ASSEMBLER

PICMICRO™ ASSEMBLER, LINKER, AND LIBRARIAN

Programming Guide

COPYRIGHT NOTICE

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

First edition: October 1998

Part no: APIC-1

WELCOME

Welcome to the PICmicro™ Assembler, Linker, and Librarian Programming Guide.

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

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

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

If you are using the command line version, refer to the *PICmicro™ Command Line Interface Guide* for general information about running the IAR Systems tools from the command line.

For information about programming with the PICmicro™ C Compiler, refer to the *PICmicro™ C Compiler Programming Guide*.

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

ABOUT THIS GUIDE

This guide consists of the following chapters:

Installation and documentation 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 PICmicro™ Assembler.

The *Tutorial* illustrates how you might use the most important features of the assembler to develop simple PICmicro™ machine-code programs. It also describes a typical development cycle using XLINK and XLIB.

Assembler options summary explains how to set the PICmicro™ Assembler options, and gives an alphabetical summary of them.

Assembler options reference then gives reference information about each option.

Environment variables gives information about using the command line options to customize your PICmicro™ Assembler configuration.

Assembler file formats describes the source format for the PICmicro™ Assembler, and the format of assembler listings.

Assembler operator summary gives a summary of the assembler operators, arranged in order of precedence.

Assembler operator reference then gives a complete alphabetical list of the PICmicro™ Assembler operators, with a full description of each one.

Assembler directives summary gives an alphabetical summary of the PICmicro™ Assembler directives.

Assembler directives reference gives complete reference information about the PICmicro™ Assembler directives, classified into groups according to their function.

Assembler instructions lists the PICmicro™ instruction mnemonics, with details of the addressing modes that can be used with each one.

XLINK Linker

XLINK Linker introduces the XLINK Linker, and describes the XLINK listing format.

XLINK options summary explains how to set the XLINK options, and gives an alphabetical summary of the options.

XLINK options reference then gives detailed information about each option.

XLINK output formats summarizes the output formats available from XLINK.

XLIB Librarian

XLIB Librarian introduces the XLIB Librarian, which is designed to allow you to create and maintain relocatable libraries of routines.

XLIB command summary gives a summary of the XLIB commands.

XLIB command reference then gives complete reference information about each XLIB command.

Diagnostics

Assembler diagnostics provides a list of error messages specific to the PICmicro™ Assembler.

XLINK diagnostics and *XLIB diagnostics* describe the error and warning messages produced by XLINK and XLIB, together with explanations and suggested courses of action in each case.

ASSUMPTIONS

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

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

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

CONVENTIONS

This guide uses the following typographical conventions:

Style	*Used for*
`computer`	Text that you type in, or that appears on the screen.
`parameter`	A label representing the actual value you should type as part of a command.
`[option]`	An optional part of a command.
`{a | b | c}`	Alternatives in a command.
bold	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.
reference	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 Workbench interface.
>_	Identifies instructions specific to the command line versions of IAR Systems tools.

CONTENTS

INSTALLATION AND DOCUMENTATION

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

INCLUDED IN THIS PACKAGE

The PICmicro™ package contains the following items:

- CD-ROM or floppy disks.
- Product documentation:

  PICmicro™ Embedded Workbench Interface Guide

  PICmicro™ Command Line Interface Guide

  PICmicro™ Assembler, Linker, and Librarian Programming Guide

  PICmicro™ C Compiler Programming Guide

  PICmicro™ C-SPY User Guide, if you have purchased the IAR C-SPY debugger.
- Licence agreement including the *Product Registration Form*, which we urge you to fill out and send us to ensure that you receive the latest release of the IAR development tools for the PICmicro™ family of microcontrollers.

INSTALLING THE EMBEDDED WORKBENCH WITH C-SPY

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

WHAT YOU NEED

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

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

INSTALLING FROM WINDOWS 95/98 OR NT 4.0

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

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

2 Click the **Start** button in the taskbar, then click **Settings** and **Control Panel**.

3 Double-click the **Add/Remove Programs** icon in the **Control Panel** folder.

4 Click **Install**, then follow the instructions on the screen.

RUNNING FROM WINDOWS 95/98 OR NT 4.0

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

2 Click the **IAR Embedded Workbench** program icon.

INSTALLING FROM WINDOWS NT 3.51

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

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

2 Double-click the **File Manager** icon in the **Main** program group.

3 Click the floppy disk icon in the **File Manager** toolbar.

4 Double-click the **setup.exe** icon, then follow the instructions on the screen.

RUNNING FROM WINDOWS NT 3.51

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

RUNNING C-SPY

Either:

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

Or:

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

INSTALLING THE COMMAND LINE TOOLS

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

WHAT YOU NEED

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

INSTALLATION

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

When the installation is complete:

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

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

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

RUNNING THE TOOLS

Type the appropriate command at the command line prompt.

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

INSTALLED FILES

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

DOCUMENTATION FILES

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

ASSEMBLER FILES

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

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

MISCELLANEOUS FILES

The `etc` subdirectory holds XLINK-related files.

