name of the file.

The result of the linking is a code file called `tutorials.d39` and a map file called `tutorials.map`.



Viewing the map file

Examine the map file to see how the segment definitions and code were placed into their physical addresses. The main points of the map file are shown in the following table:

<i>Main points</i>	<i>Description</i>
Command line	Equivalent command line.
Included XCL file	Command included in the linker command file.
Program entry	Shows the address of the program entry
Module map	Information about each module that was loaded as part of the program.
File map	Shows the name of the file from which modules were linked. Part of the program.
Module	Type and name.
Segments in the module	A list of segments in the specified module, with information about each segment.
Entries	Global symbols declared within the segment.
Next module	Information about the next module in the current file.
Segments in address order	Lists all the segments that make up the program, in the order linked.

Notice that, although the link file specified the address for all segments, many of the segments were not used. The most important information about segments is at the end, where their address and range is given.

Several entry points were described that do not appear in the original C code. The entry for ?C_EXIT is from the CSTARTUP module. The putchar entry is from the library file.

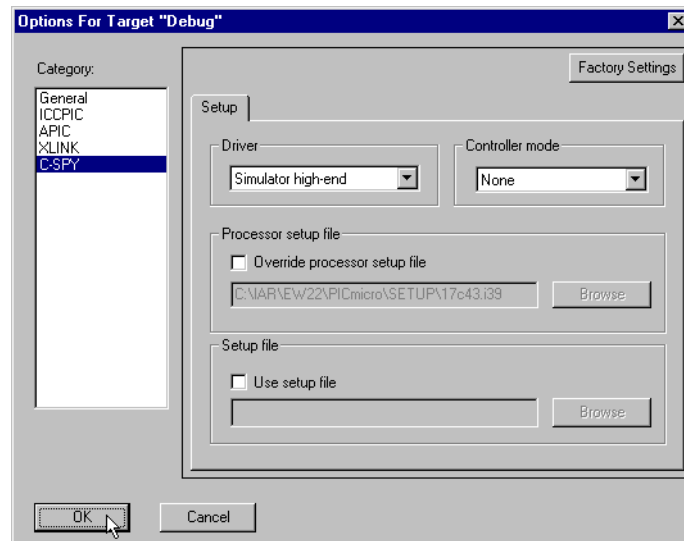
RUNNING THE PROGRAM



Running the program using the Embedded Workbench

First set up the options for the C-SPY debugger. Select the **Debug** folder icon in the **Project** window, choose **Options...** from the **Project** menu, and select **C-SPY** in the **Category** list to display the C-SPY options pages.

Select the **Simulator high-end** driver in the **Driver** list:




Then choose **OK** to save the C-SPY options.

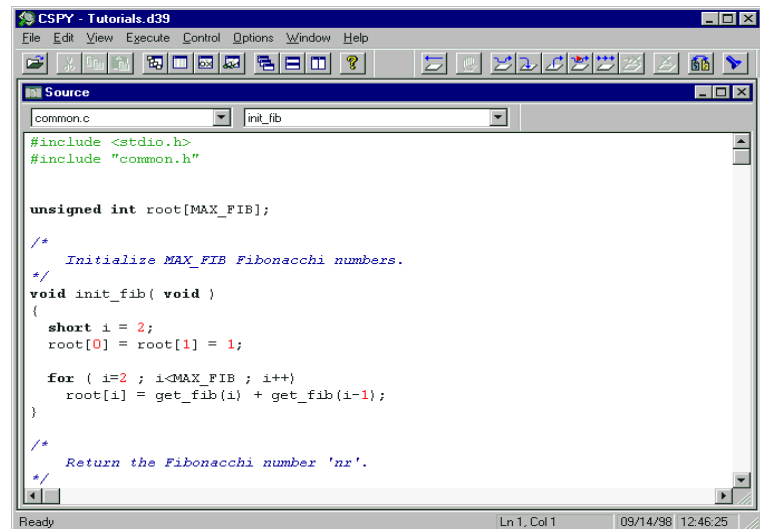
To run the program using the C-SPY debugger choose **Debugger** from the **Project** menu. The C-SPY window will be displayed.



Choose **Step** from the **Execute** menu, or click the **Step** button in the toolbar, to display the source in the Source window.

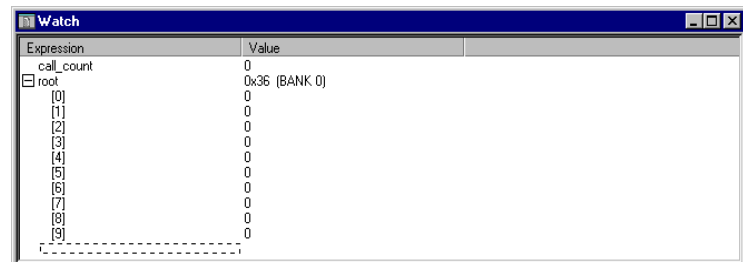
Now use the Watch window to monitor the value of `call_count` and `root` as follows. Choose **Watch** from the **Window** menu to open the Watch window.

Then place the cursor on the empty expression line and type `call_count`  to add this variable to the Watch window:

Select the file `common.c` in the file box (leftmost box) in the Source window. Next choose the function `init_fib` in the function box (rightmost box) in the Source window.



Double-click on the `root` variable. With the `root` variable marked, drag and drop the `root` variable in the Watch window. The `root` variable should be right below the `call_count` in the Watch window. Just to the left of the `root` variable there is a  sign. Click on the  to see the contents of the `root` variable array:



Step to the `init_fib()` function and choose **Step into** from the **Execute** menu. The Source window shows the `init_fib` function. As you step through the `init_fib` function you can watch the root array being filled. Notice that C-SPY lets you follow each statement in the for loop.

Open the Terminal I/O window from the **Window** menu and run by choosing **Go** from the **Execute** menu.

MODIFYING THE COMPILE AND LINK OPTIONS

Different compile or link options will produce similar output but with different memory locations.

TUTORIAL 2

This tutorial will demonstrate how to use the PICmicro™ serial port 1 using the IAR C Compiler features. The `#pragma` directive allows us to use the PICmicro™-specific language extensions. Our program will be extended to handle polled I/O.

ENTERING THE SERIAL PROGRAM

The following is a complete listing of the program. Enter it into a suitable text editor and save it as `tutor2.c`. Alternatively, a copy is provided in the tutorial subdirectory:

```
#include <stdio.h>
#include "tutor2.h"

/*****
 *          Start of code          *
 *****/
int call_count;

void init_cntr(void)
{
    SPBRG = 25;                // set 9600 baud
    TXSTA = 0x20;              // 8-bit transmission, async
mode
    RCSTA = 0x90;              // 8-bit reception, enable
serial port
}
```

```
char receive_ok(void)
{
    if ( RCIF )           // Check if we have received a
byte
    {
        RCREG;           // Read byte to enable receipt of
more
                           // bytes
        return( 1 );
    }
    return( 0 );
}

/*
    Increase the 'call_count' variable by one.
*/
void next_counter(void)
{
    call_count += 1;      /* from d_f_p */
}

void do_foreground_process(void)
{
    unsigned int fib;

    /* wait for receive data */
    if ( receive_ok() )
    {
        next_counter();
        fib = get_fib( call_count );
        put_fib( fib );
    }
    else
        putchar( '.' );
}

void main(void)
{
    /* Initialize comms channel */
    init_cntr();
}
```

```

init_fib();
/* now loop forever, taking input when ready */
while ( call_count < MAX_FIB)
    do_foreground_process();
}

```

Tutor2.c includes a header file called tutor2.h. The first lines of the tutor2.h are:

```

#include "common.h"
#include "io17c43.h"

```

The file io17c43.h includes predefined special function registers (SFRs) and interrupt routines.

COMPILING AND LINKING THE SERIAL PROGRAM



Compiling and linking using the Embedded Workbench

Create a new project containing the file tutor2.c and we will now compile and link with debug information.

Choose **Files...** from the **Project** menu. In the dialog box **Project File**, mark the file tutor.c in the **Files in Group** box. Click on the **Remove** button to remove the tutor.c file. In the **File Name** edit box select your tutor2.c file and click on the **Add** button. Now the **Files in Group** should contain the two files common.c and tutor2.c. Click on the **Done** button to exit the **Project Files** dialog box.

Then compile and link the project by choosing **Make** from the **Project** menu.



Compiling and linking the program from the command line

Compile and link the program as follows:

```

iccpic -v17c43 -r -lC . -e tutor2 -I\iar\inc\ ↵
iccpic -v17c43 -r -lC . -e common -I\iar\inc\ ↵

xlink tutor2 common -f 117c43 -rt -xms -l tutorials.map
-o tutorials ↵

```

RUNNING THE SERIAL PROGRAM



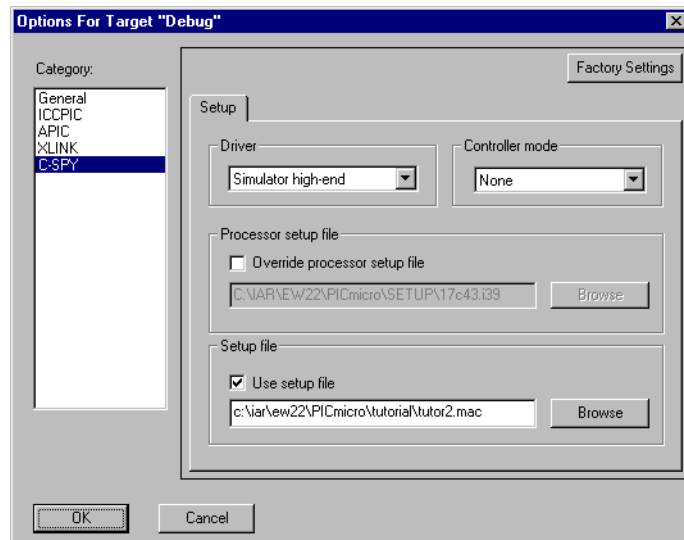
Running the program using the Embedded Workbench

First set up the options for the C-SPY. Select the **Debug** folder icon in the **Project** window, choose **Options...** from the **Project** menu, and select **C-SPY** in the **Category** list to display the C-SPY options pages.

Set the **Use setup file** to:

```
c:\iar\ewnn\picmicro\iccpic\tutor\tutor2.mac
```

(or whatever is the appropriate path for your installation). This specifies the setup file tutor2.mac, designed for this tutorial.



Check that the **Drive** is set to **Simulator high-end**.

Then choose **OK** to save the C-SPY options.

Start C-SPY and step through the program. Stop when it reaches the `while` loop, which waits for input.

To simulate different values for the RCIF register you will make a new virtual register called PIR.

Choose **Settings...** from the **Options** menu. In the **Register Setup** dialog box, click the **New** button to add a new register. Now the **Virtual Register** dialog box will appear. Enter the name of your register, for example PIR. Enter the following values in the dialog box:

Size:	1
Base:	16
Address:	116
Segment:	Bank1

Then choose **OK** in both dialog boxes. Open the Register window from the **Window** menu. PIR should be included in the displayed list of registers.

As you step through the program you can enter new values into PIR in the Register window. When bit 0 (0x01) is set in the PIR a new number should be printed. When the bit has been cleared a period (.) will be printed instead. Then choose **Close** to exit the C-SPY window.

TUTORIAL 3

We shall now modify the first tutorial program by adding an interrupt handler. The PICmicro™ C Compiler lets you write interrupt handlers directly in C using the `interrupt` keyword. The interrupt we will handle is the serial interrupt.

ENTERING THE INTERRUPT PROGRAM

The following is a complete listing of the interrupt program. The program is provided in the sample tutorials as `tutor3.c`.

```
#include <stdio.h>
#include "tutor3.h"

/*****
 *          Start of code          *
 *****/
int call_count=0;
unsigned char flag = 0;
int val;
// enable use of extended keywords
#pragma language=extended

void init_fib( void );
```



```
void init_cntr(void)
{
    SPBRG = 25;           // set 9600 baud
    TXSTA = 0x20;         // 8-bit transmission, async
mode
    RCSTA = 0x90;         // 8-bit reception, enable
serial port
    RCIE  = 1;            // Enable receive interrupt
    PEIE  = 1;            // Enable peripheral interrupts
}

// Set the interrupt vector used by tutor3
#define TUTOR_INTERRUPT_VECTOR 0x20

// do_foreground_process is now declared as
// a monitor function to disable interrupts
// during it's execution to prevent the
// interrupt routine to update the variables
// val and flag while we access them

__monitor void do_foreground_process(void)
{
    if ( flag )
    {
        put_fib( val ); // Display fetched number
        flag = 0;
    }
    else
        putchar ( '.' );
}

#pragma vector=TUTOR_INTERRUPT_VECTOR
__interrupt void tutor_interrupt(void)
{
    call_count += 1; // Increase counter
    val = RCREG;     // store fetched number
}
```

```

    flag = 1;          // and set flag
}

void main(void)
{
    // Initialize comms channel
    init_cntr();

    init_fib();
    __enable_interrupt();

    // now loop forever, taking input when ready
    while ( call_count < MAX_FIB)
        do_foreground_process();

    // Disable interrupts before exit
    __disable_interrupt();
}

```

The intrinsic include file `inpic.h` must be present to define the `enable_interrupt()` function. The `tutor3.h` header file includes the `inpic.h` header file.

```

#include "common.h"
/* Include intrinsic functions,
such as enable_interrupt() */
#include <inpic.h>

```

The interrupt function itself is defined by the following lines:

```

#define TUTOR_INTERRUPT_VECTOR 0x20

#pragma vector=TUTOR_INTERRUPT_VECTOR
__interrupt void tutor_interrupt(void)
{
    call_count += 1; // Increase counter
    val = RCREG;    // store fetched number
    flag = 1;      // and set flag
}

```

The action of this program is to store the number fetched and set the flag used by the `do_foreground_process` function which prints the fetched number.

The `__interrupt` keyword is described in the chapter *Extended keyword reference*.

COMPILING AND LINKING THE PROGRAM





Compiling and linking the program using the Embedded Workbench

Compile and link the program as before, by removing `tutor2.c` from the **Tutorials** project and adding `tutor3.c` to the project and choosing **Make** from the **Project** menu.



Compiling and linking the program from the command line

Compile and link the program as before:

```
iccpic -v17c43 -r -lC . -e tutor3 -I\iar\inc\ 
xlink tutor3 common -f l17c43 -rt -xms -l tutorials.map
-o tutorials 
```



VIEWING THE INTERRUPT PROGRAM

First set up the options for the C-SPY. Select the **Debug** folder icon in the **Project** window, choose **Options...** from the **Project** menu, and select **C-SPY** in the **Category** list to display the C-SPY options pages.

Set the **Use setup file** to:

```
c:\iar\ewnn\picmicro\iccpic\tutor\tutor3.mac
```

(or whatever is the appropriate path for your installation).

Check that the **Drive** is set to **Simulator high-end**.

Then choose **OK** to save the C-SPY options.

The `tutor3.mac` uses `tutor3.txt` as data source for the simulated input to RCREG. You must edit the `tutor3.mac` to include the actual path to `tutor3.txt`. Open the `tutor3.mac` from the Embedded Workbench by choosing **Open** from the **File** menu. The `tutor3.mac` is by default installed in `c:\iar\ewnn\picmicro\iccpic\tutor\`.

Edit the following line in `tutor3.mac` to point to the location of `tutor3.txt`:

```
__openFile(_FileHandle,
"c:\\iar\\ewnn\\picmicro\\iccpic\\tutor\\tutor3.txt", "r");
```

Then save the changes and close the file.

Start C-SPY and step through the program. Stop when it reaches the `while` loop, which waits for input.

In the Source window select `tutor3.c` in the **Source file** box and `tutor_interrupt` as the function in the **Function** box. Place the cursor on the `call_count += 1;` statement in the `tutor_interrupt` function. Set a breakpoint by selecting **Toggle Breakpoint** from the **Control** menu.

The setup file `tutor3.mac` adds a simulated interrupt to C-SPY. To view the settings, select **Interrupt...** from the **Control** menu.

The **Interrupt** dialog box will appear. We only use three of the dialog's edit boxes in this tutorial. For more information regarding the simulated interrupt, see the manufacturer's manual for your specific microcontroller and the `cwpic.txt` read-me file.

The Vector is the vector address for the target, in our case `RCIF`.

The activation time shows when the first activation is to occur and the Repeat Interval is the time in cycles from the first interrupt until the next interrupt.

Choose **Close** to exit the **Interrupt** dialog box.



Now you have a breakpoint in the interrupt function and an interrupt that will be simulated every 650 cycles. Run the program by choosing **Go** from the **Execute** menu or pressing the **Go** button. You should stop in the interrupt function. Choose **Go** again in order to see the next number being printed in the Terminal I/O window.

CONFIGURATION

This chapter describes how to configure the C compiler for different requirements.

INTRODUCTION

The PICmicro™ family has separate address spaces for code and data.

Systems based on the PICmicro™ family can vary considerably in their use of ROM and RAM, and in their stack requirements. They also differ in their need for libraries. The link options specify:

- ◆ The ROM areas: used for functions, constants, and initial values.
- ◆ The RAM areas: used for stack and variables.

Processors with no external bus have only one ROM area, located in the program address space. They also have only one RAM area, located in the internal data address space.

Processors with external data and address buses can have ROM, RAM, EEPROM, and other areas in the code address space, in addition to other internal ROM areas.

Each feature of the environment or usage is handled by one or more configurable elements of the compiler packages, as follows:

<i>Feature</i>	<i>Configurable elements</i>	<i>See</i>
Environment variables	Options settings	page 52
Processor configuration	Compiler option, XLINK, command file (including run-time library)	page 53
Memory location	XLINK command file	page 53
Non-volatile RAM	XLINK command file	page 53
Stack size	XLINK command file	page 55
putchar and getchar functions	Run-time library module	page 56
printf/scanf facilities	XLINK command file	page 57

<i>Feature</i>	<i>Configurable elements</i>	<i>See</i>
Heap size	heap.c	page 59
Hardware /memory initialization	__low_level_init module	page 60

The following sections describe each of the above features. Note that many of the configuration procedures involve editing the standard files, and you may want to make copies of the originals before beginning.

ENVIRONMENT
VARIABLES

The method of setting environment variables differ depending on the platform you are using. For information on how to set environment variables from the command line, see *Installation*, page 3. For information on how to set them on a Windows platform, see the documentation to your operating system.

The following environment variables can be used by the PICmicro™ C Compiler:

<i>Environment variable</i>	<i>Description</i>
C_INCLUDE	Specifies directories to search for include files; for example: C_INCLUDE=c:\iar\inc;c:\headers
QCCPIC	Specifies command line options; for example: QCCPIC = -lA . myfile -s9 <i>Note:</i> In Windows 95 and DOS only one equal sign (=) is allowed on the same line. The # sign can be used instead, for example: set QCCPIC=-DBUFSIZE#4
QPICINFO	Specifies the location of microcontroller set-up files (*.i39). The path must end with a backslash. For example: QPICINFO=c:\iar\setup\

For information on assembler environment variables, see the *Assembler, Linker, and Librarian Programming Guide*.

PROCESSOR GROUP

The PICmicro™ family of microcontrollers has many variants, which the PICmicro™ C Compiler currently divides into the groups described below.

SPECIFYING THE PROCESSOR OPTION

Your program may only use one processor option at a time, and the same processor option must be used by all user modules and all library modules.

To specify the processor option to the compiler when a user module is compiled, you use one of the target options listed in the file `target.txt`. See also *Processor setup file*, page 83, for a list of supported targets at the time of the printing of this user guide.

Example:

Processor option	Command line	Processors supported
16C61	-v16c61	16C61

For example, to compile `myprog` for use on the 16F84, select the **16F84** option or use the command:

```
iccpic myprog -v16f84 ↵
```

MEMORY MODEL

MEMORY LOCATION

You must specify to XLINK your hardware environment’s address ranges for ROM and RAM. You would normally do this in your copy of the XLINK command file template.

For information about how to specify the memory address ranges, see the contents of the XLINK command file template and *XLINK Linker* in the *PICmicro™ Assembler, Linker, and Librarian Programming Guide*.

NON-VOLATILE RAM

The compiler supports the declaration of variables that are to reside, or do not need to be initiated, in non-volatile RAM through the `__no_init` type modifier and the object attribute `#pragma`. The compiler places such variables in separate `no_init` segments depending on which memory keyword used. The memory segments should be 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 XLINK command file. For details of assigning a segment to a given address, see *XLINK Linker* in the *PICmicro™ Assembler, Linker, and Librarian Programming Guide*.

XLINK COMMAND FILE

To create an XLINK command file for a particular project you should first copy the appropriate supplied template from the `\iccpic` directory.

A template is supplied for each of the supported processors and specify the memory ranges for the segments and the name of the library used. The files are called `lprocessor.xcl`.

You should modify these files to specify the details of the target system's memory map. How to do this is described in the files.

RUN-TIME LIBRARY

The XLINK command line files are included and need to be modified to include the appropriate library for the options being used.

No built library modules are installed, instead you should build the modules for the appropriate microcontroller configuration using the included `Buildlib` utility. To build the libraries, you are required to specify target options for the C Compiler and Assembler as well as specify if you are using external memory. For full details, see *Building target-specific libraries*, page 88.

Pre-built library modules with standard configuration are supplied in the `\lib` directory on the CD-ROM. A library is supplied for each supported microcontroller and is named `clxxxx.r39`, where `xxxxx` corresponds to the microcontroller name.

TARGET-SPECIFIC HEADER FILES

Predefined special function registers (SFRs) are given in the following header files:

<i>File</i>	<i>Processor</i>	<i>File</i>	<i>Processor</i>
io16c61.h	16C61	io16c711.h	16C711
io16c62a.h	16C62A	io16c715.h	16C715
io16c621.h	16C621	io16c72.h	16C72
io16c622.h	16C622	io16c73.h	16C73
io16c63.h	16C63	io16c74a.h	16C74A
io16c64a.h	16C64A	io16c76.h	16C76
io16c641.h	16C641	io16c77.h	16C77
io16c642.h	16C642	io16f84.h	16F84
io16c65a.h	16C65A	io16c923.h	16C923
io16c66a.h	16C66	io16c924.h	16C924
io16c661.h	16C661	io17c42a.h	17C42A
io16c662.h	16C662	io17c43.h	17C43
io16c67.h	16C67	io17c44.h	17C44
io16c71.h	16C71	io17c752.h	17C752
io16c710.h	16C710	io17c756.h	17C756

These files are provided in the `inc` subdirectory.

STACK SIZE

Since the PICmicro™ only has a call stack, the linker can only try and make sure that the call depth does not exceed the depth of the microcontroller, 8 or 16 levels deep. The address range is not important as long as it is outside the regular address space used by the other segments.

Example for PIC17C43:

```
-Z(CODE)=CALLSTACK=100000-10001F
```

Notice that the actual size is 32 bytes in order to hold 16 word addresses.

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 `\iccpic\putchar.c` and `\iccpic\getchar.c`.

Customizing putchar

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

- ◆ 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 a fictive LCD display.

```
#include <stdio.h>
__no_init volatile __bank0 unsigned char LCD_IO @ 0x12;

int putchar(int outchar)
{
    LCD_IO = outchar;
    return ( outchar );
}
```

- ◆ Compile the modified `putchar` using the appropriate processor option and the **Make a LIBRARY module** (`--library_module`) option.

For example, if your program uses the 17C43 microcontroller, compile `putchar.c` from the command line with the command:

```
iccpic putchar --library_module -v17c43 ↵
```

This will create an optimized replacement object module file named `putchar.r39`.

- ◆ 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 small-memory-model library, use the command:

```
xlib
```

```
def-cpu pic
rep-mod putchar c117c43
exit
```

The library module `c117c43` will now have the modified `putchar` instead of the original one. (Make sure you save your original `c117c43.r39` file before you overwrite the `putchar` module.)

