



MSP430 C COMPILER

Programming Guide

COPYRIGHT NOTICE

© Copyright 1995–1996 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 trademark of IAR Systems. Windows and MS-DOS are trademarks of Microsoft Corp.

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

First edition: September 1996

Part no: ICC430–1

This documentation was produced by Human-Computer Interface.

WELCOME

Welcome to the MSP430 C Compiler Programming Guide.

This guide provides reference information about the IAR Systems C Compiler for the MSP430 microprocessor.

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 *MSP430 Windows 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 *MSP430 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 MSP430 Assembler refer to the *MSP430 Assembler, Linker, and Librarian Programming Guide*.

If your product includes the optional MSP430 C-SPY debugger refer to the *MSP430 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* provides a brief summary of the MSP430 C Compiler's features.

The *Tutorial* 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.

C compiler options summary explains how to set the C compiler options, and gives a summary of them.

C compiler options reference gives information about each C compiler option.

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

Data representation describes how the compiler represents each of the C data types and gives recommendations for efficient coding.

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

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

Language extensions summarizes the extended keywords, `#pragma` keywords, predefined symbols, and intrinsic functions specific to the MSP430 C Compiler.

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

#pragma directive reference gives reference information about the `#pragma` keywords.

Predefined symbols reference gives reference information about the predefined symbols.

Intrinsic function 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.

K&R and ANSI C language definitions describes the differences between the K&R description of the C language and the ANSI standard.

Diagnostics lists the compiler warning and error messages.

ASSUMPTIONS

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



- ◆ The MSP430 processor.
- ◆ The C programming language.
- ◆ Windows, MS-DOS, or UNIX, depending on your host system.

This guide does not attempt to describe the C language itself. For a description of the C language, *The C Programming Language* by Kernighan and Richie is recommended, of which the latest edition also covers ANSI C.

Note that the illustrations in this guide show the Embedded Workbench running with Windows 95, 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.
[<i>option</i>]	An optional part of a command.
{ a b c }	Alternatives in a command.
bold	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.
<i>reference</i>	A cross-reference 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 the IAR Systems tools.

In this guide K&R is used as an abbreviation for *The C Programming Language* by Kernighan and Richie.

CONTENTS

INSTALLATION AND DOCUMENTATION ROUTE MAP	1
Command line versions	1
Windows Workbench versions	2
UNIX versions	3
Documentation route map	4
INTRODUCTION	5
C compiler	5
TUTORIAL	7
Typical development cycle	8
Getting started	9
Creating a program	12
Using I/O	26
Adding an interrupt handler	28
C COMPILER OPTIONS SUMMARY	35
Setting C compiler options	36
Options summary	37
C COMPILER OPTIONS REFERENCE	39
Code generation	39
Debug	49
#define	50
List	52
#undef	58
Include	59
Command line	61
CONFIGURATION	65
Introduction	65
XLINK command file	65
Run-time library	66
Memory map	66
Stack size	66
Input and output	67
Register I/O	70

Heap size	71
Initialization	71
DATA REPRESENTATION	73
Data types	73
Pointers	75
Efficient coding	75
GENERAL C LIBRARY DEFINITIONS	77
Introduction	77
C LIBRARY FUNCTIONS REFERENCE	85
LANGUAGE EXTENSIONS	155
Introduction	155
Extended keywords summary	155
#pragma directive summary	156
Predefined symbols summary	157
Intrinsic function summary	157
Other extensions	158
EXTENDED KEYWORD REFERENCE	159
#PRAGMA DIRECTIVE REFERENCE	165
PREDEFINED SYMBOLS REFERENCE	173
INTRINSIC FUNCTION REFERENCE	177
ASSEMBLY LANGUAGE INTERFACE	181
Creating a shell	181
Calling convention	182
Calling assembly routines from C	184
SEGMENT REFERENCE	187
K&R AND ANSI C LANGUAGE DEFINITIONS	193
Introduction	193
Definitions	193

DIAGNOSTICS.....	199
Compilation error messages	201
Compilation warning messages	217
INDEX	227

CONTENTS

INSTALLATION AND DOCUMENTATION ROUTE MAP

This chapter explains how to install and run the command line and Windows Workbench versions of the IAR products, and gives an overview of the user guides supplied with them.

Please note that some products only exist in a command line version, and that the information may differ slightly depending on the product or platform you are using.


COMMAND LINE VERSIONS

This section describes how to install and run the command line versions of the IAR Systems tools.

WHAT YOU NEED

- ◆ DOS 4.x or later. This product is also compatible with a DOS window running under Windows 95, Windows NT 3.51 or later, or Windows 3.1x.
- ◆ At least 10 Mbytes of free disk space.
- ◆ A minimum of 4 Mbytes of RAM available for the IAR applications.

INSTALLATION

- 1 Insert the first installation disk.
- 2 At the MS-DOS prompt type:
`a:\install` 
- 3 Follow the instructions on the screen.

When the installation is complete:

- 4 Make the following changes to your `autoexec.bat` file:

Add the paths to the IAR Systems executable and user interface files to the PATH variable; for example:

```
PATH=c:\dos;c:\utils;c:\iar\exe;c:\iar\ui;
```

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

```
set C_INCLUDE=c:\iar\inc\  
set XLINK_DFLTDIR=c:\iar\lib\  

```

- 5 Reboot your computer for the changes to take effect.
- 6 Read the Read-Me file, named *product.doc*, for any information not included in the guides.

RUNNING THE TOOLS

Type the appropriate command at the MS-DOS prompt.

For more information refer to the chapter *Getting started* in the *Command Line Interface Guide*.

WINDOWS WORKBENCH VERSIONS

This section explains how to install and run the Embedded Workbench.

WHAT YOU NEED

- ◆ Windows 95, Windows NT 3.51 or later, or Windows 3.1x.
- ◆ Up to 15 Mbytes of free disk space for the Embedded Workbench.
- ◆ A minimum of 4 Mbytes of RAM for the IAR applications.

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

INSTALLING FROM WINDOWS 95 OR NT 4.0

- 1 Insert the first installation disk.
- 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 OR NT 4.0

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

INSTALLING FROM WINDOWS 3.1x OR NT 3.51

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

RUNNING FROM WINDOWS 3.1x OR NT 3.51

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

RUNNING C-SPY

Either:

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

Or:

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

UNIX VERSIONS

This section describes how to install and run the UNIX versions of the IAR Systems tools.

WHAT YOU NEED

- ◆ HP9000/700 workstation with HP-UX 9.x (minimum), or a Sun 4/SPARC workstation with SunOS 4.x (minimum) or Solaris 2.x (minimum).

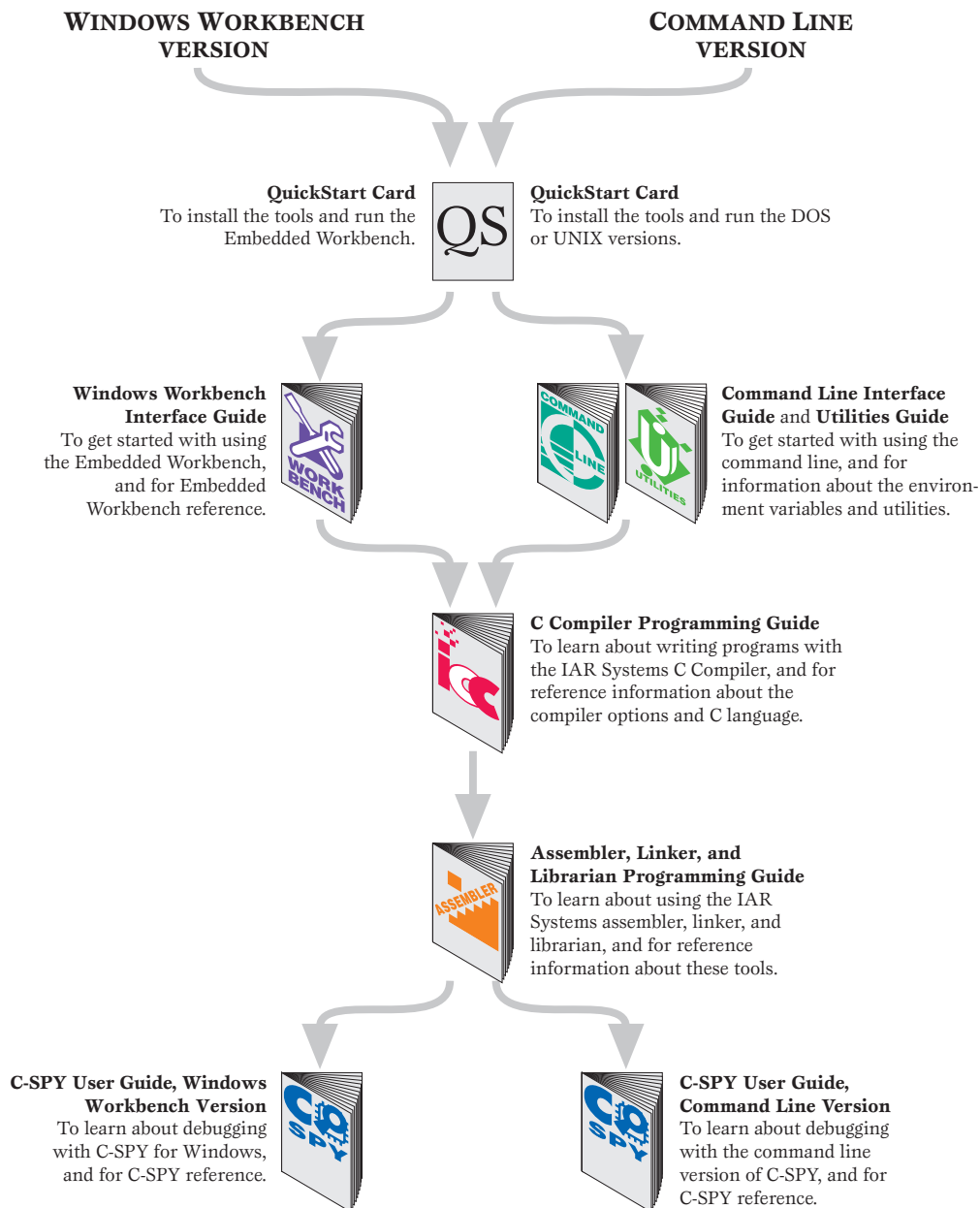
INSTALLATION

Follow the instructions provided with the media.

RUNNING THE TOOLS

Type the appropriate command at the UNIX prompt. For more information refer to the chapter *Getting started* in the *Command Line Interface Guide*.

DOCUMENTATION ROUTE MAP



INTRODUCTION

The IAR Systems MSP430 C compiler is available in two versions: a command line version, and a Windows version integrated with the IAR Systems Embedded Workbench development environment.

This guide describes both versions of the C compiler, and provides information about running it from the Embedded Workbench or from the command line, as appropriate.

C COMPILER

The IAR Systems C Compiler for the MSP430 microprocessor offers the standard features of the C language, plus many extensions designed to take advantage of the MSP430-specific facilities. The compiler is supplied with the IAR Systems Assembler for the MSP430, with which it is integrated, and shares linker and librarian manager tools.

It provides the following features:

LANGUAGE FACILITIES

- ◆ Conformance to the ANSI specification.
- ◆ Standard library of functions applicable to embedded systems, with source optionally available.
- ◆ IEEE-compatible floating-point arithmetic.
- ◆ Powerful extensions for MSP430-specific features, including efficient I/O.
- ◆ Linkage of user code with assembly routines.
- ◆ Long identifiers – up to 255 significant characters.
- ◆ Up to 32000 external symbols.

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.

- ◆ LINT-like checking of program source.

CODE GENERATION

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

TARGET SUPPORT

- ◆ Flexible variable allocation.
- ◆ Interrupt functions requiring no assembly language.
- ◆ A `#pragma` directive to maintain portability while using processor-specific extensions.

TUTORIAL

This chapter illustrates how you might use the MSP430 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 route map*.
- ◆ Be familiar with the architecture and instruction set of the MSP430 processor. For more information see the manufacturer's data book.

It is also recommended that you complete the introductory tutorial in the *MSP430 Windows Workbench Interface Guide* or *MSP430 Command Line Interface Guide*, as appropriate, to familiarize yourself with the interface you are using.

Summary of tutorial files

The following table summarizes the tutorial files used in this chapter:

<i>File</i>	<i>What it demonstrates</i>
tutor1	Compiling and running a simple C program.
tutor2	Using I/O.
tutor3	Interrupt handling.

RUNNING THE EXAMPLE PROGRAMS

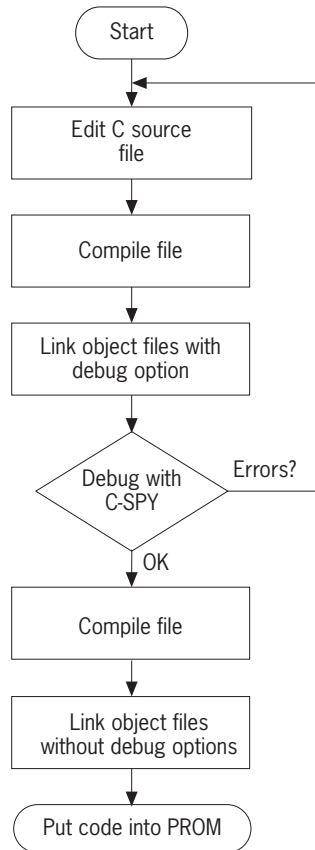
This tutorial shows how to run the example programs using the optional C-SPY simulator.

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

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

Choosing the linker command file

A suitable linker command file `lnk430.xcl` is provided in the `icc430` subdirectory.

Examine `lnk430.xcl` using a suitable text editor, such as the Embedded Workbench editor or the MS-DOS `edit` editor.

The file first contains the following XLINK command to define the CPU type as MSP430:

```
-cmsp430
```

It then contains a series of `-Z` commands to define the segments used by the compiler. The key segments are as follows:

<i>Segment type</i>	<i>Segment names</i>	<i>Address range</i>
DATA	IDATA0, UDATA0, ECSTR, CSTACK	0x0200 to 0x7FFF
CODE	RCODE, CODE, CDATA0, CONST, CSTR, CCSTR	0x8000 to 0xFFDF
CODE	INTVEC	0xFFE0 to 0xFFFF

For more information refer to the chapter *Segment reference*.

The file defines the routines to be used for `printf` and `scanf`. Finally it contains the following line to load the appropriate C library:

```
c1430
```

See *Run-time library*, page 66, for details of the different C libraries provided.

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 map*, page 66. For detailed information on choosing stack size, see *Stack size*, page 66.

CREATING A NEW PROJECT

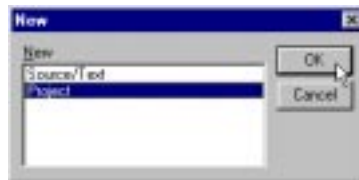
The first step is to create a new project for the tutorial programs.



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:



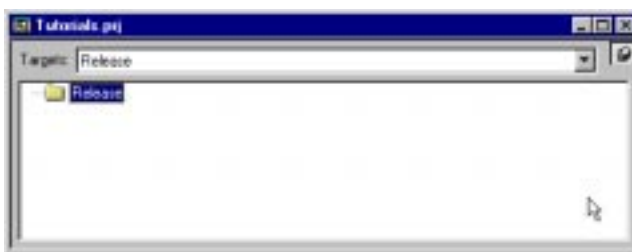
Select **Project** and choose **OK** to display the **New Project** dialog box.

Enter **Tutorials** in the **Project Filename** box, and set the **Target CPU Family** to **MSP430**:



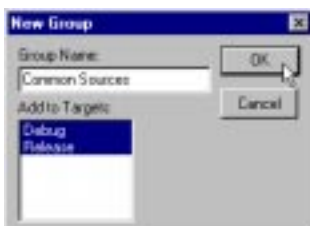
Then choose **OK** to create the new project.

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



Next, create a group to contain the tutorial source files as follows.

Choose **New Group...** from the **Project** menu and enter the name Common Sources. By default both targets are selected, so the group will be added to both targets:



Choose **OK** to create the group. It will be displayed in the Project window.



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.

The tutorial files are installed in the `icc430` directory. Select this directory by entering the command:

```
cd c:\iar\icc430 ↵
```

During this tutorial you will work in this directory, so that the files you create will reside here.

CREATING A PROGRAM The first tutorial demonstrates how to compile, link, and run a program.

ENTERING THE PROGRAM

The first program is a simple program using only standard C facilities. It repeatedly calls a function that increments a variable:

```
#include <stdio.h>
int call_count;
unsigned char my_char;
const char con_char='a';

void do_foreground_process(void)
{
    call_count++;
    putchar(my_char);
}

void main(void)
{
    int my_int=0;
    call_count=0;
    my_char=con_char;
    while (my_int<100)
    {
        do_foreground_process();
        my_int++;
    }
}
```



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 `tutor1.c`.

Alternatively, a copy of the program is provided in the C compiler 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 `tutor1.c`. Alternatively, a copy is provided in the C compiler files directory.

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

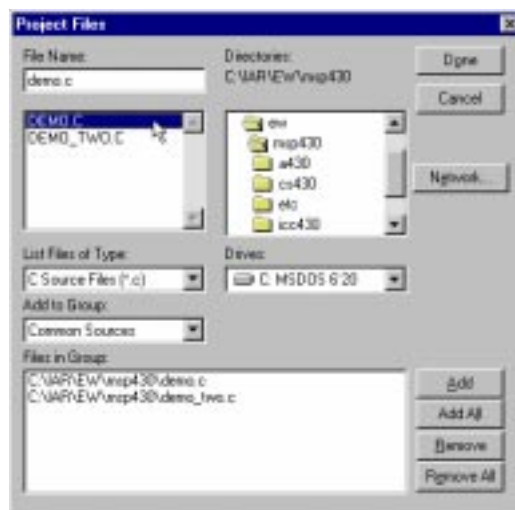
COMPILING THE PROGRAM




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 `tutor1.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:



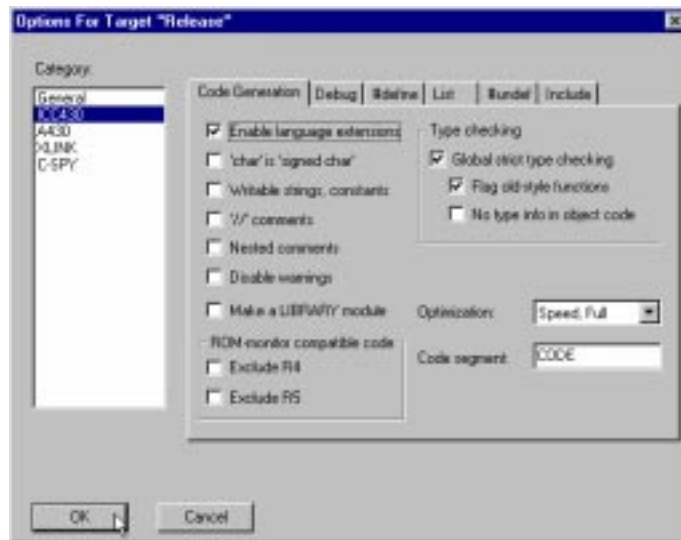
Then click **Done** to close the **Project Files** dialog box.

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 **Release** folder icon in the Project window, choose **Options...** from the **Project** menu, and select **ICC430** in the **Category** list to display the C compiler options pages:

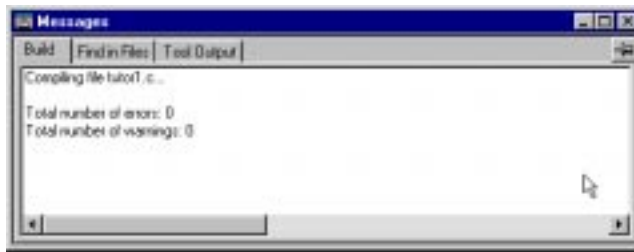


Make sure that the following options are selected on the appropriate pages of the **Options** dialog box:

<i>Page</i>	<i>Options</i>
Code generation	Enable language extensions
Debug	Generate debug information
List	List file Insert 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 listing is created in a file `tutor.lst`. Open this by choosing **Open...** from the **File** menu, and choosing `tutor1.lst` from the `release\list` directory.