EXECUTABLE FILES

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

The `bin` subdirectory holds the executable program files.

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

C COMPILER FILES

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

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

INCLUDE FILES

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

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

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

LIBRARY FILES

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

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

No library modules are installed; instead the modules should be built for the appropriate microcontroller configuration using the included Buildlib utility, see the chapter *General C library definitions* in the *PICmicro™ C Compiler Programming Guide*.

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

LINKER COMMAND FILES

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

FILE TYPES

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

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

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

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

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

DOCUMENTATION

THE OTHER GUIDES

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

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

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

PICmicro™ C Compiler Programming Guide
This guide provides programming information about the PICmicro™ C Compiler. It includes reference information about the C library functions and language extensions, and provides information about support for the target-specific options such as memory models.

You should refer to this guide when you are setting up the C compiler configuration options in the Embedded Workbench, and for information about the C language when writing and debugging C source programs.

This guide also describes the diagnostic functions and lists the PICmicro™- specific warning and error messages.

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

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

ONLINE HELP

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

READ-ME FILES

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

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

IAR ON THE WEB

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

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

INTRODUCTION

The IAR Systems PICmicro™ Assembler, and its associated tools the XLINK Linker and XLIB Librarian, are 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 these tools, and provides information about running them from the Workbench or from the command line, as appropriate.

ASSEMBLER

The IAR Systems PICmicro™ Assembler is a powerful relocating macro assembler with a versatile set of directives.

The assembler incorporates a high degree of compatibility with the microcontroller manufacturer's assembler to ensure that software originally developed using that assembler can be transferred to the IAR Systems Assembler with little or no modification.

The IAR Systems PICmicro™ Assembler provides the following features:

GENERAL

- One pass assembly, for fast execution.
- Integration with the XLINK Linker and XLIB Librarian.
- Integration with other IAR Systems software.
- Self-explanatory error messages.

ASSEMBLER FEATURES

- Support for PICmicro™-family microcontrollers.
- Up to 256 relocatable segments per module.
- 32-bit arithmetic and IEEE floating-point constants.
- 255 significant characters in symbols.
- Powerful recursive macro facilities.
- Number of symbols and program size limited only by available memory.
- Support for complex expressions with external references.

- Forward references allowed to any depth.
- Support for C language pre-processor directives and `sfr` keywords.
- Macros in Intel/Motorola style.

XLINK LINKER

The IAR Systems XLINK Linker converts one or more relocatable object files produced by the IAR Systems Assembler or C Compiler to machine code for a specified target processor. It supports a wide range of industry-standard loader formats, in addition to the IAR Systems debug format used by the C-SPY high level debugger.

XLINK supports user libraries, and will load only those modules that are actually needed by the program you are linking.

The final output produced by XLINK is an absolute, target-executable object file that can be programmed into an EPROM, down-loaded to a hardware emulator, or run directly on the host using the IAR Systems C-SPY debugger.

XLINK offers the following important features:

FEATURES OF XLINK

- Full C-level type checking across all modules.
- Full dependency resolution of all symbols in all input files, independent of input order.
- Simple override of library modules.
- Supports 255 character symbol names.
- Checks for compatible compiler settings for all modules.
- Checks that the correct version and variant of the C runtime library is used.
- Flexible segment commands allow detailed control of code and data placement.
- Link-time symbol definition enables flexible configuration control.
- Support for over 30 output formats.
- Can generate checksum of code for run-time checking.

XLIB LIBRARIAN

The IAR Systems XLIB Librarian enables you to manipulate the relocatable object files produced by the IAR Systems Assembler and the IAR C Compiler.

XLIB provides the following features:

FEATURES OF XLIB

- Support for modular programming.
- Modules can be listed, added, inserted, replaced, deleted, or renamed.
- Segments can be listed and renamed.
- Symbols can be listed and renamed.
- Modules can be changed between program and library type.
- Interactive or batch mode operation.
- A full set of library listing operations.

TUTORIAL

This tutorial illustrates how you might use the PICmicro™ Assembler to develop a series of simple machine-code programs for the PICmicro™ microcontroller, and illustrates some of the assembler's most important features.

Before reading this chapter you should:

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

It is also recommended that you complete the introductory tutorial in the *PICmicro™ Embedded Workbench Interface Guide.*

RUNNING THE EXAMPLE PROGRAMS

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

Alternatively, you can run the examples by linking them with UBROF debugging information to give a file `aout.d39`, which can be downloaded to an emulator with debugging facilities. Use the XLINK **Output format** (`-F`) option to specify a format other than the default, Intel extended.

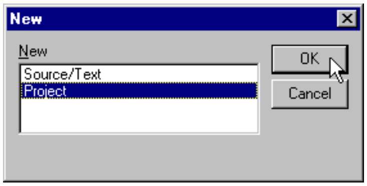
GETTING STARTED

The first step in developing an application using the assembler is to create a new project for the application files.