Note that XLINK allows you to test the modified module before installing it in the library by using the **Load as PROGRAM module** (-A) XLINK option. Place the following lines into your `.xcl` link file:

```
-A putchar
c117c43
```

This causes your version of `putchar.r39` to load instead of the one in the `c117c43` library. See the *PICmicro™ Assembler, Linker, and Librarian Programming Guide*. Note 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` and `llget.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.

REGISTER I/O

A program may access the PICmicro™ I/O system using the memory-mapped internal special function registers (SFRs).

All operators that apply to integral types may be applied to SFR registers. Predefined define declarations for the PICmicro™ family are supplied; see *Target-specific header files*, page 55.

HEAP SIZE

If the library functions `malloc` or `calloc` are used in the program, the C compiler creates a heap or memory from which their allocations are made. The default heap size is 2000 bytes, but the build lib sets it to 20 bytes in `bank0`.

The procedure for changing the heap size is described in the file `\etc\heap.c`.

You can test the modified heap module using the **Load as PROGRAM module** (-A) option, by including the following lines in the `.xcl` link file:

```
-A heap
c117c43
```

This will load your version of `heap.r39` instead of the one in the `c117c43` library.

INITIALIZATION

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

- ◆ Initializes C file-level and static variables.
- ◆ Jumps to 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. Use it for your initialisations.

The source of `__low_level_init` is provided in the file `lowinit.c`, by default located in the `iccpic` directory. To perform your own I/O initializations, create a version of this routine containing the necessary code to do the initializations to disable init segment: Pass the option `-DIGNORE_SEG_INIT` to the assembler.

Rebuild the library to include your modifications or just link in your assembled CSTARTUP and `lowinit`.

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:

- ◆ Make any required modifications to the assembler source of CSTARTUP, supplied by default in the file:


`\iccpic\cstartup.s39`

and save it under the same name.

- ◆ Assemble CSTARTUP using options that match your selected compiler options, as follows:

<i>Compiler option</i>	<i>Assembler option</i>
<i>-vsetupfile</i>	<i>-vm -vh (midrange high-end)</i>

For example, if you have compiled for the 17C43 instruction set (high-end), you must assemble with the command:

apic cstartup -vh 

CSTARTUP also includes a generic I/O file, *io.inc*. Copy the appropriate *iochip.inc* file to *io.inc*, in this case, *io17c43.inc* to *io.inc*.

This will create an object module file named *cstartup.r39*.

You should then use the following commands in the linker command file to make XLINK use the CSTARTUP module you have defined instead of the one in *library*:

-A cstartup	Load as PROGRAM module
-C c117c43	Load as LIBRARY module

The modified *cstartup* will now be used instead of the default one in the library.

The XLINK options are described in the *PICmicro™ Assembler, Linker, and Librarian Programming Guide*.



In the Embedded Workbench add the modified *cstartup* file to your project, and add -C before the library in the linker command file.

C COMPILER OPTIONS

This chapter explains how to set the C compiler options from the Embedded Workbench or the command line, and gives full reference information about each option.

The options are divided into the following sections, corresponding to the pages in the **ICCPIC** and **General** options in the Embedded Workbench:

Language	Preprocessor
Code	Diagnostics
Output	Target
List	Miscellaneous

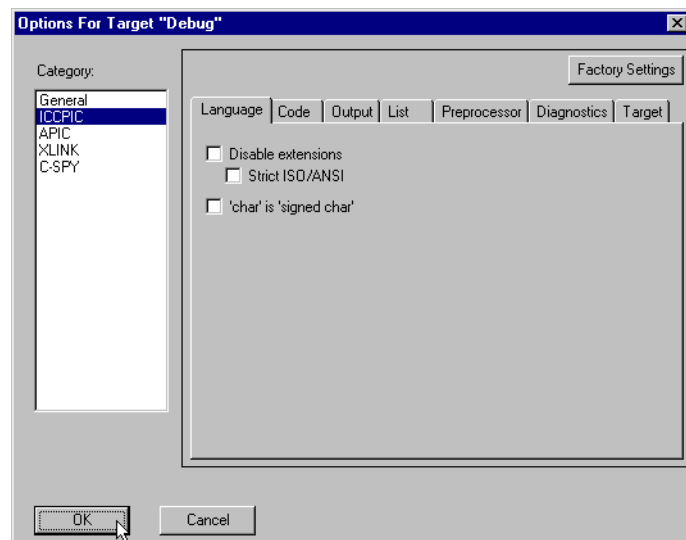
The section *Miscellaneous*, page 79, provides information about the options which are only available in the command line version of the product.

SETTING C COMPILER OPTIONS



Setting C compiler options in the Embedded Workbench

To set C compiler options in the Embedded Workbench choose **Options...** from the **Project** menu, and select **ICCPIC** in the **Category** list to display the compiler options pages:



Then click the tab corresponding to the category of options you want to view or change.



To restore all settings globally to the default factory settings, click on the **Factory Settings** button.



Setting C compiler options from the command line

To set C compiler options from the command line you include them on the command line after the `iccpic` command, either before or after the source filename. For example, when compiling the source `prog`, to generate an object file with debug information to the default object file:

```
iccpic prog -r ↵
```

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`:

```
iccpic 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 define a symbol:

```
-DDEBUG=1 ↵
```

Generally, the order of options on the command line, both relative to each other and to the source filename, is not significant. There is, however, one exception. When you use the `-I` (Include file path) option, the directories are searched in the same order as they are specified on the command line.

Specifying parameters

When a parameter is needed for a short option it can be specified either immediately following the option or as the next command line argument. For instance, an include file path of `/usr/include` can be specified either as:

```
-I/usr/include
```

or as

```
-I /usr/include
```

Note that `\` can be used instead of `/` as directory delimiter.

Additionally, output file options can take a parameter that is a directory and will then default the name and extension of the output file.

When a parameter is needed for a long option name, it can be specified either immediately after the equals sign or as the next command line argument, for example:

```
--diag_suppress=Be001
```

The option `--preprocess` is, however, an exception as the filename must be preceded by space. In the following example comments are included in the preprocessor output:

```
--preprocess=c prog
```

Options that accept multiple values may be repeated, and may also have comma-separated values (without space), for example:

```
--diag_warning=Be001,Be002
```

The current directory is specified by giving the directory `'.'`, for example:

```
iccpic prog -l .
```

A file called `'-'` is standard input/output, whichever is appropriate, for example:

```
iccpic prog -l -
```

Specifying options using the QCCPIC environment variable

Options can also be specified in the QCCPIC 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.

OPTIONS SUMMARY

The following is a summary of all the compiler options. For a full description of any option, see under the option's category name in the following sections.

<i>Command line</i>	<i>Option</i>	<i>Section</i>
<code>--char_is_signed</code>	<code>'char' is 'signed char'</code>	Language
<code>-Dsymb[=xx]</code>	Defined symbols	Preprocessor
<code>--debug</code>	Generate debug info	Output
<code>--diag_error=tag,tag...</code>	Treat these as errors	Diagnostics

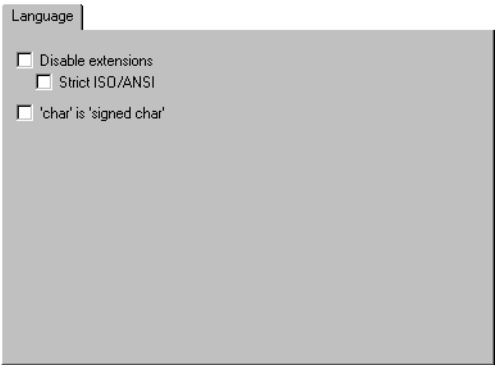
<i>Command line</i>	<i>Option</i>	<i>Section</i>
<code>--diag_remark=tag,tag...</code>	Treat these as remarks	Diagnostics
<code>--diag_suppress=tag,tag...</code>	Suppress these diagnostics	Diagnostics
<code>--diag_warning=tag,tag...</code>	Treat these as warnings	Diagnostics
<code>-e</code>	Enable language extensions	Language
<code>-Ipath</code>	Include paths	Preprocessor
<code>-l[c C a A][N] path[filename]</code>	Enable list file	List
<code>--library_module</code>	Make a library module	Output
<code>--module_name=name</code>	Object module name	Output
<code>--no_code_motion</code>	Disable code motion	Code
<code>--no_cse</code>	Disable common sub-expression elimination	Code
<code>--no_inline</code>	Disable function inlining	Code
<code>--no_unroll</code>	Disable loop unrolling	Code
<code>--no_warnings</code>	Disable warnings	Miscellaneous
<code>-o path[filename]</code>	Set object filename	Output
<code>--only_stdout</code>	Use standard output only	Miscellaneous
<code>--preprocess=[c][n][l] path[filename]</code>	Preprocessor output to file	Preprocessor
<code>-r</code>	Generate debug information	Output Directories
<code>--remarks</code>	Enable remarks	Diagnostics
<code>-s[0-9]</code>	Optimize for speed	Code
<code>--silent</code>	Set silent operation	Miscellaneous
<code>--strict_ansi</code>	Strict ISO/ANSI	Language
<code>-U symb</code>	Undefine symbol	Miscellaneous
<code>--uses_external_memory</code>	Use external memory	Target
<code>-vsetupfile</code>	Processor setup file	Target
<code>-z[0-9]</code>	Optimize for size	Code

LANGUAGE

The **Language** options enable the use of target-dependent extensions to the C language.



Embedded Workbench



Command line

<i>Option</i>	<i>Command line</i>
Enable language extensions	-e
Strict ISO/ANSI	--strict_ansi
'char' is 'signed char'	--char_is_signed

LANGUAGE EXTENSIONS

Enables target-dependent extensions to the C language.



In the Embedded Workbench language extensions are enabled by default.



Enable language extensions:

Syntax: -e

Normally, in the command line version of the PICmicro™ C Compiler, language extensions are disabled by default. If you are using language extensions in the source, you must enable them by using this option.

For details of language extensions, see the chapter *Language extensions summary*.

STRICT ISO/ANSI

Syntax: `--strict_ansi`

By default the compiler accepts a superset of ISO/ANSI C (see the chapter *IAR C extensions*). Use this option to adhere to strict ISO/ANSI.



Use `--strict_ansi` to ensure that the program conforms to the ISO/ANSI C standard.



First select **Disable extensions**, and then select **Strict ISO/ANSI** to adhere to the strict ISO/ANSI C standard.

'CHAR' IS 'SIGNED CHAR'

Syntax: `--char_is_signed`

Makes the `char` type equivalent to `signed char`.

Normally, 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.

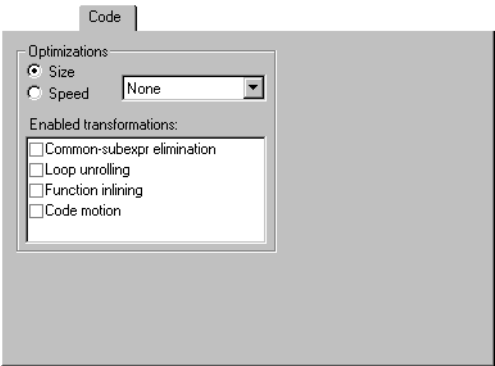
Note that the run-time library is compiled without the **'char' is 'signed char'** (`--char_is_signed`) option. If you use this option, you may get type mismatch warnings from the linker since the library uses `unsigned chars`.

CODE

The **Code** options determine the type and level of optimization for generation of object code.



Embedded Workbench



Command line

<i>Option</i>	<i>Command line</i>
Optimize for size	-z[0-9]
Optimize for speed	-s[0-9]
Disable common sub-expression elimination	--no_cse
Disable loop unrolling	--no_unroll
Disable function inlining	--no_inline
Disable code motion	--no_code_motion

OPTIMIZATIONS

By default, the compiler optimizes for maximum execution speed, not for size.

Use the optimization options to optimize for size or to change the optimization level for speed (-s) or size (-z).

The default optimization level for both speed (-s) and size (-z) is 3. You can change the level of optimization using the following modifiers.

<i>Modifier</i>	<i>Embedded Workbench option</i>	<i>Level</i>
0	None	No optimization.
1-3	Low	Fully debuggable.
4-6	Medium	Heavy optimization can make the program flow hard to follow during debug.
7-9	High	Full optimization.

Note that the **Optimize for size** and **Optimize for speed** options cannot be used at the same time.

Optimize for speed

Syntax: -s[0-9]

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

Optimize for size

Syntax: -z[0-9]

Causes the compiler to optimize the code for minimum size.

DISABLE COMMON SUB-EXPRESSION ELIMINATION

Syntax: --no_cse

Redundant re-evaluation of common sub-expressions is by default eliminated at optimization levels 4-9. This optimization normally both reduces code size and execution time. The resulting code may however be difficult to debug.



Uncheck the **Common-subexpr elimination** checkbox to disable common sub-expression elimination.



Use --no_cse to disable common sub-expression elimination.

Note that this option has no effect at optimization levels 0-3.

DISABLE LOOP UNROLLING

Syntax: `--no_unroll`

The loop body of a small loop, whose number of iterations can be determined at compile time, is duplicated to reduce the loop overhead.

This optimization, which is performed at optimization levels 7-9, normally reduces execution time, but increases code size. The resulting code may also be difficult to debug.

The compiler heuristically decides which loops to unroll. Different heuristics are used when optimizing for speed and size.



Uncheck the **Loop unrolling** checkbox to disable loop unrolling.



Use `--no_unroll` to disable loop-unrolling.

Note that this option has no effect at optimization levels 0-6.

DISABLE FUNCTION INLINING

Syntax: `--no_inline`

Function inlining means that a simple function, whose definition is known at compile-time, is integrated into the body of its caller to eliminate the overhead of the call. This optimization, which is performed at optimization levels 7-9, normally reduces execution time, but increases code size. The resulting code may also be difficult to debug.

The compiler heuristically decides which functions to inline. Different heuristics are used when optimizing for speed.



Uncheck the **Function inlining** checkbox to disable function inlining.



Use `--no_inline` to disable function in-lining.

Note that this option has no effect at optimization levels 0-6.

DISABLE CODE MOTION

Syntax: `--no_code_motion`

Evaluation of loop-invariant expressions and common sub-expressions are moved to avoid redundant re-evaluation. This optimization, which is performed at optimization levels 7-9, normally reduces code size and execution time. The resulting code may however be difficult to debug.



Uncheck the **Code motion** checkbox to disable code motion.



Use `--no_code_motion` to disable code motion.
Note that this option has no effect at optimization levels 0-6.

OUTPUT

The **Output** options determine the level of debugging information included in the object code.



Embedded Workbench

Output

☐ Make library module

☐ Object module name

☐ Generate debug info



Command line

<i>Option</i>	<i>Command line</i>
Make a library module	<code>--library_module</code>
Object module name	<code>--module_name=name</code>
Generate debug information	<code>--debug</code> <code>-r</code>

MAKE A LIBRARY MODULE

Syntax: `--library_module`


Causes the object file to be a library module rather than a program module.

A program module is always produced when linking. Use this option to make a library module which will only be included if referenced in your program.

OBJECT MODULE NAME

Syntax: --module_name=*name*

Normally, 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 --module_name=*name* option, for example:

```
iccpic prog --module_name=main 
```

This is particularly useful when several modules have the same filename, since normally the resulting duplicate module name would cause a linker error. An example is when the source file is a temporary file generated by a preprocessor. The following (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  
iccpic temp.c                 ; module name is  
                               ; always 'temp'
```

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

```
preproc %1.c temp.c           ; preprocess source,  
                               ; generating temp.c  
iccpic temp.c --module_name=%1 ; use original source  
                               ; name as module name
```

GENERATE DEBUG INFORMATION

Syntax: --debug
 -r

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

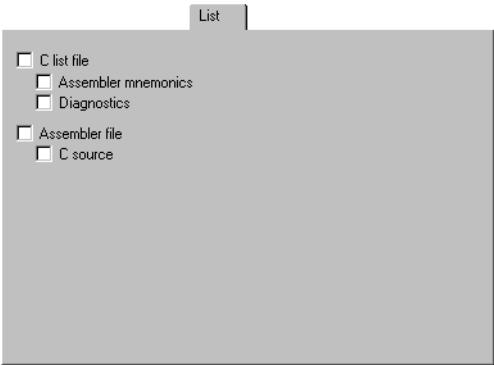
Note that including debug information will make the object files become larger than otherwise.

LIST

The **List** options determine whether a listing is produced, and the information included in the listing.



Embedded Workbench



Command line

<i>Option</i>	<i>Command line</i>
List to named file	<code>-l[c C a A][N] path[filename]</code>



LIST

Normally, the compiler does not generate a listing. To generate a listing to a file, check one of the following options:

<i>Option</i>	<i>Description</i>
C list file	Generates a C listing.
Assembler mnemonics	Includes assembler mnemonics in the C listing.
Diagnostics	Includes diagnostic information in the C listing.
Assembler file	Generates an assembler listing.
C source	Includes C source code in the assembler listing.

The file will be saved in a file whose name consists of the source filename, plus the extension `.lst` in the source file directory.


The file extension for list output to assembler source is `.s39`.



LIST TO NAMED FILE

Syntax: -l[c|C|a|A][N] *path[filename]*

Generates a listing to the named file with the default extension `.lst`.
Normally, the compiler does not generate a listing. To generate a listing to a named file, you use the `-l` option. For example, to generate a listing to the file `list.lst`, use:

`iccpic prog -l list` 

The following modifiers are available:

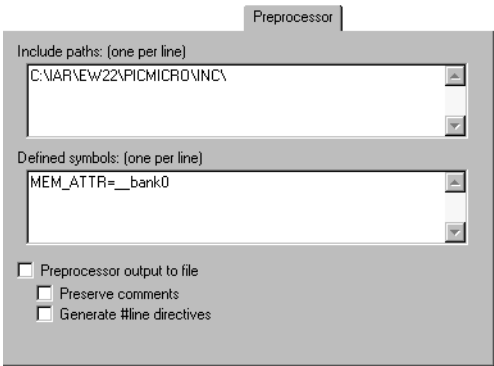
<i>Modifier</i>	<i>Command line</i>
C list file	c
C list file with assembler comments	C (default)
No diagnostics in file	N
Assembler file	a
Assembler file with C comments	A (N is implied)

PREPROCESSOR

The **Preprocessor** options allow you to define symbols and include paths for use by the C compiler.



Embedded Workbench



Command line

Option	Command line
Include paths	-I <i>path</i>
Defined symbols	-D <i>symb</i> [= <i>xx</i>]
Preprocessor output to file	--preprocess=[c][n][l] <i>filename</i>

INCLUDE PATHS

Syntax: -I*path*

Adds a path to the list of #include file paths, for example:

iccpic prog -I/mylib1 ↵

Note that both / and \ can be used as directory delimiters.

This option may be used more than once on a single command line.

Following is the full description of the compiler's #include file search procedure:

- ◆ If the #include file name is an absolute path, that file is opened.

- ◆ When the compiler encounters a `#include` file name in angle brackets such as:

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

it searches the following directories for the file to include:

1. The directories specified with the **Include paths** (`-I`) option, in the order that they were specified.
2. The directories specified using the `C_INCLUDE` environment variable, if any.

- ◆ When the compiler encounters a `#include` file name in double quotes such as:

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

it searches the directory of the source file, and then performs the same sequence as for angle-bracketed file names.

If there are nested `#include` files, the compiler starts searching in the directory of the file that was last included, iterating upwards for each included file, searching the source file directory last. Example:

```
src.c in directory dir
#include "src.h"
...
src.h in directory dir\h
#include "io.h"
...
```



When `dir\exe` is the current directory, use the following command for compilation:

```
iccpic ..\src.c -I..\dir\include
```

Then the following directories are searched for the `io.h` file, in the following order:

<code>dir\h</code>	Current file.
<code>dir</code>	File including current file.
<code>dir\include</code>	As specified with the <code>-I</code> option.

DEFINED SYMBOLS

Syntax: -D*symb*[=*xx*]

Defines a symbol with the name *symb* and the value *xx*. If no value is specified, 1 is used.

Define symbol (-D) has the same effect as a `#define` statement at the top of the source file.

-D*symb*

is equivalent to:

```
#define symb
```

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

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:

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

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

production version: iccpic prog

test version: iccpic prog -D*testver*

This option may have one or more comma-separated values, and it can be used one or more times.

PREPROCESSOR OUTPUT TO FILE

Syntax: --preprocess=[c][n][l] *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 the filename itself, optionally preceded by a pathname and optionally followed by an extension. If no extension is given, the extension `.i` is used. In the syntax description above, note that space is allowed in front of the filename.

The following preprocessor modifiers are available:

<i>Modifier</i>	<i>Command line</i>
Include comments	<code>c</code>
Preprocess only	<code>n</code>
Generate <code>#line</code> directives	<code>l</code>

DIAGNOSTICS

The **Diagnostics** options determine how diagnostics are classified and displayed. Use the diagnostics options to override the default classification of the specified diagnostics.

Note: The diagnostics cannot be suppressed for fatal errors, and that fatal errors cannot be reclassified.

See the chapter *Diagnostics* for further information about diagnostic messages.



Embedded Workbench

Diagnostics

☐ Enable remarks

Suppress these diagnostics:

Treat these as remarks:

Treat these as warnings:

Treat these as errors:



Command line

<i>Option</i>	<i>Command line</i>
Enable remarks	<code>--remarks</code>

<i>Option</i>	<i>Command line</i>
Suppress these diagnostics	<code>--diag_suppress=tag,tag...</code>
Treat these as remarks	<code>--diag_remark=tag,tag...</code>
Treat these as warnings	<code>--diag_warning=tag,tag...</code>
Treat these as errors	<code>--diag_error=tag,tag...</code>

ENABLE REMARKS

Syntax: `--remarks`

The least severe diagnostic messages are called remarks (see *Severity levels*, page 217). A remark indicates a source code construct that may cause strange behavior in the generated code.

By default no remarks are issued.

Use `--remarks` to generate remarks.

SUPPRESS THESE DIAGNOSTICS

Syntax: `--diag_suppress=tag,tag...`

Suppresses the output of diagnostics for the specified tags. The following example suppresses the warnings Pe117 and Pe177:

```
__diag_suppress=Pe117,Pe177
```

TREAT THESE AS REMARKS

Syntax: `--diag_remark=tag,tag...`

Reclassifies the specified diagnostics as remarks, which is the least severe type of diagnostic message.

A remark indicates a source code construct that may cause strange behavior in the generated code.

The following example classifies the warning Pe177 as a remark:

```
__diag_remark=Pe177
```

TREAT THESE AS WARNINGS

Syntax: `--diag_warning=tag, tag...`

Reclassifies the specified diagnostics as warnings.

A warning indicates an error or omission which is of concern, but which will not cause the compiler to stop before the compilation is completed.

The following example classifies the remark Pe826 as a warning:

```
__diag_warning=Pe826
```

TREAT THESE AS ERRORS

Syntax: `--diag_error=tag, tag...`

Reclassifies the specified diagnostics as errors.

An error indicates a violation of the C language rules, of such severity that object code will not be generated, and exit code will not be 0.