Compiling the program from the command line

To compile the program enter the command:

```
icc430 -r -L -q tutor1 -I\iar\inc ↵
```

There are several compile options used here:

<i>Option</i>	<i>Description</i>
-r	Allows the code to be debugged with C-SPY.
-L	Creates a list file.
-q	Includes assembler code with C in the listing.
-I	Specifies the pathname for include files.

This creates an object module called `tutor1.r43` and a list file called `tutor1.lst`.



Viewing the listing

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

```
#####
#
# IAR MSP430 C-Compiler Vx.xx
# Front End Vx.xx
# Global Optimizer Vx.xx
#
# Source file = tutor1.c
# List file = tutor1.lst
# Object file = tutor1.r43
# Command line = -r -L -q tutor1 -I\iar\inc
#
#
# (c) Copyright IAR Systems 1996
#####

\ 0000 NAME tutor1(16)
\ 0000 RSEG CODE(1)
\ 0000 RSEG CONST(1)
\ 0000 RSEG UDATA0(1)
\ 0000 PUBLIC call_count
\ 0000 PUBLIC con_char
\ 0000 PUBLIC do_foreground_process
\ 0000 PUBLIC main
\ 0000 PUBLIC my_char
\ 0000 EXTERN putchar
\ 0000 EXTERN ?CL430_1_00_L08
\ 0000 RSEG CODE
\ 0000 do_foreground_process:
1 #include <stdio.h>
2 int call_count;
3 unsigned char my_char;
4 const char con_char='a';
5
6 void do_foreground_process(void)
7 {
8 call_count++;
\ 0000 92530000 ADD #1,&call_count
9 putchar(my_char);
\ 0004 5C420200 MOV.B &my_char,R12
```

```

\ 0008 7CF3          AND.B  #-1,R12
\ 000A B0120000      CALL   #putchar
10      }
\ 000E 3041          RET
\ 0010              main:
11
12      void main(void)
13      {
\ 0010 0A12          PUSH   R10
14      int my_int=0;
\ 0012 0A43          MOV    #0,R10
15      call_count=0;
\ 0014 82430000      MOV    #0,&call_count
16      my_char=con_char;
\ 0018 D2420000      MOV.B  &con_char,&my_char
\ 001C 0200
\ 001E              ?0001:
17      while (my_int<100)
\ 001E 3A906400      CMP    #100,R10
\ 0022 0434          JGE    (?0000)
18      {
19      do_foreground_process();
\ 0024 B0120000      CALL   #do_foreground_process
20      my_int++;
\ 0028 1A53          ADD    #1,R10
21      }
22      }
\ 002A F93F          JMP    (?0001)
\ 002C              ?0000:
\ 002C 3A41          POP    R10
\ 002E 3041          RET
23
24
\ 0000              RSEG   CONST
\ 0000              con_char:
\ 0000 61            DCB    'a'
\ 0000              RSEG   UDATA0
\ 0000              call_count:
\ 0002              DSB    2
\ 0002              my_char:
\ 0003              DSB    1

```

```
\ 0003                                END
```

```
Errors: none  
Warnings: none  
Code size: 48  
Constant size: 1  
Static variable size: 3
```

LINKING THE PROGRAM

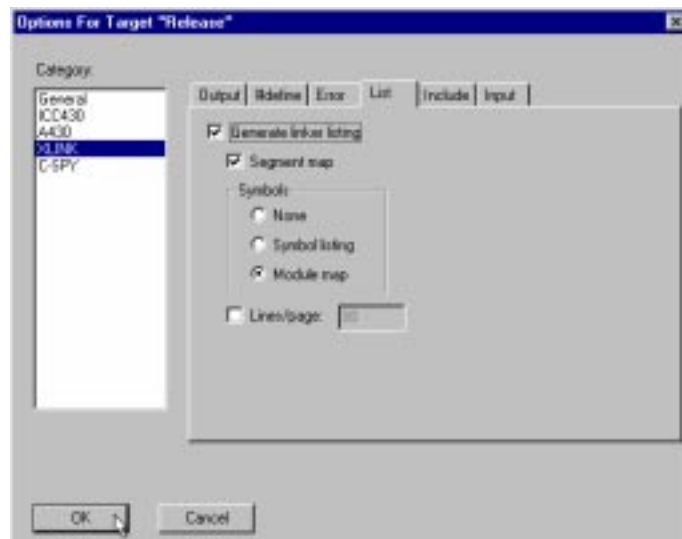


Linking the program using the Embedded Workbench

First set up the options for the XLINK Linker. Select the **Release** 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.

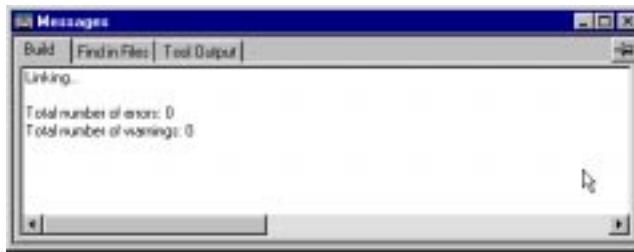
Then click **List** to display the page of list options.

Select **Generate linker listing** and **Segment map** to generate a map file to tutor1.map.



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 `tutorial.dbg` and a map file `tutorial.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 tutor1 -f lnk430 -rt -x -l tutor1.map ↵
```

The `-f` option specifies your XLINK command file `lnk430`, and the `-r` option allows the code to be debugged with C-SPY.

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

The result of linking is a code file called `about.a43` and a map file called `tutor1.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 on the following listing:

```
#####
#
# IAR Universal Linker Vx.xx
#
# Target CPU = msp430
# List file = tutor1.map
# Output file 1 = aout.d43
# Output format = debug
# Command line = tutor1 -f lnk430.xcl (-cMSP430
# -Z(CODE)RCODE,CODE,CDATA0,ZVECT,CONST,CSTR,
# CCSTR=8000-FFDF
# -Z(CODE)INTVEC=FFE0-FFFF
# -Z(DATA)IDATA0,UDATA0,ECSTR,WCSTR,TEMP,
# CSTACK+200=0200-7FFF
# -e_small_write=_formatted_write
# -e_medium_read=_formatted_read c1430.r43) -rt -x
# -l tutor1.map
#
# (c) Copyright IAR Systems 1996 #
#####
```

Command line —————
Equivalent command line.

Included XCL file —————
Commands included in the linker command file.

```
*****
*
* CROSS REFERENCE
*
*****
```

Program entry ————— Program entry at : 8030 Relocatable, from module : CSTARTUP
Shows the address of the program entry point.

```
*****
*
* MODULE MAP
*
*****
```

Module map —————
Information about each module that was loaded as part of the program.

File name

Shows the name of the file from which modules were linked.

Module

Type and name.

Segments in the module

A list of the segments in the specified module, with information about each segment.

Entries

Global symbols declared within the segment.

FILE NAME : tutor1.r43

PROGRAM MODULE, NAME : tutor1

SEGMENTS IN THE MODULE

=====

CODE

Relative segment, address : 8000 - 802F

ENTRIES	ADDRESS	REF BY MODULE
do_foreground_process	8000	Not referred to
calls direct		
main	8010	CSTARTUP
calls direct		
LOCALS	ADDRESS	
?0001	801E	
?0000	802C	

CONST

Relative segment, address : 80D2 - 80D2

ENTRIES	ADDRESS	REF BY MODULE
con_char	80D2	Not referred to

UDATA0

Relative segment, address : 0200 - 0202

ENTRIES	ADDRESS	REF BY MODULE
call_count	0200	Not referred to
my_char	0202	Not referred to

Next file

FILE NAME : c:\user\projects\iar\ti430\lib\c1430.r43

PROGRAM MODULE, NAME : CSTARTUP

SEGMENTS IN THE MODULE

=====

CODE

Relative segment, address : 8030 - 8071

ECSTR

Relative segment, address : Not in use

```
CCSTR
  Relative segment, address : Not in use
  -----

CDATA0
  Relative segment, address : Not in use
  -----

IDATA0
  Relative segment, address : Not in use
  -----

UDATA0
  Relative segment, address : Not in use
  -----

INTVEC
  Common segment, address : FFE0 - FFFF
  -----

CSTACK
  Relative segment, address : Not in use
  -----

LIBRARY MODULE, NAME : lowinit

SEGMENTS IN THE MODULE
=====

CODE
  Relative segment, address : 8072 - 8075
    ENTRIES      ADDRESS      REF BY MODULE
    ___low_level_init      8072      CSTARTUP
    -----

LIBRARY MODULE, NAME : 108
    ABSOLUTE ENTRIES      ADDRESS      REF BY MODULE
    =====
    ?CL430_1_00_L08      0001      tutor1
                                         memcpy
                                         memset
                                         putchar
    -----
```

Next module ——— LIBRARY MODULE, NAME : memcpy
Information about the next module in the current file.

SEGMENTS IN THE MODULE

=====

CODE

Relative segment, address : 8076 - 808F

ENTRIES	ADDRESS	REF BY MODULE
memcpy	8076	CSTARTUP

LIBRARY MODULE, NAME : memset

SEGMENTS IN THE MODULE

=====

CODE

Relative segment, address : 8090 - 80A7

ENTRIES	ADDRESS	REF BY MODULE
memset	8090	CSTARTUP

LIBRARY MODULE, NAME : putchar

SEGMENTS IN THE MODULE

=====

CODE

Relative segment, address : 80A8 - 80CB

ENTRIES	ADDRESS	REF BY MODULE
putchar	80AE	tutor1
calls direct		

LOCALS	ADDRESS
__low_level_put	80A8

LIBRARY MODULE, NAME : exit

SEGMENTS IN THE MODULE

=====

CODE

Relative segment, address : 80CC - 80D1

ENTRIES	ADDRESS	REF BY MODULE
exit	80CC	Not referred to
?C_EXIT	80CC	CSTARTUP

```
*****
*
* SEGMENTS IN DUMP ORDER
*
*****
```

```
*****
*
*
*      END OF CROSS REFERENCE
*
*
*****
```

24

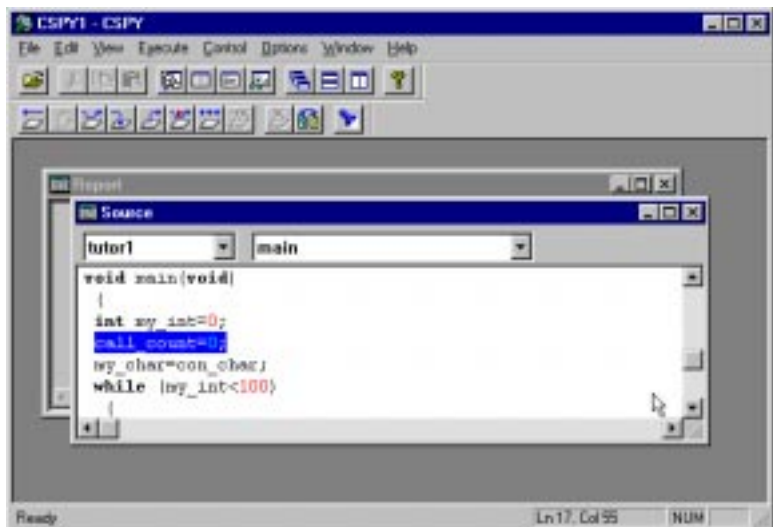
RUNNING THE PROGRAM



Running the program using the Embedded Workbench

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` as follows. Choose **Watch** from the **Window** menu, and click the **Watch** button in the Watch window toolbar:



Then type `call_count` to add this variable to the Watch window.

Choose **Step** from the **Execute** menu to step through the program until the line `do_foreground_process();` is reached, and check the value of the variable `call_count` in the Watch window. The value should be 0 since the variable has been initialized but not yet incremented.

Execute the current line and move to the next line in the loop. Examine `call_count` again – it should display 1, showing that the variable has been incremented by `do_foreground_process`.



Running the program from the command line

Execute the following command:

```
cs430 aout ↵
```

This loads the simulator and loads the program.

Type STEP or press **F2** to display the program and execute the first command.

Then display the value of `call_count` by typing:

```
call_count ↵
```

This will return the answer 0.

Then continue executing lines of the program by typing STEP or pressing **F2** until the line `my_int++` is highlighted.

Check the value of `call_count` again and it should now be 1.

If you also wish to simulate the routine `foreground_process` use the command ISTEP instead of STEP, or press **F3** instead of **F2**.

USING I/O

We shall now create a program that uses the processor's I/O ports. The resulting code will set up the LCD driver into the 4MUX mode, and then output 7 digits. This code demonstrates using the `#pragma` directive and header files.

The following is a listing of the code. Enter it into a suitable text editor and save it as `tutor2.c`. Alternatively, a copy is provided in the `icc430` subdirectory:

```
/* This example demonstrates how to display digits on the
LCD 4 MUX method */

/* enable use of extended keywords */
#pragma language=extended

/* include sfrb/sfrw definitions for I/O registers */
#include "io310.h"

char digit[10] = {
0xB7, /* "0" LCD segments a+b+c+d+e+f */
0x12, /* "1" */
```

```
0x8F, /* "2" */
0x1F, /* "3" */
0x3A, /* "4" */
0x3D, /* "5" */
0xBD, /* "6" */
0x13, /* "7" */
0xBF, /* "8" */
0x3F  /* "9" */
};

void main(void)
{
    int i;

    /* Initialize LCD driver (4Mux mode) */
    LCDCTL = 0xFF;

    /* display "6543210" */
    for (i=0; i<7; ++i)
        LCDMEM[i] = digit[i];
}
```

The first lines of the program are:

```
/* enable use of extended keywords */
#pragma language=extended
```

By default, extended keywords are not available so you must include this directive before attempting to use any. The `#pragma` directive is described in the chapter *#pragma directive reference*.

The next lines of code are:

```
/* include sfrb/sfrw definitions for I/O registers */
#include "io310.h"
```

The file `io310.h` includes definitions for all I/O registers for the 310 processors.

COMPILING AND LINKING THE PROGRAM



Compiling and linking the program using the Embedded Workbench

Choose **Files...** from the **Project** menu, and use the **Project Files** dialog box to remove the original `tutor1.c` file from the **Tutorials** project and add `tutor2.c` instead.

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 with the standard link file as follows:

```
icc430 tutor2 -r -L -q ↵
xlink tutor2 -f lnk430.xcl -r ↵
```



RUNNING THE PROGRAM

Single-step through the program using **F2** or by typing **step**.

On the real target it would be possible to attach an LCD display and watch it change. Using C-SPY it is only possible to watch the code execute.

ADDING AN INTERRUPT HANDLER

We shall now modify the previous program by adding an interrupt handler. The MSP430 C Compiler lets you write interrupt handlers directly in C using the `interrupt` keyword. The interrupt we will handle is the timer interrupt. This program sets up the timer to interrupt once a second and outputs a succession of digits to the LCD.

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

```
/* This example demonstrates how to use the basic timer
Interrupt frequency 1 Hz */

/* enable use of extended keywords */
#pragma language=extended

/* include sfr definitions for I/O registers and
intrinsic functions (_EINT) */
#include "io310.h"
#include "in430.h"
```

```

volatile int clock; /* count number of basic timer
                    interrupts */

char digit[10] = {
0xB7, /* "0" LCD segments a+b+c+d+e+f */
0x12, /* "1" */
0x8F, /* "2" */
0x1F, /* "3" */
0x3A, /* "4" */
0x3D, /* "5" */
0xBD, /* "6" */
0x13, /* "7" */
0xBF, /* "8" */
0x3F  /* "9" */
};

/* Basic Timer has vector address 0xFFE2, ie offset 2 in
INTVECT */

interrupt [0x02] void basic_timer(void)
{
    if (++clock == 10)
        clock = 0;
    /* Display 1,2,3,...,9,0,1,2,... */
    LCDMEM[0] = digit[clock];
}

void main(void)
{
    /* Initialize LCD driver (4Mux mode) */
    LCDCTL = 0xFF;
    /* Initialize Basic Timer
    Interrupt FQ is ACLK/256/128 = 1 Hz */
    BTCTL = 0xF6;
    ME2 |= 0x80; /* Set Basic Timer Module Enable */
    BTCTL &=~0x40; /* Disable Basic Timer Reset */
    IE2 |= 0x80; /* Set Basic Timer Interrupt Enable */
    clock = 0;
    /* Enable interrupts */
    _EINT();
    /* wait for interrupt */
    while (1);
}

```

The intrinsic include file must be present to define the `_EINT` function, and the I/O include must be present to define the MSP430 I/O registers:

```
/* enable use of extended keywords */
#pragma language=extended

/* include sfr definitions for I/O registers and
intrinsic functions (_EINT) */
#include "io310.h"
#include "in430.h"
```

The interrupt function itself is defined by the following lines:

```
interrupt [0x02] void basic_timer(void)
{
    if (++clock == 10)
        clock = 0;
    /* Display 1,2,3,...,9,0,1,2,... */
    LCDMEM[0] = digit[clock];
}
```

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 adding it to the **Tutorials** project and choosing **Make** from the **Project** menu.



Compiling and linking the program from the command line

Compile and link the program as before:

```
icc430 tutor3 -r -L -q ↵
```



VIEWING THE LISTING

From the listing you can see the code produced by the compiler for the interrupt function:

\	0000	NAME	tutor3(16)
\	0000	RSEG	CODE(1)
\	0000	COMMON	INTVEC(1)
\	0000	RSEG	UDATA0(1)
\	0000	RSEG	IDATA0(1)


```

\ 0000          RSEG    CDATA0(1)
\ 0000          PUBLIC  LCD_Mem
\ 0000          PUBLIC  basic_timer
\ 0000          PUBLIC  clock
\ 0000          PUBLIC  digit
\ 0000          PUBLIC  main
\ 0000          EXTERN  ?CL430_1_00_L08
\ 0000          RSEG    CODE
\ 0000          basic_timer:
1          /* This example demonstrates how to use the basic timer
2          Interrupt frequency 1 Hz */
3
4
5          /* enable use of extended keywords */
6          #pragma language=extended
7
8          /* include sfr definitions for I/O registers and intrinsic
9          functions (_EINT) */
10         #include "io310.h"
11         #include "in430.h"
12
13         volatile int clock; /* count number of basic timer
14         interrupts */
15
16         char digit[10] = {
17         0xB7, /* "0" LCD segments a+b+c+d+e+f */
18         0x12, /* "1" */
19         0x8F, /* "2" */
20         0x1F, /* "3" */
21         0x3A, /* "4" */
22         0x3D, /* "5" */
23         0xBD, /* "6" */
24         0x13, /* "7" */
25         0xBF, /* "8" */
26         0x3F /* "9" */
27         };
28
29         /* Basic Timer has vector address 0xFFE2, ie. offset 2 in
30         INTVECT */
31
32         interrupt [0x02] void basic_timer(void)

```