CREATING A NEW PROJECT

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:

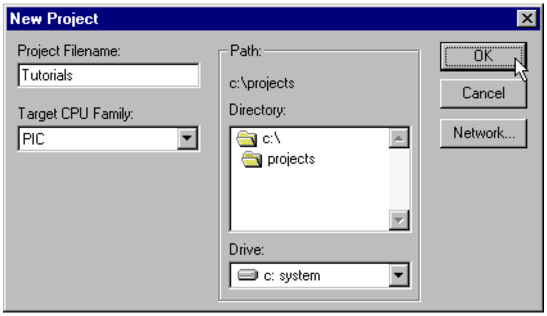


New

New: Project

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

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



New Project

Project Filename: Tutorials

Target CPU Family: PIC

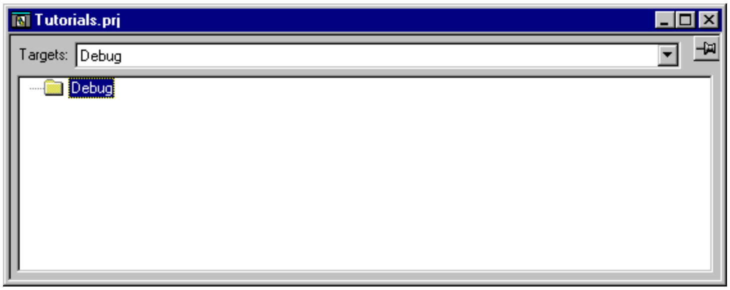
Path: c:\projects

Directory: c:\ projects

Drive: c: system

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

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

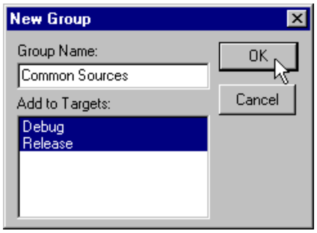


Tutorials.prj

Targets: Debug

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:



New Group

Group Name: Common Sources

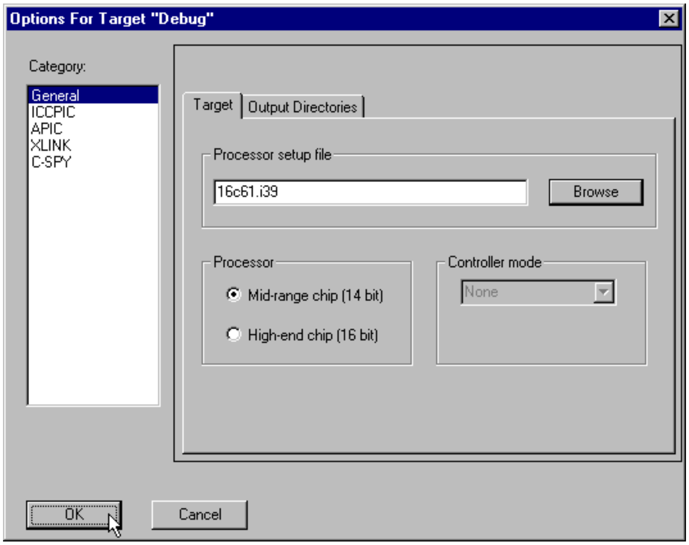
Add to Targets: Debug, Release

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

Now set up the target options to suit the chosen processor.

Select the **Debug** folder icon in the Project window, choose **Options…** from the **Project** menu and select **General** in the **Category** list. The **Target** option page is displayed.

Set the **Processor setup file** to **16C61** by selecting the file `16c61.i39` in the `...\setup\` directory. Then select the **Mid-range chip (14 bit)** processor.



Options For Target "Debug"

Category: General

Processor setup file: 16c61.i39

Processor: ● Mid-range chip (14 bit) ○ High-end chip (16 bit)

Controller mode: None

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

Creating a new project using the command line

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

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

```
cd iar\apic ↵
```

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

CREATING A PROGRAM

The first tutorial illustrates how you write a basic assembler program, and how you then assemble, link, and run it.

WRITING A PROGRAM

The first example program is a simple count loop which counts up the register files `0x0D` and `0x0C` in binary coded decimal:

```
        name    first

#define DIGIT1  0x0D         ; alias register file 0Ch
                             with symbol DIGIT1
#define DIGIT2  0x0C         ; alias register file 0Dh
                             with symbol DIGIT2
#define STATUS  0x03
#define CARRY   0            ; Carry flag in status reg
#define Z       2            ; Zero flag in status reg


        ORG     0x00         ; Set location counter at 0,
                             which is the reset vector

reset:  GOTO    start        ; Assemble symbolic address
                             to start of program

        ORG     0x50         ; Set new location for
                             program start

start:
        CLRF    DIGIT1       ; Make shure register files
                             are zero
        CLRF    DIGIT2       ;

count_loop:
        MOVLW   1            ; Load and add #1 to DIGIT1
        ADDWF   DIGIT1, 1    ;

        MOVLW   0x0A
        SUBWF   DIGIT1,0     ; Simulate a compare to #10
        BTFSS   STATUS,Z     ; skip over goto if DIGIT1 -
                             10 == 0
        GOTO    no_digit_carry; No decimal carry

        CLRF    DIGIT1       ; Start over from zero
        MOVLW   1            ; Load and add 1 to DIGIT2
        ADDWF   DIGIT2,1

no_digit_carry:

        MOVLW   0x0A            ; Compare against #10
        SUBWF   DIGIT2,0
        BTFSS   STATUS, Z       ; Skip if equal (Z == 1)
        GOTO    count_loop
        NOP

        END
```

The `ORG` directive sets the program start to address `0h`, the PIC 16C61 start address upon reset.