The following example classifies warning Pe117 as an error:

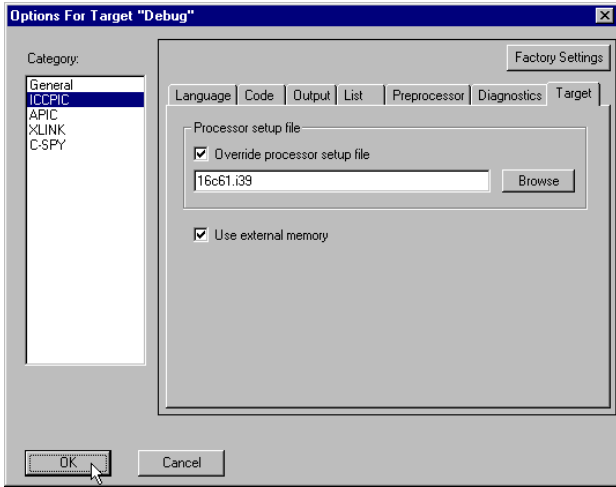
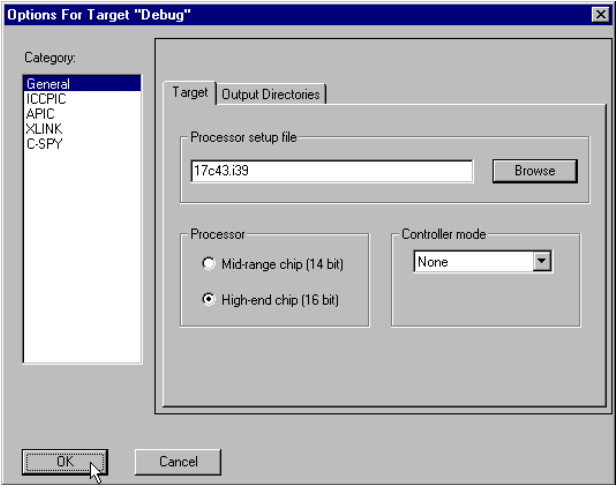
```
__diag_error=Pe117
```

TARGET

The **Target** option specifies the processor setup file. The target should normally be set under the **General** category and will affect both the PICmicro™ C compiler and optional PICmicro™ C-SPY debugger. The **Target** options for **ICCPIC** allow for overriding the general target processor setup file and use of external memory addresses.



Embedded Workbench





Command line

<i>Option</i>	<i>Command line</i>
Processor setup file	<code>-vsetupfile</code>
Use external memory (Only high-end processors)	<code>--uses_external_memory</code>

PROCESSOR SETUP FILE

Syntax: `-vsetupfile`

Selects the target processor setup file. To select a processor setup file, click the **Browse** button to display a standard **Open** dialog box. Select a setup file and click **Open**. The default location for the processor setup files is the `iar\setup\` directory. The following processors are available:

<i>File</i>	<i>Processor name</i>	<i>File</i>	<i>Processor name</i>
16c61.i39	16C61	16c711.i39	16C711
16c62a.i39	16C62A	16c715.i39	16C715
16c621.i39	16C621	16c72.i39	16C72
16c622.i39	16C622	16c73.i39	16C73
16c63.i39	16C63	16c74a.i39	16C74A
16c64a.i39	16C64A	16c76.i39	16C76
16c641.i39	16C641	16c77.i39	16C77
16c642.i39	16C642	16f84.i39	16F84
16c65a.i39	16C65A	16c923.i39	16C923
16c66a.i39	16C66	16c924.i39	16C924
16c661.i39	16C661	17c42a.i39	17C42A
16c662.i39	16C662	17c43.i39	17C43
16c67.i39	16C67	17c44.i39	17C44
16c71.i39	16C71	17c752.i39	17C752
16c710.i39	16C710	17c756.i39	17C756

If no **Processor** is specified, the C compiler uses 16C61 by default.



The location of the processor set-up files must be set using the environment variable QPICINFO. See *Environment variables*, page 52. The location is by default \iar\setup\.

USE EXTERNAL MEMORY

Syntax: --uses_external_memory

Enables external memory up to 64 kWord for high-end processors (17CXX). This option forces the use of long calls and jumps. If enabled, the C library must use the assembler option -D__HAS_PAGES.

OUTPUT DIRECTORIES

The **Output directories** options in the **General** category allow you to specify output directories for executable files, object files, and list files.



Embedded Workbench

Output Directories

Executables:

Object files:

List files:



Command line

Option

Command line

Set object filename

-o path[filename]



EXECUTABLES

Use this option to override the default directory for executable files.

Enter the name of the directory where you want to save executable files for the project.



OBJECT FILES

Use this option to override the default directory for object files.

Enter the name of the directory where you want to save object files for the project.



LIST FILES

Use this option to override the default directory for list files.

Enter the name of the directory where you want to save list files for the project.



SET OBJECT FILENAME

Syntax: -o path[filename]

Use the -o option to specify a directory for the output file. The filename may include a pathname. For example, to store it in the file obj.39 in the mypath directory, you would use:

```
iccpic prog -o /mypath/obj ↵
```

Note that both / and \ can be used as directory delimiters.

If this option is not used, the compiler stores the object code in a file whose name consists of the source filename, excluding the path, plus the extension .r39. The file will be stored in the current active directory.

MISCELLANEOUS

The following additional options are available from the command line:

<i>Option</i>	<i>Command line</i>
Disable warnings	--no_warnings
Use standard output only	--only_stdout
Set silent operation	--silent
Undefine symbol	-U symb



DISABLE WARNINGS

Syntax: `--no_warnings`

Normally, the compiler issues standard warning messages. To disable all warning messages, you use the **Disable Warnings** (`--no_warnings`) option.



USE STANDARD OUTPUT ONLY

Syntax: `--only_stdout`

Causes the compiler to use only standard output. There will be no console output on `stderr`.



SET SILENT OPERATION

Syntax: `--silent`

Causes the compiler to operate without sending unnecessary messages to standard output (normally the screen).

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

This option is related with the **Message Filtering Level** option in the Embedded Workbench.



UNDEFINE SYMBOL

Syntax: `-U symb`

Removes the definition of the named symbol.

Normally, the compiler provides various pre-defined 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 undefine symbol (`-U`) option.

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

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

```
iccpic prog -U__VER__ ↵
```

GENERAL C LIBRARY DEFINITIONS

This chapter gives an introduction to the C library functions, summarizes them according to header file and describes how to build a target-specific library.

INTRODUCTION

The PICmicro™ 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, for user programs. These are documented in this chapter.
- ◆ CSTARTUP, the single program module containing the start-up code. It is described in *Initialization*, page 60.
- ◆ Intrinsic functions, allowing low-level use of PICmicro™ features. See the chapter *Intrinsic functions reference*.

LIBRARY OBJECT FILES

You must build the appropriate library object file for your chosen microcontroller. The 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 supplied. There are some I/O-oriented routines (such as `putchar` and `getchar`) that you may need to customize for your target application.

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 covering a particular functional area, letting you include just 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.

BUILDING TARGET-SPECIFIC LIBRARIES

The Buildlib utility consists of DOS batch files for each supported processor. The batch files are found in the `...picmicro\clib` directory and are named `blxxxxx.bat`, where `xxxxx` corresponds to the processor name. By default the batch files build library modules for use without external memory. In order to build for use with external memory, two lines in the `blxxxxx.bat` file must be modified:

Default (without external memory):

```
@SET APIC_EXTRA=
@SET COMP_OPTIONS=--only_stdout
```

With external memory:

```
@SET APIC_EXTRA=-D__HAS_PAGES
@SET COMP_OPTIONS=--only_stdout --uses_external_memory
```

The resulting file `clxxxxx.r39` will be placed in the `...picmicro\lib` directory.

Note: External memory is only available on high-end processors.

BUILDING A LIBRARY MODULE



First, you must set the `QPICINFO` environment variable, see *Environment variables*, page 52.



If you are using the command line version you should already have set the `QPICINFO` environment variable, see *Installation*, page 3.

If the `QPICINFO` environment variable is not set an error message is shown.

Then run the appropriate `blxxxxx.bat` file from the command line. For example to build a library module for the 17C43 move to the `...picmicro\clib` directory and enter:

```
bl17c43 ↵
```

The Buildlib utility will start. As this is a complex process, the build might take couple of minutes.

Note: The Buildlib utility requires at least 512 bytes of free environment space. To set the environment space in DOS or Windows 95/98, add the following line to your config.sys file:

```
SHELL=C:\COMMAND.COM /E:1024 /P
```

LIBRARY DEFINITIONS SUMMARY

This section lists the header files and summarizes the functions included in each of them. Header files may additionally contain target-specific definitions; these are documented in the chapter *Language extensions summary*.

CHARACTER HANDLING – ctype.h

isalnum	int isalnum(int c)	Letter or digit equality.
isalpha	int isalpha(int c)	Letter equality.
iscntrl	int iscntrl(int c)	Control code equality.
isdigit	int isdigit(int c)	Digit equality.
isgraph	int isgraph(int c)	Printable non-space character equality.
islower	int islower(int c)	Lower case equality.
isprint	int isprint(int c)	Printable character equality.
ispunct	int ispunct(int c)	Punctuation character equality.
isspace	int isspace(int c)	White-space character equality.
isupper	int isupper(int c)	Upper case equality.
isxdigit	int isxdigit(int c)	Hex digit equality.
tolower	int tolower(int c)	Converts to lower case.
toupper	int toupper(int c)	Converts to upper case.

LOW-LEVEL ROUTINES – icclbutl.h

<code>_formatted_read</code>	<code>int _formatted_read (const char **line, const char **format, va_list ap)</code>	Reads formatted data.
<code>_formatted_write</code>	<code>int _formatted_write (const char* format, void outputf (char, void *), void *sp, va_list ap)</code>	Formats and writes data.
<code>_medium_read</code>	<code>int _formatted_read (const char **line, const char **format, va_list ap)</code>	Reads formatted data excluding floating-point numbers.
<code>_medium_write</code>	<code>int _formatted_write (const char* format, void outputf (char, void *), void *sp, va_list ap)</code>	Writes formatted data excluding floating-point numbers.
<code>_small_write</code>	<code>int _formatted_write (const char* format, void outputf (char, void *), void *sp, va_list ap)</code>	Small formatted data write routine.

MATHEMATICS – math.h

<code>acos</code>	<code>double acos(double arg)</code>	Arc cosine.
<code>asin</code>	<code>double asin(double arg)</code>	Arc sine.
<code>atan</code>	<code>double atan(double arg)</code>	Arc tangent.
<code>atan2</code>	<code>double atan2(double arg1, double arg2)</code>	Arc tangent with quadrant.
<code>ceil</code>	<code>double ceil(double arg)</code>	Smallest integer greater than or equal to arg.
<code>cos</code>	<code>double cos(double arg)</code>	Cosine.
<code>cosh</code>	<code>double cosh(double arg)</code>	Hyperbolic cosine.
<code>exp</code>	<code>double exp(double arg)</code>	Exponential.
<code>fabs</code>	<code>double fabs(double arg)</code>	Double-precision floating-point absolute.
<code>floor</code>	<code>double floor(double arg)</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.
<code>tanh</code>	<code>double tanh(double <i>arg</i>)</code>	Hyperbolic tangent.

NON-LOCAL JUMPS – `setjmp.h`

Not supported on PICmicro™.

VARIABLE ARGUMENTS – `stdarg.h`

<code>va_arg</code>	<code>type va_arg(va_list <i>ap</i>, <i>mode</i>)</code>	Next argument in function call.
<code>va_end</code>	<code>void va_end(va_list <i>ap</i>)</code>	Ends reading function call arguments.
<code>va_list</code>	<code>char *va_list[1]</code>	Argument list type.
<code>va_start</code>	<code>void va_start(va_list <i>ap</i>, <i>parmN</i>)</code>	Starts reading function call arguments

INPUT/OUTPUT – `stdio.h`

<code>getchar</code>	<code>int getchar(void)</code>	Gets character.
<code>gets</code>	<code>char *gets(char *s)</code>	Gets string.
<code>printf</code>	<code>int printf(const char *format, ...)</code>	Writes formatted data.
<code>putchar</code>	<code>int putchar(int value)</code>	Puts character.
<code>puts</code>	<code>int puts(const char *s)</code>	Puts string.
<code>scanf</code>	<code>int scanf(const char *format, ...)</code>	Reads formatted data.
<code>sprintf</code>	<code>int sprintf(char *s, const char *format, ...)</code>	Writes formatted data to a string.
<code>sscanf</code>	<code>int sscanf(const char *s, const char *format, ...)</code>	Reads formatted data from a string.
<code>vprint</code>	<code>int vprintf(const char *format, va_list argptr)</code>	Writes formatted data to standard output
<code>vsprint</code>	<code>int vsprintf(char *s, const char *format, va_list argptr)</code>	Writes formatted data to a buffer

GENERAL UTILITIES – `stdlib.h`

<code>abort</code>	<code>void abort(void)</code>	Terminates the program abnormally.
<code>abs</code>	<code>int abs(int j)</code>	Absolute value.
<code>atof</code>	<code>double atof(const char *nptr)</code>	Converts ASCII to double.
<code>atoi</code>	<code>int atoi(const char *nptr)</code>	Converts ASCII to int.
<code>atol</code>	<code>long atol(const char *nptr)</code>	Converts ASCII to long int.
<code>bsearch</code>	<code>void *bsearch(const void *key, const void *base, size_t nmemb, size_t size, int (*compare)(const void *_key, const void *_base));</code>	Makes a generic search in an array.
<code>calloc</code>	<code>void *calloc(size_t nelem, size_t elsize)</code>	Allocates memory for an array of objects.

<code>div</code>	<code>div_t div(int <i>numer</i>, int <i>denom</i>)</code>	Divide.
<code>exit</code>	<code>void exit(int <i>status</i>)</code>	Terminates the program.
<code>free</code>	<code>void free(void *<i>ptr</i>)</code>	Frees memory.
<code>labs</code>	<code>long int labs(long int <i>j</i>)</code>	Long absolute.
<code>ldiv</code>	<code>ldiv_t ldiv(long int <i>numer</i>, long int <i>denom</i>)</code>	Long division.
<code>malloc</code>	<code>void *malloc(size_t <i>size</i>)</code>	Allocates memory.
<code>qsort</code>	<code>void qsort(const void *<i>base</i>, size_t <i>nmem</i>, size_t <i>size</i>, int (*<i>compare</i>) (const void *<i>_key</i>, const void *<i>_base</i>));</code>	Makes a generic sort of an array.
<code>rand</code>	<code>int rand(void)</code>	Random number.
<code>realloc</code>	<code>void *realloc(void *<i>ptr</i>, size_t <i>size</i>)</code>	Reallocates memory.
<code>srand</code>	<code>void srand(unsigned int <i>seed</i>)</code>	Sets random number sequence.
<code>strtod</code>	<code>double strtod(const char *<i>nptr</i>, char **<i>endptr</i>)</code>	Converts a string to double.
<code>strtol</code>	<code>long int strtol(const char *<i>nptr</i>, char **<i>endptr</i>, int <i>base</i>)</code>	Converts a string to a long integer.
<code>strtoul</code>	<code>unsigned long int strtoul(const char *<i>nptr</i>, char **<i>endptr</i>, int <i>base</i>)</code>	Converts a string to an unsigned long integer.

STRING HANDLING – `string.h`

<code>memchr</code>	<code>void *memchr(const void *<i>s</i>, int <i>c</i>, size_t <i>n</i>)</code>	Searches for a character in memory.
<code>memcmp</code>	<code>int memcmp(const void *<i>s1</i>, const void *<i>s2</i>, size_t <i>n</i>)</code>	Compares memory.
<code>memcpy</code>	<code>void *memcpy(void *<i>s1</i>, const void *<i>s2</i>, size_t <i>n</i>)</code>	Copies memory.
<code>memmove</code>	<code>void *memmove(void *<i>s1</i>, const void *<i>s2</i>, size_t <i>n</i>)</code>	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.
strspn	size_t strspn(const char *s1, const char *s2)	Spans characters in a string.
strstr	char *strstr(const char *s1, const char *s2)	Searches for a substring.
strtok	char *strtok(char *s1, const char *s2)	Breaks a string into tokens.
strxfrm	size_t strxfrm(char *s1, const char *s2, size_t n)	Transforms a string and returns the length.

COMMON DEFINITIONS – `stddef.h`

No functions (various definitions including `size_t`, `NULL`, `ptrdiff_t`, `offsetof`, etc).

INTEGRAL TYPES – `limits.h`

No functions (various limits and sizes of integral types).

FLOATING-POINT TYPES – `float.h`

No functions (various limits and sizes of floating-point types).

ERRORS – `errno.h`

No functions (various error return values).

ASSERT – `assert.h`

<code>assert</code>	<code>void assert(int <i>expression</i>)</code> Checks an expression.
---------------------	---

C LIBRARY FUNCTIONS

REFERENCE

This section gives an alphabetical list of the C library functions, with a full description of their operation, and the options available for each one.

The format of each function description is as follows:

	Function name	Header filename
Brief description	atoi	stdlib.h
		Converts ASCII to int.
Declaration		DECLARATION int atoi(const char *nptr)
Parameters		PARAMETERS nptr A pointer to a string containing a number in ASCII form.
Return value		RETURN VALUE The int number found in the string.
Description		DESCRIPTION Converts the ASCII string pointed to by nptr to an integer, skipping white space and terminating upon reaching any unrecognized character.
Examples		EXAMPLES " -3K" gives -3 "6" gives 6 "149" gives 149

FUNCTION NAME

The name of the C library function.

HEADER FILENAME

The function header filename.

BRIEF DESCRIPTION

A brief summary of the function.

DECLARATION

The C library declaration.

PARAMETERS

Details of each parameter in the declaration.

RETURN VALUE

The value, if any, returned by the function.

DESCRIPTION

A detailed description covering the function's most general use. This includes information about what the function is useful for, and a discussion of any special conditions and common pitfalls.

EXAMPLES

One or more examples illustrating the function's use.

abort

`stdlib.h`

Terminates the program abnormally.

DECLARATION

`void abort(void)`

PARAMETERS

None.

RETURN VALUE

None.

DESCRIPTION

Terminates the program abnormally and does not return to the caller. This function calls the `exit` function, and by default the entry for this resides in `CSTARTUP`.

abs

stdlib.h

Absolute value.

DECLARATION`int abs(int j)`**PARAMETERS***j* An int value.**RETURN VALUE**An int having the absolute value of *j*.**DESCRIPTION**Computes the absolute value of *j*.

acos

math.h

Arc cosine.

DECLARATION`double acos(double arg)`**PARAMETERS***arg* A double in the range [-1,+1].**RETURN VALUE**The double arc cosine of *arg*, in the range [0,pi].**DESCRIPTION**Computes the principal value in radians of the arc cosine of *arg*.

asin`math.h`

Arc sine.

DECLARATION`double asin(double arg)`**PARAMETERS***arg* A double in the range $[-1, +1]$.**RETURN VALUE**The double arc sine of *arg*, in the range $[-\pi/2, +\pi/2]$.**DESCRIPTION**Computes the principal value in radians of the arc sine of *arg*.

assert`assert.h`

Checks an expression.

DECLARATION`void assert (int expression)`**PARAMETERS***expression* An expression to be checked.**RETURN VALUE**

None.

DESCRIPTION

This is a macro that checks an expression. If it is false it prints a message to `stderr` and calls `abort`.

The message has the following format:

File *name*; line *num* # Assertion failure "*expression*"

To ignore `assert` calls put a `#define NDEBUG` statement before the `#include <assert.h>` statement.

atan`math.h`

Arc tangent.

DECLARATION`double atan(double arg)`**PARAMETERS***arg* A double value.**RETURN VALUE**The double arc tangent of *arg*, in the range $[-\pi/2, \pi/2]$.**DESCRIPTION**Computes the arc tangent of *arg*.

atan2`math.h`

Arc tangent with quadrant.

DECLARATION`double atan2(double arg1, double arg2)`**PARAMETERS***arg1* A double value.*arg2* A double value.**RETURN VALUE**The double arc tangent of *arg1/arg2*, in the range $[-\pi, \pi]$.**DESCRIPTION**Computes the arc tangent of *arg1/arg2*, using the signs of both arguments to determine the quadrant of the return value.

atof

stdlib.h

Converts ASCII to double.

DECLARATIONdouble atof(const char **nptr*)**PARAMETERS**

nptr A pointer to a string containing a number in ASCII form.

RETURN VALUE

The double number found in the string.

DESCRIPTION

Converts the string pointed to by *nptr* to a double-precision floating-point number, skipping white space and terminating upon reaching any unrecognized character.

EXAMPLES

" -3K" gives -3.00

".0006" gives 0.0006

"1e-4" gives 0.0001

atoi

stdlib.h

Converts ASCII to int.

DECLARATIONint atoi(const char **nptr*)**PARAMETERS**

nptr A pointer to a string containing a number in ASCII form.

RETURN VALUE

The int number found in the string.

DESCRIPTION

Converts the ASCII string pointed to by *nptr* to an integer, skipping white space and terminating upon reaching any unrecognized character.

EXAMPLES

" -3K" gives -3

"6" gives 6

"149" gives 149

atol

stdlib.h

Converts ASCII to long int.

DECLARATION

long atol(const char **nptr*)

PARAMETERS

nptr A pointer to a string containing a number in ASCII form.

RETURN VALUE

The long number found in the string.

DESCRIPTION

Converts the number found in the ASCII string pointed to by *nptr* to a long integer value, skipping white space and terminating upon reaching any unrecognized character.

EXAMPLES

" -3K" gives -3

"6" gives 6

"149" gives 149

bsearch

stdlib.h

Makes a generic search in an array.

DECLARATION

```
void *bsearch(const void *key, const void *base, size_t
nmemb, size_t size, int (*compare) (const void *_key,
const void *_base));
```

PARAMETERS

<i>key</i>	Pointer to the searched for object.
<i>base</i>	Pointer to the array to search.
<i>nmemb</i>	Dimension of the array pointed to by <i>base</i> .
<i>size</i>	Size of the array elements.
<i>compare</i>	The comparison function which takes two arguments and returns: <div>< 0 (negative value) if <i>_key</i> is less than <i>_base</i>. 0 if <i>_key</i> equals <i>_base</i>. > 0 (positive value) if <i>_key</i> is greater than <i>_base</i>.</div>

RETURN VALUE

<i>Result</i>	<i>Value</i>
Successful	A pointer to the element of the array that matches the key.
Unsuccessful	Null.

DESCRIPTION

Searches an array of *nmemb* objects, pointed to by *base*, for an element that matches the object pointed to by *key*.

calloc

stdlib.h

Allocates memory for an array of objects.

DECLARATION

```
void *calloc(size_t nelm, size_t elsize)
```

PARAMETERS

<i>nlem</i>	The number of objects.
<i>elsize</i>	A value of type <i>size_t</i> specifying the size of each object.

RETURN VALUE

<i>Result</i>	<i>Value</i>
Successful	A pointer to the start (lowest address) of the memory block.
Unsuccessful	Zero if there is no memory block of the required size or greater available.

DESCRIPTION

Allocates a memory block for an array of objects of the given size. To ensure portability, the size is not given in absolute units of memory such as bytes, but in terms of a size or sizes returned by the `sizeof` function.

The availability of memory depends on the default heap size, see *Heap size*, page 59.

ceil

`math.h`

Smallest integer greater than or equal to *arg*.

DECLARATION

```
double ceil(double arg)
```

PARAMETERS

<i>arg</i>	A double value.
------------	-----------------

RETURN VALUE

A double having the smallest integral value greater than or equal to *arg*.

DESCRIPTION

Computes the smallest integral value greater than or equal to *arg*.

cos

math.h

Cosine.

DECLARATION

double cos(double *arg*)

PARAMETERS

arg A double value in radians.

RETURN VALUE

The double cosine of *arg*.

DESCRIPTION

Computes the cosine of *arg* radians.

cosh

math.h

Hyperbolic cosine.

DECLARATION

double cosh(double *arg*)

PARAMETERS

arg A double value in radians.

RETURN VALUE

The double hyperbolic cosine of *arg*.

DESCRIPTION

Computes the hyperbolic cosine of *arg* radians.

div

stdlib.h

Divide.

DECLARATION`div_t div(int numer, int denom)`**PARAMETERS***numer* The int numerator.*denom* The int denominator.**RETURN VALUE**

A structure of type `div_t` holding the quotient and remainder results of the division.

DESCRIPTION

Divides the numerator *numer* by the denominator *denom*. The type `div_t` is defined in `stdlib.h`.