```

30      {
\ 0000 0C12          PUSH    R12
31      if (++clock == 10)
\ 0002 92530000      ADD     #1,&clock
\ 0006 B2900A00      CMP     #10,&clock
\ 000A 0000
\ 000C 0220          JNE     (?0001)
32      clock = 0;
\ 000E 82430000      MOV     #0,&clock
\ 0012          ?0001:
33      /* Display 1,2,3,...,9,0,1,2,... */
34      LCDMEM[0] = digit[clock];
\ 0012 1C420000      MOV     &clock,R12
\ 0016 D24C0200      MOV.B   digit(R12),&49
\ 001A 3100
35      }
\ 001C 3C41          POP     R12
\ 001E 0013          RETI
\ 0020          main:
36
37
38      void main(void)
39      {
40      /* Initialize LCD driver (4Mux mode) */
41      LCDCTL = 0xFF;
\ 0020 F2433000      MOV.B   #255,&48
42      /* Initialize Basic Timer
43      Interrupt FQ is ACLK/256/128 = 1 Hz */
44
45
46      BTCTL = 0xF6;
\ 0024 F240F600      MOV.B   #246,&64
\ 0028 4000
47      ME2 |= 0x80; /* Set Basic Timer Module Enable */
\ 002A F2D08000      BIS.B   #128,&5
\ 002E 0500
48      BTCTL &=~0x40; /* Disable Basic Timer Reset */
\ 0030 F2F0BF00      AND.B   #191,&64
\ 0034 4000
49      IE2 |= 0x80; /* Set Basic Timer Interrupt Enable */
\ 0036 F2D08000      BIS.B   #128,&1
\ 003A 0100

```

```

50          clock = 0;
\ 003C 82430000      MOV    #0,&clock
51          /* Enable interrupts */
52          _EINT();
\ 0040 32D2          EINT
\ 0042              ?0003:
53          /* wait for interrupt */
54          while (1);
55          }
\ 0042 FF3F          JMP    (?0003)
56
57
58
59
60
61
\ 0000              COMMON INTVEC
\ 0002              DSB    2
\ 0002 0000          DCW    basic_timer
\ 0000              RSEG   UDATA0
\ 0000              clock:
\ 0002              DSB    2
\ 0000              RSEG   IDATA0
\ 0000              LCD_Mem:
\ 0002              DSB    2
\ 0002              digit:
\ 000C              DSB    10
\ 0000              RSEG   CDATA0
\ 0000 3100          DCW    49
\ 0002 B7            DCB    183
\ 0003 12            DCB    18
\ 0004 8F            DCB    143
\ 0005 1F            DCB    31
\ 0006 3A            DCB    ':'
\ 0007 3D            DCB    '-'
\ 0008 BD            DCB    189
\ 0009 13            DCB    19
\ 000A BF            DCB    191
\ 000B 3F            DCB    '?'
\ 000C              END

```

C COMPILER OPTIONS

SUMMARY

This chapter gives a summary of the C compiler options, and explains how to set the options from the Embedded Workbench or the command line.



The options are divided into the following sections, corresponding to the pages in the **ICC430** options in the Embedded Workbench version:

Code generation	#undef
Debug	Include
#define	Target
List	

The *Command line* section provides information on those options which are only available in the command line version.

For full reference about each option refer to the following chapter, *C compiler options reference*.

These chapters use the following symbols:

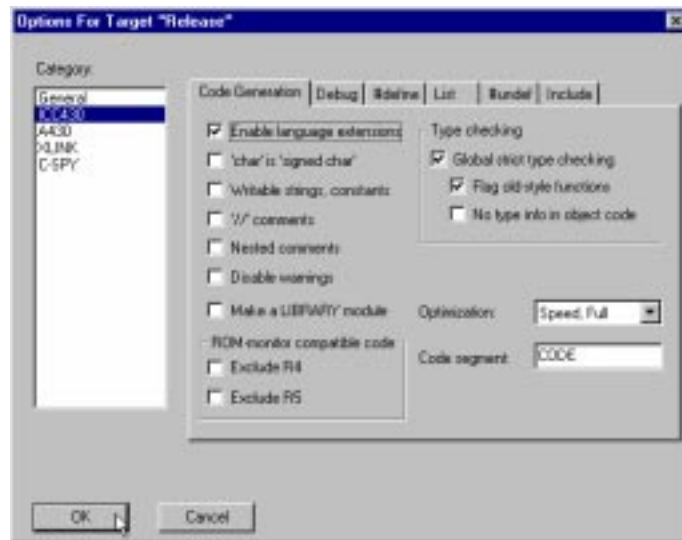
<i>Style</i>	<i>Used for</i>
	Identifies instructions specific to the versions of the IAR Systems tools for the Embedded Workbench interface.
	Identifies instructions specific to the command line versions of the IAR Systems tools.

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 **ICC430** 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.



Setting C compiler options from the command line

To set C compiler options you include them on the command line after the `icc430` command, either before or after the source filename. For example, when compiling the source `prog`, to generate a listing to the default listing filename (`prog.lst`):

```
icc430 prog -L ↵
```

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

```
icc430 prog -l list.lst ↵
```

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

```
icc430 prog -Llist ↵
```

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

Options can also be specified in the QCC430 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 next chapter, *C compiler options reference*.

<i>Option</i>	<i>Description</i>	<i>Section</i>
<code>-Aprefix</code>	Assembly output to prefixed filename.	List
<code>-a filename</code>	Assembly output to named file.	List
<code>-b</code>	Make object a library module.	Command line
<code>-C</code>	Nested comments.	Code generation
<code>-c</code>	Char is signed char.	Code generation
<code>-Dsymb [=xx]</code>	Defined symbols.	<code>#define</code>
<code>-e</code>	Enable language extensions.	Code generation
<code>-F</code>	Form-feed after function.	List
<code>-ffilename</code>	Extend the command line.	Command line
<code>-G</code>	Open standard input as source.	Command line
<code>-g</code>	Global strict type check.	Code generation
<code>-gA</code>	Flag old-style functions.	Code generation
<code>-g0</code>	No type info in object code.	Code generation
<code>-Hname</code>	Set object module name.	Command line
<code>-Iprefix</code>	Include paths.	Include
<code>-i</code>	Add <code>#include</code> file lines.	List
<code>-K</code>	<code>//</code> comments.	Code generation

<i>Option</i>	<i>Description</i>	<i>Section</i>
-L[<i>prefix</i>]	List to prefixed source name.	List
-l <i>filename</i>	List to named file.	List
-N <i>prefix</i>	Preprocessor to prefixed filename.	List
-n <i>filename</i>	Preprocessor to named file.	List
-O <i>prefix</i>	Set object filename prefix.	Command line
-o <i>filename</i>	Set object filename.	Command line
-p <i>nn</i>	Lines/page.	List
-q	Insert mnemonics.	List
-P	Generate promable code.	Command line
-R <i>name</i>	Set code segment name.	Code generation
-r[012][i][n][r]	Generate debug information.	Debug
-S	Set silent operation.	Command line
-s[0-9]	Optimize for speed.	Code generation
-T	Active lines only.	List
-t <i>n</i>	Tab spacing.	List
-Usymb	Undefine symbol.	#undef
-ur[4][5]	ROM-monitor compatible code.	Code generation
-w	Disable warnings.	Code generation
-X	Explain C declarations.	List
-x[DFT2]	Cross reference.	List
-z[0-9]	Optimize for size.	Code generation

C COMPILER OPTIONS REFERENCE

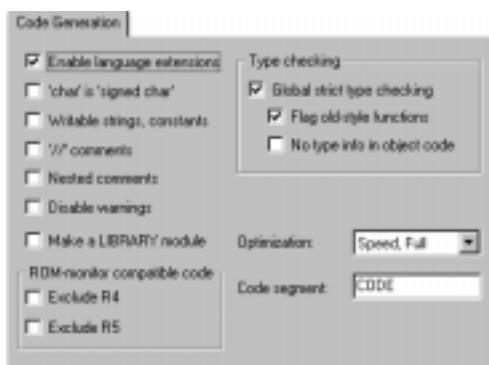
This chapter gives detailed information on each of the MSP430 C Compiler options, divided into functional categories.

CODE GENERATION

The code generation options determine the interpretation of the source program and the generation of object code.



Embedded Workbench



Command line

- | | |
|-----|------------------------------|
| -e | Enable language extensions. |
| -c | 'char' is 'signed char'. |
| -y | Writable strings, constants. |
| -K | '/' comments. |
| -C | Nested comments. |
| -w | Disable warnings. |
| -b | Make a LIBRARY module. |
| -g | Global strict type checking. |
| -gA | Flag old-style functions. |
| -g0 | No type info in object code. |

-z[0-9]	Optimize for size.
-s[0-9]	Optimize for speed.
-Rname	Code segment.
-ur[4][5]	ROM-monitor compatible code.

ENABLE LANGUAGE EXTENSIONS (-e)

Syntax: -e

Enables target dependent extensions to the C language.

Normally, language extensions are disabled to preserve compatibility. If you are using language extensions in the source, you must enable them by including this option.

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

‘CHAR’ IS ‘SIGNED CHAR’ (-c)

Syntax: -c

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: the run-time library is compiled without the **Char is signed char** (-c) option, so if you use this option for your program and enable type checking with the **Global strict type check** (-g) or **Generate debug information** (-r) options, you may get type mismatch warnings from the linker.

WRITABLE STRINGS, CONSTANTS (-y)

Syntax: -y

Causes the compiler to compile string literals as writable variables.

Normally, string literals are compiled as read-only. If you want to be able to write to string literals, you use the **Writable strings, constants** (-y) option, causing strings to be compiled as writable variables.

Note that arrays initialized with strings (ie `char c[] = "string"`) are always compiled as initialized variables, and are not affected by the **Writable strings, constants** (-y) option.

‘//’ COMMENTS (-K)

Syntax: -K

Enables comments in C++ style, that is, comments introduced by ‘//’ and extending to the end of the line.

Normally for compatibility the compiler does not accept C + + style comments. If your source includes C + + style comments, you must use the ‘//’ **comments** (-K) option for them to be accepted.

NESTED COMMENTS (-C)

Syntax: -C

Enables nested comments.

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

DISABLE WARNINGS (-w)

Syntax: -w

Disables compiler warning messages.

Normally, the compiler issues standard warning messages, and any additional warning messages enabled with the **Global strict type check** (-g) option. To disable all warning messages, you use the **Disable warnings** (-w) option.

MAKE A LIBRARY MODULE (-b)

Syntax: -b

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

The compiler normally produces a program module ready for linking with XLINK. If instead you want a library module for inclusion in a library with XLIB, you use the **Make a LIBRARY module** (-b) option.

GLOBAL STRICT TYPE CHECKING (-g)

Syntax: -g[A][0]

Enable checking of type information throughout the source.

There is a class of conditions in the source that indicate possible programming faults but which for compatibility the compiler and linker normally ignore. To cause the compiler and linker to issue a warning each time they encounter such a condition, use the **Global strict type checking** (-g) option.

FLAG OLD-STYLE FUNCTIONS (-gA)

Syntax: -gA

Normally, the **Global strict type checking** (-g) option does not warn of old-style K&R functions. To enable such warnings, use the **Flag old-style functions** (-gA) option.

NO TYPE INFO IN OBJECT CODE (-g0)

Syntax: -g0

Normally, the **Global strict type checking** (-g) option includes type information in the object module, increasing its size and link time, allowing the linker to issue type check warnings. To exclude this information, avoiding this increase in size and link time but inhibiting linker type check warnings, use the **No type info in object code** (-g0) option.

When linking multiple modules, note that objects in a module compiled without type information, that is without any -g option or with a -g option with 0 modifier, are considered typeless. Hence there will never be any warning of a type mismatch from a declaration from a module compiled without type information, even if the module with a corresponding declaration has been compiled with type information.

The conditions checked by the **Global strict type checking** (-g) option are:

- ◆ Calls to undeclared functions.
- ◆ Undeclared K&R formal parameters.
- ◆ Missing return values in non-void functions.

- ◆ Unreferenced local or formal parameters.
- ◆ Unreferenced goto labels.
- ◆ Unreachable code.
- ◆ Unmatching or varying parameters to K&R functions.
- ◆ #undef on unknown symbols.
- ◆ Valid but ambiguous initializers.
- ◆ Constant array indexing out of range.

Examples

The following examples illustrate each of these types of error.

Calls to undeclared functions

Program:

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

Error:

```
my_func();        /* mis-spelt my_fun gives undeclared
                  function warning */
-----^
"undecfn.c",5  Warning[23]: Undeclared function
'my_func'; assumed "extern" "int"
```

Undeclared K&R formal parameters

Program:

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

Error:

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

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

Missing return values in non-void functions

Program:

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

Error:

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

Unreferenced local or formal parameters

Program:

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

Error:

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

Unreferenced goto labels

Program:

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

```
    return 0;
}
```

Error:

```
}
^
```

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

Unreachable code

Program:

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

Error:

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

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

Unmatching or varying parameters to K&R functions

Program:

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

Error:

```
my_fun(buffer,99) ;      /* wrong order of parameters */
-----^
"varyparm.c",14 Warning[26]: Inconsistent use of K&R
function - changing type of parameter
my_fun(buffer,99) ;      /* wrong order of parameters */
-----^
"varyparm.c",14 Warning[26]: Inconsistent use of K&R
function - changing type of parameter
my_fun(99) ;            /* missing parameter */
-----^
"varyparm.c",15 Warning[25]: Inconsistent use of K&R
function - varying number of parameters
```

#undef on unknown symbols

Program:

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

Error:

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

Valid but ambiguous initializers

Program:

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


Error:

```
type3 example = {99, {42,1,2}, 37} ;
-----^
"ambigini.c",4 Warning[12]: Incompletely bracketed
initializer
```

Constant array indexing out of range

Program:

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

Error:

```
buffer[500] = 'a' ;    /* Constant index out of range */
-----^
"arrindex.c",5 Warning[28]: Constant [index] outside
array bounds
```

OPTIMIZE FOR SIZE (-z)

Syntax: -z[0-9]

Causes the compiler to optimize the code for minimum size.

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

<i>Modifier</i>	<i>Level</i>
0	No optimization.
1-3	Fully debuggable.
4-6	Some constructs are not debuggable.
7-9	Full optimization. Some constructs are not debuggable.

OPTIMIZE FOR SPEED (-s)**Syntax:** -s[0-9]

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

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

<i>Modifier</i>	<i>Level</i>
0	No optimization.
1-3	Fully debuggable.
4-6	Some constructs are not debuggable.
7-9	Full optimization. Some constructs are not debuggable.

CODE SEGMENT (-R)**Syntax:** -R*name*

Sets the name of the code segment.

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

ROM-MONITOR COMPATIBLE CODE (-ur)**Syntax:** -ur[4][5]

Causes the compiler to generate ROM-monitor compatible code by not using register R4 and/or R5.

DEBUG

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



Embedded Workbench



Command line

-r[012][i][n][r] Generate debug information.

GENERATE DEBUG INFORMATION (-r)

Syntax: -r[012][i][n][r]

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

Normally the compiler does not include debugging information, for code efficiency. To make code debuggable with C-SPY, you simply include the option with no modifiers.

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

<i>Option</i>	<i>Command line</i>
Add #include file information.	i
Suppress source in object code.	n
No register variables.	r
Code added to statements.	0, 1, 2

Normally the **Generate debug information** (-r) option does not include `#include` file debugging information, because this is usually of little interest, and most debuggers other than C-SPY do not support debugging inside `#include` files well. If you want to debug inside `#include` files, for example if the `#include` files contain function definitions rather than the more usual function declarations, you use the **Add #include file information** (-ri) modifier. A side effect is that source line records contain the global (= total) line count which can affect source line displays in some debuggers other than C-SPY.

The **Generate debug information** (-r) option usually includes C source lines in the object file, so they can be displayed during debugging. If you want to suppress this to reduce the size of the object file, you use the **Suppress source in object code** (-rn) modifier. Use this option for most other debuggers that do not include specific information about how to use IAR Systems C compilers.

Normally, the compiler tries to put locals as register variables. However, some debuggers cannot handle register variables; to suppress the use of register variables use the **No register variables** (-rr) modifier.

The **Code added to statements** options add one (-r1) or two (-r2) NOPs to the code generated for each statement. Only use one of these options if your debugging tool specifically requires you to do so.

#define

The **#define** option allows you to define symbols for use by the C compiler.



Embedded Workbench



**Command line**

-D Defined symbols.

DEFINED SYMBOLS (-D)

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

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

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

-D*symp* is equivalent to `#define symp`

The **Defined symbols** (-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:

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

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

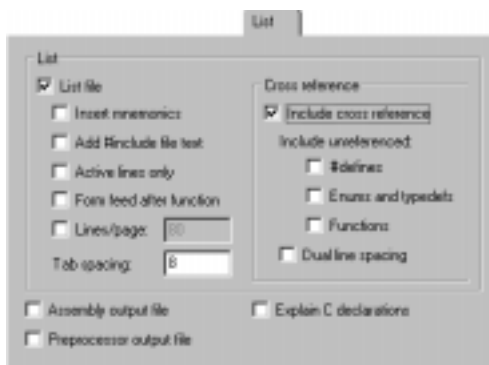
```
production version:  icc430 prog
test version:         icc430 prog -Dtestver
```

LIST

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



Embedded Workbench



Command line

- L[*prefix*] List to prefixed source name.
- l *filename* List to named file.
- q Insert mnemonics.
- i Add #include file text.
- T Active lines only.
- F Form feed after function.
- p*nn* Lines/page.
- t*n* Tab spacing.
- x[DFT2] Cross reference.
- A*prefix* Assembly output to prefixed filename.
- a *filename* Assembly output to named file.
- N*prefix* Preprocessor to prefixed filename.
- n *filename* Preprocessor to named file.
- X Explain C declarations.



LIST FILE

List to prefixed source name (-L)

Syntax: `-L[prefix]`

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

Normally, the compiler does not generate a listing. To simply generate a listing, you use the `-L` option without a prefix. For example, to generate a listing in the file `prog.lst`, you use:

```
icc430 prog -L
```



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

```
icc430 prog -Llist
```

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

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



List to named file (-l)

Syntax: `-l 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:

```
icc430 prog -l list
```

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

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

INSERT MNEMONICS (-q)

Syntax: `-q`

Includes generated assembly lines in the listing.

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

Note that this option is only available if a listing is specified.

See also options `-a`, `-A`, `-l`, and `-L`.

ADD #INCLUDE FILE TEXT (-i)

Syntax: `-i`

Causes the listing to include `#include` files.

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

ACTIVE LINES ONLY (-T)

Syntax: `-T`

Causes the compiler to list only active source lines.

Normally the compiler lists all source lines. To save listing space by eliminating inactive lines, such as those in false `#if` structures, you use the **Active lines only** (`-T`) option.

FORM-FEED AFTER FUNCTION (-F)

Syntax: `-F`

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

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

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

LINES/PAGE (-p)

Syntax: `-pnn`

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

Normally, the listing is not formatted into pages. To format it into pages with a form feed at every page, you use the **Lines/page** (-p) option. For example, for a printer with 50 lines per page:

```
icc430 prog -p50 
```

TAB SPACING (-t)

Syntax: -t*n*

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

Normally, the listing is formatted with a tab spacing of 8 characters. If you want a different tab spacing, you set it with the **Tab spacing** (-t) option.

CROSS REFERENCE (-x)

Syntax: -x[*DFT2*]

Includes a cross-reference list in the listing.

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

When you select **Cross reference** the following options become available:

<i>Command line</i>	<i>Option</i>
D	Show unreferenced <code>#defines</code> .
F	Show unreferenced functions.
T	Show unreferenced typedefs and enum constants.
2	Dual line spacing.



ASSEMBLY OUTPUT FILE

Assembly output to prefixed filename (-A)

Syntax: `-Aprefix`

Generates assembler source to *prefix*.s43.

By default the compiler does not generate an assembler source. To send assembler source to the file with the same name as the source leafname but with the extension .s43, use -A without an argument. For example:

```
icc430 prog -A
```

generates an assembly source to the file prog.s43.



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

```
icc430 prog -Aasm
```

generates an assembly source in the file asm\prog.s43.

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

If the -l or -L option is also used, the C source lines are included in the assembly source file as comments.

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



Assembly output to named file (-a)

Syntax: `-a filename`

Generates assembler source to *filename*.s43.

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

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

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

If the -l or -L option is also used, the C source lines are included in the assembly source file as comments.

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

PREPROCESSOR TO PREFIXED FILENAME (-N)

Syntax: -N*prefix*

Generates preprocessor output to *prefix source.i*.

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