Writing the program using the Embedded Workbench

Run the Embedded Workbench, and 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 `first.s39`. The files associated with the PICmicro™ Assembler have extensions `.s39`, `.a39`, `.d39`, and `.r39` to identify them. A copy of the program is provided in the `iar\ew`*`nn`*`\picmicro\apic\tutor` 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 `first.s39`. The files associated with the PICmicro™ Assembler have extensions `.s39`, `.a39`, `.d39`, and `.r39` to identify them. A copy is provided in the `iar\apic\tutor` directory.

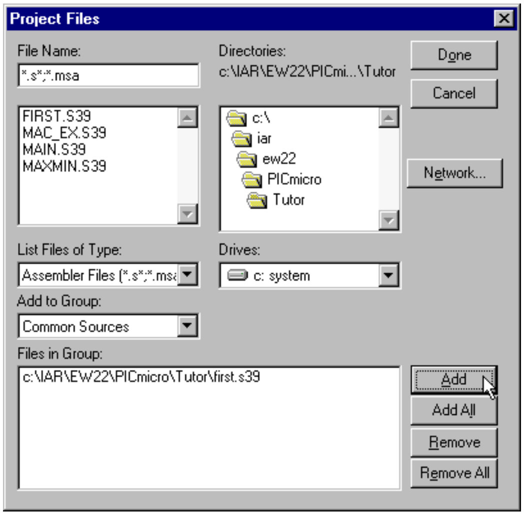
You now have a source file which is ready to assemble.

ASSEMBLING THE PROGRAM

Assembling the program using the Embedded Workbench

To assemble 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 `first.s39` in the file selection list in the upper half of the dialog box, and choose **Add** to add it to the **Common Sources** group:



Project Files

File Name: *.s*;*.msa

Directories: c:\IAR\EW22\PICmi...\Tutor

FIRST.S39
MAC_EX.S39
MAIN.S39
MAXMIN.S39

c:\
iar
ew22
PICmicro
Tutor

List Files of Type: Assembler Files (*.s*;*.msa

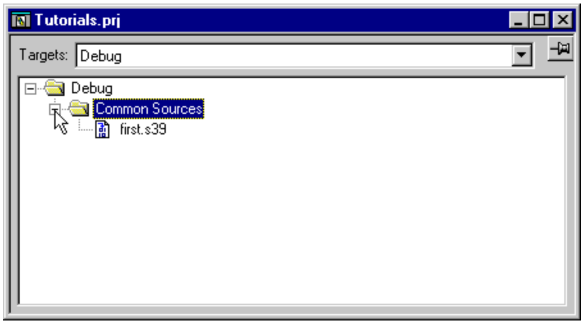
Drives: c: system

Add to Group: Common Sources

Files in Group: c:\IAR\EW22\PICmicro\Tutor\first.s39

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

Click the ⊞ symbol to display the file in the Project window tree display:



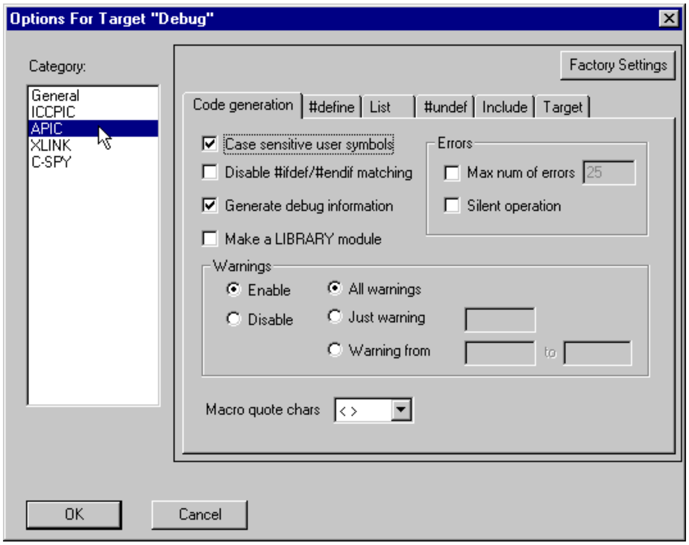
Tutorials.prj

Targets: Debug

- Debug
  - Common Sources
    - first.s39

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

Select the **Debug** folder in the Project window. Then choose **Options…** from the **Project** menu and select **APIC** in the **Category** list to display the assembler options pages:



Options For Target "Debug"

Category:
- General
- ICCPIC
- APIC
- XLINK
- C-SPY

Factory Settings

Code generation | #define | List | #undef | Include | Target

☑ Case sensitive user symbols

☐ Disable #ifdef/#endif matching

☑ Generate debug information

☐ Make a LIBRARY module

Errors

☐ Max num of errors 25

☐ Silent operation

Warnings

● Enable ● All warnings

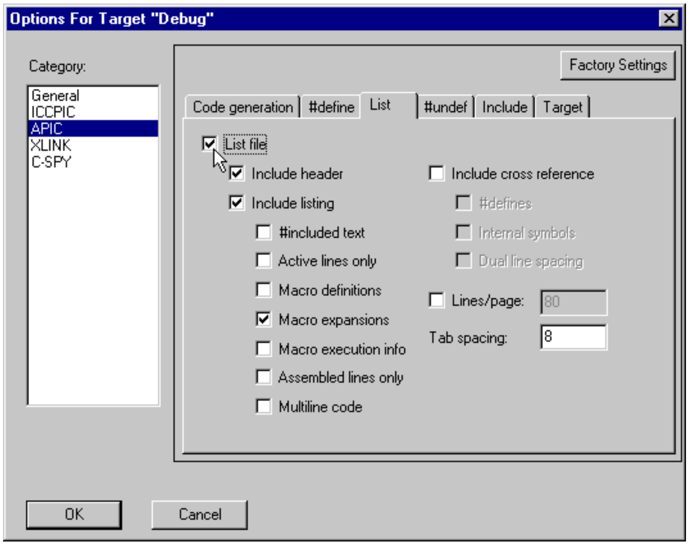
○ Disable ○ Just warning

○ Warning from to

Macro quote chars < >

OK Cancel

Click **List**, to display the page of list options, and select **List file** to produce an assembler list file. This will enable you to examine the code generated by the assembler:



Options For Target "Debug"

Category: General, ICCPIC, APIC, XLINK, C-SPY

Factory Settings

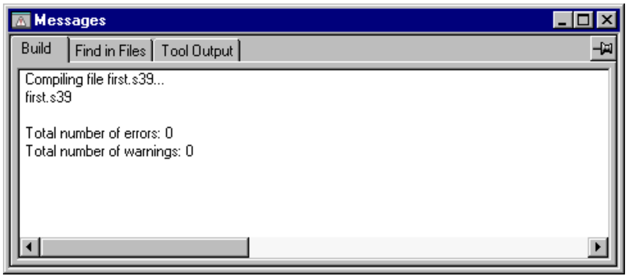
Code generation | #define | List | #undef | Include | Target

☑ List file
☑ Include header
☑ Include listing
☐ #included text
☐ Active lines only
☐ Macro definitions
☑ Macro expansions
☐ Macro execution info
☐ Assembled lines only
☐ Multiline code
☐ Include cross reference
☐ #defines
☐ Internal symbols
☐ Dual line spacing
☐ Lines/page: 80
Tab spacing: 8

OK Cancel

Choose **OK** to close the **Options** dialog box.

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



Messages

Build | Find in Files | Tool Output

```
Compiling file first.s39...
first.s39

Total number of errors: 0
Total number of warnings: 0
```

The listing is created in a file `first.lst` in the folder specified in the **General** options page; by default this is `Debug\list`. Open this file by choosing **Open…** from the **File** menu, and choosing `first.lst` from the appropriate folder.

Assembling the file using the command line

To assemble the file, type the following command at the prompt:

```
apic first -r -L ↵
```

This will send a listing to the file `first.lst`.

Viewing the listing

If you look at the list file you will see that it contains the following (the header will be slightly different if you are using the command line):