If the division is inexact, the quotient is the integer of lesser magnitude that is the nearest to the algebraic quotient. The results are defined such that:

$$\text{quot} * \text{denom} + \text{rem} == \text{numer}$$

exit

stdlib.h

Terminates the program.

DECLARATION`void exit(int status)`**PARAMETERS***status* An int status value.**RETURN VALUE**

None.

DESCRIPTION

Terminates the program normally. This function does not return to the caller. This function entry resides by default in CSTARTUP.

exp

math.h

Exponential.

DECLARATION

double exp(double *arg*)

PARAMETERS

arg A double value.

RETURN VALUE

A double with the value of the exponential function of *arg*.

DESCRIPTION

Computes the exponential function of *arg*.

exp10

math.h

Exponential.

DECLARATION

double exp10(double *arg*)

PARAMETERS

arg A double value.

RETURN VALUE

A double with the value of 10^{arg} .

DESCRIPTION

Computes the the value of 10^{arg} .

fabs`math.h`

Double-precision floating-point absolute.

DECLARATION`double fabs(double arg)`**PARAMETERS**

arg A double value.

RETURN VALUE

The double absolute value of *arg*.

DESCRIPTION

Computes the absolute value of the floating-point number *arg*.

floor`math.h`

Largest integer less than or equal.

DECLARATION`double floor(double arg)`**PARAMETERS**

arg A double value.

RETURN VALUE

A double with the value of the largest integer less than or equal to *arg*.

DESCRIPTION

Computes the largest integral value less than or equal to *arg*.

fmod

math.h

Floating-point remainder.

DECLARATION

```
double fmod(double arg1, double arg2)
```

PARAMETERS

arg1 The double numerator.

arg2 The double denominator.

RETURN VALUE

The double remainder of the division *arg1/arg2*.

DESCRIPTION

Computes the remainder of *arg1/arg2*, ie the value *arg1* - *i* * *arg2*, for some integer *i* such that, if *arg2* is non-zero, the result has the same sign as *arg1* and magnitude less than the magnitude of *arg2*.

free

stdlib.h

Frees memory.

DECLARATION

```
void free(void *ptr)
```

PARAMETERS

ptr A pointer to a memory block previously allocated by malloc, calloc, or realloc.

RETURN VALUE

None.

DESCRIPTION

Frees the memory used by the object pointed to by *ptr*. *ptr* must earlier have been assigned a value from malloc, calloc, or realloc.

frexp`math.h`

Splits a floating-point number into two parts.

DECLARATION

```
double frexp(double arg1, int *arg2)
```

PARAMETERS

arg1 Floating-point number to be split.

arg2 Pointer to an integer to contain the exponent of *arg1*.

RETURN VALUE

The double mantissa of *arg1*, in the range 0.5 to 1.0.

DESCRIPTION

Splits the floating-point number *arg1* into an exponent stored in **arg2*, and a mantissa which is returned as the value of the function.

The values are as follows:

$$\text{mantissa} * 2^{\text{exponent}} = \text{value}$$

getchar`stdio.h`

Gets character.

DECLARATION

```
int getchar(void)
```

PARAMETERS

None.

RETURN VALUE

An int with the ASCII value of the next character from the standard input stream.

DESCRIPTION

Gets the next character from the standard input stream.

You should customize this function for the particular target hardware configuration. The function is supplied in source format in the file `getchar.c`.

gets

`stdio.h`

Gets string.

DECLARATION

`char *gets(char *s)`

PARAMETERS

s A pointer to the string that is to receive the input.

RETURN VALUE

<i>Result</i>	<i>Value</i>
Successful	A pointer equal to <i>s</i> .
Unsuccessful	Null.

DESCRIPTION

Gets the next string from standard input and places it in the string pointed to. The string is terminated by end of line or end of file. The end-of-line character is replaced by zero.

This function calls `getchar`, which must be adapted for the particular target hardware configuration.

isalnum

`ctype.h`

Letter or digit equality.

DECLARATION

`int isalnum(int c)`

PARAMETERS

c An `int` representing a character.

RETURN VALUE

An int which is non-zero if *c* is a letter or digit, else zero.

DESCRIPTION

Tests whether a character is a letter or digit.

isalpha

ctype.h

Letter equality.

DECLARATION

```
int isalpha(int c)
```

PARAMETERS

c An int representing a character.

RETURN VALUE

An int which is non-zero if *c* is letter, else zero.

DESCRIPTION

Tests whether a character is a letter.

isctrl

ctype.h

Control code equality.

DECLARATION

```
int isctrl(int c)
```

PARAMETERS

c An int representing a character.

RETURN VALUE

An int which is non-zero if *c* is a control code, else zero.

DESCRIPTION

Tests whether a character is a control character.

isdigit

ctype.h

Digit equality.

DECLARATION

```
int isdigit(int c)
```

PARAMETERS

c An int representing a character.

RETURN VALUE

An int which is non-zero if *c* is a digit, else zero.

DESCRIPTION

Tests whether a character is a decimal digit.

isgraph

ctype.h

Printable non-space character equality.

DECLARATION

```
int isgraph(int c)
```

PARAMETERS

c An int representing a character.

RETURN VALUE

An int which is non-zero if *c* is a printable character other than space, else zero.

DESCRIPTION

Tests whether a character is a printable character other than space.

islower`ctype.h`

Lower case equality.

DECLARATION

```
int islower(int c)
```

PARAMETERS

`c` An int representing a character.

RETURN VALUE

An int which is non-zero if `c` is lower case, else zero.

DESCRIPTION

Tests whether a character is a lower case letter.

isprint`ctype.h`

Printable character equality.

DECLARATION

```
int isprint(int c)
```

PARAMETERS

`c` An int representing a character.

RETURN VALUE

An int which is non-zero if `c` is a printable character, including space, else zero.

DESCRIPTION

Tests whether a character is a printable character, including space.

ispunct

ctype.h
Punctuation character equality.

DECLARATION

int ispunct(int c)

PARAMETERS

c An int representing a character.

RETURN VALUE

An int which is non-zero if c is printable character other than space, digit, or letter, else zero.

DESCRIPTION

Tests whether a character is a printable character other than space, digit, or letter.

isspace

ctype.h
White-space character equality.

DECLARATION

int isspace (int c)

PARAMETERS

c An int representing a character.

RETURN VALUE

An int which is non-zero if c is a white-space character, else zero.

DESCRIPTION

Tests whether a character is a white-space character, that is, one of the following:

<i>Character</i>	<i>Symbol</i>
Space	' '

<i>Character</i>	<i>Symbol</i>
Formfeed	\f
Newline	\n
Carriage return	\r
Horizontal tab	\t
Vertical tab	\v

isupper

ctype.h

Upper case equality.

DECLARATION

```
int isupper(int c)
```

PARAMETERS

c An int representing a character.

RETURN VALUE

An int which is non-zero if *c* is upper case, else zero.

DESCRIPTION

Tests whether a character is an upper case letter.

isxdigit

ctype.h

Hex digit equality.

DECLARATION

```
int isxdigit(int c)
```

PARAMETERS

c An int representing a character.

RETURN VALUE

An `int` which is non-zero if `c` is a digit in upper or lower case, else zero.

DESCRIPTION

Tests whether the character is a hexadecimal digit in upper or lower case, that is, one of 0-9, a-f, or A-F.

labs

`stdlib.h`

Long absolute.

DECLARATION

`long int labs(long int j)`

PARAMETERS

j A long int value.

RETURN VALUE

The long int absolute value of *j*.

DESCRIPTION

Computes the absolute value of the long integer *j*.

ldexp

`math.h`

Multiply by power of two.

DECLARATION

`double ldexp(double arg1, int arg2)`

PARAMETERS

arg1 The double multiplier value.

arg2 The int power value.

RETURN VALUE

The double value of *arg1* multiplied by two raised to the power of *arg2*.

DESCRIPTION

Computes the value of the floating-point number multiplied by 2 raised to a power.

ldiv

stdlib.h

Long division

DECLARATION

ldiv_t ldiv(long int *numer*, long int *denom*)

Parameters

numer The long int numerator.

denom The long int denominator.

RETURN VALUE

A struct of type ldiv_t holding the quotient and remainder of the division.

DESCRIPTION

Divides the numerator *numer* by the denominator *denom*. The type ldiv_t is defined in stdlib.h.

If the division is inexact, the quotient is the integer of lesser magnitude that is the nearest to the algebraic quotient. The results are defined such that:

$\text{quot} * \text{denom} + \text{rem} == \text{numer}$

log

math.h

Natural logarithm.

DECLARATION

double log(double *arg*)

PARAMETERS

arg A double value.

RETURN VALUE

The double natural logarithm of *arg*.

DESCRIPTION

Computes the natural logarithm of a number.

log10

math.h

Base-10 logarithm.

DECLARATION

double log10(double *arg*)

PARAMETERS

arg A double number.

RETURN VALUE

The double base-10 logarithm of *arg*.

DESCRIPTION

Computes the base-10 logarithm of a number.

malloc

stdlib.h

Allocates memory.

DECLARATION

void *malloc(size_t *size*)

PARAMETERS

size A size_t object specifying the size of the object.

RETURN VALUE

<i>Result</i>	<i>Value</i>
Successful	A pointer to the start (lowest byte address) of the memory block.
Unsuccessful	Zero, if there is no memory block of the required size or greater available.

DESCRIPTION

Allocates a memory block for an object of the specified size.

The availability of memory depends on the size of the heap. For more information about changing the heap size refer to *Heap size*, page 59.

memchr

string.h

Searches for a character in memory.

DECLARATION

```
void *memchr(const void *s, int c, size_t n)
```

PARAMETERS

<i>s</i>	A pointer to an object.
<i>c</i>	An int representing a character.
<i>n</i>	A value of type size_t specifying the size of each object.

RETURN VALUE

<i>Result</i>	<i>Value</i>
Successful	A pointer to the first occurrence of <i>c</i> in the <i>n</i> characters pointed to by <i>s</i> .
Unsuccessful	Null.

DESCRIPTION

Searches for the first occurrence of a character in a pointed-to region of memory of a given size.

Both the single character and the characters in the object are treated as unsigned.

memcmp

string.h

Compares memory.

DECLARATION

int memcmp(const void *s1, const void *s2, size_t n)

PARAMETERS

s1

A pointer to the first object.

s2

A pointer to the second object.

n

A value of type size_t specifying the size of each object.

RETURN VALUE

An integer indicating the result of comparison of the first *n* characters of the object pointed to by *s1* with the first *n* characters of the object pointed to by *s2*:

Return value	Meaning
>0	s1 > s2
=0	s1 = s2
<0	s1 < s2

DESCRIPTION

Compares the first *n* characters of two objects.

memcpy

string.h

Copies memory.

DECLARATION

void *memcpy(void *s1, const void *s2, size_t n)

PARAMETERS

s1

A pointer to the destination object.

s2

A pointer to the source object.

n The number of characters to be copied.

RETURN VALUE

s1.

DESCRIPTION

Copies a specified number of characters from a source object to a destination object.

If the objects overlap, the result is undefined, so `memmove` should be used instead.

memmove

`string.h`

Moves memory.

DECLARATION

```
void *memmove(void *s1, const void *s2, size_t n)
```

PARAMETERS

s1 A pointer to the destination object.

s2 A pointer to the source object.

n The number of characters to be copied.

RETURN VALUE

s1.

DESCRIPTION

Copies a specified number of characters from a source object to a destination object.

Copying takes place as if the source characters are first copied into a temporary array that does not overlap either object, and then the characters from the temporary array are copied into the destination object.

memset

string.h

Sets memory.

DECLARATION

```
void *memset(void *s, int c, size_t n)
```

PARAMETERS

s A pointer to the destination object.

c An int representing a character.

n The size of the object.

RETURN VALUE

s.

DESCRIPTION

Copies a character (converted to an unsigned char) into each of the first specified number of characters of the destination object.

modf

math.h

Fractional and integer parts.

DECLARATION

```
double modf(double value, double *iptr)
```

PARAMETERS

value A double value.

iptr A pointer to the double that is to receive the integral part of value.

RETURN VALUE

The fractional part of *value*.

DESCRIPTION

Computes the fractional and integer parts of *value*. The sign of both parts is the same as the sign of *value*.

pow

math.h

Raises to the power.

DECLARATION

double pow(double *arg1*, double *arg2*)

PARAMETERS

arg1 The double number.

arg2 The double power.

RETURN VALUE

arg1 raised to the power of *arg2*.

DESCRIPTION

Computes a number raised to a power.

printf

stdio.h

Writes formatted data.

DECLARATION

int printf(const char **format*, ...)

PARAMETERS

format A pointer to the format string.

... The optional values that are to be printed under the control of *format*.

RETURN VALUE

<i>Result</i>	<i>Value</i>
---------------	--------------

Successful	The number of characters written.
------------	-----------------------------------

Unsuccessful	A negative value, if an error occurred.
--------------	---

DESCRIPTION

Writes formatted data to the standard output stream, returning the number of characters written or a negative value if an error occurred.

Since a complete formatter demands a lot of space there are several different formatters to choose. For more information see the chapter *Configuration*.

format is a string consisting of a sequence of characters to be printed and conversion specifications. Each conversion specification causes the next successive argument following the *format* string to be evaluated, converted, and written.

The form of a conversion specification is as follows:

% [*flags*] [*field_width*] [*.precision*] [*length_modifier*]
conversion

Items inside [] are optional.

Flags

The *flags* are as follows:

Flag	Effect
-	Left adjusted field.
+	Signed values will always begin with plus or minus sign.
space	Values will always begin with minus or space.
#	Alternate form:

Specifier	Effect
octal	First digit will always be a zero.
G g	Decimal point printed and trailing zeros kept.
E e f	Decimal point printed.
X	Non-zero values prefixed with 0X.
X	Non-zero values prefixed with 0X.

<i>Flag</i>	<i>Effect</i>
0	Zero padding to field width (for d, i, o, u, x, X, e, E, f, g, and G specifiers).

Field width

The `field_width` is the number of characters to be printed in the field. The field will be padded with space if needed. A negative value indicates a left-adjusted field. A field width of `*` stands for the value of the next successive argument, which should be an integer.

Precision

The `precision` is the number of digits to print for integers (d, i, o, u, x, and X), the number of decimals printed for floating-point values (e, E, and f), and the number of significant digits for g and G conversions. A field width of `*` stands for the value of the next successive argument, which should be an integer.

Length modifier

The effect of each *length_modifier* is as follows:

<i>Modifier</i>	<i>Use</i>
h	Before d, i, u, x, X, or o specifiers to denote a short int or unsigned short int value.
l	Before d, i, u, x, X, or o specifiers to denote a long integer or unsigned long value.
L	Before e, E, f, g, or G specifiers to denote a long double value.

Conversion

The result of each value of *conversion* is as follows:

<i>Conversion</i>	<i>Result</i>
d	Signed decimal value.
i	Signed decimal value.
o	Unsigned octal value.
u	Unsigned decimal value.

<i>Conversion</i>	<i>Result</i>
x	Unsigned hexadecimal value, using lower case (0-9, a-f).
X	Unsigned hexadecimal value, using upper case (0-9, A-F).
e	Double value in the style [-]d.ddde+dd.
E	Double value in the style [-]d.dddE+dd.
f	Double value in the style [-]ddd.ddd.
g	Double value in the style of f or e, whichever is the more appropriate.
G	Double value in the style of F or E, whichever is the more appropriate.
C	Single character constant.
s	String constant.
p	Pointer value (address).
n	No output, but store the number of characters written so far in the integer pointed to by the next argument.
%	% character.

Note that promotion rules convert all `char` and short `int` arguments to `int` while `floats` are converted to `double`.

`printf` calls the library function `putchar`, which must be adapted for the target hardware configuration.

The source of `printf` is provided in the file `printf.c`. The source of a reduced version that uses less program space and stack is provided in the file `intwri.c`.

EXAMPLES

After the following C statements:

```
int i=6, j=-6;
char *p = "ABC";
long l=100000;
```

```
float f1 = 0.0000001;
f2 = 750000;
double d = 2.2;
```

the effect of different printf function calls is shown in the following table where ° represents space:

<i>Statement</i>	<i>Output</i>	<i>Characters output</i>
printf("%c",p[1])	B	1
printf("%d",i)	6	1
printf("%3d",i)	°6	3
printf("%.3d",i)	°°6	3
printf("%-10.3d",i)	006°°°°°°°°	10
printf("%10.3d",i)	°°°°°°°°006	10
printf("Value=%+3d",i)	Value=°+6	9
printf("%10.*d",i,j)	°°°-000006	10
printf("String=%s",p)	String=[ABC]	12
printf("Value=%lX",l)	Value=186A0	11
printf("%f",f1)	0.000000	8
printf("%f",f2)	750000.000000	13
printf("%e",f1)	1.000000e-07	12
printf("%16e",d)	°°°°2.200000e+00	16
printf("%.4e",d)	2.2000e+00	10
printf("%g",f1)	1e-07	5
printf("%g",f2)	750000	6
printf("%g",d)	2.2	3

putchar

stdio.h

Puts character.

DECLARATION

int putchar(int value)

PARAMETERS

value

The int representing the character to be put.

RETURN VALUE

Result	Value
Successful	value.
Unsuccessful	The EOF macro.

DESCRIPTION

Writes a character to standard output.

You should customize this function for the particular target hardware configuration. The function is supplied in source format in the file putchar.c.

This function is called by printf.

puts

stdio.h

Puts string.

DECLARATION

int puts(const char *s)

PARAMETERS

s

A pointer to the string to be put.

130

RETURN VALUE

<i>Result</i>	<i>Value</i>
Successful	A non-negative value.
Unsuccessful	-1 if an error occurred.

DESCRIPTION

Writes a string followed by a new-line character to the standard output stream.

qsort

stdlib.h

Makes a generic sort of an array.

DECLARATION

```
void qsort (const void *base, size_t nmemb, size_t size,
int (*compare) (const void *_key, const void *_base));
```

PARAMETERS

<i>base</i>	Pointer to the array to sort.
<i>nmemb</i>	Dimension of the array pointed to by <i>base</i> .
<i>size</i>	Size of the array elements.
<i>compare</i>	The comparison function, which takes two arguments and returns: < 0 (negative value)if <i>_key</i> is less than <i>_base</i> . 0 if <i>_key</i> equals <i>_base</i> . > 0 (positive value)if <i>_key</i> is greater than <i>_base</i> .

RETURN VALUE

None.

DESCRIPTION

Sorts an array of *nmemb* objects pointed to by *base*.

rand

stdlib.h

Random number.

DECLARATION

int rand(void)

PARAMETERS

None.

RETURN VALUE

The next int in the random number sequence.

DESCRIPTION

Computes the next in the current sequence of pseudo-random integers, converted to lie in the range `[0, RAND_MAX]`.

See *srand*, page 139, for a description of how to seed the pseudo-random sequence.

realloc

stdlib.h

Reallocates memory.

DECLARATION

void *realloc(void *ptr, size_t size)

PARAMETERS

ptr A pointer to the start of the memory block.

size A value of type `size_t` specifying the size of the object.

RETURN VALUE

<i>Result</i>	<i>Value</i>
Successful	A pointer to the start (lowest address) of the memory block.
Unsuccessful	Null, if no memory block of the required size or greater was available.

DESCRIPTION

Changes the *size* of a memory block (which must be allocated by `malloc`, `calloc`, or `realloc`).

scanf

`stdio.h`

Reads formatted data.

DECLARATION

```
int scanf(const char *format, ...)
```

PARAMETERS

<i>format</i>	A pointer to a format string.
<i>...</i>	Optional pointers to the variables that are to receive values.

RETURN VALUE

<i>Result</i>	<i>Value</i>
Successful	The number of successful conversions.
Unsuccessful	-1 if the input was exhausted.

DESCRIPTION

Reads formatted data from standard input.

Since a complete formatter demands a lot of space there are several different formatters to choose. For more information, see *Input and output*, page 56.

format is a string consisting of a sequence of ordinary characters and conversion specifications. Each ordinary character reads a matching character from the input. Each conversion specification accepts input meeting the specification, converts it, and assigns it to the object pointed to by the next successive argument following *format*.

If the format string contains white-space characters, input is scanned until a non-white-space character is found.

The form of a conversion specification is as follows:

% [*assign_suppress*] [*field_width*] [*length_modifier*]
conversion

Items inside [] are optional.

Assign suppress

If a * is included in this position, the field is scanned but no assignment is carried out.

field_width

The *field_width* is the maximum field to be scanned. The default is until no match occurs.

length_modifier

The effect of each *length_modifier* is as follows:

<i>Length modifier</i>	<i>Before</i>	<i>Meaning</i>
l	d, i, or n	long int as opposed to int.
	o, u, or x	unsigned long int as opposed to unsigned int.
	e, E, g, G, or f	double operand as opposed to <i>float</i> .
h	d, i, or n	short int as opposed to int.
	o, u, or x	unsigned short int as opposed to unsigned int.
L	e, E, g, G, or f	long double operand as opposed to float.

Conversion

The meaning of each conversion is as follows:

Conversion Meaning

d	Optionally signed decimal integer value.
i	Optionally signed integer value in standard C notation, that is, is decimal, octal (0n) or hexadecimal (0xn, 0Xn).
o	Optionally signed octal integer.
u	Unsigned decimal integer.
x	Optionally signed hexadecimal integer.
X	Optionally signed hexadecimal integer (equivalent to x).
f	Floating-point constant.
e E g G	Floating-point constant (equivalent to f).
s	Character string.
c	One or <code>field_width</code> characters.
n	No read, but store number of characters read so far in the integer pointed to by the next argument.
p	Pointer value (address).
[Any number of characters matching any of the characters before the terminating <code>]</code> . For example, <code>[abc]</code> means a, b, or c.
[]	Any number of characters matching <code>]</code> or any of the characters before the further, terminating <code>]</code> . For example, <code>[]abc</code> means <code>]</code> , a, b, or c.
[^	Any number of characters not matching any of the characters before the terminating <code>]</code> . For example, <code>[^abc</code> means not a, b, or c.
[^]	Any number of characters not matching <code>]</code> or any of the characters before the further, terminating <code>]</code> . For example, <code>[^]abc</code> means not <code>]</code> , a, b, or c.
%	% character.

In all conversions except `c`, `n`, and all varieties of `['`, leading white-space characters are skipped.

`scanf` indirectly calls `getchar`, which must be adapted for the actual target hardware configuration.

EXAMPLES

For example, after the following program:

```
int n, i;
char name[50];
float x;
n = scanf("%d%f%s", &i, &x, name)
```

this input line:

```
25 54.32E-1 Hello World
```

will set the variables as follows:

```
n = 3, i = 25, x = 5.432, name="Hello World"
```

and this function:

```
scanf("%2d%f*d %[0123456789]", &i, &x, name)
```

with this input line:

```
56789 0123 56a72
```

will set the variables as follows:

```
i = 56, x = 789.0, name="56" (0123 unassigned)
```

sin

`math.h`

Sine.

DECLARATION

```
double sin(double arg)
```

PARAMETERS

arg A double value in radians.

RETURN VALUE

The double sine of *arg*.

DESCRIPTION

Computes the sine of a number.

sinh

math.h

Hyperbolic sine.

DECLARATION

double sinh(double *arg*)

PARAMETERS

arg A double value in radians.

RETURN VALUE

The double hyperbolic sine of *arg*.

DESCRIPTION

Computes the hyperbolic sine of *arg* radians.

sprintf

stdio.h

Writes formatted data to a string.

DECLARATION

int sprintf(char **s*, const char **format*, ...)

PARAMETERS

s A pointer to the string that is to receive the formatted data.

format A pointer to the format string.

... The optional values that are to be printed under the control of *format*.

RETURN VALUE

<i>Result</i>	<i>Value</i>
Successful	The number of characters written.
Unsuccessful	A negative value if an error occurred.