```
icc430 prog -N ↵
```

generates preprocessor output to the file *prog.i*.

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

```
icc430 prog -Npreproc\ ↵
```

generates an assembly source in the file *preproc\prog.i*.

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

PREPROCESSOR TO NAMED FILE (-n)

Syntax: -n *filename*

Generates preprocessor output to *filename.i*.

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

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

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

EXPLAIN C DECLARATIONS (-X)

Syntax: -X

Displays an English description of each C declaration in the file.

To obtain English descriptions of the C declarations, for example to aid the investigation of error messages, you use the **List C declarations** (*-X*) option.

For example, the declaration:

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

gives the description:

storage class: extern

[func_attr:0210] prototyped ?cptr0 function returning

[attribute:0110] ?dptr0 - ?cptr0 code pointer to

[func_attr:0210] prototyped ?cptr0 function

returning

[attribute:0110] ?dptr0 - void

and having following parameter(s):

storage class : auto

[attribute:0110] ?dptr0 - int

and having following parameter(s):

storage class: auto

[attribute:0110] ?dptr0 - int

storage class : auto

[attribute:0110] ?dptr0 - ?cptr0 code pointer to

[func_attr:0210] ?cptr0 function returning

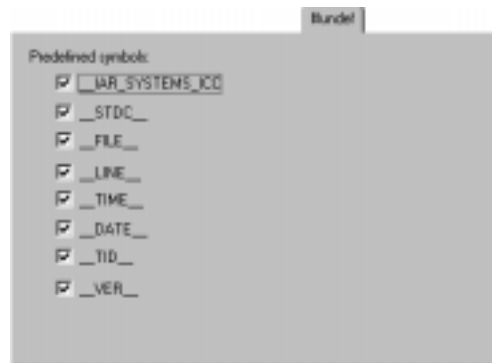
[attribute:0110] ?dptr0 - void

#undef

The **#undef** option allows you to undefine predefined symbols.



Embedded Workbench



Command line

-U*symb* Predefined symbols.

PREDEFINED SYMBOLS (-U)

Syntax: `-Usymb`

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 **Predefined symbols** (-U) option.

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



To undefine a symbol, deselect it in the **Predefined symbols** list.



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

```
icc430 prog -U__VER__
```

INCLUDE

The **Include** option allows you to define the include path for the C compiler.



Embedded Workbench



Command line

`-Iprefix` Include paths.

INCLUDE PATHS (-I)

Syntax: `-Iprefix`

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

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

For example:

```
icc430 prog -I\mylib\ ↵
```

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

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

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

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

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

it performs the following search sequence:

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

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

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

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

COMMAND LINE

The following additional options are available from the command line.

<code>-f filename</code>	Extend the command line.
<code>-G</code>	Open standard input as source.
<code>-H name</code>	Set object module name.
<code>-O prefix</code>	Set object filename prefix.
<code>-o filename</code>	Set object filename.
<code>-P</code>	Generate PROMable code.
<code>-S</code>	Set silent operation.



EXTEND THE COMMAND LINE (-f)

Syntax: `-f filename`

Reads command line options from the named file, with the default extension `.xcl`.

Normally, the compiler accepts command parameters only from the command line itself and the `QCC430` environment variable. To make long command lines more manageable, and to avoid any operating system command line length limit, you use the `-f` option to specify a command file, from which the compiler reads command line items as if they had been entered at the position of the option.

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

For example, you could replace the command line:

```
icc430 prog -r -L -Dtestver "-Dusername=John Smith"
-Duserid=463760 ↵
```

with

```
icc430 prog -r -L -Dtestver -f userinfo ↵
```

and the file `userinfo.xcl` containing:

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

**OPEN STANDARD INPUT AS SOURCE (-G)****Syntax:** -G

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

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

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

**SET OBJECT MODULE NAME (-H)****Syntax:** -H*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 -H option, for example:

```
icc430 prog -Hmain
```

This is particularly useful when several modules have the same filename, since normally the resulting duplicate module name would cause a linker error. 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
icc430 temp.c             ; module name is always 'temp'
```

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

```
preproc %1.c temp.c      ; preprocess source, generating
                           temp.c
icc430 temp.c -H%1       ; use original source name as
                           module name
```

**SET OBJECT FILENAME PREFIX (-O)****Syntax:** -O*prefix*

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

Normally (and unless the `-o` option is used) the object is stored with the filename corresponding to the source filename, but with the extension `.r43`. To store the object in a different directory, you use the `-O` option.

For example, to store the object in the `\obj` directory, use:

```
icc430 prog -O\obj\ ↵
```

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



SET OBJECT FILENAME (-o)

Syntax: `-o filename`

Set the filename in which the object module will be stored. The filename consists of an optional pathname, obligatory leafname, and optional extension (default `.r43`).

Normally the compiler stores the object code in a file whose name is:

- ◆ The prefix specified by `-o`, plus
- ◆ The leafname of the source, plus
- ◆ The extension `.r43`.

To store the object in a different filename, you use the `-o` option. For example, to store it in the file `obj.r43`, you would use:

```
icc430 prog -o obj ↵
```

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

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



GENERATE PROMABLE CODE (-P)

Syntax: `-P`

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

This option is included for compatibility with other IAR compilers, but in the MSP430 C Compiler is always active.



SET SILENT OPERATION (-S)

Syntax: -S

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 -S option. This does not affect the display of error and warning messages.

CONFIGURATION

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

INTRODUCTION

Systems based on the MSP430 microprocessor can vary considerably in their requirements.

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 element</i>	<i>See page</i>
Memory map	XLINK command file.	66
Non-volatile RAM	XLINK command file.	66
Stack size	XLINK command file.	66
putchar and getchar functions	Run-time library module.	67
printf/scanf facilities	XLINK command file.	69, 70
Heap size	Heap library module.	71
Hardware/memory initialization	__low_level_init module.	71

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.

XLINK COMMAND FILE

To create an XLINK command file for a particular project you should first copy the file `lnk430.xcl` from `c:\iar\icc430`. You should then modify this file, as described within the file, to specify the details of the target system's memory map.

RUN-TIME LIBRARY

The XLINK command file refers to the library file `cl430.r43`. This should not normally be changed.

MEMORY MAP

You need to 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.

The link options specify:

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

For details of specifying the memory address ranges, see the contents of the XLINK command file template and the XLINK section of the *MSP430 Assembler, Linker, and Librarian Programming Guide*.

NON-VOLATILE RAM

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

To assign the `NO_INIT` segment to the address of the non-volatile RAM, you need to modify the XLINK command file. For details of assigning a segment to a given address, see the XLINK section of the *MSP430 Assembler, Linker, and Librarian Programming Guide*.

STACK SIZE

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

ESTIMATING THE REQUIRED STACK SIZE

The stack is used for the following:

- ◆ Preserving register variables across function calls.

- ◆ Storing local variables and parameters.
- ◆ Storing temporary results in expressions.
- ◆ Storing temporary values in run-time library routines.
- ◆ Saving the return address of function calls.
- ◆ Saving the processor state during interrupts.

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

CHANGING THE STACK SIZE

The default stack size is set to 512 (200h) bytes in the linker command files, with the expression `CSTACK+200` in the linker command:

```
-Z(DATA)CSTACK+200
```

To change the stack size edit the linker command file and replace 200 by the size of the stack you want to use.

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 `c:\iar\icc430\putchar.c` and `c:\iar\icc430\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 back under the same name (or create your own routine using `putchar.c` as a model).

For example, the code below uses memory-mapped I/O to write to an LCD display:

```
int putchar(int outchar)
/* a very basic LCD putchar routine */
/* pos must be initialized to 15, the */
/* LCD must be initialized and character_map */
```

```
/* must be set up with the lookup table. */  
/* It will then display the first 15 characters */  
/* supplied */  
{if (pos>0)  
    LCDMEM[-pos]=character_map(outchar);  
}
```

- ◆ Compile the modified putchar using the appropriate processor option.

```
icc430 putchar -b
```

This will create an optimized replacement object module file named `putchar.r43`.

- ◆ Add the new putchar module to the appropriate run-time library module, replacing the original. For example, to add the new putchar module to the standard library, use the command:

```
xlib  
def-cpu MSP430  
rep-mod putchar c14308ss  
exit
```

The library module `c1430` will now have the modified putchar instead of the original. (Be sure to save your original `c1430.r43` 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 `-A` option. Place the following lines into your `.xcl` link file:

```
-A putchar  
c1430
```

This causes your version of `putchar.r43` to load instead of the one in the `c1430` library. See the *MSP430 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 as two C files, `getchar.c` and `llget.c`.

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 `int` 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 THE 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 MSP430 I/O system using the memory-mapped internal special-function registers (SFRs).

All operators that apply to integral types except the unary `&` (address) operator may be applied to SFR registers. Predefined `sfrb/sfrw` declarations for the MSP430 family are supplied; see *Run-time library*, page 66, and the chapter *Extended keyword reference*.

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

<i>Processor</i>	<i>Header file</i>
------------------	--------------------

MSP430x31x	<code>io310.h</code>
------------	----------------------

MSP430x32x	<code>io320.h</code>
------------	----------------------

MSP430x33x	<code>io330.h</code>
------------	----------------------

These files are provided in the `icc430` subdirectory.

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.

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

You can test the modified heap module by including the following lines in the `.xcl` link file:

```
-A heap
c14301
```

This will load your version of `heap.r43` instead of the one in the `c14301` library.

INITIALIZATION

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

- ◆ Initializes the stack pointer.
- ◆ Initializes C file-level and static variables.
- ◆ Calls the user program function `main`.

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

VARIABLE AND I/O INITIALIZATION

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

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

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

The source of `__low_level_init` is provided in the file `lowinit.c`, by default located in the `icc430` directory. To perform your own I/O initializations, create a version of this routine containing the necessary code to do the initializations. If you also want to disable the initialization of data segments, make the routine return 0. Compile the customized routine and link it with the rest of your code.

MODIFYING CSTARTUP

If you want to modify CSTARTUP itself you will need to reassemble CSTARTUP with options which match your selected compilation 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 `c:\iar\icc430\cstartup.s43`, and save it under the same name.
- ◆ Assemble CSTARTUP.

This will create an object module file named `cstartup.r43`.

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  
-C library
```



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

DATA REPRESENTATION

This chapter describes how the MSP430 C Compiler represents each of the C data types, and gives recommendations for efficient coding.

DATA TYPES

The MSP430 C Compiler supports all ANSI C basic elements. Variables are stored with the least significant part located at the low memory address.

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

<i>Data type</i>	<i>Bytes</i>	<i>Range</i>	<i>Notes</i>
sfrb, sfrw	1		See the chapter <i>Extended keyword reference</i> .
char (by default)	1	0 to 255	Equivalent to unsigned char
char (using -c option)	1	-128 to 127	Equivalent to signed char
signed char	1	-128 to 127	
unsigned char	1	0 to 255	
short, int	2	-2^{15} to $2^{15}-1$	-32768 to 32767
unsigned short, unsigned int	2	0 to $2^{16}-1$	0 to 65535
long	4	-2^{31} to $2^{31}-1$	-2147483648 to 2147483647
unsigned long	4	0 to $2^{32}-1$	0 to 4294967295
pointer	2		See <i>Pointers</i> , page 75.
enum	1 to 4		See below.
float	4	$\pm 1.18\text{E}-38$ to $\pm 3.39\text{E}+38$	

<i>Data type</i>	<i>Bytes</i>	<i>Range</i>	<i>Notes</i>
double, long double	4	$\pm 1.18\text{E-}38$ to $\pm 3.39\text{E}+38$ (same as float)	

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

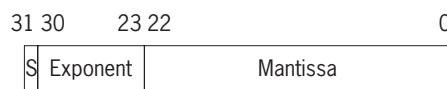
The char type is, by default, unsigned in the compiler, but the **Char is signed char** (-c) option allows you to make it signed. Note, however, that the library is compiled with char types as unsigned.

FLOATING POINT

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

4-byte floating-point format

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



The value of the number is:

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

Zero is represented by 4 bytes of zeros.

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

SPECIAL FUNCTION REGISTER VARIABLES

Special Function Register (sfr) variables are located in direct internal RAM locations. sfrb instructions have a range 0x00 to 0xFF and sfrw from 0x100 to 0x1FF. The sfrb type allows a symbolic name to be associated with a byte in this range. The register at that address can be addressed symbolically, but no memory space is allocated.

BITFIELDS

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

Bitfields with base type char, short, and long are extensions to ANSI C integer bitfields.

Bitfield variables are packed in elements of the specified type starting at the LSB position.

POINTERS

This section describes the MSP430 C Compiler's use of code pointers and data pointers.

CODE POINTERS

The size of code pointers is 2 bytes and can refer to memory in the range 0x0000 to 0xFFFF.

DATA POINTERS

The size of data pointers is 2 bytes, and can point to memory in the range 0x0000 to 0xFFFF.

EFFICIENT CODING

It is important to appreciate the limitations of the MSP430 architecture in order to avoid the use of inefficient language constructs. The following is a list of recommendations on how best to use the MSP430 C Compiler.

- ◆ Bitfield types should be used only to conserve data memory space as they execute slowly on the MSP430. Use a bit mask on unsigned char or unsigned int instead of bitfields. If you must use bitfields, use unsigned for efficiency.
- ◆ Variables that are not used outside their module should be declared as static, as this improves the possibility of temporarily keeping them in a register.
- ◆ Use unsigned data types, when possible. Sometimes unsigned operations execute more efficiently than the signed counterparts. This especially applies to division and modulo.

- ◆ Use ANSI prototypes. Function calls to ANSI functions are performed more efficiently than K&R-style functions; see the chapter *K&R and ANSI C language definitions*.
- ◆ The MSP430 operates most efficiently on 16-bit data types (eg short and unsigned short). In general, the use of 8-bit data types saves data space, but does not reduce code size. 32-bit data types have no direct support in the architecture and are therefore less efficient.
- ◆ Scalar auto variables are normally allocated in registers. Therefore use autos rather than statics whenever possible.
- ◆ The first two parameters of functions are passed in registers; see *Calling convention*, page 182. It is thus more efficient to pass arguments to a function in parameters than in static variables.
- ◆ Copying structs and unions are costly operations. Avoid run-time assignment of structs/unions, functions with struct/union parameters, and functions returning structs/unions. Prefer operating on pointers to structs/unions whenever possible.
- ◆ Non-scalar auto variables (structs, unions and arrays) with initial values results in run-time copying each time the function, in which they are declared, is called. For constant variables, this could be avoided by using the storage class “static const”.

GENERAL C LIBRARY DEFINITIONS

This chapter gives an introduction to the C library functions, and summarizes them according to header file.

INTRODUCTION

The IAR C Compiler package provides most of the important C library definitions that apply to PROM-based embedded systems. These are of three types:

- ◆ Standard C library definitions, for user programs. These are documented in this chapter.
- ◆ CSTARTUP, the single program module containing the start-up code.
- ◆ Intrinsic functions, allowing low-level use of MSP430 features.

LIBRARY OBJECT FILES

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.

The library object files are supplied having been compiled with the **Flag old-style functions** (`-gA`) option.

HEADER FILES

The user program gains access to library definitions through header files, which it incorporates using the `#include` directive. To avoid wasting time at compilation, the definitions are divided into a number of different header files each 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.

LIBRARY DEFINITIONS SUMMARY

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

CHARACTER HANDLING – `ctype.h`

<code>isalnum</code>	<code>int isalnum(int c)</code>	Letter or digit equality.
<code>isalpha</code>	<code>int isalpha(int c)</code>	Letter equality.
<code>iscntrl</code>	<code>int iscntrl(int c)</code>	Control code equality.
<code>isdigit</code>	<code>int isdigit(int c)</code>	Digit equality.
<code>isgraph</code>	<code>int isgraph(int c)</code>	Printable non-space character equality.
<code>islower</code>	<code>int islower(int c)</code>	Lower case equality.
<code>isprint</code>	<code>int isprint(int c)</code>	Printable character equality.
<code>ispunct</code>	<code>int ispunct(int c)</code>	Punctuation character equality.
<code>isspace</code>	<code>int isspace(int c)</code>	White-space character equality.
<code>isupper</code>	<code>int isupper(int c)</code>	Upper case equality.
<code>isxdigit</code>	<code>int isxdigit(int c)</code>	Hex digit equality.
<code>tolower</code>	<code>int tolower(int c)</code>	Converts to lower case.
<code>toupper</code>	<code>int toupper(int c)</code>	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* <i>format</i>, void <i>outputf</i> (char, void *), void *<i>sp</i>, va_list <i>ap</i>)</code>	Writes formatted data excluding floating-point numbers.
<code>_small_write</code>	<code>int _formatted_write (const char* <i>format</i>, void <i>outputf</i> (char, void *), void *<i>sp</i>, va_list <i>ap</i>)</code>	Small formatted data write routine.

MATHEMATICS – `math.h`

<code>acos</code>	<code>double acos(double <i>arg</i>)</code>	Arc cosine.
<code>asin</code>	<code>double asin(double <i>arg</i>)</code>	Arc sine.
<code>atan</code>	<code>double atan(double <i>arg</i>)</code>	Arc tangent.
<code>atan2</code>	<code>double atan2(double <i>arg1</i>, double <i>arg2</i>)</code>	Arc tangent with quadrant.
<code>ceil</code>	<code>double ceil(double <i>arg</i>)</code>	Smallest integer greater than or equal to <i>arg</i> .
<code>cos</code>	<code>double cos(double <i>arg</i>)</code>	Cosine.
<code>cosh</code>	<code>double cosh(double <i>arg</i>)</code>	Hyperbolic cosine.
<code>exp</code>	<code>double exp(double <i>arg</i>)</code>	Exponential.
<code>fabs</code>	<code>double fabs(double <i>arg</i>)</code>	Double-precision floating-point absolute.
<code>floor</code>	<code>double floor(double <i>arg</i>)</code>	Largest integer less than or equal.
<code>fmod</code>	<code>double fmod(double <i>arg1</i>, double <i>arg2</i>)</code>	Floating-point remainder.
<code>frexp</code>	<code>double frexp(double <i>arg1</i>, int *<i>arg2</i>)</code>	Splits a floating-point number into two parts.
<code>ldexp</code>	<code>double ldexp(double <i>arg1</i>, int <i>arg2</i>)</code>	Multiply by power of two.
<code>log</code>	<code>double log(double <i>arg</i>)</code>	Natural logarithm.
<code>log10</code>	<code>double log10(double <i>arg</i>)</code>	Base-10 logarithm.
<code>modf</code>	<code>double modf(double <i>value</i>, double *<i>iptr</i>)</code>	Fractional and integer parts.

<code>pow</code>	<code>double pow(double <i>arg1</i>, double <i>arg2</i>)</code>	Raises to the power.
<code>sin</code>	<code>double sin(double <i>arg</i>)</code>	Sine.
<code>sinh</code>	<code>double sinh(double <i>arg</i>)</code>	Hyperbolic sine.
<code>sqrt</code>	<code>double sqrt(double <i>arg</i>)</code>	Square root.
<code>tan</code>	<code>double tan(double <i>x</i>)</code>	Tangent.
<code>tanh</code>	<code>double tanh(double <i>arg</i>)</code>	Hyperbolic tangent.

NON-LOCAL JUMPS – `setjmp.h`

<code>longjmp</code>	<code>void longjmp(jmp_buf <i>env</i>, int <i>val</i>)</code>	Long jump.
<code>setjmp</code>	<code>int setjmp(jmp_buf <i>env</i>)</code>	Sets up a jump return point.

VARIABLE ARGUMENTS – `stdarg.h`

<code>va_arg</code>	<code>type va_arg(va_list <i>ap</i>, mode)</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>, parmN)</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 <i>value</i>)</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.

sprintf	int sprintf(char *s, const char *format,)	Writes formatted data to a string.
sscanf	int sscanf(const char *s, const char *format, ...)	Reads formatted data from a string.

GENERAL UTILITIES – stdlib.h

abort	void abort(void)	Terminates the program abnormally.
abs	int abs(int j)	Absolute value.
atof	double atof(const char *nptr)	Converts ASCII to double.
atoi	int atoi(const char *nptr)	Converts ASCII to int.
atol	long atol(const char *nptr)	Converts ASCII to long int.
bsearch	void *bsearch(const void *key, const void *base, size_t nmemb, size_t size, int (*compare) (const void *_key, const void *_base));	Makes a generic search in an array.
calloc	void *calloc(size_t nelem, size_t elsize)	Allocates memory for an array of objects.
div	div_t div(int numer, int denom)	Divide.
exit	void exit(int status)	Terminates the program.
free	void free(void *ptr)	Frees memory.
labs	long int labs(long int j)	Long absolute.
ldiv	ldiv_t ldiv(long int numer, long int denom)	Long division.
malloc	void *malloc(size_t size)	Allocates memory.
qsort	void qsort(const void *base, size_t nmemb, size_t size, int (*compare) (const void *_key, const void *_base));	Makes a generic sort of an array.
rand	int rand(void)	Random number.

<code>realloc</code>	<code>void *realloc(void *ptr, size_t size)</code>	Reallocates memory.
<code>srand</code>	<code>void srand(unsigned int seed)</code>	Sets random number sequence.
<code>strtod</code>	<code>double strtod(const char *nptr, char **endptr)</code>	Converts a string to double.
<code>strtol</code>	<code>long int strtol(const char *nptr, char **endptr, int base)</code>	Converts a string to a long integer.
<code>strtoul</code>	<code>unsigned long int strtoul (const char *nptr, char **endptr, base int)</code>	Converts a string to an unsigned long integer.

STRING HANDLING – `string.h`

<code>memchr</code>	<code>void *memchr(const void *s, int c, size_t n)</code>	Searches for a character in memory.
<code>memcmp</code>	<code>int memcmp(const void *s1, const void *s2, size_t n)</code>	Compares memory.
<code>memcpy</code>	<code>void *memcpy(void *s1, const void *s2, size_t n)</code>	Copies memory.
<code>memmove</code>	<code>void *memmove(void *s1, const void *s2, size_t n)</code>	Moves memory.
<code>memset</code>	<code>void *memset(void *s, int c, size_t n)</code>	Sets memory.
<code>strcat</code>	<code>char *strcat(char *s1, const char *s2)</code>	Concatenates strings.
<code>strchr</code>	<code>char *strchr(const char *s, int c)</code>	Searches for a character in a string.
<code>strcmp</code>	<code>int strcmp(const char *s1, const char *s2)</code>	Compares two strings.
<code>strcoll</code>	<code>int strcoll(const char *s1, const char *s2)</code>	Compares strings.
<code>strcpy</code>	<code>char *strcpy(char *s1, const char *s2)</code>	Copies string.

<code>strcspn</code>	<code>size_t strcspn(const char *s1, const char *s2)</code>	Spans excluded characters in string.
<code>strerror</code>	<code>char *strerror(int <i>errnum</i>)</code>	Gives an error message string.
<code>strlen</code>	<code>size_t strlen(const char *s)</code>	String length.
<code>strncat</code>	<code>char *strncat(char *s1, const char *s2, size_t <i>n</i>)</code>	Concatenates a specified number of characters with a string.
<code>strncmp</code>	<code>int strncmp(const char *s1, const char *s2, size_t <i>n</i>)</code>	Compares a specified number of characters with a string.
<code>strncpy</code>	<code>char *strncpy(char *s1, const char *s2, size_t <i>n</i>)</code>	Copies a specified number of characters from a string.
<code>strpbrk</code>	<code>char *strpbrk(const char *s1, const char *s2)</code>	Finds any one of specified characters in a string.
<code>strrchr</code>	<code>char *strrchr(const char *s, int <i>c</i>)</code>	Finds character from right of string.
<code>strspn</code>	<code>size_t strspn(const char *s1, const char *s2)</code>	Spans characters in a string.
<code>strstr</code>	<code>char *strstr(const char *s1, const char *s2)</code>	Searches for a substring.
<code>strtok</code>	<code>char *strtok(char *s1, const char *s2)</code>	Breaks a string into tokens.
<code>strxfrm</code>	<code>size_t strxfrm(char *s1, const char *s2, size_t <i>n</i>)</code>	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`

`assert`

`void assert(int expression)` 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 <i>nptr</i> 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 <i>nptr</i> 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.

DECLARATION

double 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.

DECLARATION

int 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: < 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

<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 nelem, size_t elsize)
```