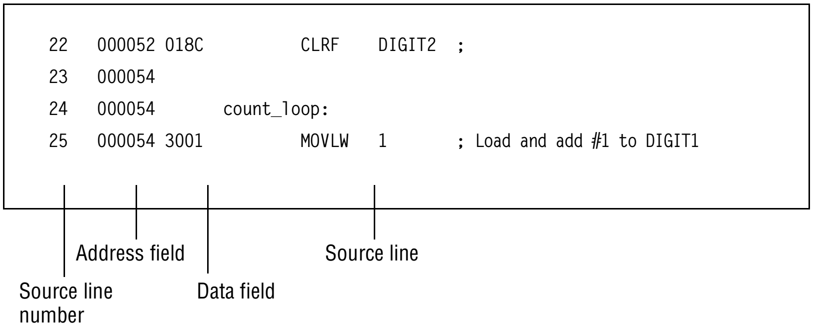
```
##########################################################################
#                                                                        #
#     IAR Systems PIC micro Assembler VX.X dd/Mmm/yyyy  hh:mm:ss         #
#                                                                        #
#                                                                        #
#           Target option =  Midrange  - 16C61 and above                 #
#           Source file   =  first.s39                                   #
#           List file     =  first.lst                                   #
#           Object file   =  first.r39                                   #
#           Command line  =  first -r -L                                 #
#                                                                        #
#                                         (c) Copyright IAR Systems 1998 #
##########################################################################
   1    000000               name    first
   2    000000
   3    000000       #define DIGIT1  0x0D      ; alias register file 0Ch
   4    000000                                 ; with symbol DIGIT1
   5    000000       #define DIGIT2  0x0C      ; alias register file 0Dh
   6    000000                                 ; with symbol DIGIT2
   7    000000       #define STATUS  0x03
   8    000000       #define CARRY   0         ; Carry flag in status reg
   9    000000       #define Z       2         ; Zero flag in status reg
  10    000000
  11    000000               ORG     0x00      ; Set location counter at
  12    000000                                 ; 0, which is the reset
  13    000000                                 ; vector
  14    000000
  15    000000 2828  reset:  GOTO    start     ; Jump to start of program
```

```
   16   000002
   17   000050               ORG     0x50      ; Set new location for
   18   000050                                 ; program start
   19   000050       start:
   20   000050 018D          CLRF    DIGIT1    ; Make sure register files
   21   000052                                 ; are zero
   22   000052 018C          CLRF    DIGIT2    ;
   23   000054
   24   000054       count_loop:
   25   000054 3001          MOVLW   1         ; Load and add #1 to DIGIT1
   26   000056 078D          ADDWF   DIGIT1, 1 ;
   27   000058 300A          MOVLW   0x0A
   28   00005A 028D          SUBWF   DIGIT1,0  ; Simulate a compare to #10
   29   00005C 1D03          BTFSS   STATUS,Z  ; skip over goto iff
   30   00005E                                 ; DIGIT1 - 10 == 0
   31   00005E 2833          GOTO    no_digit_carry ; No decimal carry
   32   000060 018D          CLRF    DIGIT1    ; Start over from zero
   33   000062 3001          MOVLW   1         ; Load and add 1 to DIGIT2
   34   000064 078C          ADDWF   DIGIT2,1
   35   000066
   36   000066       no_digit_carry:
   37   000066 300A          MOVLW   0x0A      ; Compare against #10
   38   000068 028C          SUBWF   DIGIT2,0
   39   00006A 1D03          BTFSS   STATUS, Z  ; Skip if equal (Z == 1)
   40   00006C 282A          GOTO    count_loop
   41   00006E 0000          NOP
   42   000070
   43   000070               END
##############################
#          CRC:D003         #
#        Errors:   0        #
#        Warnings: 0        #
#         Bytes: 34         #
###################################
```

This shows the machine-code instructions generated by each of the source code statements.

Note that the CRC number depends on the date of assembly, and may vary.

The format of the listing is as follows:



```
22   000052 018C          CLRF    DIGIT2  ;
23   000054
24   000054       count_loop:
25   000054 3001          MOVLW   1       ; Load and add #1 to DIGIT1
```

Assuming that the source assembled successfully, the file `first.r39`, will also be created, containing the linkable object code.

If you made any errors when entering the program, these will be displayed on the screen during the assembly. If this happens, return to the editor, check carefully through the source code to locate and correct all the mistakes, resave the source file using the same name, and try assembling it again.

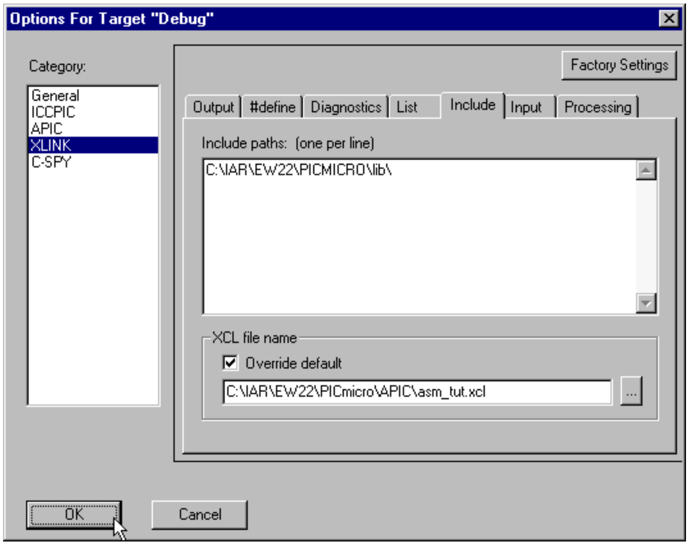
LINKING THE PROGRAM

Linking the program using the Embedded Workbench
Before linking the program you need to set up the linker options for the project.

Select the **Debug** folder in the Project window. Then choose **Options…** from the **Project** menu, and select **XLINK** in the **Category** list. Select the **Include** tab. In the XCL file name options, select **Override default**, and set the XCL file name to:

`\iar\ew`*`nn`*`\picmicro\apic\asm_tut.xcl`

(or whatever is the appropriate path for your installation). This specifies the simple linker command file `asm_tut.xcl`, designed for assembler-only projects.