DESCRIPTION

Operates exactly as `printf` except the output is directed to a string. See *printf*, page 125, for details.

`sprintf` does not use the function `putchar`, and therefore can be used even if `putchar` is not available for the target configuration.

Since a complete formatter demands a lot of space there are several different formatters to choose. For more information, see *Input and output*, page 56.

sqrt

`math.h`

Square root.

DECLARATION

`double sqrt(double arg)`

PARAMETERS

arg A double value.

RETURN VALUE

The double square root of *arg*.

DESCRIPTION

Computes the square root of a number.

srand

stdlib.h

Sets random number sequence.

DECLARATION

```
void srand(unsigned int seed)
```

PARAMETERS

seed An unsigned int value identifying the particular random number sequence.

RETURN VALUE

None.

DESCRIPTION

Selects a repeatable sequence of pseudo-random numbers.

The function rand is used to get successive random numbers from the sequence. If rand is called before any calls to srand have been made, the sequence generated is that which is generated after srand(1).

sscanf

stdio.h

Reads formatted data from a string.

DECLARATION

```
int sscanf(const char *s, const char *format, ...)
```

PARAMETERS

s A pointer to the string containing the data.

format A pointer to a format string.

... Optional pointers to the variables that are to receive values.

RETURN VALUE

<i>Result</i>	<i>Value</i>
Successful	The number of characters written.
Unsuccessful	A negative value if an error occurred.

DESCRIPTION

Operates exactly as `scanf` except the input is taken from the string `s`. See `scanf` for details.

The function `sscanf` does not use `getchar`, and so can be used even when `getchar` is not available for the target configuration.

Since a complete formatter demands a lot of space there are several different formatters to choose. For more information see the chapter *Configuration*.

strcat

`string.h`

Concatenates strings.

DECLARATION

`char *strcat(char *s1, const char *s2)`

PARAMETERS

<i>s1</i>	A pointer to the first string.
<i>s2</i>	A pointer to the second string.

RETURN VALUE

s1.

DESCRIPTION

Appends a copy of the second string to the end of the first string. The initial character of the second string overwrites the terminating null character of the first string.

strchr

string.h

Searches for a character in a string.

DECLARATION

char *strchr(const char *s, int c)

PARAMETERS

c An int representation of a character.
s A pointer to a string.

RETURN VALUE

If successful, a pointer to the first occurrence of *c* (converted to a char) in the string pointed to by *s*.

If unsuccessful due to *c* not being found, null.

DESCRIPTION

Finds the first occurrence of a character (converted to a char) in a string. The terminating null character is considered to be part of the string.

strcmp

string.h

Compares two strings.

DECLARATION

int strcmp(const char *s1, const char *s2)

PARAMETERS

s1 A pointer to the first string.
s2 A pointer to the second string.

RETURN VALUE

The int result of comparing the two strings:

<i>Return value</i>	<i>Meaning</i>
>0	s1 > s2

<i>Return value</i>	<i>Meaning</i>
=0	s1 = s2
<0	s1 < s2

DESCRIPTION

Compares the two strings.

strcoll

string.h

Compares strings.

DECLARATION

int strcoll(const char *s1, const char *s2)

PARAMETERS

- s1 A pointer to the first string.
- s2 A pointer to the second string.

RETURN VALUE

The int result of comparing the two strings:

<i>Return value</i>	<i>Meaning</i>
>0	s1 > s2
=0	s1 = s2
<0	s1 < s2

DESCRIPTION

Compares the two strings. This function operates identically to strcmp and is provided for compatibility only.

strcpy

string.h

Copies string.

DECLARATION

char *strcpy(char *s1, const char *s2)

PARAMETERS*s1* A pointer to the destination object.*s2* A pointer to the source string.**RETURN VALUE***s1*.**DESCRIPTION**Copies a string into an object.

strcspn

string.h

Spans excluded characters in string.

DECLARATION

size_t strcspn(const char *s1, const char *s2)

PARAMETERS*s1* A pointer to the subject string.*s2* A pointer to the object string.**RETURN VALUE**

The int length of the maximum initial segment of the string pointed to by *s1* that consists entirely of characters *not* from the string pointed to by *s2*.

DESCRIPTION

Finds the maximum initial segment of a subject string that consists entirely of characters *not* from an object string.

strerror

string.h

Gives an error message string.

DECLARATION

char * strerror (int *errnum*)

PARAMETERS

errnum

The error message to return.

RETURN VALUE

The function returns the following strings.

<i>errnum</i>	<i>String returned</i>
EZERO	"no error"
EDOM	"domain error"
ERANGE	"range error"
<i>errnum</i> < 0 <i>errnum</i> > Max_err_num	"unknown error"
All other numbers	"error No. <i>errnum</i> "

DESCRIPTION

Returns an error message string.

strlen

string.h

String length.

DECLARATION

size_t strlen(const char *s)

PARAMETERS

s

A pointer to a string.

RETURN VALUE

An object of type `size_t` indicating the length of the string.

DESCRIPTION

Finds the number of characters in a string, not including the terminating null character.

strncat

string.h

Concatenates a specified number of characters with a string.

DECLARATION

```
char *strncat(char *s1, const char *s2, size_t n)
```

PARAMETERS

s1 A pointer to the destination string.
s2 A pointer to the source string.
n The number of characters of the source string to use.

RETURN VALUE

s1.

DESCRIPTION

Appends not more than *n* initial characters from the source string to the end of the destination string.

strncmp

string.h

Compares a specified number of characters with a string.

DECLARATION

```
int strncmp(const char *s1, const char *s2, size_t n)
```

PARAMETERS

s1 A pointer to the first string.
s2 A pointer to the second string.
n The number of characters of the source string to compare.

RETURN VALUE

The `int` result of the comparison of not more than n initial characters of the two strings:

<i>Return value</i>	<i>Meaning</i>
>0	$s1 > s2$
$=0$	$s1 = s2$
<0	$s1 < s2$

DESCRIPTION

Compares not more than n initial characters of the two strings.

strncpy

`string.h`

Copies a specified number of characters from a string.

DECLARATION

`char *strncpy(char *s1, const char *s2, size_t n)`

PARAMETERS

- $s1$ A pointer to the destination object.
- $s2$ A pointer to the source string.
- n The number of characters of the source string to copy.

RETURN VALUE

$s1$.

DESCRIPTION

Copies not more than n initial characters from the source string into the destination object.

strpbrk

string.h

Finds any one of specified characters in a string.

DECLARATION

char *strpbrk(const char *s1, const char *s2)

PARAMETERS*s1* A pointer to the subject string.*s2* A pointer to the object string.**RETURN VALUE**

<i>Result</i>	<i>Value</i>
---------------	--------------

Successful	A pointer to the first occurrence in the subject string of any character from the object string.
------------	--

Unsuccessful	Null if none were found.
--------------	--------------------------

DESCRIPTION

Searches one string for any occurrence of any character from a second string.

strrchr

string.h

Finds character from right of string.

DECLARATION

char *strrchr(const char *s, int c)

PARAMETERS*s* A pointer to a string.*c* An int representing a character.**RETURN VALUE**If successful, a pointer to the last occurrence of *c* in the string pointed to by *s*.

DESCRIPTION

Searches for the last occurrence of a character (converted to a `char`) in a string. The terminating null character is considered to be part of the string.

strspn

`string.h`

Spans characters in a string.

DECLARATION

```
size_t strspn(const char *s1, const char *s2)
```

PARAMETERS

s1 A pointer to the subject string.

s2 A pointer to the object string.

RETURN VALUE

The length of the maximum initial segment of the string pointed to by *s1* that consists entirely of characters from the string pointed to by *s2*.

DESCRIPTION

Finds the maximum initial segment of a subject string that consists entirely of characters from an object string.

strstr

`string.h`

Searches for a substring.

DECLARATION

```
char *strstr(const char *s1, const char *s2)
```

PARAMETERS

s1 A pointer to the subject string.

s2 A pointer to the object string.

RETURN VALUE

<i>Result</i>	<i>Value</i>
Successful	A pointer to the first occurrence in the string pointed to by <i>s1</i> of the sequence of characters (excluding the terminating null character) in the string pointed to by <i>s2</i> .
Unsuccessful	Null if the string was not found. <i>s1</i> if <i>s2</i> is pointing to a string with zero length.

DESCRIPTION

Searches one string for an occurrence of a second string.

strtod

stdlib.h

Converts a string to double.

DECLARATION

double strtod(const char **nptr*, char ***endptr*)

PARAMETERS

<i>nptr</i>	A pointer to a string.
<i>endptr</i>	A pointer to a pointer to a string.

RETURN VALUE

<i>Result</i>	<i>Value</i>
Successful	The double result of converting the ASCII representation of an floating-point constant in the string pointed to by <i>nptr</i> , leaving <i>endptr</i> pointing to the first character after the constant.
Unsuccessful	Zero, leaving <i>endptr</i> indicating the first non-space character.

DESCRIPTION

Converts the ASCII representation of a number into a double, stripping any leading white space.

strtok

string.h

Breaks a string into tokens.

DECLARATION

char *strtok(char *s1, const char *s2)

PARAMETERS*s1* A pointer to a string to be broken into tokens.*s2* A pointer to a string of delimiters.**RETURN VALUE**

<i>Result</i>	<i>Value</i>
Successful	A pointer to the token.
Unsuccessful	Zero.

DESCRIPTION

Finds the next token in the string *s1*, separated by one or more characters from the string of delimiters *s2*.

The first time you call `strtok`, *s1* should be the string you want to break into tokens. `strtok` saves this string. On each subsequent call, *s1* should be NULL. `strtok` searches for the next token in the string it saved. *s2* can be different from call to call.

If `strtok` finds a token, it returns a pointer to the first character in it. Otherwise it returns NULL. If the token is not at the end of the string, `strtok` replaces the delimiter with a null character (`\0`).

strtol

stdlib.h

Converts a string to a long integer.

DECLARATION

long int strtol(const char *nptr, char **endptr, int base)

PARAMETERS

<i>nptr</i>	A pointer to a string.
<i>endptr</i>	A pointer to a pointer to a string.
<i>base</i>	An <code>int</code> value specifying the base.

RETURN VALUE

<i>Result</i>	<i>Value</i>
Successful	The <code>long int</code> result of converting the ASCII representation of an integer constant in the string pointed to by <i>nptr</i> , leaving <i>endptr</i> pointing to the first character after the constant.
Unsuccessful	Zero, leaving <i>endptr</i> indicating the first non-space character.

DESCRIPTION

Converts the ASCII representation of a number into a `long int` using the specified base, and stripping any leading white space.

If the base is zero the sequence expected is an ordinary integer. Otherwise the expected sequence consists of digits and letters representing an integer with the radix specified by *base* (must be between 2 and 36). The letters `[a, z]` and `[A, Z]` are ascribed the values 10 to 35. If the base is 16, the `0x` portion of a hex integer is allowed as the initial sequence.

strtoul

`stdlib.h`

Converts a string to an unsigned long integer.

DECLARATION

```
unsigned long int strtoul(const char *nptr, char
**endptr, base int)
```

PARAMETERS

<i>nptr</i>	A pointer to a string.
<i>endptr</i>	A pointer to a pointer to a string.
<i>base</i>	An <code>int</code> value specifying the base.

RETURN VALUE

<i>Result</i>	<i>Value</i>
Successful	The unsigned long int result of converting the ASCII representation of an integer constant in the string pointed to by <i>nptr</i> , leaving <i>endptr</i> pointing to the first character after the constant.
Unsuccessful	Zero, leaving <i>endptr</i> indicating the first non-space character.

DESCRIPTION

Converts the ASCII representation of a number into an unsigned long int using the specified base, stripping any leading white space.

If the base is zero the sequence expected is an ordinary integer. Otherwise the expected sequence consists of digits and letters representing an integer with the radix specified by *base* (must be between 2 and 36). The letters [a, z] and [A, Z] are ascribed the values 10 to 35. If the base is 16, the 0x portion of a hex integer is allowed as the initial sequence.

strxfrm

string.h

Transforms a string and returns the length.

DECLARATION

size_t strxfrm(char *s1, const char *s2, size_t n)

PARAMETERS

<i>s1</i>	Return location of the transformed string.
<i>s2</i>	String to transform.
<i>n</i>	Maximum number of characters to be placed in <i>s1</i> .

RETURN VALUE

The length of the transformed string, not including the terminating null character.

DESCRIPTION

The transformation is such that if the `strcmp` function is applied to two transformed strings, it returns a value corresponding to the result of the `strcoll` function applied to the same two original strings.

tan

`math.h`

Tangent.

DECLARATION

`double tan(double arg)`

PARAMETERS

arg A double value in radians.

RETURN VALUE

The double tangent of *arg*.

DESCRIPTION

Computes the tangent of *arg* radians.

tanh

`math.h`

Hyperbolic tangent.

DECLARATION

`double tanh(double arg)`

PARAMETERS

arg A double value in radians.

RETURN VALUE

The double hyperbolic tangent of *arg*.

DESCRIPTION

Computes the hyperbolic tangent of *arg* radians.

tolower`ctype.h`

Converts to lower case.

DECLARATION

```
int tolower(int c)
```

PARAMETERS

`c` The int representation of a character.

RETURN VALUE

The int representation of the lower case character corresponding to `c`.

DESCRIPTION

Converts a character into lower case.

toupper`ctype.h`

Converts to upper case.

DECLARATION

```
int toupper(int c)
```

PARAMETERS

`c` The int representation of a character.

RETURN VALUE

The int representation of the upper case character corresponding to `c`.

DESCRIPTION

Converts a character into upper case.

va_arg

stdarg.h

Next argument in function call.

DECLARATIONtype va_arg(va_list *ap*, *mode*)**PARAMETERS***ap* A value of type va_list.*mode* A type name such that the type of a pointer to an object that has the specified type can be obtained simply by postfixing a * to type.**RETURN VALUE**

See below.

DESCRIPTION

A macro that expands to an expression with the type and value of the next argument in the function call. After initialization by va_start, this is the argument after that specified by parmN. va_arg advances *ap* to deliver successive arguments in order.

For an example of the use of va_arg and associated macros, see the files printf.c and intwri.c.

va_end

stdarg.h

Ends reading function call arguments.

DECLARATIONvoid va_end(va_list *ap*)**PARAMETERS***ap* A pointer of type va_list to the variable-argument list.**RETURN VALUE**

See below.

DESCRIPTION

A macro that facilitates normal return from the function whose variable argument list was referenced by the expansion `va_start` that initialized `va_list ap`.

va_list

`stdarg.h`

Argument list type.

DECLARATION

`void *va_list[1]`

PARAMETERS

None.

RETURN VALUE

See below.

DESCRIPTION

An array type suitable for holding information needed by `va_arg` and `va_end`.

va_start

`stdarg.h`

Starts reading function call arguments.

DECLARATION

`void va_start(va_list ap, parmN)`

PARAMETERS

ap A pointer of type `va_list` to the variable-argument list.

parmN The identifier of the rightmost parameter in the variable parameter list in the function definition.

RETURN VALUE

See below.

DESCRIPTION

A macro that initializes *ap* for use by *va_arg* and *va_end*.

vprintf

stdio.h

Writes formatted data to standard output.

DECLARATION

int vprintf(const char **format*, va_list *argptr*)

PARAMETERS

<i>format</i>	A pointer to the format string.
<i>argptr</i>	List of arguments.

RETURN VALUE

<i>Result</i>	<i>Value</i>
Successful	The number of characters written.
Unsuccessful	A negative value, if an error occurred.

DESCRIPTION

Performs the same function as *printf*, but accepts a pointer to a list of arguments rather than the arguments themselves. For format details see *printf*, page 125, and for argument list details see *va_list*, page 156.

vsprintf

stdio.h

Writes formatted data to a buffer.

DECLARATION

int vsprintf(char **s*, const char **format*, va_list *argptr*)

PARAMETERS

<i>s</i>	A pointer to the string that is to receive the formatted data.
<i>format</i>	A pointer to the format string.
<i>argptr</i>	List of arguments.

RETURN VALUE

<i>Result</i>	<i>Value</i>
Successful	The number of characters written.
Unsuccessful	A negative value, if an error occurred.

DESCRIPTION

Performs the same function as `sprintf`, but accepts a pointer to a list of arguments rather than the arguments themselves. For details of *s* and *format* see *sprintf*, page 137, and for argument list details see *va_list*, page 156.

`_formatted_read`

`icclbutl.h`

Reads formatted data.

DECLARATION

```
int _formatted_read (const char **line, const char
**format, va_list ap)
```

PARAMETERS

<i>line</i>	A pointer to a pointer to the data to scan.
<i>format</i>	A pointer to a pointer to a standard <code>scanf</code> format specification string.
<i>ap</i>	A pointer of type <code>va_list</code> to the variable argument list.

RETURN VALUE

The number of successful conversions.

DESCRIPTION

Reads formatted data. This function is the basic formatter of `scanf`.

`_formatted_read` is concurrently reusable (reentrant).

Note that the use of `_formatted_read` requires the special ANSI-defined macros in the file `stdarg.h`, described above. In particular:

- ◆ There must be a variable *ap* of type `va_list`.

- ◆ There must be a call to `va_start` before calling `_formatted_read`.
- ◆ There must be a call to `va_end` before leaving the current context.
- ◆ The argument to `va_start` must be the formal parameter immediately to the left of the variable argument list.

_formatted_write

icclbut1.h

Formats and writes data.

DECLARATION

```
int _formatted_write (const char *format, void outputf
(char, void *), void *sp, va_list ap)
```

PARAMETERS

<i>format</i>	A pointer to standard printf/sprintf format specification string.
<i>outputf</i>	A function pointer to a routine that actually writes a single character created by <code>_formatted_write</code> . The first parameter to this function contains the actual character value and the second a pointer whose value is always equivalent to the third parameter of <code>_formatted_write</code> .
<i>sp</i>	A pointer to some type of data structure that the low-level output function may need. If there is no need for anything more than just the character value, this parameter must still be specified with <code>(void *) 0</code> as well as declared in the output function.
<i>ap</i>	A pointer of type <code>va_list</code> to the variable-argument list.

RETURN VALUE

The number of characters written.

DESCRIPTION

Formats write data. This function is the basic formatter of `printf` and `sprintf`, but through its universal interface can easily be adapted for writing to non-standard display devices.

Since a complete formatter demands a lot of space there are several different formatters to choose. For more information see the chapter *Configuration*.

`_formatted_write` is concurrently reusable (reentrant).

Note that the use of `_formatted_write` requires the special ANSI-defined macros in the file `stdarg.h`, described above. In particular:

- ◆ There must be a variable *ap* of type `va_list`.
- ◆ There must be a call to `va_start` before calling `_formatted_write`.
- ◆ There must be a call to `va_end` before leaving the current context.
- ◆ The argument to `va_start` must be the formal parameter immediately to the left of the variable argument list.

For an example of how to use `_formatted_write`, see the file `printf.c`.

_medium_read

`icclbutl.h`

Reads formatted data excluding floating-point numbers.

DECLARATION

```
int _medium_read (const char **line, const char **format,
va_list ap)
```

PARAMETERS

- | | |
|---------------|--|
| <i>line</i> | A pointer to a pointer to the data to scan. |
| <i>format</i> | A pointer to a pointer to a standard <code>scanf</code> format specification string. |
| <i>ap</i> | A pointer of type <code>va_list</code> to the variable argument list. |

RETURN VALUE

The number of successful conversions.

DESCRIPTION

A reduced version of `_formatted_read` which is half the size, but does not support floating-point numbers.

For further information see *_formatted_read*, page 158.

`_medium_write`

`icclbutl.h`

Writes formatted data excluding floating-point numbers.

DECLARATION

```
int _medium_write (const char *format, void outputf(char,
void *), void *sp, va_list ap)
```

PARAMETERS

<i>format</i>	A pointer to standard <code>printf/sprintf</code> format specification string.
<i>outputf</i>	A function pointer to a routine that actually writes a single character created by <code>_formatted_write</code> . The first parameter to this function contains the actual character value and the second a pointer whose value is always equivalent to the third parameter of <code>_formatted_write</code> .
<i>sp</i>	A pointer to some type of data structure that the low-level output function may need. If there is no need for anything more than just the character value, this parameter must still be specified with <code>(void *) 0</code> as well as declared in the output function.
<i>ap</i>	A pointer of type <code>va_list</code> to the variable-argument list.

RETURN VALUE

The number of characters written.

DESCRIPTION

A reduced version of `_formatted_write` which is half the size, but does not support floating-point numbers.

For further information see *_formatted_write*, page 159.

_small_write

icclbut1.h

Small formatted data write routine.

DECLARATION

```
int _small_write (const char *format, void outputf(char,  
void *), void *sp, va_list ap)
```

PARAMETERS

<i>format</i>	A pointer to standard printf/sprintf format specification string.
<i>outputf</i>	A function pointer to a routine that actually writes a single character created by <code>_formatted_write</code> . The first parameter to this function contains the actual character value and the second a pointer whose value is always equivalent to the third parameter of <code>_formatted_write</code> .
<i>sp</i>	A pointer to some type of data structure that the low-level output function may need. If there is no need for anything more than just the character value, this parameter must still be specified with <code>(void *) 0</code> as well as declared in the output function.
<i>ap</i>	A pointer of type <code>va_list</code> to the variable-argument list.

RETURN VALUE

The number of characters written.

DESCRIPTION

This is a small version of `_formatted_write` which is about a quarter of the size.

The `_small_write` formatter supports only the following specifiers for int objects:

%%, %d, %o, %c, %s, and %x

It does not support field width or precision arguments, and no diagnostics will be produced if unsupported specifiers or modifiers are used. For further information see `_formatted_write`, page 159.

EXTENDED KEYWORDS

REFERENCE

This chapter describes the non-standard keywords that support specific features of the PICmicro™ microcontroller:

- ◆ `__bankN` which controls the storage of variables and the representation of pointers for the SFR/GPR banks.
- ◆ `__nonbanked` which controls the storage of variables and the representation of pointers for memory addresses that are not bank-switched in the SFR/GPR area.
- ◆ `__bank` which represents a generic bank pointer that is able to point to all variables located in the SFR/GPR area.
- ◆ `__eeprom` which controls the storage of variables located in EEPROM memory and the associated pointer type.
- ◆ `__constptr` which represents a pointer that is able to point to variables declared as `const` and located in the program memory as a series of RETLW instructions.
- ◆ `__ptable` and `__rtable` which controls the storage of variables and the representation of pointers for TABLES located in program memory.
- ◆ `__dptr` which represents the default pointer type and is able to point to all memories represented by the above storage keywords.
- ◆ `__no_init` tells the compiler to place the variable in a special `no_init` segment for the above storage type which will not be initialized at startup.
- ◆ `__bankN_func` that tells the compiler which bank a function should use for the parameters and auto variables. Also used as a function pointer.
- ◆ `__interrupt`, which supports interrupt functions.
- ◆ `__setN`, which specifies save area for interrupt functions.
- ◆ `__monitor`, which supports atomic execution of a function.

STORAGE

By default the compiler places variables in bank 0. Variables declared `const` will be placed in code memory using `RETLW` instructions. The default is overridden by the following keywords:

- ◆ `__bankN`, to place the variable in the GPR area of the specified bank, which is accessed using efficient 7/8-bit addressing using a minimum of bank switching.
- ◆ `__eeprom`, to place the variable in the EEPROM area for processors that have such an area. It is accessed through library routines, which the user should modify to conform to the hardware used.
- ◆ `__ptable` and `__rtable`, to place a variable in code memory to be accessed through the `TABLE` instruction set. All access is done through library routines, which the user should modify to conform to the hardware used. Variables defined as `__ptable` and `__rtable` can only be of integer types (`char`, `short`, `long`, `float`), not array nor `struct`.