PARAMETERS

<i>nelem</i>	The number of objects.
<i>elsize</i>	A value of type <code>size_t</code> 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 71.

ceil

math.h

Smallest integer greater than or equal to *arg*.**DECLARATION**

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

PARAMETERS

arg 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.

demon 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:

$$quot * denom + rem == numer$$

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*.

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.

DECLARATIONdouble 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.

DECLARATIONvoid 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	' '
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.

isxdigit

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:

$$quot * denom + rem == numer$$

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.

longjmp

setjmp.h

Long jump.

DECLARATION

```
void longjmp(jmp_buf env, int val)
```

PARAMETERS

env A struct of type jmp_buf holding the environment, set by setjmp.

val The int value to be returned by the corresponding setjmp.

RETURN VALUE

None.

DESCRIPTION

Restores the environment previously saved by setjmp. This causes program execution to continue as a return from the corresponding setjmp, returning the value *val*.

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, see *Heap size*, page 71.

memchr

string.h

Searches for a character in memory.

DECLARATION

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

PARAMETERS

s A pointer to an object.

c An `int` representing a character.

n 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

<i>s1</i>	A pointer to the first object.
<i>s2</i>	A pointer to the second object.
<i>n</i>	A value of type <code>size_t</code> 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*:

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

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

<i>format</i>	A pointer to the format string.
...	The optional values that are to be printed under the control of <i>format</i> .

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:

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

<i>Conversion</i>	<i>Result</i>
<code>n</code>	No output, but store the number of characters written so far in the integer pointed to by the next argument.
<code>%</code>	<code>%</code> 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; Δ 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)	006	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.

puts

RETURN VALUE

<i>Result</i>	<i>Value</i>
Successful	<i>value</i> .
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.

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.

DESCRIPTIONSorts 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` 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

<i>ptr</i>	A pointer to the start of the memory block.
<i>size</i>	A value of type <code>size_t</code> 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

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 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 67.

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 float.
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:

<i>Conversion</i>	<i>Meaning</i>
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.

<i>Conversion</i>	<i>Meaning</i>
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)
```

setjmp

setjmp.h

Sets up a jump return point.

DECLARATION

```
int setjmp(jmp_buf env)
```

PARAMETERS

env An object of type `jmp_buf` into which `setjmp` is to store the environment.

RETURN VALUE

Zero.

Execution of a corresponding `longjmp` causes execution to continue as if it was a return from `setjmp`, in which case the value of the `int` value given in the `longjmp` is returned.

DESCRIPTION

Saves the environment in *env* for later use by `longjmp`.

Note that `setjmp` must always be used in the same function or at a higher nesting level than the corresponding call to `longjmp`.

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

<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>...</i>	The optional values that are to be printed under the control of <i>format</i> .

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` 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 67.

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 successful conversions.
------------	---------------------------------------

Unsuccessful	-1 if the input was exhausted.
--------------	--------------------------------

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

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

=0	<i>s1</i> = <i>s2</i>
----	-----------------------

<0	<i>s1</i> < <i>s2</i>
----	-----------------------

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>
<code>>0</code>	<code>s1 > s2</code>
<code>=0</code>	<code>s1 = s2</code>
<code><0</code>	<code>s1 < s2</code>

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**

strerror is an implementation-defined function. In the MSP430 C Compiler it returns the following strings.

<i>errnum</i>	<i>String returned</i>
EZERO	"no error"
EDOM	"domain error"
ERANGE	"range error"
errnum < 0 errnum > 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	<i>s1</i> > <i>s2</i>
=0	<i>s1</i> = <i>s2</i>
<0	<i>s1</i> < <i>s2</i>

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 a 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

Result	Value
--------	-------

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

- nptr* A pointer to a string.
- endptr* A pointer to a pointer to a string.
- base* An *int* value specifying the base.

RETURN VALUE

<i>Result</i>	<i>Value</i>
Successful	The 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 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

- nptr A pointer to a string.
- endptr A pointer to a pointer to a string.
- base An int value specifying the base.

RETURN VALUE

Result	Value
Successful	The unsigned long int result of converting the ASCII representation of an integer constant in the string pointed to by nptr, leaving endptr pointing to the first character after the constant.
Unsuccessful	Zero, leaving endptr 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

s1 Return location of the transformed string.

s2 String to transform.

n Maximum number of characters to be placed in *s1*.

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.

DECLARATION

```
type 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.

DECLARATION

```
void 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

```
char *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`.

_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 scanf 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 <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

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`.

`_medium_write`

icc1but1.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 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

A reduced version of `_formatted_write` which is half the size, but does not support floating-point numbers.

For further information see `_formatted_write`.

_small_write

icclbutl.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

A small version of `_formatted_write` which is about a quarter of the size, and uses only about 15 bytes of RAM.

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`.

LANGUAGE EXTENSIONS

This chapter summarizes the extensions provided in the MSP430 C Compiler to support specific features of the MSP430 microprocessor.

INTRODUCTION

The extensions are provided in three ways:

- ◆ As extended keywords. By default, the compiler conforms to the ANSI specifications and MSP430 extensions are not available. The command line option `-e` makes the extended keywords available, and hence reserves them so that they cannot be used as variable names.
- ◆ As `#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.
- ◆ As intrinsic functions. These provide direct access to very low-level processor details. To enable intrinsic functions include the file `in430.h`.

EXTENDED KEYWORDS SUMMARY

The extended keywords provide the following facilities:

I/O ACCESS

The program may access the MSP430 I/O system using the following data types:

`sfrb, sfrw.`

NON-VOLATILE RAM

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

`no_init.`

FUNCTIONS

To override the default mechanism by which the compiler calls a function use one of the following function modifiers:

`interrupt`, `monitor`.

#PRAGMA DIRECTIVE SUMMARY

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

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

The following categories of `#pragma` functions are available:

BITFIELD ORIENTATION

```
#pragma bitfield=default  
#pragma bitfield=reversed
```

CODE SEGMENT

```
#pragma codeseg(seg_name)
```

EXTENSION CONTROL

```
#pragma language=default  
#pragma language=extended
```

FUNCTION ATTRIBUTE

```
#pragma function=default  
#pragma function=interrupt  
#pragma function=monitor
```

MEMORY USAGE

```
#pragma memory=constseg(seg-name)[:type]  
#pragma memory=dataseg(seg-name)[:type]  
#pragma memory=default  
#pragma memory=no_init
```

WARNING MESSAGE CONTROL

```
#pragma warnings=default
#pragma warnings=off
#pragma warnings=on
```

PREDEFINED SYMBOLS SUMMARY	Predefined symbols allow inspection of the compile-time environment.	
	<i>Function</i>	<i>Description</i>
	__DATE__	Current date in Mmm dd yyyy format.
	__FILE__	Current source filename.
	__IAR_SYSTEMS_ICC	IAR C compiler identifier.
	__LINE__	Current source line number.
	__STDC__	ANSI C compiler identifier.
	__TID__	Target identifier.
	__TIME__	Current time in hh:mm:ss format.
	__VER__	Returns the version number as an int.

INTRINSIC FUNCTION SUMMARY	Intrinsic functions allow very low-level control of the MSP430 microprocessor. To use them in a C application, include the header file <code>in430.h</code> . 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 MSP430 processor.	
	<i>Intrinsic</i>	<i>Description</i>
	_args\$	Returns an array of the parameters to a function.
	_argt\$	Returns the type of parameter.
	_NOP	The nop instruction.
	_EINT	Enables interrupts.

<i>Intrinsic</i>	<i>Description</i>
<code>_DINT</code>	Disables interrupts.
<code>_BIS_SR</code>	Set a bit in the status register.
<code>_BIC_SR</code>	Clears a bit in the status register.
<code>_OPC</code>	Inserts a DW const directive.

OTHER EXTENSIONS

\$ CHARACTER

The character `$` has been added to the set of valid characters in identifiers for compatibility with DEC/VMS C.

USE OF SIZEOF AT COMPILE TIME

The ANSI-specified restriction that the `sizeof` operator cannot be used in `#if` and `#elif` expressions has been eliminated.

EXTENDED KEYWORD REFERENCE

This chapter describes the extended keywords in alphabetical order.

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

<i>Parameter</i>	<i>What it means</i>
<i>storage-class</i>	Denotes an optional keyword <code>extern</code> or <code>static</code> .
<i>declarator</i>	Denotes a standard C variable or function declarator.

interrupt

Declare interrupt function.

SYNTAX

storage-class `interrupt` *function-declarator*
storage-class `interrupt` [*vector*] *function-declarator*

PARAMETERS

function-declarator A void function declarator having no arguments.

[*vector*] A square-bracketed constant expression yielding the vector address.

DESCRIPTION

The `interrupt` keyword declares a function that is called upon a processor interrupt.

The function must be void and have no arguments.

If a vector is specified, the address of the function is inserted in that vector. If no vector is specified, the user must provide an appropriate entry in the vector table (preferably placed in the `cstartup` module) for the interrupt function.

EXAMPLES

Several include files are provided that define specific interrupt functions; see *Run-time library*, page 66. To use a predefined interrupt define it with a statement:

```
interrupt void name(void)
{ }
```

where *name* is the name of the interrupt function.

```
interrupt [0x18] void UART_handler(void)
{
    if(TCCTL & 4)
        receive();
    else
        transmit();
}
```

The vector address (0x18 in this example) is offset into the INTVEC segment (0xFFE0). This example will place the vector in location 0xFFFF8.

monitor

Makes a function atomic.

SYNTAX

storage-class **monitor** *function-declarator*

DESCRIPTION

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 **monitor** is equivalent to a normal function in all other respects.

EXAMPLES

The example `got_flag` below disables interrupts while a flag is tested. If the flag is not set the function sets it. Interrupts are set to their previous state when the functions exits.

```

char printer_free;           /* printer-free
                             semaphore */
monitor int got_flag(char *flag) /* With no interruption */
{
    if (!*flag)              /* test if available */
    {
        return (*flag = 1);  /* yes - take */
    }
    return (0);              /* no - do not take */
}

void f(void)
{
    if (got_flag(&printer_free)) /* act only if
                                printer is free */
        .... action code ....
}

```

no_init

Type modifier for non-volatile variables.

SYNTAX

storage-class no_init *declarator*

DESCRIPTION

By default, the compiler places variables in main, volatile RAM and initializes them on start-up. The `no_init` type modifier causes the compiler to place the variable in non-volatile RAM (or EEPROM) and not to initialize it on start-up.

`no_init` variable declarations may not include initializers.

If non-volatile memory is used, it is essential for the program to be linked to refer to the non-volatile RAM area, which must be specified to be in the address range 0x0000 to 0xFFFF. For details, see *Non-volatile RAM*, page 66.

The practical usable range, however, is slightly smaller (0x0200 to 0xFFDF).

EXAMPLES

The examples below show valid and invalid uses of the `no_init` modifier.

```
no_init int settings[50];    /* array of non-volatile
                             settings */
((no_init far i ;))          /* conflicting type
                             modifiers - invalid */
no_init int i = 1 ;          /* initializer included -
                             invalid */
```

sfrb

Declare object of one-byte I/O data type.

SYNTAX

`sfrb identifier = constant-expression`

DESCRIPTION

`sfrb` denotes an I/O register which:

- ◆ Is equivalent to unsigned char.
- ◆ Can only be directly addressable; ie the `&` operator cannot be used.
- ◆ Resides in a fixed location anywhere in the address range 0x00 to 0xFF.

The value of an `sfrb` variable is the contents of the SFR register at the address *constant-expression*. All operators that apply to integral types except the unary `&` (address) operator may be applied to `sfrb` variables.

EXAMPLES

```
sfrb P0OUT = 0x11;           /* Defines P0OUT */
sfrb P0IN = 0x10;
void func()
{
    P0OUT = 4;                /* Sets entire variable
                              P0OUT = 00000100 */
    P0OUT |= 4;               /* Sets one bit
                              P0OUT = XXXXX1XX */
}
```

```

POOUT &= ~8                /* Clears one bit
                           POOUT = XXXX0XXX */
if (POIN & 2) printf("ON"); /* Read entire POIN and
                           mask bit 2 */
}

```

sfrw

Declare object of two-byte I/O data type.

SYNTAX

sfrw identifier = constant-expression

DESCRIPTION

sfrw denotes an I/O register which:

- ◆ Is equivalent to unsigned short.
- ◆ Can only be directly addressable; ie the & operator cannot be used.
- ◆ Resides at a fixed location anywhere in the address range 0x100 to 0x1FF.

The value of an sfrw variable is the contents of the SFR register at the address *constant-expression*. All operators that apply to integral types except the unary & (address) operator may be applied to sfrw variables.

EXAMPLES

```

sfrw    WDTCTL = 0x120;        /* Defines watchdog time
                               control register */

void func(void)
{
    WDTCTL = 0x5A08;           /* Set watchdog time
                               control register */
}

```

#PRAGMA DIRECTIVE REFERENCE

This chapter describes the `#pragma` directives in alphabetical order.

bitfields = default

Restores default order of storage of bitfields.

SYNTAX

```
#pragma bitfields = default
```

DESCRIPTION

Causes the compiler to allocate bitfields in its normal order. See `bitfields=reversed`.

bitfields = reversed

Reverses order of storage of bitfields.

SYNTAX

```
#pragma bitfields=reversed
```

DESCRIPTION

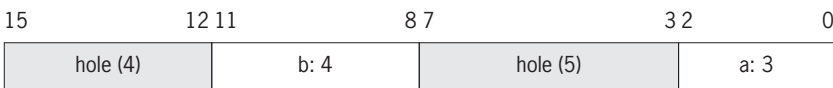
Causes the compiler to allocate bitfields starting at the most significant bit of the field, instead of at the least significant bit. The ANSI standard allows the storage order to be implementation dependent, so you can use this keyword to avoid portability problems.

EXAMPLES

The default layout of the following structure in memory is given in the diagram below:

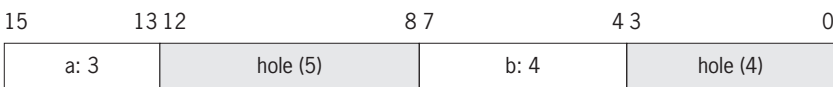
```
struct
{
    short a:3;      /* a is 3 bits */
    short :5;       /* this reserves a hole of 5 bits */
}
```

```
short b:4;      /* b is 4 bits */
} bits;        /* bits is 16 bits */
```



For comparison, the following structure has the layout shown in the diagram below:

```
#pragma bitfields=reversed
struct
{
short a:3;      /* a is 3 bits */
short :5;      /* this reserves a hole of 5 bits */
short b:4;      /* b is 4 bits */
} bits;        /* bits is 16 bits */
```



codeseg

Sets the code segment name.

SYNTAX

```
#pragma codeseg( seg_name)
```

where *seg_name* specifies the segment name, which must not conflict with data segments.

DESCRIPTION

This directive places subsequent code in the named segment and is equivalent to using the -R option. The pragma can only be executed once by the compiler.