Options For Target "Debug"

Category: XLINK

Include paths: (one per line)

C:\IAR\EW22\PICMICRO\lib\

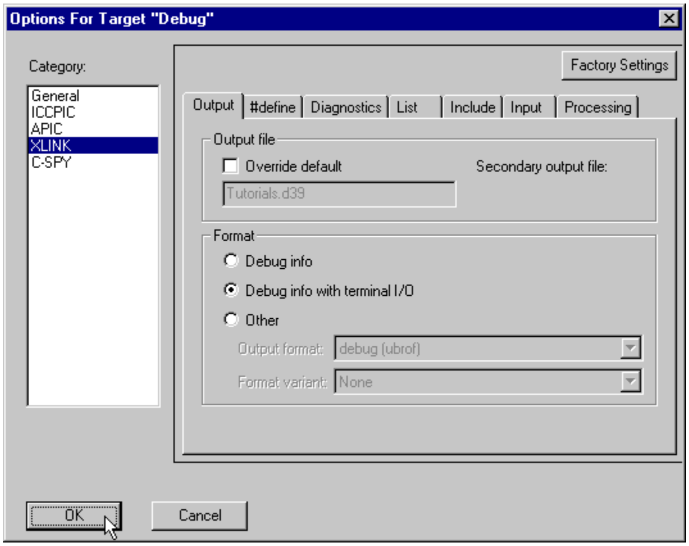
XCL file name

☑ Override default

C:\IAR\EW22\PICmicro\APIC\asm_tut.xcl

Click **Output** to display the output options.

Check that the **Format** option is set to **Debug info with terminal I/O**, to generate a file for debugging with C-SPY.



Options For Target "Debug"

Category: XLINK

Output file

□ Override default

Tutorials.d39

Secondary output file:

Format

○ Debug info

● Debug info with terminal I/O

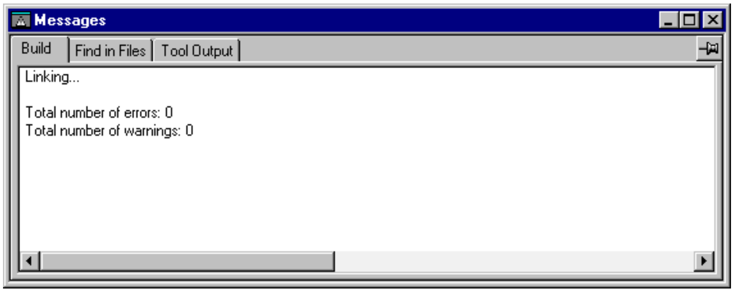
○ Other

Output format: debug (ubrof)

Format variant: None

Then choose **OK** to close the **Options** dialog box.

To link the file choose **Link** from the **Project** menu. As before, the progress during linking is shown in the Messages window.



```
Linking...

Total number of errors: 0
Total number of warnings: 0
```

The code will be placed in a file `tutorials.d39`.

Linking the program using the command line

To link the object file to produce code that can be executed, enter the command:

```
xlink first -cPIC -r ↵
```

The `-c` option specifies the target processor, and the `-r` option includes debugging information.

By default, the output code will be placed in a file `aout.d39`.

RUNNING THE PROGRAM

Running the program using the Embedded Workbench

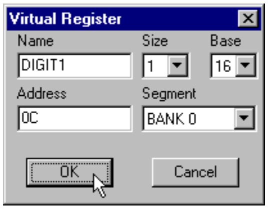
To run the example program using the C-SPY debugger choose **Debugger** from the **Project** menu.

The following warning messages will be displayed in the Report window:

```
Exit label missing.
No break on program exit.
```

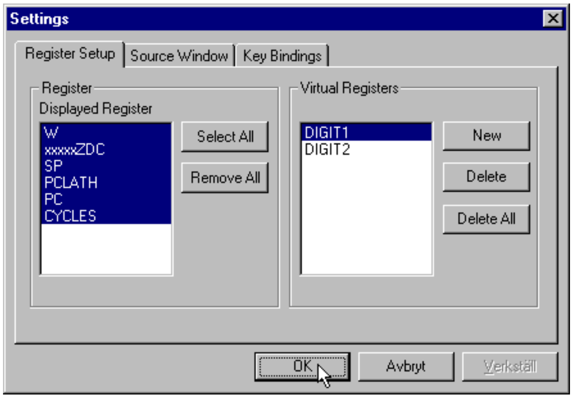
You can ignore these warnings.

In C-SPY select **Settings...** from the **Options** menu to add `DIGIT1` and `DIGIT2` as virtual registers. Click **New** to open the **Virtual Register** dialog box:



Name: DIGIT1
Size: 1
Base: 16
Address: 0C
Segment: BANK 0

Add register `DIGIT1` at address `0x0D` in `Bank 0`. Then click **OK**. Add register `DIGIT2` in the same manner, at address `0X0C` in `Bank 0`. For both registers **Size** is 1, and **Base** is 16. Click **OK**.