Different representations are used for pointers into different memories. A `__bankN` and `__eeprom` pointer is 8 bits in size whereas `__bank`, `__ptable`, `__rtable` and `__constptr` are 16-bits wide and `__dptr` is 24-bit wide. They also differ in the way the access memory.

The keywords follow the same syntax as the type qualifiers `const` and `volatile`. The following declaration places the variable `i` in a `__bank1` memory segment:

```
__bank1 int i;
```

Note however the difference between placing the keyword before and after the type specifier. When the keyword is placed before the type, it affects all the identifiers of the declaration. Otherwise, the keyword only affects the identifier that follows immediately. In the following example, `a`, `b`, and `c` are thus placed in `__bank1` memory, whereas `d` is not affected by the keyword:

```
__bank1 short a, b;
short __bank1 c, d;
```

A keyword that is followed by an asterisk (*), affects the type of the pointer being declared. A pointer to `__bank2` memory is thus declared by:

```
char __bank2 * p;
```

Note that the location of the pointer variable `p` is not affected by the keyword. In the following example, however, the pointer variable `p2` is placed in `__bank1` memory. Like `p`, `p2` points to a character in `__bank2` memory.

```
__bank1 char __bank2 *p2;
```

Storage can also be specified using typedefs. The following two declarations are equivalent:

```
typedef char __bank1 Byte;
typedef Byte *BytePtr;
Byte b;
BytePtr bp;
```

and

```
__bank1 char b;
char __bank1 *bp;
```

It is possible to avoid the non-standard keywords in declarations by using `#pragmas`. The `#pragma type_attribute` controls the storage of variables. See the chapter *#pragma directives reference* for a complete description of the pragmas.

The previous example may be rewritten using the `#pragma type_attribute`:

```
#pragma type_attribute = __bank1
typedef char Byte;
typedef Byte *BytePtr;
...
```

`Pragma type_attribute` only affects the declaration of the identifier that follows immediately. The following two declarations are therefore equivalent:

```
#pragma type_attribute = __bank1
short c, d;
```

and

```
short __bank1 c, d;
```

That is, only `c` is affected by the keyword.

It is, for obvious reasons, impossible to place a variable in more than one memory segment. It is therefore not feasible to specify more than one of the keywords in a declaration. Multiple keywords result in a diagnostic message. The keyword that is specified "closest" to the identifier is used in this case. In the following declarations `x1` and `y1` are placed in `__bank1` memory, while `x2` and `y2` are placed in `__bank2` memory.

```
__bank1 int x1, __bank2 x2;  
__bank2 int __bank1 y1, y2;
```

Direct usage of keywords, as in the above example, overrides a keyword that is specified in a `#pragma`.

CONST AND __CONSTPTR

The following keywords change the access to a variable:

- ◆ `const` which declares a variable as `const`.
- ◆ `__constptr` which points to a `const` declared variable.

The keywords follow the same syntax as `__bank1`, and `__bank2`, for instance:

```
const int settings[10];
```

A pointer to a `const` variable is declared by:

```
int __constptr *ptr;
```

__NO_INIT

The following keywords change the definition of a variable:

- ◆ `__no_init`, to place a variable in a non-volatile memory segment and to suppress initialization at start-up.

The keywords are placed in front of the type, for instance to place settings in non-volatile memory:

```
__no_init int settings[10];
```

`#pragma object_attribute` can also be used. The following declaration of settings is equivalent with the previous one:

```
#pragma object_attribute = __no_init  
int settings[10];
```

Note that `__no_init` cannot be used in typedefs.

Unlike the keywords that specify storage and access of a variable, it is not necessary to specify `__no_init` in declarations. The following example declares settings without any keyword (e.g. in a header file:)

```
extern int settings[];
```

The definition of settings however specifies that it is placed in non-volatile memory:

```
__no_init int settings[10];
```

If a keyword is specified in a declaration, it is however used in the subsequent definition of the variable, for instance:

```
extern __no_init int settings[];
...
int settings[10];
```

ABSOLUTE LOCATION

It is possible to specify the location of a variable (its absolute address) using either of the following two constructs:

- ◆ the `@` operator followed by a constant-expression
- ◆ `pragma location`

The following declaration locates `PORT1` at address 45h:

```
char PORT1 @ 0x45;
```

The equivalent declaration using `pragma location` is:

```
#pragma location = 0x45
char PORT1;
```

Located objects are by default `__no_init` and can not be initialized.

FUNCTIONS

The following keywords control the calling convention of a function:

- ◆ `__bankN_func`, which informs the compiler which bank to use for parameters and auto variables.
- ◆ `__interrupt`, which supports interrupt functions.
- ◆ `__monitor`, which supports atomic execution of a function.

The keywords are specified before the return type:

```
__bank0_func void foo(void);
```

A keyword that is followed by an asterisk (*) affects the type of the pointer being declared. A pointer to a `__bank0_func` function is declared in the following example:

```
void (__bank0_func * fptr) (void);
```

It is possible to avoid the non-standard keywords in declarations by using `#pragmas`. The `#pragma type_attribute` controls the function calling convention. See the chapter *#pragma directives reference* for a complete description of the `#pragmas`.

The previous declaration of `foo` may be rewritten using `#pragma type_attribute`:

```
#pragma type_attribute = __bank0_func  
void foo(void);
```

INTERRUPT AND TRAP FUNCTIONS

The following example declares an interrupt function with an interrupt vector at address 8h offset in the INTVEC segment:

```
#pragma vector = 0x08  
__interrupt void my_handler(void);
```

The compiler generates code at the vector address that saves the register environment and then jumps to the interrupt function. If `#pragma vector` is omitted, you must write the code at the interrupt vector and at least save WREG and STATUS/ALUSTA and BSR, and PCLATH if necessary. If no interrupt vector is assigned, the interrupt function will not end in a RETFIE instruction, but in an ordinary RETURN instruction to return the control to the user function. The RETURN instruction assumes that the above mentioned registers are saved prior to entering the interrupt function.

By default the compiler uses a predefined set of reserved addresses, located in all banks present, to save WREG and PCLATH before it switches banks. To specify another set than the default, use the keyword `__setN` where *N* is a number between 0 and 3. This can be the case if you, for example, have more than one interrupt function that can occur at the same time.

MONITOR FUNCTIONS

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.

A function declared with the `__monitor` keyword is equivalent to a normal function in all other respects.

In the following example a semaphore is implemented. The semaphore is tested, and if the resource is available it is claimed by setting the flag. The routine then returns indicating if the requested resource can be used and clears the interrupt mask. This function is short (as all monitor functions should be) so it does not interfere with the operation of other interrupt routines.

```
char printer_free;                /* printer-free
__monitor int got_flag(char *flag) semaphore */
/* With no danger of
/* test if available */
/* yes - take */
/* no - do not take */
void f(void)
{
    if (got_flag(&printer_free)) /* act only if
        .... action code .... printer is free */
}
```

INTRINSIC

The `__intrinsic` keyword is used for IAR Systems internal purposes only.

#PRAGMA DIRECTIVES

REFERENCE

This chapter describes the `#pragma` directives of the PICmicro™ microcontroller.

The `#pragma` directives are preprocessed, which means that macros are substituted in the `#pragma` directive. The `#pragma` directives `warnings`, `codeseg`, `bitfields`, `baseaddr`, and `alignment` are recognized and will give a diagnostic message but will not work.

All `#pragma` directives should be entered like:

```
#pragma pragmaname = pragmavalue
```

TYPE ATTRIBUTE

The `#pragma type_attribute` affects the declaration of the identifier that immediately follows the `#pragma`. In the following example, `myBuffer` is placed in a `__bank1` segment, whereas the variable `i` is not affected by the `#pragma`.

```
#pragma type_attribute=__bank1
char inBuffer[10];
int i;
```

The following declarations, which use extended keywords, are equivalent. See the chapter *Extended keywords reference* for more details.

```
__bank1 char inBuffer[10];
int i;
```

The `#pragma type_attribute` modifies the next variable or the next function.

The following keywords can be used with `#pragma type_attribute` for a variable:

- ◆ One of `__bankN`, `__ptable`, `__rtable` and `__eeprom`.

The following keywords can be used with `#pragma type_attribute` for a function:

◆ `__bankN_func`

Several keywords are specified using the following syntax:

```
#pragma type_attribute=__no_init __bank1
```

DATASEG

Use the following syntax to place variables in a named segment:

```
#pragma dataseg=MY_SEGMENT
char myBuffer[10];
#pragma dataseg=default
```

The segment name must not be a predefined segment, see *Segment reference* for more information. The variable `myBuffer` will not be initialized at start-up and must thus not have any initializer.

The memory in which the segment resides is optionally specified using the following syntax:

```
#pragma dataseg=__bank1 MyOtherSeg
```

All variables in `MyOtherSeg` will be accessed using `__bank1` addressing.

CONSTSEG

Use the following syntax to place constant variables in a names segment:

```
#pragma constseg=MY_CONSTANTS
const int factorySettings[] = {42, 15, -128, 0};
#pragma constseg=default
```

The segment name must not be a predefined segment, see *Segment reference* for more information.

The memory in which the segment resides is optionally specified using the following syntax:

```
#pragma constseg=__eeprom MyOtherSeg
```

All variables in `MyOtherSeg` will be accessed using `__eeprom` addressing.

LOCATION

The `#pragma location` specifies the location (absolute address) of the variable, whose declaration follows the `#pragma`. For example:

```
#pragma location = 0x45
char PORT1; // PORT1 is located at address 45h
```

FUNCTION

The `#pragma function` controls the calling convention of subsequent function declarations and definitions. It may have one of the following forms:

```
#pragma function=__bankN_func
#pragma function=__monitor
#pragma function=__interrupt
#pragma function=default
```

The following example declares an interrupt function:

```
#pragma function=__interrupt
void my_handler(void);
#pragma function=default
```

See the chapter *Extended keywords reference* for more details.

VECTOR

The `#pragma vector` specifies the interrupt vector of an interrupt function whose declaration follows the `#pragma`, for example:

```
#pragma vector=0x10
__interrupt void my_handler(void);
```

DIAGNOSTICS

The following #pragmas are available for reclassifying, restoring, and suppressing diagnostics:

DIAG_REMARK

Syntax: #pragma diag_remark=tag,tag,...

Changes the severity level to remark for the specified diagnostics.

DIAG_WARNING

Syntax: #pragma diag_warning=tag,tag,...

Changes the severity level to warning for the specified diagnostics.

DIAG_ERROR

Syntax: #pragma diag_error=tag,tag,...

Changes the severity level to error for the specified diagnostics.

DIAG_DEFAULT

Syntax: #pragma diag_default=tag,tag,...

Changes the severity level back to default or as defined on the command line for the diagnostic messages with the specified tags.

DIAG_SUPPRESS

Syntax: #pragma diag_suppress=tag,tag,...

Suppresses the diagnostic messages with the specified tags.

See the chapter *Diagnostics* for more information about diagnostic messages.

LANGUAGE

Syntax: #pragma language=[extended | default]

extended Turns on the IAR extensions and turns off the strict_ansi option.

default Uses the settings specified on the command line.

PREDEFINED SYMBOLS

REFERENCE

This chapter gives reference information about the symbols predefined by the compiler.

__DATE__

Current date.

SYNTAX

__DATE__

DESCRIPTION

The date of compilation is returned in the form Mmm dd yyyy.

__FILE__

Current source filename.

SYNTAX

__FILE__

DESCRIPTION

The name of the file currently being compiled is returned.

__IAR_SYSTEMS_ICC__

IAR C compiler identifier.

SYNTAX

__IAR_SYSTEMS_ICC__

DESCRIPTION

The number 2 is returned. This symbol can be tested with `#ifdef` to detect that it was compiled by an IAR Systems C compiler.

__LINE__

Current source line number.

SYNTAX

__LINE__

DESCRIPTION

The current line number of the file currently being compiled is returned.

__STDC__

ISO/ANSI standard C identifier.

SYNTAX

__STDC__

DESCRIPTION

The number 1 is returned. This symbol can be tested with `#ifdef` that the compiler used adheres to detect that it was compiled by an ANSI C compiler.

__STDC__VERSION__

ISO/ANSI Standard C and version identifier.

SYNTAX

__STDC__VERSION__

DESCRIPTION

The number 1994092 is returned.

__TID__

Target identifier.

SYNTAX

__TID__

DESCRIPTION

The target identifier contains a number unique for each IAR Systems C Compiler (ie unique for each target), the intrinsic flag, the value of the **Processor setup file** (-v) option:

For the PICmicro™ processors the TARGET_IDENT is 39.

The __TID__ value is constructed as:

$((t \ll 8) \mid (v \ll 4))$

You can extract the values as follows:

```
t = (__TID__ >> 8) & 0x7F; /* target identifier*/  
v = (__TID__ >> 4) & 0xF;  /* processor option*/
```

Note that there are two underscores at each end of the macro name.

To find the value of Target_IDENT for the current compiler, execute:

```
printf("%ld", (__TID__>>8)&0x7F)
```

For an example of the use of __TID__, see the file stdarg.h.

__TIME__

Current time.

SYNTAX

__TIME__

DESCRIPTION

The time of compilation is returned in the form hh:mm:ss.

__VER__

Returns the compiler version number.

SYNTAX

__VER__

DESCRIPTION

The version number of the compiler is returned 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

REFERENCE

This chapter gives reference information about the intrinsic functions. To use the intrinsic functions, include the header file `intrinsic.h`.

asm

Inserts an assembler statement.

SYNTAX

```
void asm(const unsigned char *string)
```

DESCRIPTION

Assembles and inserts the supplied assembler statement in-line. The statement can include instruction mnemonics, register mnemonics, constants, and/or a reference to a global variable. Optimizations depending on control-flow analysis, register contents tracking, etc will be disabled using this function.

__disable_interrupt

Disable global interrupts.

SYNTAX

```
void __disable_interrupt(void);
```

DESCRIPTION

Disables interrupts.

__enable_interrupt

Enable global interrupts.

SYNTAX

```
void __enable_interrupt(void);
```

DESCRIPTION

Enables interrupts.

__no_operation

Inserts a no instruction.

SYNTAX

```
void __no_operation(void);
```

DESCRIPTION

Generates a NOP instruction.

__option

Load option register (only on on mid-range processors).

SYNTAX

```
void __option(void);
```

DESCRIPTION

Generates an OPTION instruction.

__set_configuration_word

Sets the controller configuration word.

SYNTAX

```
void __set_configuration_word(unsigned short);
```

DESCRIPTION

Sets the configuration word for the controller.

Predefined values which are to be ANDed together are defined in the `ioxxx.h` file.

EXAMPLE

To disable the brown-out detect circuit and set the processor mode to microcontroller for the 17c756 microcontroller you would use:

```
__set_configuration_word(MC_MODE & BODEN_OFF)
```

__sleep

Puts the controller in sleep mode.

SYNTAX

```
void __sleep(void);
```

DESCRIPTION

Generates a SLEEP instruction.

__tris

Load TRIS register (only on on mid-range processors).

SYNTAX

```
void __tris(unsigned char);
```

DESCRIPTION

Generates a TRIS instruction.

ASSEMBLY LANGUAGE INTERFACE

The PICmicro™ C Compiler allows assembly language modules to be combined with compiled C modules. This is particularly used for small, time-critical routines that need to be written in assembly language and then called from a C main program. This chapter describes the interface between a C main program and assembly language routines.

CREATING A SHELL

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

The shell source 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 */
    return 0x1234;
}
void f(void)
{
    foo(4,5); /* Call to foo */
}
```



Compiling the program using the Embedded Workbench

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

<i>Category</i>	<i>Option</i>
List	Assembly output file
List	List file
	Insert mnemonics



Compiling the program using the command line

This program should be compiled as follows:

```
iccpic shell -lA . -s0
```

The `-lA .` option creates an assembly list file with C code and the `-s0` option prevents optimization.

The result is the assembler source `shell.s39` containing the declarations, function call, function return, variable accesses, and a listing file `shell.lst`.

The following section describes the interface in detail.

CALLING CONVENTION

The compiler uses a static overlay model to handle auto variables and parameters. This means that instead of passing arguments on a stack there is an area for the parameters of the function called which needs to be filled in before the call. If you compiled the `shell.c` file, you would find in the function `f`, the function call to function `foo` with parameters 4 and 5.

The actual code would look like this:

```

        RSEG CODE:CODE(2)
;      12
;      13 void f( void )
f:
        FUNCALL f, foo
        LOCFRAME CALLSTACK, 2, STACK
        ARGFRAME OVERLAY0, 4, STATIC
        REQUIRE ?CLPIC17C43_0_95_L00
;      14 {
;      15   foo( 4,5 );
```

```

        MOVLW    0                ;
        MOVLR    0                ;
        MOVWF    PRM(foo,OVERLAY0,2) ; (Param:2 Offset:0)
        MOVLW    5                ;
        MOVLR    0                ;
        MOVWF    PRM(foo,OVERLAY0,3) ; (Param:2 Offset:1)
        MOVLW    0                ;
        MOVLR    0                ;
        MOVWF    PRM(foo,OVERLAY0,0) ; (Param:1 Offset:0)
        MOVLW    4                ;
        MOVLR    0                ;
        MOVWF    PRM(foo,OVERLAY0,1) ; (Param:1 Offset:1)
; Total param area: 4 bytes
        CALL     A:foo            ;
;    16 }
; Overlay area size: 0
        RETURN                    ;

```

At the beginning of the function you see a set of assembler directives:

```
FUNCALL f, foo
```

It tells the assembler that function `f` calls function `foo`.

```
LOCFRAME CALLSTACK, 2, STACK
```

Function `foo` uses 2 bytes of the call stack (for the return address)

```
ARGFRAME OVERLAY0, 4, STATIC
```

The arguments to `foo` are placed in the `OVERLAY0` segment and is 4 bytes in size.

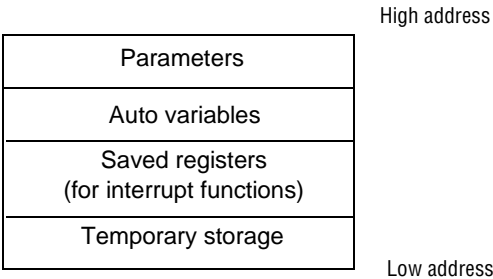
The actual parameter initialization occurs in the `MOVWF` statements:

```
MOVWF PRM(foo,OVERLAY0,1)
```

`PRM` tells the assembler that the address is a parameter which belongs to the function `foo`, that it is placed in the segment `OVERLAY0` and is located at offset 1 from the beginning of the parameter area. The compiler also generates assembler comments for the parameters telling which parameter and the byte offset of the parameter.

OVERLAY FRAMES

Each function have an overlay frame as follows:



If we take a look at the code generated for the foo function we can see how it works.

```
CALL    A:foo    ;
; 16 }

PUBLIC  foo
FUNCTION foo
ARGFRAME OVERLAY0, 4, STATIC
LOCFRAME CALLSTACK, 4294967294, STACK
LOCFRAME OVERLAY0, 1, STATIC

RSEG    CODE:CODE(2)
; 2
; 3 int foo(int i, int j )
foo:
    REQUIRE ?CLPIC17C43_0_95_L00
; 4 {
; 5   char c;
; 6   i++;
    MOVLW 1 ;
    MOVLW 0 ;
    ADDWF  LOC(foo,OVERLAY0,2),1 ; (i+1,1)
    MOVLW 0 ;
    ADDWFC LOC(foo,OVERLAY0,1),1 ; (i,1)
; 7   j++;
```

```

        MOVLW    1                      ;
        ADDWF    LOC(foo,OVERLAY0,4),1 ; (j+1,1)
        MOVLW    0                      ;
        ADDWFC   LOC(foo,OVERLAY0,3),1 ; (j,1)
;   8   c++;
        INCF     LOC(foo,OVERLAY0,0),1 ; (c,1)
;   9   k++;
        MOVLW    1                      ;
        ADDWF    k+1,1                  ; (k+1,1)
        MOVLW    0                      ;
        ADDWFC   k,1                   ; (k,1)
;  10   return( 0x1234 );
        MOVLW    18                    ;
        MOVWF    ?A1                   ; (?A1)
        MOVLW    52                    ;
        MOVWF    ?A0                   ; (?A0)
; Overlay area size: 5
        RETURN                          ;

```

In `foo` there is no temporary storage area. The overlay area consists of the two `int` parameters `i` and `j` plus the auto variable `c`. A total of 5 bytes reported at the end of the function. As described above the auto storage is at the beginning of the overlay area, `c` is located at offset 0. Above the auto area we have the parameter area. `i` is located at offset 1 and `j` is located at offset 3.

Note that instead of the `PRM` directive we now use `LOC` (local). The syntax is the same as for the `PRM` directive.

At the top is the declaration of the function `foo`:

```
FUNCTION foo
```

Tells the assembler that the `foo` label is a function.

```
ARGFRAME OVERLAY0, 4, STATIC
```

Tells the assembler that the argument area is four bytes in size, located in the segment `OVERLAY0`.

```
LOCFRAME CALLSTACK, 4294967294, STACK
```

Tells the assembler that the function uses the `CALLSTACK` segment, the numbers are flags generated by the compiler describing aspects of the function. Do not edit this field.

```
LOCFRAME OVERLAY0, 1, STATIC
```

Tells the compiler that the auto area is located in segment OVERLAY0 and is one byte in size.

The return value is passed in a predefined set of addresses:

- ◆ A byte is passed in ?A0
- ◆ A word is passed in ?A1 (high byte) and ?A0 (low byte)
- ◆ A 3 byte pointer is passed in ?A2(High byte), ?A1 (mid byte) and ?A0 (low byte)
- ◆ A long is passed in ?A3, ?A2, ?A1 and ?A0.

?A0 to ?A3 belongs to a set of predefined work registers located in bank 0.

Other work registers are ?B0 to ?B3, ?C0 to ?C3, ?D0 to ?D3. These are internally used in the assembly libraries but is used as temporary registers by the compiler when not calling library routines.

ASSEMBLER STATIC OVERLAY DIRECTIVES

The following shows how to use the assembler static overlay directives.

Prototyping an external function

```
FUNCTION name, 0202H
```

```
ARGFRAME OVERLAY0, param-size, STATIC
```

The FUNCTION defines the name *name* of the function and the 0202H are the bits telling the linker properties of the function. The easiest way is to compile the function header in the C-compiler and generate an assembler listing and extract the proper lines.

ARGFRAME defines in which segment the parameters should be placed, the size of the parameter frame and the type (STATIC). There are four segments to choose from: OVERLAY0 to OVERLAY3. They correspond to the compiler keywords `__bankx_func`.

Prototyping a function

```
FUNCTION name, 0203H
```

```
ARGFRAME OVERLAY0, size, STATIC
```

```
LOCFRAME CALLSTACK, 2, STACK
```

```
LOCFRAME OVERLAY0, 1, STATIC
```

FUNCTION again tells the name of the function, and the flagbits is 0203H for ordinary functions.

Syntax:

```
FUNCTION name, flags
```

If you are unsure of the flagbits, declare a dummy function in C and compile it with **Optimization** set to **None** (-z0) and look at the generated assembler file.

ARGFRAME again declares the size of the parameter area and which segment it belongs to.

Syntax:

```
ARGFRAME segment, size, type
```

type is always STATIC for the PICmicro™.

We need two LOCFRAME lines:

```
LOCFRAME CALLSTACK, 2, STACK
```

declares the STACK usage.