EXAMPLES

The following example defines the code segment as ROM:

```
#pragma codeseg(ROM)
```

function = default

Restores function definitions to the default type.

SYNTAX

```
#pragma function=default
```

DESCRIPTION

Cancels `function=interrupt` and `function=monitor` directives.

EXAMPLES

The example below specifies that an external function `f1` is an interrupt function, while `f3` is a normal function.

```
#pragma function=interrupt
extern void f1();
#pragma function=default
extern int f3();          /* Default function type */
```

function = interrupt

Makes function definitions interrupt.

SYNTAX

```
#pragma function=interrupt
```

DESCRIPTION

This directive makes subsequent function definitions of interrupt type. It is an alternative to the function attribute `interrupt`.

Note that `#pragma function=interrupt` does not offer a vector option.

EXAMPLES

The example below shows an interrupt function `process_int` (the address of this function must be placed into the `INTVEC` table).

```
#pragma function=interrupt
void process_int()      /* an interrupt function */
{
    ...
}
```

function = monitor

```
}  
#pragma function=default
```

function = monitor

Makes function definitions atomic (non-interruptible).

SYNTAX

```
#pragma function=monitor
```

DESCRIPTION

Makes subsequent function definitions of monitor type. It is an alternative to the function attribute monitor.

EXAMPLES

The function f2 below will execute with interrupts temporarily disabled.

```
#pragma function=monitor  
void f2()          /* Will make f2 a monitor function */  
{  
    "..."  
}  
#pragma function=default
```

language = default

Restores availability of extended keywords to default.

SYNTAX

```
#pragma language=default
```

DESCRIPTION

Returns extended keyword availability to the default set by the C compiler -e option. See language=extended.

EXAMPLES

See the example language=extended below.

language = extended

Makes extended keywords available.

SYNTAX

```
#pragma language=extended
```

DESCRIPTION

Makes the extended keywords available regardless of the state of the C compiler -e option; see *Enable language extensions (-e)*, page 40.

EXAMPLE

In the example below, the extended language modifier is specified to disable interrupts.

```
#pragma language=extended
no_init int sys_para;          /* put in non-volatile RAM */
#pragma language=default
int mycount;
```

memory = constseg

Directs constants to the named segment by default.

SYNTAX

```
#pragma memory=constseg(seg_name)
```

DESCRIPTION

Directs constants to the named segment by default. Subsequent declarations implicitly get storage class const. Placement can be overridden by the keywords no_init and const.

The segment must not be one of the compiler's reserved segment names.

EXAMPLE

The example below places the constant array arr into the ROM segment TABLE.

```
#pragma memory=constseg(TABLE)
char arr[] = {6, 9, 2, -5, 0};
#pragma memory = default
```

memory = dataseg

Directs variables to the named segment by default.

SYNTAX

```
#pragma memory=dataseg(seg_name)
```

DESCRIPTION

Directs variables to the named segment by default. The default is overridden by the keywords `no_init` and `const`.

If omitted, variables will be placed in UDATA0 (uninitialized variables) or IDATA0 (initialized variables).

No initial values may be supplied in the variable definitions. Up to 10 different alternate data segments can be defined in any given module. You can switch to any previously defined data segment name at any point in the program.

EXAMPLE

The example below places three variables into the read/write area called USART.

```
#pragma memory=dataseg(USART)
char USART_data;           /* offset 0 */
char USART_control;        /* offset 1 */
int USART_rate;            /* offset 2, 3 */
#pragma memory = default
```

memory = default

Restores memory allocation of objects to the default area.

SYNTAX

```
#pragma memory=default
```

DESCRIPTION

Restores memory allocation of objects to the default area.

Subsequently uninitialized data is allocated in UDATA0, and initialized data in IDATA0.

memory = no_init

Directs variables to the NO_INIT segment by default.

SYNTAX

```
#pragma memory=no_init
```

DESCRIPTION

Directs variables to the NO_INIT segment, so that they will not be initialized and will reside in non-volatile RAM. It is an alternative to the memory attribute `no_init`. The default may be overridden by the keyword `const`.

The NO_INIT segment must be linked to coincide with the physical address of non-volatile RAM; see the chapter *Configuration* for details.

EXAMPLES

The example below places the variable `buffer` into non-initialized memory. Variables `i` and `j` are placed into the DATA area.

```
#pragma memory=no_init
char buffer[1000];      /* in uninitialized memory */
#pragma memory=default
int i,j;                /* default memory type */
```

Note that a non-default memory `#pragma` will generate error messages if function declarators are encountered. Local variables and parameters cannot reside in any other segment than their default segment, the stack.

warnings = default

Restores compiler warning output to default state.

SYNTAX

```
#pragma warnings=default
```

DESCRIPTION

Returns the output of compiler warning messages to the default set by the C compiler `-w` option. See `#pragma warnings=on` and `#pragma warnings=off`.

warnings = off

warnings = off

Turns off output of compiler warnings.

SYNTAX

```
#pragma warnings=off
```

DESCRIPTION

Disables output of compiler warning messages regardless of the state of the C compiler -w option; see *Disable warnings (-w)*, page 41.

warnings = on

Turns on output of compiler warnings.

SYNTAX

```
#pragma warnings=on
```

DESCRIPTION

Enables output of compiler warning messages regardless of the state of the C compiler -w option; see *Disable warnings (-w)*, page 41.

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 1 is returned. This symbol can be tested with `#ifdef` to detect being compiled by an IAR Systems C Compiler.

__LINE__

__LINE__

Current source line number.

SYNTAX

__LINE__

DESCRIPTION

The current line number of the file currently being compiled is returned as an `int`.

__STDC__

IAR C compiler identifier.

SYNTAX

__STDC__

DESCRIPTION

The number 1 is returned. This symbol can be tested with `#ifdef` to detect being compiled by an ANSI C compiler.

__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 `-v` option, and the value corresponding to the `-m` option:

31	16	15	14	8	7	4	3	0
(not used)		Intrinsic support	Target_IDENT, unique to each target processor	-v option value, if supported		-m option value, if supported		

For the MSP430 the `Target_IDENT` is 43.

The `__TID__` value is constructed as:

```
(0x8000 | (t << 8) | (v << 4) | m)
```

You can extract the values as follows:

```
f = (__TID__) & 0x8000;  
t = (__TID__ >> 8) & 0x7F;  
v = (__TID__ >> 4) & 0xF;  
m = __TID__ & 0x0F;
```

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`.

The highest bit 0x8000 is set in the MSP430 C Compiler to indicate that the compiler recognizes intrinsic functions. This may affect how you write header files.

__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 `int`.

EXAMPLE

The example below prints a message for version 3.34.

```
#if __VER__ == 334
#message "Compiler version 3.34"
#endif
```

INTRINSIC FUNCTION REFERENCE

This chapter gives reference information about the intrinsic functions. To use the intrinsic functions include the header file `in430.h`.

`_args$`

Returns an array of the parameters to a function.

SYNTAX

`_args$`

DESCRIPTION

`_args$` is a reserved word that returns a char array (`char *`) containing a list of descriptions of the formal parameters of the current function:

<i>Offset</i>	<i>Contents</i>
0	Parameter 1 type in <code>_argt\$</code> format.
1	Parameter 1 size in bytes.
2	Parameter 2 type in <code>_argt\$</code> format.
3	Parameter 2 size in bytes.
2n-2	Parameter n type in <code>_argt\$</code> format.
2n-1	Parameter n size in bytes.
2n	<code>\0</code>

Sizes greater than 127 are reported as 127.

`_args$` may be used only inside function definitions. For an example of the use of `_args$`, see the file `stdarg.h`.

If a variable length (`varargs`) parameter list was specified then the parameter list is deemed to terminate at the final explicit parameter; you cannot easily determine the types or sizes of the optional parameters.

_argt\$

Returns the type of the parameter.

SYNTAX

`_argt$(v)`

DESCRIPTION

The returned values and their corresponding meanings are shown in the following table.

<i>Value</i>	<i>Type</i>
1	unsigned char
2	char
3	unsigned short
4	short
5	unsigned int
6	int
7	unsigned long
8	long
9	float
10	double
11	long double
12	pointer/address
13	union
14	struct

EXAMPLE

The example below uses `_argt$` and tests for integer or long parameter types.

```
switch (_argt$(i))
{
    case 6:
        printf("int %d\n", i);
        break;
    case 8:
        printf("long %ld\n", i);
        break;
    default:
```

_BIC_SR

Clears bits in the status register.

SYNTAX

unsigned short _BIC_SR(unsigned short *mask*)

DESCRIPTION

Clears bits using a BIC *mask*, SR instruction. Returns the contents of SR prior to update.

EXAMPLE

```
/* disable interrupts */  
old_SR=_BIC_SR(0x08);  
/* restore interrupts */  
_BIS_SR(old_SR);
```

_BIS_SR

Sets bits in the status register.

SYNTAX

unsigned short _BIS_SR(unsigned short *mask*)

DESCRIPTION

Sets bits using a BIS *mask*, SR instruction. Returns the contents of SR prior to update.

EXAMPLE

```
/* enter low power mode LPM3 */  
_BIS_SR(0xC0);
```

_DINT

Disables interrupts.

SYNTAX

_DINT(void)

`_EINT`

DESCRIPTION

Disables interrupts using a DINT instruction.

`_EINT`

Enables interrupts.

SYNTAX

`_EINT(void)`

DESCRIPTION

Enables interrupts using an EINT instruction.

`_NOP`

Executes the NOP instruction.

SYNTAX

`_NOP(void)`

DESCRIPTION

Executes the NOP instruction.

`_OPC`

Inserts a DW constant directive.

SYNTAX

`_OPC(const unsigned char)`

DESCRIPTION

Inserts a DW constant.

ASSEMBLY LANGUAGE INTERFACE

The MSP430 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 i;
}
void f(void)
{
    foo(4,5);     /* Call to foo */
}
```

This program is then compiled as follows:

```
icc430 shell -A -q -L
```

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

The result is the assembler source `shell.s43` containing the declarations, function call, function return, variable accesses, and a listing file `shell.lst`.

The following sections describe the interface in detail.

CALLING CONVENTION REGISTER USAGE

The compiler uses two groups of processor registers.

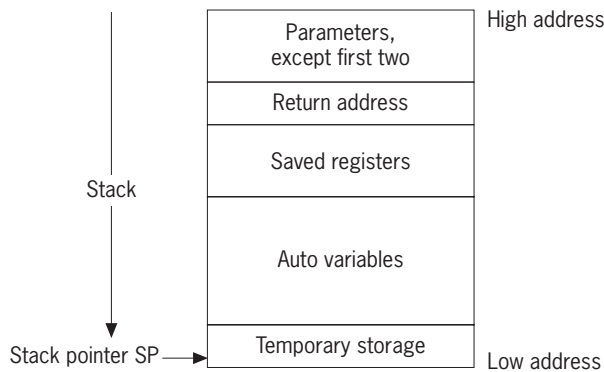
The scratch registers R12 to R15 are used for parameter passing and hence are not normally preserved across the call.

The other general purpose registers, R4 to R11, are mainly used for register variables and temporary results, and must be preserved across a call.

Note that the `-ur` option prevents the compiler from using registers R4 and/or R5.

STACK FRAMES AND PARAMETER PASSING

Each function call creates a stack frame as follows:



Parameters are passed to an assembler routine in a right to left order. The leftmost two parameters are passed in registers unless they are a struct or union, in which case they are also passed on the stack. The remaining parameters are always passed on the stack. Take as an

example the following call:

```
f(w,x,y,z)
```

Since the arguments are dealt with in a right to left order, z will be loaded onto the stack first, followed by y. x will either be in R14, R15:R14, or on the stack, depending on its type, as will w. The result is returned in R12 (or R13:R12 for a 32 bit type) and in a special area pointed to by R12 if it is a struct/union.

This is summarized in the following table:

<i>Argument</i>	<i>< 32 bit type</i>	<i>32 bit type</i>	<i>struct/union</i>
4th (z)	On the stack	On the stack	On the stack
3rd (y)	On the stack	On the stack	On the stack
2nd (x)	R14	R15:R14	On the stack
1st (w)	R12	R13:R12	On the stack
Result	R12	R13:R12	Special area

INTERRUPT FUNCTIONS

Interrupt functions preserve the scratch registers and SR (status register) as well as registers R4 to R11.

The status register is saved as part of the interrupt calling process. Any registers used by the routine are then saved using PUSH Rxx instructions. On exit these registers are recovered using POP Rxx instructions and the RTI instruction is used to reload the status register and return from the interrupt.

MONITOR FUNCTIONS

In the case of a monitor instruction, on entry the compiler saves the processor status using PUSH SR, and disables interrupts using DINT. On exit it uses RTI to reload the status register (and the previous interrupt enable flag) and to return.

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 may allocate it in one or more of the following ways:

- ◆ On the hardware stack.
- ◆ In static workspace, provided of course that the routine is not required to be simultaneously re-usable (“re-entrant”).

Functions can always use R12 to R15 without saving them, and R6 to R11 provided they are pushed before use. R4 and R5 should not be used for ROM monitor compatible code.

If the C code is compiled with `-ur45`, but the application will not be run in the ROM monitor, then it is possible to use R4 and R5 in the assembler routine without saving them, since the C code will never use 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 scratch registers R12–R15.
- ◆ The routine must exit using `RTI`.
- ◆ The routine must treat all flags as undefined.

DEFINING INTERRUPT VECTORS

As an alternative to defining a C interrupt function in assembly language as described above, the user is free to assemble an interrupt routine and install it directly in the interrupt vector.

The interrupt vectors are located in the INTVEC segment.

SEGMENT REFERENCE

The MSP430 C Compiler places code and data into named segments which are referred to by XLINK. Details of the segments are required for programming assembly language modules, and are also useful when interpreting the assembly language output of the compiler.

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

- ◆ The name of the segment.
- ◆ A brief description of the contents.
- ◆ Whether the segment is read/write or read-only.
- ◆ Whether the segment may be accessed from the assembly language (assembly-accessible) or from the compiler only.
- ◆ A fuller description of the segment contents and use.

MEMORY MAP DIAGRAMS

The diagrams on the following pages show the MSP430 memory map and the allocation of segments within each memory area.

SEGMENT REFERENCE

FFFF	INTVEC	Vectors
FFEO		
FFDF	CCSTR	String literal initializers when the -y option is used
	CSTR	C string literals
	CONST	Used for storing const objects
	CDATA0	Variable initializers for variable in IDATA0
8000	CODE	Holds the user program
	CODE	
7FFF	CSTACK	Stack
	NO_INIT	Holds no-init variables
	ECSTR	Writable copies of string literals
	UDATA0	Variables which are not specifically initialized
0000	IDATA0	Holds variables which are initialized from CDATA0
	DATA	

CCSTR

String literals.

TYPE

Read-only.

DESCRIPTION

Assembly-accessible.

Holds C string literals. This segment is copied to ECSTR at startup. For more information refer to the C compiler **Writable strings** (-y) option; see *Code generation*, page 39. See also *CSTR*, page 190, and *ECSTR*, page 191.

CDATA0

Initialization constants for variables located in IDATA0 by CSTARTUP.

TYPE

Read-only.

DESCRIPTION

Assembly-accessible.

CSTARTUP copies initialization values from this segment to the IDATA0 segments.

CODE

Code.

TYPE

Read-only.

DESCRIPTION

Assembly-accessible.

Holds user program code and various library routines. 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 184.

CONST

Constants.

TYPE

Read-only.

DESCRIPTION

Assembly-accessible.

Used for storing const objects. Can be used in assembly language routines for declaring constant data.

CSTACK

Stack.

TYPE

Read/write.

DESCRIPTION

Assembly-accessible.

Holds the internal stack.

This segment and length is normally defined in the XLINK file by the command:

$-Z(DATA)CSTACK + nn = start$

where nn is the length and $start$ is the location.

CSTR

String literals.

TYPE

Read only.

DESCRIPTION

Assembly-accessible.

Holds C string literals when the C compiler **Writable strings** (-y) option is not active (default). For more information refer to *Writable strings, constants* (-y), page 40. See also *CCSTR*, page 189, and *ECSTR*, page 191.

ECSTR

Writable copies of string literals.

TYPE

Read/write.

DESCRIPTION

Assembly-accessible.

Holds C string literals. For more information refer to *Writable strings, constants* (-y), page 40. See also *CCSTR*, page 189, and *CSTR*, page 190.

IDATA0

Initialized static data for variables.

TYPE

Read-write.

DESCRIPTION

Assembly-accessible.

Holds static variables in internal data memory that are automatically initialized from CDATA0 in `cstartup.s`⁴³. See also CDATA0 above.

INTVEC

Interrupt vectors.

TYPE

Read-only.

DESCRIPTION

Assembly-accessible.

Holds the interrupt vector table generated by the use of the `interrupt` extended keyword (which can also be used for user-written interrupt vector table entries). It must be located at address `0xFFE0`.

NO_INIT

Non-volatile variables.

TYPE

Read/write.

DESCRIPTION

Assembly-accessible.

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

UDATA0

Uninitialized static data.

TYPE

Read/write.

DESCRIPTION

Assembly-accessible.

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

K&R AND ANSI C LANGUAGE DEFINITIONS

This chapter describes the differences between the K&R description of the C language and the ANSI standard.

INTRODUCTION

There are two major standard C language definitions:

- ◆ Kernighan & Richie, commonly abbreviated to K&R.

This is the original definition by the authors of the C language, and is described in their book *The C Programming Language*.

- ◆ ANSI.

The ANSI definition is a development of the original K&R definition. It adds facilities that enhance portability and parameter checking, and removes a small number of redundant keywords. The IAR Systems C Compiler follows the ANSI approved standard X3.159-1989.

Both standards are described in depth in the latest edition of *The C Programming Language* by Kernighan & Richie. This chapter summarizes the differences between the standards, and is particularly useful to programmers who are familiar with K&R C but would like to use the new ANSI facilities.

DEFINITIONS

ENTRY KEYWORD

In ANSI C the entry keyword is removed, so allowing entry to be a user-defined symbol.

CONST KEYWORD

ANSI C adds `const`, an attribute indicating that a declared object is unmodifiable and hence may be compiled into a read-only memory segment. For example:

```
const int i;           /* constant int */
const int *ip;         /* variable pointer to
                        constant int */
```

```
int *const ip;                /* constant pointer to
                              variable int */
typedef struct                /* define the struct
                              'cmd_entry' */
{
    char *command;
    void (*function)(void);
}
cmd_entry
const cmd_entry table[] =    /* declare a constant object
                              of type 'cmd_entry' */
{
    "help", do_help,
    "reset", do_reset,
    "quit", do_quit
};
```

VOLATILE KEYWORD

ANSI C adds `volatile`, an attribute indicating that the object may be modified by hardware and hence any access should not be removed by optimization.

SIGNED KEYWORD

ANSI C adds `signed`, an attribute indicating that an integer type is signed. It is the counterpart of `unsigned` and can be used before any integer type-specifier.

VOID KEYWORD

ANSI C adds `void`, a type-specifier that can be used to declare function return values, function parameters, and generic pointers. For example:

```
void f();                    /* a function without return
                              value */
type_spec f(void);          /* a function with no parameters
                              */
void *p;                    /* a generic pointer which can be
                              /* cast to any other pointer and
                              is assignment-compatible with any
                              pointer type */
```

ENUM KEYWORD

ANSI C adds enum, a keyword that conveniently defines successive named integer constants with successive values. For example:

```
enum {zero,one,two,step=6,seven,eight};
```

DATA TYPES

In ANSI C the complete set of basic data types is:

```
{unsigned | signed} char
{unsigned | signed} int
{unsigned | signed} short
{unsigned | signed} long
float
double
long double
*                      /* Pointer */
```

FUNCTION DEFINITION PARAMETERS

In K&R C, function parameters are declared by conventional declaration statements before the body of the function. In ANSI C, each parameter in the parameter list is preceded by its type identifiers. For example:

<i>K&R</i>	<i>ANSI</i>
long int g(s)	long int g (char * s)
char * s;	
{	{

The arguments of ANSI-type functions are always type-checked. The IAR Systems C Compiler checks the arguments of K&R-type functions only if the **Global strict type check** (-g) option is used.

FUNCTION DECLARATIONS

In K&R C, function declarations do not include parameters. In ANSI C they do. For example:

<i>Type</i>	<i>Example</i>
K&R	<code>extern int f();</code>
ANSI (named form)	<code>extern int f(long int val);</code>
ANSI (unnamed form)	<code>extern int f(long int);</code>

In the K&R case, a call to the function via the declaration cannot have its parameter types checked, and if there is a parameter-type mismatch, the call will fail.

In the ANSI C case, the types of function arguments are checked against those of the parameters in the declaration. If necessary, a parameter of a function call is cast to the type of the parameter in the declaration, in the same way as an argument to an assignment operator might be. Parameter names are optional in the declaration.

ANSI also specifies that to denote a variable number of arguments, an ellipsis (three dots) is included as a final formal parameter.

If external or forward references to ANSI-type functions are used, a function declaration should appear before the call. It is unsafe to mix ANSI and K&R type declarations since they are not compatible for promoted parameters (`char` or `float`).

A function that takes a variable number of arguments must be declared prior to the call. It is unsafe to call such a function, unless it has been declared using a prototype containing an ellipsis. The header file `stdio.h` contains declarations of the standard functions `printf`, `scanf`, etc.

Note that in the IAR Systems C Compiler, the `-g` option will find all compatibility problems among function calls and declarations, including between modules.

HEXADECIMAL STRING CONSTANTS

ANSI allows hexadecimal constants denoted by backslash followed by x and any number of hexadecimal digits. For example:

```
#define Escape_C "\x1b\x43" /* Escape 'C' \0 */
```

\x43 represents ASCII C which, if included directly, would be interpreted as part of the hexadecimal constant.

STRUCTURE AND UNION ASSIGNMENTS

In K&R C, functions and the assignment operator may have arguments that are pointers to struct or union objects, but not struct or union objects themselves.

ANSI C allows functions and the assignment operator to have arguments that are struct or union objects, or pointers to them. Functions may also return structures or unions:

```
struct s a,b;                /* struct s declared earlier
                               */
struct s f(struct s parm);    /* declare function
                               accepting and returning
                               structs */
a = f(b);                    /* call it */
```

To increase the usability of structures further, ANSI allows auto structures to be initialized.

SHARED VARIABLE OBJECTS

Various C compilers differ in their handling of variable objects shared among modules. The IAR Systems C Compiler uses the scheme called *Strict REF/DEF*, recommended in the ANSI supplementary document *Rationale For C*. It requires that all modules except one use the keyword *extern* before the variable declaration. For example:

<i>Module #1</i>	<i>Module #2</i>	<i>Module #3</i>
int i;	extern int i;	extern int i;
int j=4;	extern int j;	extern int j;

#elif

ANSI C's new #elif directive allows more compact nested else-if structures.

```
#elif expression
```

```
...
```

is equivalent to:

```
#else
```

```
#if expression
```

```
...
```

```
#endif
```

#error

The #error directive is provided for use in conjunction with conditional compilation. When the #error directive is found, the compiler issues an error message and terminates.

DIAGNOSTICS

The diagnostic error and warning messages fall into six categories:

- ◆ Command line error messages.
- ◆ Compilation error messages.
- ◆ Compilation warning messages.
- ◆ Compilation fatal error messages.
- ◆ Compilation memory overflow message.
- ◆ Compilation internal error messages.

COMMAND LINE ERROR MESSAGES

Command line errors occur when the compiler finds a fault in the parameters given on the command line. In this case, the compiler issues a self-explanatory message.

COMPILATION ERROR MESSAGES

Compilation error messages are produced when the compiler has found a construct which clearly violates the C language rules, such that code cannot be produced.

The IAR C Compiler is more strict on compatibility issues than many other C compilers. In particular pointers and integers are considered as incompatible when not explicitly casted.

COMPILATION WARNING MESSAGES

Compilation warning messages are 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.

COMPILATION FATAL ERROR MESSAGES

Compilation fatal error messages are produced when the compiler has found a condition that not only prevents code generation, but which makes further processing of the source not meaningful. After the message has been issued, compilation terminates. Compilation fatal error messages are described in *Compilation error messages* in this chapter, and marked as fatal.

COMPILATION MEMORY OVERFLOW MESSAGE

When the compiler runs out of memory, it issues the special message:

```
* * * C O M P I L E R   O U T   O F   M E M O R Y * * *  
      Dynamic memory used: nnnnnn bytes
```

If this error occurs, the cure is either to add system memory or to split source files into smaller modules. Also note that the following options cause the compiler to use more memory (but not -rn):

<i>Option</i>	<i>Command line</i>
Insert mnemonics.	-q
Cross-reference.	-x
Assembly output to prefixed filename.	-A
Generate PROMable code.	-P
Generate debug information.	-r

See the *MSP430 Command Line Interface Guide* for more information.

COMPILATION INTERNAL ERROR MESSAGES

A compiler internal error message indicates that there has been a serious and unexpected failure due to a fault in the compiler itself, for example, the failure of an internal consistency check. After issuing a self-explanatory message, the compiler terminates.

Internal errors should normally not occur and should be reported to the IAR Systems technical support group. Your report should include all possible information about the problem and preferably also a disk with the program that generated the internal error.

COMPILATION ERROR MESSAGES

The following table lists the compilation error messages:

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
0	Invalid syntax	The compiler could not decode the statement or declaration.
1	Too deep #include nesting (max is 10)	Fatal. The compiler limit for nesting of #include files was exceeded. One possible cause is an inadvertently recursive #include file.
2	Failed to open #include file 'name'	Fatal. The compiler could not open an #include file. Possible causes are that the file does not exist in the specified directories (possibly due to a faulty -I prefix or C_INCLUDE path) or is disabled for reading.
3	Invalid #include filename	Fatal. The #include filename was invalid. Note that the #include filename must be written <file> or "file".
4	Unexpected end of file encountered	Fatal. The end of file was encountered within a declaration, function definition, or during macro expansion. The probable cause is bad () or { } nesting.
5	Too long source line (max is 512 chars); truncated	The source line length exceeds the compiler limit.
6	Hexadecimal constant without digits	The prefix 0x or 0X of hexadecimal constant was found without following hexadecimal digits.
7	Character constant larger than "long"	A character constant contained too many characters to fit in the space of a long integer.

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
8	Invalid character encountered: '\xhh'; ignored	A character not included in the C character set was found.
9	Invalid floating point constant	A floating-point constant was found to be too large or have invalid syntax. See the ANSI standard for legal forms.
10	Invalid digits in octal constant	The compiler found a non-octal digit in an octal constant. Valid octal digits are: 0–7.
11	Missing delimiter in literal or character constant	No closing delimiter ' or " was found in character or literal constant.
12	String too long (max is 509)	The limit for the length of a single or concatenated string was exceeded.
13	Argument to #define too long (max is 512)	Lines terminated by \ resulted in a #define line that was too long.
14	Too many formal parameters for #define (max is 127)	Fatal. Too many formal parameters were found in a macro definition (#define directive).
15	',' or ')' expected	The compiler found an invalid syntax of a function definition header or macro definition.
16	Identifier expected	An identifier was missing from a declarator, goto statement, or pre-processor line.
17	Space or tab expected	Pre-processor arguments must be separated from the directive with tab or space characters.
18	Macro parameter 'name' redefined	The formal parameter of a symbol in a #define statement was repeated.

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
19	Unmatched <code>#else</code> , <code>#endif</code> or <code>#elif</code>	Fatal. A <code>#if</code> , <code>#ifdef</code> , or <code>#ifndef</code> was missing.
20	No such pre-processor command: 'name'	<code>#</code> was followed by an unknown identifier.
21	Unexpected token found in pre-processor line	A pre-processor line was not empty after the argument part was read.
22	Too many nested parameterized macros (max is 50)	Fatal. The pre-processor limit was exceeded.
23	Too many active macro parameters (max is 256)	Fatal. The pre-processor limit was exceeded.
24	Too deep macro nesting (max is 100)	Fatal. The pre-processor limit was exceeded.
25	Macro 'name' called with too many parameters	Fatal. A parameterized <code>#define</code> macro was called with more arguments than declared.
26	Actual macro parameter too long (max is 512)	A single macro argument may not exceed the length of a source line.
27	Macro 'name' called with too few parameters	A parameterized <code>#define</code> macro was called with fewer arguments than declared.
28	Missing <code>#endif</code>	Fatal. The end of file was encountered during skipping of text after a false condition.
29	Type specifier expected	A type description was missing. This could happen in <code>struct</code> , <code>union</code> , prototyped function definitions/declarations, or in K&R function formal parameter declarations.

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
30	Identifier unexpected	There was an invalid identifier. This could be an identifier in a type name definition like: sizeof(int*ident); or two consecutive identifiers.
31	Identifier 'name' redeclared	There was a redeclaration of a declarator identifier.
32	Invalid declaration syntax	There was an undecodable declarator.
33	Unbalanced '(' or ')' in declarator	There was a parenthesis error in a declarator.
34	C statement or func-def in #include file, add "i" to the "-r" switch	To get proper C source line stepping for #include code when the C-SPY debugger is used, the -ri option must be specified. Other source code debuggers (that do not use the UBROF output format) may not work with code in #include files.
35	Invalid declaration of "struct", "union" or "enum" type	A struct, union, or enum was followed by an invalid token(s).
36	Tag identifier 'name' redeclared	A struct, union, or enum tag is already defined in the current scope.
37	Function 'name' declared within "struct" or "union"	A function was declared as a member of struct or union.
38	Invalid width of field (max is nn)	The declared width of field exceeds the size of an integer (nn is 16 or 32 depending on the target processor).

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
39	<code>','</code> or <code>;'</code> expected	There was a missing <code>,</code> or <code>;</code> at the end of declarator.
40	Array dimension outside of "unsigned int" bounds	Array dimension negative or larger than can be represented in an unsigned integer.
41	Member 'name' of "struct" or "union" redeclared	A member of struct or union was redeclared.
42	Empty "struct" or "union"	There was a declaration of struct or union containing no members.
43	Object cannot be initialized	There was an attempted initialization of typedef declarator or struct or union member.
44	<code>;'</code> expected	A statement or declaration needs a terminating semicolon.
45	<code>']'</code> expected	There was a bad array declaration or array expression.
46	<code>':'</code> expected	There was a missing colon after default, case label, or in <code>?</code> -operator.
47	<code>'('</code> expected	The probable cause is a misformed <code>for</code> , <code>if</code> , or <code>while</code> statement.
48	<code>'),'</code> expected	The probable cause is a misformed <code>for</code> , <code>if</code> , or <code>while</code> statement or expression.
49	<code>','</code> expected	There was an invalid declaration.
50	<code>'{'</code> expected	There was an invalid declaration or initializer.
51	<code>'}'</code> expected	There was an invalid declaration or initializer.

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
52	Too many local variables and formal parameters (max is 1024)	Fatal. The compiler limit was exceeded.
53	Declarator too complex (max is 128 '(' and/or '*')	The declarator contained too many (,), or *
54	Invalid storage class	An invalid storage-class for the object was specified.
55	Too deep block nesting (max is 50)	Fatal. The {} nesting in a function definition was too deep.
56	Array of functions	An attempt was made to declare an array of functions. The valid form is array of pointers to functions: <pre>int array [5] (); /* Invalid */ int (*array [5]) (); /* Valid */</pre>
57	Missing array dimension specifier	There was a multi-dimensional array declarator with a missing specified dimension. Only the first dimension can be excluded (in declarations of extern arrays and function formal parameters).
58	Identifier 'name' redefined	There was a redefinition of a declarator identifier.
59	Function returning array	Functions cannot return arrays.
60	Function definition expected	A K&R function header was found without a following function definition, for example: <pre>int f(i); /* Invalid */</pre>
61	Missing identifier in declaration	A declarator lacked an identifier.
62	Simple variable or array of a "void" type	Only pointers, functions, and formal parameters can be of void type.

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
63	Function returning function	A function cannot return a function, as in: <code>int f()(); /* Invalid */</code>
64	Unknown size of variable object 'name'	The defined object has unknown size. This could be an external array with no dimension given or an object of an only partially (forward) declared struct or union.
65	Too many errors encountered (>100)	Fatal. The compiler aborts after a certain number of diagnostic messages.
66	Function 'name' redefined	Multiple definitions of a function were encountered.
67	Tag 'name' undefined	There was a definition of variable of enum type with type undefined or a reference to undefined struct or union type in a function prototype or as a sizeof argument.
68	"case" outside "switch"	There was a case without any active switch statement.
69	"interrupt" function may not be referred or called	An interrupt function call was included in the program. Interrupt functions can be called by the run-time system only.
70	Duplicated "case" label: nn	The same constant value was used more than once as a case label.
71	"default" outside "switch"	There was a default without any active switch statement.
72	Multiple "default" within "switch"	More than one default in one switch statement.

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
73	Missing "while" in "do" - "while" statement	Probable cause is missing {} around multiple statements.
74	Label 'name' redefined	A label was defined more than once in the same function.
75	"continue" outside iteration statement	There was a continue outside any active while, do ... while, or for statement.
76	"break" outside "switch" or iteration statement	There was a break outside any active switch, while, do ... while, or for statement.
77	Undefined label 'name'	There is a goto label with no label: definition within the function body.
78	Pointer to a field not allowed struct { int *f:6; /* Invalid */ }	There is a pointer to a field member of struct or union:
79	Argument of binary operator missing	The first or second argument of a binary operator is missing.
80	Statement expected	One of ? : ,] or } was found where statement was expected.
81	Declaration after statement This could be due to an unwanted ; for example: int i;; char c; /* Invalid */ Since the second ; is a statement it causes a declaration after a statement.	A declaration was found after a statement.

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
82	"else" without preceding "if"	The probable cause is bad {} nesting.
83	"enum" constant(s) outside "int" or "unsigned" "int" range	An enumeration constant was created too small or too large.
84	Function name not allowed in this context	An attempt was made to use a function name as an indirect address.
85	Empty "struct", "union" or "enum"	There is a definition of struct or union that contains no members or a definition of enum that contains no enumeration constants.
86	Invalid formal parameter	There is a fault with the formal parameter in a function declaration.
Possible causes are:		
	int f();	/* valid K&R declaration */
	int f(i);	/* invalid K&R declaration */
	int f(int i);	/* valid ANSI declaration */
	int f(i);	/* invalid ANSI declaration */
87	Redeclared formal parameter: 'name'	A formal parameter in a K&R function definition was declared more than once.
88	Contradictory function declaration	void appears in a function parameter type list together with other type of specifiers.
89	"..." without previous parameter(s)	... cannot be the only parameter description specified.
For example:		
	int f(...);	/* Invalid */
	int f(int, ...);	/* Valid */

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
90	Formal parameter identifier missing For example: <pre>int f(int *p, char, float ff) /* Invalid - second parameter has no name */ { /* function body */ }</pre>	An identifier of a parameter was missing in the header of a prototyped function definition.
91	Redeclared number of formal parameters For example: <pre>int f(int,char); /* first declaration -valid */ int f(int); /* fewer parameters -invalid */ int f(int,char,float);/* more parameters -invalid */</pre>	A prototyped function was declared with a different number of parameters than the first declaration.
92	Prototype appeared after reference	A prototyped declaration of a function appeared after it was defined or referenced as a K&R function.
93	Initializer to field of width nn (bits) out of range	A bit-field was initialized with a constant too large to fit in the field space.
94	Fields of width 0 must not be named	Zero length fields are only used to align fields to the next int boundary and cannot be accessed via an identifier.
95	Second operand for division or modulo is zero	An attempt was made to divide by zero.

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
96	Unknown size of object pointed to	An incomplete pointer type is used within an expression where size must be known.
97	Undefined "static" function 'name'	A function was declared with static storage class but never defined.
98	Primary expression expected	An expression was missing.
99	Extended keyword not allowed in this context	An extended processor-specific keyword occurred in an illegal context; eg <code>interrupt int i</code> .
100	Undeclared identifier: 'name'	There was a reference to an identifier that had not been declared.
101	First argument of '.' operator must be of "struct" or "union" type	The dot operator <code>.</code> was applied to an argument that was not struct or union.
102	First argument of '->' was not pointer to "struct" or "union"	The arrow operator <code>-></code> was applied to an argument that was not a pointer to a struct or union.
103	Invalid argument of "sizeof" operator	The <code>sizeof</code> operator was applied to a bit-field, function, or extern array of unknown size.
104	Initializer "string" exceeds array dimension	An array of char with explicit dimension was initialized with a string exceeding array size.
	For example: <pre>char array [4] = "abcde"; /* invalid */</pre>	
105	Language feature not implemented	A constant argument or constant pointer is required for the in-line functions.

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
106	Too many function parameters (max is 127)	Fatal. There were too many parameters in function declaration/definition.
107	Function parameter 'name' already declared	A formal parameter in a function definition header was declared more than once. For example: <pre>/* K&R function */ int myfunc(i, i) /* invalid */ int i; { } /* Prototyped function */ int myfunc(int i, int i) /* invalid */ { }</pre>
108	Function parameter 'name' declared but not found in header	In a K&R function definition, the parameter was declared but not specified in the function header. For example: <pre>int myfunc(i) int i, j /* invalid - j is not specified in the function header */ { }</pre>
109	';' unexpected	An unexpected delimiter was encountered.
110	')' unexpected	An unexpected delimiter was encountered.
111	'{' unexpected	An unexpected delimiter was encountered.
112	',' unexpected	An unexpected delimiter was encountered.

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
113	'.' unexpected	An unexpected delimiter was encountered.
114	'[' unexpected	An unexpected delimiter was encountered.
115	'(' unexpected	An unexpected delimiter was encountered.
116	Integral expression required	The evaluated expression yielded a result of the wrong type.
117	Floating point expression required	The evaluated expression yielded a result of the wrong type.
118	Scalar expression required	The evaluated expression yielded a result of the wrong type.
119	Pointer expression required	The evaluated expression yielded a result of the wrong type.
120	Arithmetic expression required	The evaluated expression yielded a result of the wrong type.
121	Lvalue required	The expression result was not a memory address.
122	Modifiable lvalue required	The expression result was not a variable object or a const.
123	Prototyped function argument number mismatch	A prototyped function was called with a number of arguments different from the number declared.
124	Unknown "struct" or "union" member: 'name'	An attempt was made to reference a non-existent member of a struct or union.
125	Attempt to take address of field	The & operator may not be used on bit-fields.
126	Attempt to take address of "register" variable	The & operator may not be used on objects with register storage class.

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
127	Incompatible pointers	There must be full compatibility of objects that pointers point to. In particular, if pointers point (directly or indirectly) to prototyped functions, the code performs a compatibility test on return values and also on the number of parameters and their types. This means that incompatibility can be hidden quite deeply, for example: <pre>char (*(p1)[8])(int); char (*(p2)[8])(float); /* p1 and p2 are incompatible - the function parameters have incompatible types */</pre> The compatibility test also includes checking of array dimensions if they appear in the description of the objects pointed to, for example: <pre>int (*p1)[8]; int (*p2)[9]; /* p1 and p2 are incompatible - array dimensions differ */</pre>
128	Function argument incompatible with its declaration	A function argument is incompatible with the argument in the declaration.
129	Incompatible operands of binary operator	The type of one or more operands to a binary operator was incompatible with the operator.
130	Incompatible operands of '=' operator	The type of one or more operands to = was incompatible with =.
131	Incompatible "return" expression	The result of the expression is incompatible with the return value declaration.
132	Incompatible initializer	The result of the initializer expression is incompatible with the object to be initialized.

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
133	Constant value required	The expression in a case label, <code>#if</code> , <code>#elif</code> , bit-field declarator, array declarator, or static initializer was not constant.
134	Unmatching "struct" or "union" arguments to '?' operator	The second and third argument of the ? operator are different.
135	" pointer + pointer" operation	Pointers may not be added.
136	Redeclaration error	The current declaration is inconsistent with earlier declarations of the same object.
137	Reference to member of undefined "struct" or "union"	The only allowed reference to undefined struct or union declarators is a pointer.
138	"- pointer" expression	The - operator may be used on pointers only if both operators are pointers, that is, <code>pointer - pointer</code> . This error means that an expression of the form <code>non-pointer - pointer</code> was found.
139	Too many "extern" symbols declared (max is 32767)	Fatal. The compiler limit was exceeded.
140	"void" pointer not allowed in this context	A pointer expression such as an indexing expression involved a void pointer (element size unknown).
141	<code>#error 'any message'</code>	Fatal. The pre-processor directive <code>#error</code> was found, notifying that something must be defined on the command line in order to compile this module.

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
142	"interrupt" function can only be "void" and have no arguments	An interrupt function declaration had a non-void result and/or arguments, neither of which are allowed.
143	Too large, negative or overlapping "interrupt" [value] in name	Check the [vector] values of the declared interrupt functions.
144	Bad context for storage modifier (storage-class or function)	The no_init keyword can only be used to declare variables with static storage-class. That is, no_init cannot be used in typedef statements or applied to auto variables of functions. An active #pragma memory=no_init can cause such errors when function declarations are found.
145	Bad context for function call modifier	The keywords interrupt, banked, non_banked, or monitor can be applied only to function declarations.
146	Unknown #pragma identifier	An unknown pragma identifier was found. This error will terminate object code generation only if the -g option is in use.
147	Extension keyword "name" is already defined by user	Upon executing: #pragma language=extended the compiler found that the named identifier has the same name as an extension keyword. This error is only issued when compiler is executing in ANSI mode.
148	'=' expected	An sfrb-declared identifier must be followed by =value.

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
149	Attempt to take address of "sfrb" or "bit" variable	The & operator may not be applied to variables declared as bit or as sfrb.
150	Illegal range for "sfrb" or "bit" address	The address expression is not a valid bit or sfrb address.
151	Too many functions defined in a single module.	There may not be more than 256 functions in use in a module. Note that there are no limits to the number of declared functions.
152	'.' expected	The . was missing from a bit declaration.
153	Illegal context for extended specifier	

MSP430-SPECIFIC ERROR MESSAGES

<i>No</i>	<i>Warning message</i>	<i>Suggestion</i>
154	Constant argument required	A non-constant argument is used with the _OPC intrinsic.

COMPILATION
WARNING MESSAGES

The following table lists the compilation warning messages:

<i>No</i>	<i>Warning message</i>	<i>Suggestion</i>
0	Macro 'name' redefined	A symbol defined with #define was redeclared with a different argument or formal list.
1	Macro formal parameter 'name' is never referenced	A #define formal parameter never appeared in the argument string.
2	Macro 'name' is already #undef	#undef was applied to a symbol that was not a macro.

<i>No</i>	<i>Warning message</i>	<i>Suggestion</i>
3	Macro 'name' called with empty parameter(s)	A parameterized macro defined in a #define statement was called with a zero-length argument.
4	Macro 'name' is called recursively; not expanded	A recursive macro call makes the pre-processor stop further expansion of that macro.
5	Undefined symbol 'name' in #if or #elif; assumed zero	It is considered as bad programming practice to assume that non-macro symbols should be treated as zeros in #if and #elif expressions. Use either: #ifdef symbol or #if defined (symbol)
6	Unknown escape sequence ('\c'); assumed 'c'	A backslash (\) found in a character constant or string literal was followed by an unknown escape character.
7	Nested comment found without using the '-C' option	The character sequence /* was found within a comment, and ignored.
8	Invalid type-specifier for field; assumed "int"	In this implementation, bitfields may be specified only as int or unsigned int.
9	Undeclared function parameter 'name'; assumed "int"	An undeclared identifier in the header of a K&R function definition is by default given the type int.
10	Dimension of array ignored; array assumed pointer	An array with an explicit dimension was specified as a formal parameter, and the compiler treated it as a pointer to object.

<i>No</i>	<i>Warning message</i>	<i>Suggestion</i>
11	Storage class "static" ignored; 'name' declared "extern"	An object or function was first declared as extern (explicitly or by default) and later declared as static. The static declaration is ignored.
12	Incompletely bracketed initializer	To avoid ambiguity, initializers should either use only one level of {} brackets or be completely surrounded by {} brackets.
13	Unreferenced label 'name'	Label was defined but never referenced.
14	Type specifier missing; assumed "int"	No type specifier given in declaration – assumed to be int.
15	Wrong usage of string operator ('#' or '##'); ignored	This implementation restricts usage of # and ## operators to the token-field of parameterized macros.
In addition the # operator must precede a formal parameter:		
<pre>#define mac(p1) #p1 /* Becomes "p1" */ #define mac(p1,p2) p1+p2##add_this /* Merged p2 */</pre>		
16	Non-void function: "return" with <expression>; expected	A non-void function definition should exit with a defined return value in all places.
17	Invalid storage class for function; assumed to be "extern"	Invalid storage class for function – ignored. Valid classes are extern, static, or typedef.
18	Redeclared parameter's storage class	Storage class of a function formal parameter was changed from register to auto or vice versa in a subsequent declaration/definition.
19	Storage class "extern" ignored; 'name' was first declared as "static"	An identifier declared as static was later explicitly or implicitly declared as extern. The extern declaration is ignored.

<i>No</i>	<i>Warning message</i>	<i>Suggestion</i>
20	Unreachable statement(s)	One or more statements were preceded by an unconditional jump or return such that the statement or statements would never be executed.
	For example: break; i = 2;	/* Never executed */
21	Unreachable statement(s) at unreferenced label 'name'	One or more labeled statements were preceded by an unconditional jump or return but the label was never referenced, so the statement or statements would never be executed.
	For example: break; here: i = 2;	/* Never executed */
22	Non-void function: explicit "return" <expression>; expected	A non-void function generated an implicit return. This could be the result of an unexpected exit from a loop or switch. Note that a switch without default is always considered by the compiler to be 'exitable' regardless of any case constructs.
23	Undeclared function 'name'; assumed "extern" "int"	A reference to an undeclared function causes a default declaration to be used. The function is assumed to be of K&R type, have extern storage class, and return int.

<i>No</i>	<i>Warning message</i>	<i>Suggestion</i>
24	Static memory option converts local "auto" or "register" to "static"	A command line option for static memory allocation caused auto and register declarations to be treated as static.
25	Inconsistent use of K&R function - varying number of parameters	A K&R function was called with a varying number of parameters.
26	Inconsistent use of K&R function - changing type of parameter For example: myfunc (34); myfunc(34.6);	A K&R function was called with changing types of parameters. /* int argument */ /* float argument */
27	Size of "extern" object 'name' is unknown	extern arrays should be declared with size.
28	Constant [index] outside array bounds	There was a constant index outside the declared array bounds.
29	Hexadecimal escape sequence larger than "char"	The escape sequence is truncated to fit into char.

<i>No</i>	<i>Warning message</i>	<i>Suggestion</i>
30	Attribute ignored	Since const or volatile are attributes of objects they are ignored when given with a structure, union, or enumeration tag definition that has no objects declared at the same time. Also, functions are considered as being unable to return const or volatile.

For example:

```
const struct s
{
    ...
}; /* no object declared, const ignored - warning */
const int myfunc(void);
/* function returning const int - warning */
const int (*fp)(void); /* pointer to function
returning const int - warning */
int (*const fp)(void);
/* const pointer to function returning int - OK,
no warning */
```

31	Incompatible parameters of K&R functions	Pointers (possibly indirect) to functions or K&R function declarators have incompatible parameter types.
----	--	--

The pointer was used in one of following contexts:

```
pointer - pointer,
expression ? ptr : ptr,
pointer relational_op pointer
pointer equality_op pointer
pointer = pointer
formal parameter vs actual parameter
```

<i>No</i>	<i>Warning message</i>	<i>Suggestion</i>
32	Incompatible numbers of parameters of K&R functions	Pointers (possibly indirect) to functions or K&R function declarators have a different number of parameters.
	The pointer is directly used in one of following contexts:	
	pointer - pointer expression ? ptr : ptr pointer relational_op pointerpointer equality_op pointer pointer = pointer formal parameter vs actual parameter	
33	Local or formal 'name' was never referenced	A formal parameter or local variable object is unused in the function definition.
34	Non-printable character '\xhh' found in literal or character constant	It is considered as bad programming practice to use non-printable characters in string literals or character constants. Use \0xhhh to get the same result.
35	Old-style (K&R) type of function declarator	An old style K&R function declarator was found. This warning is issued only if the -gA option is in use.
36	Floating point constant out of range	A floating-point value is too large or too small to be represented by the floating-point system of the target.
37	Illegal float operation: division by zero not allowed	During constant arithmetic a zero divide was found.
38	Tag identifier 'name' was never defined	

<i>No</i>	<i>Warning message</i>	<i>Suggestion</i>
39	Dummy statement. Optimized away!	<p>Redundant code found. This usually indicates a typing mistake in the user code or it might also be generated when using macros which are a little bit too generic (which is not a fault).</p> <p>For example:</p> <pre>a+b;</pre>
40	Possible bug! "If" statement terminated	<p>This usually indicates a typing mistake in the user code.</p> <p>For example:</p> <pre>if (a==b); { <if body> }</pre>
41	Possible bug! Uninitialized variable	<p>A variable is used before initialization (the variable has a random value).</p> <p>For example:</p> <pre>void func (p1) { short a; p1+=a; }</pre>
42		This message is discarded.

<i>No</i>	<i>Warning message</i>	<i>Suggestion</i>
43	Possible bug! Integer promotion may cause problems. Use cast to avoid it	The rule of integer promotion says that all integer operations must generate a result as if they were of int type if they have a small precision than int and this can sometimes lead to unexpected results. For example: <pre>short tst(unsigned char a) { if (-a) return (1); else return (-1); }</pre> <p>This example will always return the value 1 even with the value 0xff. The reason is that the integer promotion casts the variable a to 0x00ff first and then preforms a bit not.</p> <p>Integer promotion is ignored by many other C compilers, so this warning may be generated when recompiling an existing program with the IAR Systems compiler.</p>
44	Possible bug! Single '=' instead of '==' used in "if" statement	This usually indicates a typing mistake in the user code. For example: <pre>if (a=1) { <if body> }</pre>
45	Redundant expression. Example: Multiply with 1, add with 0	This might indicate a typing mistake in the user code, but it can also be a result of stupid code generated by a case tool.

<i>No</i>	<i>Warning message</i>	<i>Suggestion</i>
46	Possible bug! Strange or faulty expression. Example: Division by zero	This usually indicates a bug in the user code.
47	Unreachable code deleted by the global optimizer	Redundant code block in the user code. It might be a result of a bug but is usually only a sign of incomplete code.
48	Unreachable returns. The function will never return	The function will never be able to return to the calling function. This might be a result of a bug, but is usually generated when you have never ending loops in a RTOS system.
49	Unsigned compare always true/false	<p>This indicates a bug in the user code! A common reason is a missing -c compiler switch.</p> <p>For example:</p> <pre>for (uc=10; uc>=0; uc--) { <loop body> }</pre> <p>This is a never ending loop because an unsigned value is always larger than or equal to zero.</p>
51	Signed compare always true/false	This indicates a bug in the user code!

MSP430-SPECIFIC WARNING MESSAGES

None.

E		G		intrinsic functions (<i>continued</i>)	
ECSTR (segment)	191	getchar (library function)	67, 99	_BIS_SR	179
efficient coding	75	gets (library function)	100	_DINT	179
Embedded Workbench				_EINT	180
installing	2, 3			_NOP	180
running	2			_OPC	180
entry (keyword)	193	H		INTVEC (segment)	191
enum (keyword)	74, 195	header files	77	isalnum (library function)	101
errno.h (header file)	84	ctype.h	78	isalpha (library function)	101
error messages	201	errno.h	84	isctrl (library function)	102
exit (library function)	96	float.h	83	isdigit (library function)	102
exp (library function)	96	icclbutl.h	78	isgraph (library function)	103
extended keyword summary	155	limits.h	83	islower (library function)	103
extended keywords	159	math.h	79	isprint (library function)	104
interrupt	159	setjmp.h	80	ispunct (library function)	104
monitor	160	stdarg.h	80	isspace (library function)	105
sfrb	162	stddef.h	83	isupper (library function)	105
sfrw	163	stdio.h	80	isxdigit (library function)	106
extensions	155	stdlib.h	81		
		string.h	82	K	
		heap size	71	K&R definition	v
		hexadecimal string constants	197	Kernighan & Richie definition	193
F				keywords	
fabs (library function)	97	I		const	193
features, C compiler	5	icclbutl.h (header file)	78	entry	193
float.h (header file)	83	IDATA0 (segment)	191	enum	74, 195
floating point precision, XLINK		include options	59	signed	194
command file	65	initialization	71	struct	197
floating-point format	74	input and output	67	union	197
4-byte	74	installation, requirements	1	void	194
floor (library function)	97	interrupt (extended keyword)	159	volatile	194
fmod (library function)	98	interrupt functions	183, 184		
free (library function)	98	interrupt vectors	185	L	
frexp (library function)	99	intrinsic function summary	157	labs (library function)	106
function = default (#pragma		intrinsic function support	175	language extensions	155
directive)	167	intrinsic functions		language = default (#pragma	
function = interrupt (#pragma		_arg\$	177	directive)	168
directive)	167	_argt\$	178		
function = monitor (#pragma		_BIC_SR	179		
directive)	168				

language = extended (#pragma directive)	169	library functions (<i>continued</i>)	104	strlen	135
ldexp (library function)	107	isprint	104	strncat	135
ldiv (library function)	107	ispunct	105	strncmp	136
library functions		isspace	105	strncpy	137
formatted_read	149	isupper	106	strpbrk	137
formatted_write	150	isxdigit	106	strrchr	138
medium_read	151	labs	107	strspn	139
medium_write	152	ldexp	107	strstr	139
small_write	153	ldiv	108	strtod	140
abort	86	log	109	strtok	141
abs	87	log10	109	strtol	142
acos	87	longjmp	110	strtoul	143
asin	88	malloc	110	strxfrm	144
assert	88	memchr	111	tan	144
atan	89	memcmp	112	tanh	145
atan2	89	memcpy	112	tolower	145
atof	90	memmove	113	toupper	146
atoi	90	memset	114	va_arg	146
atol	91	modf	114	va_end	147
bsearch	92	pow	115	va_list	147
calloc	93	printf	119	va_start	148
ceil	93	putchar	120	library functions summary	78
cos	94	puts	121	limits.h (header file)	83
cosh	94	qsort	121	linker command file	65
div	95	rand	122	list options	52
exit	96	realloc	123	listings, formatting	54
exp	96	scanf	126	log (library function)	108
fabs	97	setjmp	127	log10 (library function)	109
floor	97	sin	127	longjmp (library function)	109
fmod	98	sinh	128		
free	98	sprintf	128		
frexp	99	sqrt	129	M	
getchar	99	srand	130	malloc (library function)	110
gets	100	sscanf	130	math.h (header file)	79
isalnum	101	strcat	131	memchr (library function)	110
isalpha	101	strchr	132	memcmp (library function)	111
iscntrl	102	strcmp	132	memcpy (library function)	112
isdigit	102	strcoll	133	memmove (library function)	112
isgraph	103	strcpy	134	memory map	66, 187
islower	103	strcspn	134		
		strerror			

230

strncat (library function) 135
 strncmp (library function) 136
 strncpy (library function) 137
 strpbrk (library function) 137
 strrchr (library function) 138
 strspn (library function) 139
 strstr (library function) 139
 strtod (library function) 140
 strtok (library function) 141
 strtol (library function) 142
 strtoul (library function) 143
 struct (keyword) 197
 strxfrm (library function) 144
 symbols, undefining 59

T

tab spacing 55
 tan (library function) 144
 tanh (library function) 145
 target identifier 174
 tolower (library function) 145
 toupper (library function) 146
 tutorial
 adding an interrupt handler 28
 configuring to suit the target
 program 9
 using #pragma directives 26
 tutorial (command line)
 running a program 26
 using C-SPY 26
 tutorial files 7
 type check 42

U

UDATA0 (segment) 192
 #undef options 58
 union (keyword) 197

V

va_arg (library function) 146
 va_end (library function) 147
 va_list (library function) 147
 va_start (library function) 148
 void (keyword) 194
 volatile (keyword) 194

W

warning messages 217, 226
 warnings = default (#pragma
 directive) 171
 warnings = off (#pragma
 directive) 172
 warnings = on (#pragma
 directive) 172
 Workbench
 installing 3
 running 2

X

XLINK command file 65
 XLINK options, -A 68

SYMBOLS

#pragma (directive) 165
 #pragma directive summary 156
 #pragma directives
 alignment 165
 bitfields = default 165
 bitfields = reversed 165
 codeseg 166
 function = default 167
 function = interrupt 167
 function = monitor 168

#pragma directives (continued)

 language = default 168
 language = extended 169
 memory = constseg 169
 memory = dataseg 170
 memory = default 170
 memory = no_init 171
 warnings = default 171
 warnings = off 172
 warnings = on 172
 \$ character 158
 -A (C compiler option) 56, 181
 -a (C compiler option) 56
 -A (XLINK option) 68
 -b (C compiler option) 41
 -C (C compiler option) 41
 -c (C compiler option) 40
 -D (C compiler option) 51
 -e (C compiler option) 40, 168
 -F (C compiler option) 54
 -f (C compiler option) 61
 -G (C compiler option) 62
 -g (C compiler option) 42, 77
 -H (C compiler option) 62
 -I (C compiler option) 60
 -i (C compiler option) 54
 -K (C compiler option) 41
 -L (C compiler option) 53, 181
 -l (C compiler option) 53
 -m (C compiler option) 174
 -N (C compiler option) 57
 -n (C compiler option) 57
 -O (C compiler option) 62
 -o (C compiler option) 63
 -P (C compiler option) 15, 63
 -p (C compiler option) 54
 -q (C compiler option) 15, 53, 181
 -R (C compiler option) 48
 -r (C compiler option) 15, 49
 -S (C compiler option) 64
 -s (C compiler option) 48

INDEX

-T (C compiler option)	54	__LINE__ (predefined symbol)	174	_formatted_write (library	
-t (C compiler option)	55	__STDC__ (predefined symbol)	174	function)	69, 150
-U (C compiler option)	59	__TID__ (predefined symbol)	174	_medium_read (library	
-ur (C compiler option)	48	__TIME__ (predefined symbol)	175	function)	70, 151
-v (C compiler option)	174	__VER__ (predefined symbol)	175	_medium_write (library	
-w (C compiler option)	41, 171, 172	_args\$ (intrinsic function)	177	function)	69, 152
-X (C compiler option)	57	_argt\$ (intrinsic function)	178	_NOP (intrinsic function)	180
-x (C compiler option)	55	_BIC_SR (intrinsic function)	179	_OPC (intrinsic function)	180
-y (C compiler option)	40, 189, 191	_BIS_SR (intrinsic function)	179	_small_write (library	
-z (C compiler option)	47, 48	_DINT (intrinsic function)	179	function)	69, 153
__DATE__ (predefined symbol)	173	_EINT (intrinsic function)	180		
__FILE__ (predefined symbol)	173	_formatted_read (library			
__IAR_SYSTEMS_ICC		function)	70, 149		
(predefined symbol)	173				