```
LOCFRAME OVERLAYx, size, STATIC
```

is used to declare local variables with parameters segment, size and type.

Normally it should be two LOCFRAME lines unless you call any assembler routines that do not have a LOCFRAME declared. The size should be the maximum call stack used.

Syntax:

```
LOCFRAME segment, size, type (STACK for CALLSTACK, STATIC for OVERLAYx)
```

If your function calls any other function, you need to repeat the prototype prior to the actual code, with the only difference being that FUNCTION is replaced with FUNCALL.

Syntax:

```
FUNCALL caller, callee [,callee,...]
```

Accessing overlayed variables

When you are calling a function, you must use the PRM keyword to place the parameters in the parameter area of the called function.

Syntax:

```
PRM(function, segment, offset)
```

The PRM keyword describes offset into the ARGFRAME area.

For example:

```
MOVWF PRM(test, OVERLAY0, 1 )
```

When accessing parameters and local variables from inside the function you would use the LOC keyword which has the same syntax as PRM.

To access the ARGFRAME area you add the size of the LOCFRAME area when computing the offset to the parameters.

Syntax:

```
LOC( function, segment, size )
```

INTERRUPT FUNCTIONS

If a vector is assigned to the interrupt function, the compiler will save flags and registers used and then generate the interrupt code for the interrupt vector.

If no vector is assigned, you must generate the INTVEC code to save WREG, PCLATH, STATUS/ALUSTA and BSR, and then call the interrupt function. The compiler will only save the work register used. The compiler will generate an ordinary RETURN instruction after it has restored the work registers, returning control to the user code so it can restore WREG.

MONITOR FUNCTIONS

The interrupt status register is saved and global interrupts are disabled at entry. At exit the interrupt register is restored.

CALLING ASSEMBLY ROUTINES FROM C

An assembler routine that is to be called from C must:

- ◆ Conform to the calling convention described above.
- ◆ Have a `PUBLIC` entry-point label.
- ◆ Be declared as external before any call, to allow type checking and optional promotion of parameters, as in `extern int foo()` or `extern int foo(int i, int j)`.

LOCAL STORAGE ALLOCATION

If the routine needs local storage, it must be allocated in the overlay area.

Functions can always use `?A0` to `?A3`, `?B0` to `?B3`, `?C0` to `?C3` and `?D0` to `?D3` without saving them.

INTERRUPT FUNCTIONS

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

- ◆ The routine must preserve all used registers, including global registers.
- ◆ The routine must treat all flags as undefined.

DEFINING INTERRUPT VECTORS

The interrupt routines should be inserted in the `INTVEC` function in `CSTARTUP` if the user has chosen not to use the vector program.

The interrupt vectors are located in the `INTVEC` segment.

SEGMENT REFERENCE

The PICmicro™ C Compiler places code and data into named segments which are referred to by the IAR XLINK Linker. 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 section provides memory map diagrams and 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.

The segments for the PICmicro™ compiler can be divided into groups. One is the code, which consists of the segment CODE, RCODE and SCODE. One is the interrupt vector which is INTVEC. One is the constant segment CONST and the last group is the variables segments.

The variable segment names consists of a prefix named after the storage keyword plus a suffix that tells what kind of segment it is. For example:

`__bank0`

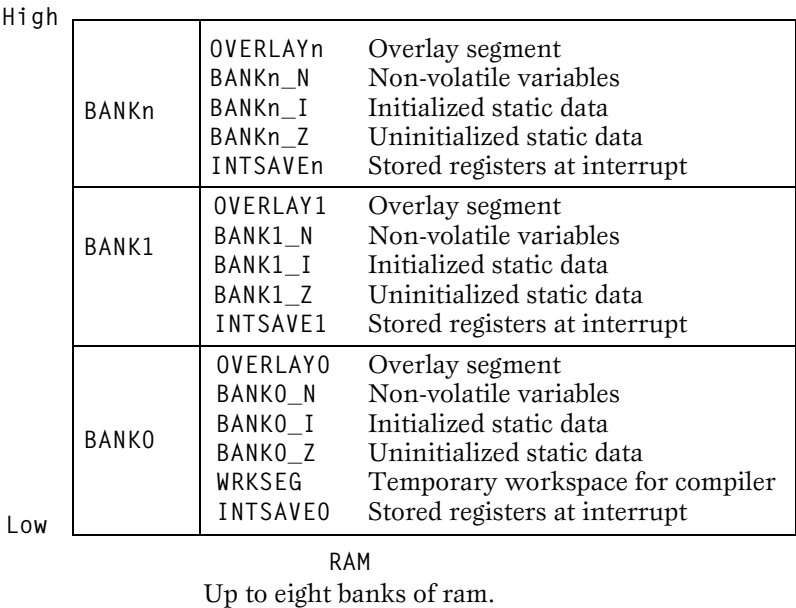
would yield the prefix BANK0.

The suffix states the kind of segment. The following suffixes are available:

<i>Suffix</i>	<i>Description</i>
<code>_Z</code>	Zero initialized memory.
<code>_I</code>	Initialized memory.
<code>_ID</code>	Initialize data for <code>_I</code> (Located in code memory).
<code>_N</code>	<code>__no_init</code> , do not need to be initialized.
<code>_A</code>	Absolute addressed segments.

MEMORY MAP DIAGRAMS

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



High

PTABLE_N	Non-volatile variables for PTABLE
PTABLE_I	Initialized static data for PTABLE
PTABLE_Z	Uninitialized static data for PTABLE
CONST	Constants
PTABLE_ID	Initialization constants for PTABLE
EEPROM_ID	Initialization constants for EEPROM
BANKn_ID	Initialization constants for BANK <i>n</i>
HCODE	Code
LCODE	Code
CODE	Code
RCODE	Root code
SCODE	Startup code
INTVEC	Interrupt vectors

Low

CODE memory

High

CODE	Code
RTABLE_Z	Uninitialized static data for RTABLE
RTABLE_I	Initialized static data for RTABLE
RTABLE_N	Non-volatile variables for RTABLE

Low

External memory

High

EEPROM_Z	Uninitialized static data for EEPROM
EEPROM_I	Initialized static data for EEPROM
EEPROM_N	Non-volatile variables for EEPROM

Low

EEPROM memory

BANKN_ID, EEPROM_ID, PTABLE_ID, RTABLE_ID	<p>Initialization constants for BANK<i>n</i>, EEPROM, PTABLE and RTABLE data respectively.</p> <p>TYPE</p> <p>Read-only. Located as RETLW instructions in code memory.</p> <p>DESCRIPTION</p> <p>Assembly-accessible.</p> <p>CSTARTUP copies initialization values from this segment to the xxx_I segment.</p> <p>You should not declare data in assembly language to be stored in xxx_ID segments.</p>
BANKN_I, EEPROM_I, PTABLE_I, RTABLE_I	<p>Initialized static data for BANK<i>n</i>, EEPROM, PTABLE and RTABLE data respectively.</p> <p>TYPE</p> <p>Read-write.</p> <p>DESCRIPTION</p> <p>Assembly-accessible.</p> <p>Holds static variables in internal data memory that are automatically initialized from BANK<i>n</i>_ID, EEPROM_ID, PTABLE_ID and RTABLE_ID in cstartup.s39. See also BANK<i>n</i>_ID, EEPROM_ID, PTABLE_ID, RTABLE_ID.</p>
CALLSTACK	<p>Call stack.</p> <p>TYPE</p> <p>Read/write.</p> <p>DESCRIPTION</p> <p>Holds the call stack.</p> <p>This segment and length is normally defined in the XLINK file by the command:</p>

`-Z(DATA)CSTACK = start-end`

where *start* is the location and *end* is the end.

CODE

Code.

TYPE

Read-only.

DESCRIPTION

Assembly-accessible.

Holds user program code and various library routines that is smaller than 256 words and can be placed anywhere within a 256 word page. Note that any assembly language routines called from C must meet the calling convention of the memory model in use. For more information see *Calling assembly routines from C*, page 191.

CONST

Constants.

TYPE

Read-only.

DESCRIPTION

Assembly-accessible. Consists of RETLW instructions.

Used for storing constant objects. Can be used in assembly language routines for declaring constant data.

<p>HCODE</p>	<p>Code.</p> <p>TYPE</p> <p>Read-only.</p> <p>DESCRIPTION</p> <p>Assembly-accessible.</p> <p>Holds user program code and various library routines that are larger or equal to the code page size. Note that any assembly language routines called from C must meet the calling convention of the memory model in use. For more information see <i>Calling assembly routines from C</i>, page 191.</p>
<p>INTVEC</p>	<p>Interrupt vectors.</p> <p>TYPE</p> <p>Read-only.</p> <p>DESCRIPTION</p> <p>Holds the interrupt vector table generated by the use of the <code>interrupt</code> extended keyword (which can also be used for user-written interrupt vector table entries).</p>
<p>LCODE</p>	<p>Code.</p> <p>TYPE</p> <p>Read-only.</p> <p>DESCRIPTION</p> <p>Assembly-accessible.</p> <p>Holds user program code and various library routines that are larger or equal to 256 words but is smaller than the code page size. Note that any assembly language routines called from C must meet the calling convention of the memory model in use. For more information see <i>Calling assembly routines from C</i>, page 191.</p>

OVERLAY N

The overlay segment for bank n .

TYPE

read/write

DESCRIPTION

This is where the linker puts parameters and autos.

**R
RLCODE**

Root code.

TYPE

Read-only.

DESCRIPTION

Assembly-accessible.

Used for interrupt handlers and internal library functions. Must be placed in the first code page.

SCODE

Startup code.

TYPE

Read only.

DESCRIPTION

Not assembly-accessible.

Holds startup code.

xxx_A	<p>Absolute addressed segments.</p> <p>TYPE</p> <p>read/(write depending on memory)</p> <p>DESCRIPTION</p> <p>Typically used for located variables and SFRs. Cannot be initialized by CSTARTUP.</p> <p>The compiler only allows located variables in the File register banks but can also be used from assembler.</p>
xxx_N	<p>Non-volatile variables.</p> <p>TYPE</p> <p>Read/write.</p> <p>DESCRIPTION</p> <p>Assembly-accessible.</p> <p>Holds variables to be placed in the specified memory. These will have been allocated by the compiler, declared <code>__no_init</code> or created <code>__no_init</code> by use of the memory <code>#pragma</code>, or created manually from assembly language source.</p>
xxx_Z	<p>Uninitialized static data.</p> <p>TYPE</p> <p>Read/write.</p> <p>DESCRIPTION</p> <p>Assembly-accessible.</p> <p>Holds variables in memory that are not explicitly initialized; these are implicitly initialized to all zero, which is performed by CSTARTUP.</p>

IMPLEMENTATION-DEFINED BEHAVIOR

ISO 9899:1990, the International Organization for Standardization standard - *Programming Language - C* (revision and redesign of ANSI X3.159-1989, American National Standard), changed by ISO Amendment 1:1994 and Technical corrigendum 1 and 2, contains an appendix called *Portability Issues*. The ISO appendix lists areas of the C language that ISO leaves open to each particular implementation. This chapter describes how IAR C handles these implementation-defined areas of the C language.

Note that IAR C adheres to a freestanding implementation of the ISO standard for the programming language - C. This means that parts of a standard library can be excluded in the implementation. IAR does not support the following parts of the standard library: `locale`, `files` (but streams `stdio` and `stdout`), `time`, and `signal`.

This chapter follows the same order as the ISO appendix. Each item covered includes references to the ISO chapter and section (in parenthesis) that explains the implementation-defined behavior.

TRANSLATION

DIAGNOSTICS (5.1.1.3)

IAR C produces diagnostics in the form:

level[tag] file-name line-number: message

where *level* is the level of seriousness of the message (remark, warning, error, or fatal error); *tag* is a unique tag that identifies the message; *file-name* is the name of the source file in which the error was encountered; *line-number* is the line number at which the compiler detected the error; and ; *message* is an explanatory message, possibly several lines.

ENVIRONMENT**ARGUMENTS TO MAIN (5.1.2.2.1)**

In IAR C, the function called at program startup is called `main`. There is no prototype declared for `main`, and the only definition supported for `main` is:

```
void main(void)
```

To change this behavior, see the CSTARTUP description.

INTERACTIVE DEVICES (5.1.2.3)

IAR C treats the streams `stdio` and `stdout` as interactive devices.

IDENTIFIERS**SIGNIFICANT CHARACTERS WITHOUT EXTERNAL LINKAGE (6.1.2)**

The number of significant initial characters in an identifier without external linkage are 200.

SIGNIFICANT CHARACTERS WITH EXTERNAL LINKAGE (6.1.2)

The number of significant initial characters in an identifier with external linkage are 200.

CASE DISTINCTIONS ARE SIGNIFICANT (6.1.2)

IAR C treats identifiers with external linkage as case sensitive.

CHARACTERS**SOURCE AND EXECUTION CHARACTER SETS (5.2.1)**

The source character set is the set of legal characters that can appear in source files. In IAR C, the source character set is the standard ASCII character set.

The execution character set is the set of legal characters that can appear in the execution environment. In IAR C, the execution character set is the standard ASCII character set.

BITS PER CHARACTER IN EXECUTION CHARACTER SET (5.2.4.2.1)

The number of bits in a character is represented by the manifest constant `CHAR_BIT`. The standard include file `limits.h` defines `CHAR_BIT` as 8.

MAPPING OF CHARACTERS (6.1.3.4)

The mapping of members of the source character set (in character and string literals) to members of the execution character set are done in a one-to-one way, i.e. using the same representation value for each member in the character sets, except for the escape sequences listed in the ISO standard.

UNREPRESENTED CHARACTER CONSTANTS (6.1.3.4)

The value of an integer character constant that contains a character or escape sequence not represented in the basic execution character set or in the extended character set for a wide character constant, generates a diagnostic and will be truncated to fit the execution character set.

CHARACTER CONSTANT WITH MORE THAN ONE CHARACTER (6.1.3.4)

An integer character constant that contains more than one character will be treated as an integer constant. The value will be calculated by treating the leftmost character as the most significant character, and the rightmost character as the least significant character, in an integer constant. A diagnostic will be issued if the value cannot be represented in an integer constant.

A wide character constant that contains more than one multibyte character, generates a diagnostic.

CONVERTING MULTIBYTE CHARACTERS (6.1.3.4)

The current and only locale supported in IAR C is the "C" locale. The library does not support any multibyte functions, or for that matter any locale functions.

RANGE OF 'PLAIN' CHAR (6.2.1.1)

A *'plain'* char has the same range as an `unsigned char`.

INTEGERS

RANGE OF INTEGER VALUES (6.1.2.5)

The representation of integer values are in two's-complement form. The most-significant bit holds the sign; 1 for negative, 0 for positive and zero.

See *Integer types*, page 10, for information about the ranges for the different integer types: `char`, `short int`, `int`, and `long int`.

DEMOTION OF INTEGERS (6.2.1.2)

Converting an integer to a shorter signed integer is made by truncation. If the value cannot be represented when converting an unsigned integer to a signed integer of equal length the bit-pattern remains the same, i.e. a large enough value will be converted into a negative value.

SIGNED BITWISE OPERATIONS (6.3)

Bitwise operations on signed integers work the same as bitwise operations on unsigned integers, i.e. the sign-bit will be treated as any other bit.

SIGN OF THE REMAINDER ON INTEGER DIVISION (6.3.5)

The sign of the remainder on integer division is the same as the sign of the dividend.

NEGATIVE VALUED SIGNED RIGHT SHIFTS (6.3.7)

The result of a right shift of a negative-valued signed integral type, preserves the sign-bit. Example shifting `0xFF00` down one step yields `0xFF80`.

FLOATING POINT

REPRESENTATION OF FLOATING POINT VALUES (6.1.2.5)

The representation and sets of the various floating-point numbers adheres to IEEE 854-1987. A typical floating point number is built up of a sign-bit (s), a biased exponent (e), and a mantissa (m).

See *Floating point types*, page 11, form information about the ranges and sizes for the different floating-point types: `float`, `double`, and `long double`.

CONVERTING INTEGER VALUES TO FLOATING-POINT VALUES (6.2.1.3)

When an integral number is cast to a floating-point value that cannot exactly represent the value, the value is rounded (up or down) to the nearest suitable value.

DEMOTING FLOATING-POINT VALUES (6.2.1.4)

When a floating-point value is converted to a floating-point value of narrower type that cannot exactly represent the value, the value is rounded(up or down) to the nearest suitable value.

**ARRAYS AND
POINTERS****SIZE_T (6.3.3.4, 7.1.1)**

See *size_t*, page 14, for information about *size_t* in IAR C.

CONVERSION FROM/TO POINTERS (6.3.4)

See *Casting*, page 14, for information about casting of data pointers and function pointers.

PTRDIFF_T (6.3.6, 7.1.1)

See *ptrdiff_t*, page 14, for information about the *ptrdiff_t* in IAR C.

REGISTERS**HONORING THE REGISTER KEYWORD (6.5.1)**

IAR C does not honor user requests for register variables. Instead it makes its own choices when optimizing.

**STRUCTURES,
UNIONS,
ENUMERATIONS,
AND BIT-FIELDS****IMPROPER ACCESS TO A UNION (6.3.2.3)**

If a union gets its value stored through a member and is then accessed using a member of a different type, the result is solely dependent on the internal storage of the first member.

PADDING AND ALIGNMENT OF STRUCTURE MEMBERS (6.5.2.1)

See the section *Data representation*, page 10, for information about the alignment requirement for data objects in IAR C.

SIGN OF 'PLAIN' BIT-FIELDS (6.5.2.1)

A 'plain' `int` bit-field is treated as an unsigned `int` bit-field. All integer types are allowed as bit-fields.

ALLOCATION ORDER OF BIT-FIELDS WITHIN A UNIT (6.5.2.1)

Bit-fields are allocated within an integer from least-significant to most-significant bit.

CAN BIT-FIELDS STRADDLE A STORAGE-UNIT BOUNDARY (6.5.2.1)

Bit-fields cannot straddle a storage-unit boundary for the bit-field integer type chosen.

INTEGER TYPE CHOSEN TO REPRESENT ENUMERATION TYPES (6.5.2.2)

The chosen integer type for a specific enumeration type depends on the enumeration constants defined for the enumeration type. The chosen integer type is the smallest possible.

QUALIFIERS

ACCESS TO VOLATILE OBJECTS (6.5.3)

Any reference to an object with volatile qualified type is an access.

DECLARATORS

MAXIMUM NUMBERS OF DECLARATORS (6.5.4)

IAR C does not limit the number of declarators. The number is limited only by available memory.

STATEMENTS**MAXIMUM NUMBER OF CASE STATEMENTS (6.6.4.2)**

IAR C does not limit the number of case statements (case values) in a switch statement. The number is limited only by available memory.

**PREPROCESSING
DIRECTIVES****CHARACTER CONSTANTS AND CONDITIONAL
INCLUSION (6.8.1)**

The character set used in the preprocessor directives is the same as the execution character set. The preprocessor recognizes negative character values if a 'plain' character is treated as a signed character.

INCLUDING BRACKETED FILENAMES (6.8.2)

For file specifications enclosed in angle brackets, the preprocessor does not search directories of the parent files. A "parent" file is the file that has the `#include` directive. Instead, it begins by searching for the file in the directories specified on the compiler command line.

INCLUDING QUOTED FILENAMES (6.8.2)

For file specifications enclosed in quotes, the preprocessors directory search begins with the directories of the parent file, then proceeds through the directories of any grandparent files. Thus, searching begins relative to the directory containing the source file currently being processed. If there is no grandparent file and the file has not been found, the search continues as if the filename were enclosed in angle brackets.

CHARACTER SEQUENCES (6.8.2)

Preprocessor directives use the source character set, with the exception of escape sequences. Thus to specify a path for an include file, use only one backslash:

```
#include "mydirectory\myfile"
```

Within source code, two backslashes are necessary:

```
file = fopen("mydirectory\\myfile","rt");
```

RECOGNIZED #PRAGMA DIRECTIVES (6.8.6)

The following #pragmas are recognized in IAR C:

```
constseg  
dataseg  
diag_default  
diag_error  
diag_remark  
diag_suppress  
diag_warning  
function  
language  
location  
memory  
type_attribute  
vector
```

DEFAULT __DATE__ AND __TIME__ (6.8.8)

The definitions for __TIME__ and __DATE__ are always available.

LIBRARY FUNCTIONS NULL MACRO (7.1.6)

The NULL macro is defined to (void *) 0.

DIAGNOSTIC PRINTED BY THE ASSERT FUNCTION (7.2)

The assert() function prints:

Assertion failed: *expression*, file *filename*, line
linenumber

when the parameter evaluates to zero.

DOMAIN ERRORS (7.5.1)

HUGE_VAL, largest representable value in a double floating-point type, will be returned by the mathematic functions on domain errors.

**UNDERFLOW OF FLOATING-POINT VALUES SETS
ERRNO TO ERANGE (7.5.1)**

The mathematics functions set the integer expression `errno` to `ERANGE` (a macro in `errno.h`) on underflow range errors.

FMOD() FUNCTIONALITY (7.5.6.4)

If the second argument to `fmod()` is zero, the function returns zero (it does not change the integer expression `errno`).

SIGNAL() (7.7.1.1)

IAR C does not support the signal part of the library.

TERMINATING NEWLINE CHARACTER (7.9.2)

`Stdout` stream functions recognize either `newline` or `end of file` (`EOF`) as the terminating character for a line.

BLANK LINES (7.9.2)

Space characters written out to the `stdout` stream immediately before a newline character are preserved. There is no way to read in the line through the stream `stdin` that was written out through the stream `stdout` in IAR C.

**NULL CHARACTERS APPENDED TO DATA WRITTEN
TO BINARY STREAMS (7.9.2)**

There are no binary streams implemented in IAR C.

FILES (7.9.3)

There are no streams other than `stdin` and `stdout` in IAR C. This means that a file system is not implemented.

REMOVE() (7.9.4.1)

There are no streams other than `stdin` and `stdout` in IAR C. This means that a file system is not implemented.

RENAME() (7.9.4.2)

There are no streams other than `stdin` and `stdout` in IAR C. This means that a file system is not implemented.

%P IN PRINTF() (7.9.6.1)

The argument to a %p conversion specifier, print pointer, to printf() is treated as having the type 'char *'. The value will be printed as a hexadecimal number, similar to using the %x conversion specifier.

%P IN SCANF() (7.9.6.2)

The %p conversion specifier, scan pointer, to scanf() reads a hexadecimal number and converts that into a value with the type 'void *'.

READING RANGES IN SCANF() (7.9.6.2)

A - character is always treated explicitly as a - charatcer.

FILE POSITION ERRORS (7.9.9.1, 7.9.9.4)

There are no streams other than stdin and stdout in IAR C. This means that a file system is not implemented.

MESSAGE GENERATED BY PERROR() (7.9.10.4)

perror() is not supported in IAR C.

ALLOCATING ZERO BYTES OF MEMORY (7.10.3)

The calloc(), malloc(), and realloc() functions accept zero as an argument. Memory will be allocated, and a valid pointer to that memory is returned, and the memory block can be modified later by realloc.

BEHAVIOR OF ABORT() (7.10.4.1)

The abort() function does not flush stream buffers, and it does not handle files, since this is an unsupported feature in IAR C.

BEHAVIOR OF EXIT() (7.10.4.3)

The exit() function does not return in IAR C.

ENVIRONMENT (7.10.4.4)

An environment is not supported in IAR C.

SYSTEM() (7.10.4.5)

A system is not supported in IAR C.

MESSAGE RETURNED BY STRERROR() (7.11.6.2)

The messages returned by `strerror()` depending on the argument is:

<i>Argument</i>	<i>Message</i>
EZERO	no error
EDOM	domain error
ERANGE	range error
<0 >99	unknown error
all others	error No.xx

THE TIME ZONE (7.12.1)

Time is not supported in IAR C.

CLOCK() (7.12.2.1)

Time is not supported in IAR C.

IAR C EXTENSIONS

This chapter describes extensions, implemented in IAR C, to the ISO standard for the programming language C. The extensions can be turned off by using the compiler command-line option `--strict_ansi`. The following extensions are available:

- ◆ Memory, type, and object attributes. The attributes follow the syntax for qualifiers but not the semantics.
- ◆ Absolute addressed data object operator '@'.
- ◆ A translation unit (input file) can contain no declarations.
- ◆ Comment text can appear at the ends of preprocessing directives.
- ◆ `__ALIGNOF__` is similar to `sizeof`, but returns the alignment requirement value for a type, or 1 if there is no alignment requirement. It may be followed by a type or expression in parenthesis, `__ALIGNOF__(type)`, or `__ALIGNOF__(expression)`. The expression in the second form is not evaluated.
- ◆ Bit fields may have base types that are enums or integral types besides `int` and `unsigned int`. This matches G.5.8 in the *ISO Portability* issues appendix.
- ◆ The last member of a struct may have an incomplete array type. It may not be the only member of the struct (otherwise, the struct would have zero size).
- ◆ A file-scope array may have an incomplete struct, union, or enum type as its element type. The type must be completed before the array is subscripted (if it is), and by the end of the compilation if the array is not external.
- ◆ Static functions may be declared in function and block scopes. Their declarations are moved to the file scope.
- ◆ enum tags may be incomplete: one may define the tag name and resolve it (by specifying the brace-enclosed list) later.
- ◆ The values of enumeration constants may be given by expressions that evaluate to unsigned quantities that fit in the `unsigned int` range but not in the `int` range. A warning is issued for suspicious cases.

- ◆ An extra comma is allowed at the end of an enum list. A remark is issued.
- ◆ The final semicolon preceeding the closing `}` of a `struct` or union specifier may be omitted. A warning is issued.
- ◆ A label definition may be immediately followed by a right brace (normally a statement must follow a label definition). A warning is issued.
- ◆ An empty declaration (a semicolon with nothing before it) is allowed. A remark is issued.
- ◆ An initializer expression that is a single value and is used to initialize an entire static array, `struct`, or union need not be enclosed in braces. ISO C requires the braces.
- ◆ In an initializer, a pointer constant value may be cast to an integral type if the integral type is big enough to contain it.
- ◆ The address of a variable with register storage class may be taken. A warning is issued.
- ◆ In an integral constant expression, an integer constant may be cast to a pointer type and then back to an integral type.
- ◆ In duplicate size and sign specifiers (e.g. `short short` or `unsigned unsigned`) the redundancy is ignored. A warning is issued.
- ◆ `long float` is accepted as synonym of `double`.
- ◆ Benign redeclarations of `typedef` names are allowed. That is, a `typedef` name may be redeclared in the same scope as the same type. A warning is issued.
- ◆ Dollar signs are accepted in identifiers.
- ◆ Numbers are scanned according to the syntax for numbers rather than the pp-number syntax. Thus, `0x123e+1` is scanned as three tokens instead of one invalid token (if `--strict_ansi` is specified, of course, the pp-number syntax is used).
- ◆ Assignment and pointer difference is allowed between pointers to types that are interchangeable but not identical, for example, `unsigned char *` and `char *`. This includes pointers to same-sized integral types (e.g. typically, `short *` and `int *`). A warning is issued. Assignment of a string constant to a pointer to any kind of character is allowed without a warning.

- ◆ Assignment of pointer types is allowed in cases where the destination type has added type qualifiers that are not at the top level (e.g. `int **` to `int const **`). Comparisons and pointer difference of such pairs of pointer types are also allowed. A warning is issued.
- ◆ In operations on pointers, a pointer to `void` is always implicitly converted to another type if necessary, and a `null` pointer constant is always implicitly converted to a `null` pointer of the right type if necessary. In ISO C, some operators allow such things, and then others (generally, where it does not make sense) do not allow them.
- ◆ `asm` statements are accepted, like `asm("MOV R1,#5");`. This is disabled in strict ISO C mode (with the command line option `--strict_ansi`).
- ◆ Anonymous structs and unions (similar to the C++ anonymous unions) are allowed. An anonymous structure type defines an unnamed object (and not a type) whose member names are promoted to the surrounding scope. The member names must be unique in the surrounding scope. External anonymous structure types are allowed.
- ◆ External entities declared in other scopes are visible. A warning is issued. Example:

```
void f1(void) { extern void f(); }  
void f2() { f(); }
```
- ◆ End of line comments (`//`, as in C++) are allowed.
- ◆ A non-lvalue array expression is converted to a pointer to the first element of the array when it is subscripted or similarly used.

DIAGNOSTICS

A normal diagnostic from the compiler is produced in the form:

```
level[tag] file-name line-number: message
```

where *level* is the level of seriousness of the message (remark, warning, error, or fatal error); *tag* is a unique tag that identifies the message; *file-name* is the name of the source file in which the error was encountered; *line-number* is the line number at which the compiler detected the error; and ; *message* is an explanatory message, possibly several lines.

SEVERITY LEVELS

The diagnostics are divided into different levels of severity:

Remark

A diagnostic that is produced when the compiler finds a source code construct that can possibly lead to erroneous behaviour in the generated code. Remarks are by default not issued but can be enabled by using the command-line option `--remark`.

Warning

A diagnostic that is produced when the compiler finds a programming error or omission which is of concern but not so severe as to prevent the completion of compilation. Warnings can be disabled by using the command line option `-w`. The command line option `--warnings_affect_exit_code` will make the compiler produce a non-zero exit-code if a warning was encountered.

Error

A diagnostic that is produced when the compiler has found a construct which clearly violates the C language rules, such that code cannot be produced.

Fatal error

A diagnostic that is produced when the compiler has found a condition that not only prevents code generation, but which makes further processing of the source code pointless. After the diagnostic has been issued, compilation terminates. A fatal error will produce a non-zero exit-code.

SETTING THE SEVERITY LEVEL

The diagnostic can be suppressed or the severity level can be changed for all diagnostics except fatal errors and some of the regular errors.

See *Diagnostics*, page 68, for a description of the options that are available for setting severity levels.

See *Diagnostics*, page 174, for a description of the `#pragma` directives that are available for setting severity levels.

INTERNAL ERROR

A diagnostic that signals that there has been a serious and unexpected failure due to a fault in the compiler itself is the internal error. It is produced using the following form:

Internal error: *message*

where *message* is an explanatory message. Internal errors should not occur and should be reported to your software vendor. Your report should include all possible information about the problem and preferably a minimal source file that generates the internal error in electronic form.

MESSAGES

The following messages show examples of diagnostic messages:

Og0001 Assembler error: *string*

Indicates erroneous syntax in an `asm` operation. The string specifies the actual error.

Be0001 Target error: *string*

Target description.

Be0002 Target message: *string*

Target description.

Be0003 A located initialized variable must be constant: *full symbol name and place*

Specifying absolute address variables that must be initialized at start up is not allowed.

PICMICRO™-SPECIFIC REMARKS

In addition to the general remarks, the PICmicro™ compiler can generate the following remarks:

Pic008 Located variable is implicitly `__no_init`

A located variable is by default declared as `__no_init`.

PICMICRO™-SPECIFIC WARNING MESSAGES

In addition to the general warnings, the PICmicro™ C compiler can generate the following warnings:

Pic017 WREG, PCLATH, STATUS/ALUSTA and BSR will not be saved automatically unless a vector is assigned to the interrupt function.

An interrupt function with no vector assigned is found.

PICMICRO™-SPECIFIC ERROR MESSAGES

In addition to the general errors, the PICmicro™ compiler can generate the following errors:

Pic001 Mixing memory attributes for segment *segment*

Different attributes are set for the segment *segment*.

Pic003 SFR *sfr* is located on different address than preset for this processor

The address of the user-defined *sfr* does not match the address in the setup file for the processor.

Pic006 Illegal cast: A variable of type *type* cannot be cast to a `__dptr` pointer

A non-pointer variable cannot be cast to a `__dptr` pointer.

Pic007 A value of type *xxx* cannot be assigned to an entity of type *yyy*

You try to cast a pointer to *xxx* that cannot be represented by *yyy* or you are trying to downcast from a `__bank` or a `__dptr` in which case you can turn the error to a warning by using the pragma `diag_warning`.

Pic009 `__no_init` objects cannot be initiated

An object declared as `__no_init` must not have an initializer.

Pic010 Auto variables cannot be located

Auto variables must not be located using the @ operator.

Pic011 Located objects cannot be initiated

Located objects are by default declared `__no_init` and cannot have an initializer.

Pic012 Interrupt functions cannot take parameters

An interrupt function must not take parameters.

Pic013 Interrupt functions cannot return a value

Interrupt functions have no caller and cannot return a value.

Pic014 Function must be declared `_interrupt` to have a vector

The vector pragma is only allowed on interrupt functions.

Pic015 `__setN` keyword can only be used on functions declared `__interrupt`

The `__setN` keyword is only applicable to interrupt functions.

Pic016 `__setN` can only be used once a function

Only one set can be defined for an interrupt function.

Pic018 `__monitor` and `__interrupt` can not be declared for the same function

A function can either be `__monitor` or `__interrupt`, not both.

Pic019 Calls to interrupt functions are not allowed

An interrupt function cannot be called.

Pic020 Illegal parameter to `__tris`

The argument to the `__tris` intrinsic must be a constant or the address of a variable located in a bank and casted to an unsigned char. Example:
`__tris((unsigned char) & c);`

PICMICRO™-SPECIFIC FATAL ERROR MESSAGES

In addition to the general fatal errors, the PICmicro™ compiler can generate the following errors:

Pic002 Dongle error: *string*

The dongle (hardware lock) needed to run the compiler was not found. Check if it is connected or if the driver necessary for communication with the dongle is installed.

Pic004 `--uses_external_memory` **option not allowed for mid-range processors**

`--uses_external_memory` is only allowed for high-end processors (17CXX).

Pic005 Target Error:*message*

A target-defined message.

A		B			
abort (library function)	98	BANKN_I (segment)	196	processor setup file	83
abs (library function)	99	BANKN_ID (segment)	196	strict ISO/ANSI	68
absolute location	18, 167	bin subdirectory	4	suppress these diagnostics	80
acos (library function)	99	Bitfields	11	treat these as errors	81
adding an interrupt handler	21	bsearch (library function)	104	treat these as remarks	80
address space	51	building target-specific libraries	88	treat these as warnings	81
apic subdirectory	4	buildlib utility	88	use external memory	84
apic\tutor subdirectory	4			-D	78
APIC_INC				-e	67
(environment variable)	5			-I	76
ARGFRAME (Assembler static		C		-l	75, 184
overlay directive)	188			-o	73, 85
asin (library function)	100	C compiler		-r	73
assembler file (C compiler option)	74	files	5	-s	70, 184
assembler files	4	C compiler options		-U	86
assembler interface	183	assembler file	74	-v	83, 177
assembler interface shell	183	assembler mnemonics	74	-z	70
assembler mnemonics		C list file	74	--char_is_signed	68
(C compiler option)	74	C source	74	--debug	73
Assembler static overlay directives		'char is 'signed char'	68	--diag_error	81
ARGFRAME	188	code motion	71	--diag_remark	80
FUNCALL	189	common sub-expression		--diag_suppress	80
FUNCTION	188	elimination	70	--diag_warning	81
LOC	190	defined symbols	78	--library_module	72
LOCFRAME	189	diagnostics	74	--module_name	73
PRM	189	enable remarks	80	--no_code_motion	71
assembler, inline	19	executables	84	--no_cse	70
Assembler, Linker, and Librarian		function inlining	71	--no_inline	71
Programming Guide	7	generate debug info	73	--no_unroll	71
assert (library function)	100	include paths	76	--no_warnings	86
assert.h (header file)	95	language extensions	67	--only_stdout	86
assumptions	v	list files	85	--preprocess	78
atan (library function)	101	loop unrolling	71	--remarks	80
atan2 (library function)	101	make a library module	72	--silent	86
atof (library function)	102	object files	85	--strict_ansi	68
atoi (library function)	102	objekt module name	73	--uses_external_memory	84
atol (library function)	103	optimize for size	70	C compiler, features	9
		optimize for speed	70	C list file (C compiler option)	74
		output directories	84	C source (C compiler option)	74
		preprocessor to output file	78	Calling convention	184
				calloc (library function)	59, 104

.d39	6	cctype.h	89	intrinsic functions	
.h	6	errno.h	95	__asm	179
.inc	6	float.h	95	__disable_interrupt	179
.ini	7	icclbutl.h	90	__enable_interrupt	179
.i39	6	limits.h	95	__no_operation	180
.lst	6	math.h	90	__option	180
.mac	6	setjmp.h	91	__set_configuration_word	180
.map	6	stdarg.h	91	__sleep	181
.mem	6	stdio.h	91, 92	__tris	181
.prj	6	stdlib.h	92	INTVEC (segment)	198
.r39	6	string.h	93	isalnum (library function)	112
.s39	6	heap size	59	isalpha (library function)	113
.xcl	6	help	8	iscntrl (library function)	113
floating-point format	11	hints, programming	16	isdigit (library function)	114
4-byte	12			isgraph (library function)	114
float.h (header file)	95			islower (library function)	115
floor (library function)	109	I		isprint (library function)	115
fmod (library function)	110	IAR web site	8	ispunct (library function)	116
free (library function)	110	icclbutl.h (header file)	90	isspace (library function)	116
frexp (library function)	111	iccpic subdirectory	5	isupper (library function)	117
FUNCALL (Assembler static		iccpic\tutor subdirectory	5	isxdigit (library function)	117
overlay directive)	189	inc subdirectory	5		
function inlining		include files	5	L	
(C compiler option)	71	include paths (C compiler option)	76	labs (library function)	118
FUNCTION (Assembler static		information, product		language extensions	17
overlay directive)	188	inline assembler		language extensions	
function (#pragma directive)	173	installation		(C compiler option)	67
		requirements		language options	67
G		installed files		language (generic pragma)	174
generate debug info		assembler		LCODE (segment)	198
(C compiler option)	73	C compiler		ldexp (library function)	118
getchar (library function)	56, 111	documentation		ldiv (library function)	119
gets (library function)	112	executable		lib subdirectory	5
		include		libraries, target-specific	88
		library		library files	5
		linker command		library functions	
		miscellaneous		abort	98
H		integer types	10	abs	99
HCODE (segment)	198	internal error	218	acos	99
header files	87	intrinsic function summary	18		
assert.h	95				

INDEX

asin	100	memcmp	122	tolower	154
assert	100	memcpy	122	toupper	154
atan	101	memmove	123	va_arg	155
atan2	101	memset	124	va_end	155
atof	102	modf	124	va_list	156
atoi	102	pow	125	va_start	156
atol	103	printf	125	vprintf	157
bsearch	104	putchar	130	vsprintf	157
calloc	104	puts	130	_formatted_read	158
ceil	105	qsort	131	_formatted_write	159
cos	106	rand	132	_medium_read	160
cosh	106	realloc	132	_medium_write	161
div	107	scanf	133	_small_write	162
exit	107	sin	136	library functions summary	89
exp	108	sinh	137	limits.h (header file)	95
exp 10	108	sprintf	137	linker command files	5
fabs	109	sqrt	138	list files (C compiler option)	85
floor	109	srand	139	list options	74
fmod	110	sscanf	139	list to named file	75
free	110	strcat	140	LOC (Assembler static	
frexp	111	strchr	141	overlay directive)	190
getchar	111	strcmp	141	location (#pragma directive)	173
gets	112	strcoll	142	LOCFRAME (Assembler static	
isalnum	112	strcpy	143	overlay directive)	189
isalpha	113	strcspn	143	log (library function)	119
iscntrl	113	strerror	144	log10 (library function)	120
isdigit	114	strlen	144	loop unrolling (C compiler option)	71
isgraph	114	strncat	145		
islower	115	strncmp	145		
isprint	115	strncpy	146	M	
ispunct	116	strpbrk	147	make a library module	
isspace	116	strrchr	147	(C compiler option)	72
isupper	117	strspn	148	malloc (library function)	59, 120
isxdigit	117	strstr	148	math.h (header file)	90
labs	118	strtod	149	memchr (library function)	121
ldexp	118	strtok	150	memcmp (library function)	122
ldiv	119	strtol	150	memcpy (library function)	122
log	119	strtol	151	memmove (library function)	123
log10	120	strxfrm	152	memset (library function)	124
malloc	120	tan	153	miscellaneous files	4
memchr	121	tanh	153		

miscellaneous options 85
 modf (library function) 124

N

non-volatile RAM 53
 NO_INIT (segment) 54, 200

O

object files (C compiler option) 85
 objekt module name
 (C compiler option) 73
 online help 8
 optimize for size
 (C compiler option) 70
 optimize for speed
 (C compiler option) 70
 options, see C compiler options
 output directories 84
 output options 72
 OVERLAYn (segment) 199

P

PATH variable 3, 4
 pointers
 casting 14
 __constptr 164
 pointers
 code 12
 data 13
 pow (library function) 125
 pragma directives 171
 pragmas
 constseg 172
 dataseg 172
 diagnostic 174
 diag_default 174

diag_error 174
 diag_remark 174
 diag_suppress 174
 diag_warning 174
 function 173
 language 174
 location 173
 vector 173
 predefined symbols
 __DATE__ 175
 __FILE__ 175
 __IAR_SYSTEMS_ICC__ 175
 __LINE__ 176
 __STDC__ 176
 __TID__ 177
 __TIME__ 177
 __VER__ 178
 preprocessor options 76
 preprocessor to output file
 (C compiler option) 78
 printf (library function) 57, 125
 PRM (Assembler static
 overlay directive) 189
 processor setup file
 (C compiler option) 83
 programming hints 16
 PTABLE_I (segment) 196
 PTABLE_ID (segment) 196
 ptrdiff_t 14
 putchar (library function) 56, 130
 puts (library function) 130

Q

QCCPIC (environment variable) 52
 QPICINFO (environment variable) 52
 qsort (library function) 131

R

rand (library function) 132
 RCODE (segment) 199
 Read-Me files 8
 realloc (library function) 132
 register I/O 59
 remark 217
 requirements 3
 RTABLE_I (segment) 196
 RTABLE_ID (segment) 196
 running
 C-SPY 3
 Embedded Workbench 2

S

scanf
 (library function) 58
 SCODE (segment) 199
 segments 193
 BANKN_I 196
 BANKN_ID 196
 CALLSTACK 196
 CODE 197
 CONST 197
 EEPROM_I 196
 EEPROM_ID 196
 HCODE 198
 INTVEC 198
 LCODE 198
 NO_INIT 54, 200
 OVERLAYn 199
 PTABLE_I 196
 PTABLE_ID 196
 RCODE 199
 RTABLE_I 196
 RTABLE_ID 196
 SCODE 199
 xxx_A 200

xxx_N	200	strchr (library function)	147		
xxx_Z	200	strspn (library function)	148	U	
set object filename		strstr (library function)	148	undefine symbol	
(C compiler option)	85	strtod (library function)	149	(C compiler option)	86
setjmp.h (header file)	91	strtok (library function)	150	use external memory	
setting C compiler options	63	strtol (library function)	150	(C compiler option)	84
setting options	63	strtoul (library function)	151	use standard output only	
severity level		strxfrm (library function)	152	(C compiler option)	86
specifying	218	support, technical	8		
severity levels	217	suppress these diagnostics			
shell for interfacing to		(C compiler option)	80	V	
assembler	183			va_arg (library function)	155
silent operation				va_end (library function)	155
(C compiler option)	86	T		va_list (library function)	156
sin (library function)	136	tan (library function)	153	va_start (library function)	156
sinh (library function)	137	tanh (library function)	153	vector (#pragma directive)	173
size_t	14	target identifier	177	vprintf (library function)	157
sprintf (library function)	57, 137	target options	82	vsprintf (library function)	157
sqrt (library function)	138	target-specific libraries	88		
srand (library function)	139	technical support	8		
sscanf (library function)	139	tolower (library function)	154	W	
stack size	55	toupper (library function)	154	warning	217
stdarg.h (header file)	91	treat these as errors		web	8
stdio.h (header file)	91, 92	(C compiler option)	81	Workbench	
stdlib.h (header file)	92	treat these as remarks	80	installing	2
storage	164	(C compiler option)		running	2
strcat (library function)	140	treat these as warnings	81		
strchr (library function)	141	(C compiler option)			
strcmp (library function)	141	tutorial	21		
strcoll (library function)	142	adding an interrupt handler	22	X	
strcpy (library function)	143	configuring	23	XLINK	5
strcspn (library function)	143	using #pragma directives	21	XLINK command file	
strerror (library function)	144	tutorial files	21, 22	memory model	54
strict ISO/ANSI		tutor2.mac	45	XLINK options, -A	57
(C compiler option)	68	tutor3.mac	49	XLINK_DFLTDIR	5
string.h (header file)	93	tutor3.txt	49	xxx_A (segment)	200
strlen (library function)	144	type_attribute		xxx_N (segment)	200
strncat (library function)	145	(#pragma directive)	171	xxx_Z (segment)	200
strncmp (library function)	145				
strncpy (library function)	146				
strpbrk (library function)	147				

Symbols

-A (XLINK option)	57	.h (file type)	6	__FILE__	
-D (C compiler option)	78	.inc (file type)	6	(predefined symbol)	175
-e (C compiler option)	32, 67	.ini (file type)	7	__IAR_SYSTEMS_ICC__	
-I (C compiler option)	76	.i39 (file type)	6	(predefined symbol)	175
-l (C compiler option)	75, 184	.lst (file type)	6	__interrupt (extended keyword)	168
-o (C compiler option)	73, 85	.mac (file type)	6	__intrinsic (extended keyword)	169
-P (C compiler option)	32	.map (file type)	6	__intrinsic (IAR keyword)	169
-r (C compiler option)	32, 73	.mem (file type)	6	__LINE__ (predefined symbol)	176
-s (C compiler option)	70, 184	.prj (file type)	6	__monitor (extended keyword)	168
-U (C compiler option)	86	.r39 (file type)	6	__no_init (extended keyword)	166
-v (C compiler option)	83, 177	.s39 (file type)	6	__no_operation	
-z (C compiler option)	70	.xcl (file type)	6	(intrinsic function)	180
--char_is_signed		@ operator	167	__option	
(C compiler option)	68	_formatted_read		(intrinsic function)	180
--debug (C compiler option)	73	(library function)	58, 158	__ptable (extended keyword)	164
--diag_error (C compiler option)	81	_formatted_write		__rtable (extended keyword)	164
--diag_remark (C compiler option)	80	(library function)	57, 159	__setN (extended keyword)	169
--diag_suppress (C compiler option)	80	_medium_read		__set_configuration_word	
--diag_warning (C compiler option)	81	(library function)	160	(intrinsic function)	180
--library module		_medium_write		__sleep	
(C compiler option)	72	(library function)	57, 161	(intrinsic function)	181
--module_name (C compiler option)	73	_small_write		__STDC__ (predefined symbol)	176
--no_code_motion		(library function)	58	__TID__ (predefined symbol)	177
(C compiler option)	71	__asm (inline assembler)	19	__TIME__ (predefined symbol)	177
--no_cse (C compiler option)	70	__asm (intrinsic function)	179	__tris	
--no_inline (C compiler option)	71	__bankN (extended keyword)	164	(intrinsic function)	181
--no_unroll (C compiler option)	71	__bankN_func		__VER__ (predefined symbol)	178
--no_warnings (C compiler option)	86	(extended keyword)	168		
--only_stdout (C compiler option)	86	__const (extended keyword)	166		
--preprocess (C compiler option)	78	__constptr	164		
--remarks (C compiler option)	80	__constptr (extended keyword)	166		
--silent (C compiler option)	86	__DATE__			
--strict_ansi (C compiler option)	68	(predefined symbol)	175		
--uses_external_memory		__disable_interrupt			
(C compiler option)	84	(intrinsic function)	179		
.a39 (file type)	6	__eeprom (extended keyword)	164		
.c (file type)	6	__enable_interrupt			
.cfg (file type)	7	(intrinsic function)	179		
.d39 (file type)	6				