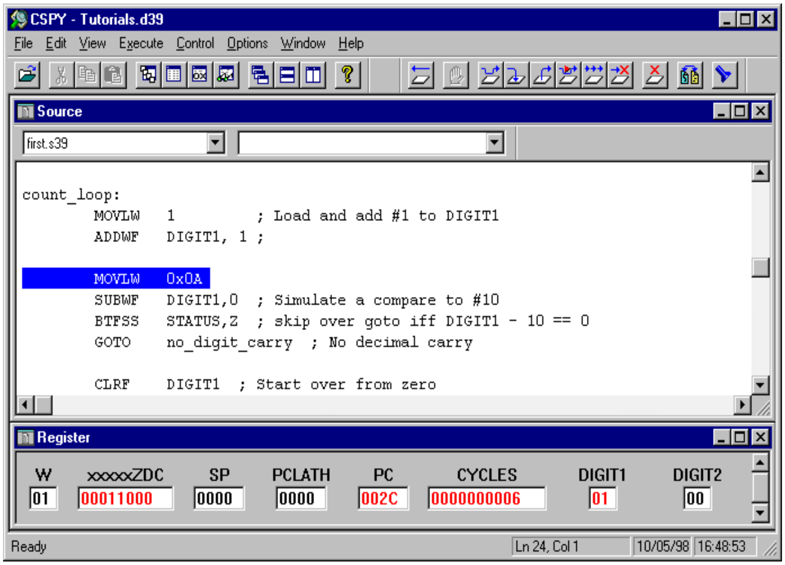
Displayed Register:
W
xxxxxZDC
SP
PCLATH
PC
CYCLES

Virtual Registers:
DIGIT1
DIGIT2

Then click **OK** to close the Settings window.

Open the Register window, by choosing **Register** from the **Window** menu.

Then choose **Step** from the **Execute** menu, or click the **Step** button in the debug bar, to step through the program and watch the `DIGIT1` and `DIGIT2` registers count in binary-coded decimal format.



```
count_loop:
        MOVLW   1         ; Load and add #1 to DIGIT1
        ADDWF   DIGIT1, 1 ;

        MOVLW   0x0A
        SUBWF   DIGIT1,0  ; Simulate a compare to #10
        BTFSS   STATUS,Z  ; skip over goto iff DIGIT1 - 10 == 0
        GOTO    no_digit_carry  ; No decimal carry

        CLRF    DIGIT1  ; Start over from zero
```

W	xxxxxZDC	SP	PCLATH	PC	CYCLES	DIGIT1	DIGIT2
01	00011000	0000	0000	002C	0000000006	01	00

Running the program using the command line

To run the example program, download `aout.d39` to the IAR C-SPY Debugger using the command line:

```
cwnn -d SPIC16 aout ↵
```

and follow the instructions above.

USING MACROS

The second example will demonstrate the use of simple macros. It shows how to read an I/O port and count the occurrences of a specific bit pattern.

For a complete explanation of the assembler's macro features see *Macro processing directives*, page 114.

The program below defines two simple, yet useful, macros:

The first macro, cmp_eq, compares a register file and a literal constant and sets the zero flag in the status register accordingly:

```
cmp_eq  MACRO   F,K
        MOVLW   K            ; load K
        SUBWF   F, 0         ; Subtract K from register
                             file and store in W (and set
                             Z accordingly)
        ENDM
```

This macro would be called with a statement such as:

```
        cmp_eq  A,0x81
```

The second macro, jnz, jumps to a specified label if the zero flag is not set. jnz has the following definition:

```
jnz     MACRO   label
        BTFSS   STATUS, Z    ; check Z flag and ...
        GOTO    label        ; branch if Z cleared, else
                             skip
        ENDM
```

and is used like:

```
        jnz     label
```

The full program is as follows:

```
#define STATUS  3
#define Z       2
#define RP0     5
#define TEMP    0x0C
#define A       0x0D
#define B       0x0E
#define PORTB   0x06         ; PIC16C84 specific ...
#define TRISB   0x06         ; address
```

```
; macro to compare a register file and a literal
cmp_eq  MACRO                F,K
        MOVLW   K            ; load K
        SUBWF   F, 0         ; Subtract K from register
                             file and store in W (and set
                             Z accordingly)
        ENDM


; macro that test the Z flag and jumps to <label> if Z==1

jnz     MACRO   label
        BTFSS   STATUS, Z    ; check Z flag and ...
        GOTO    label        ; branch if Z cleared, else
                             skip
        ENDM

; program that initializes PORTB and read and counts some
                             values
        ORG     0
        GOTO    main         ;reset vector

        ORG     0x50

main:
        BSF     STATUS, RP0  ; Switch register file bank

        MOVLW   0xFF         ; Configure PORTB ...
        MOVWF   TRISB        ; as input

        BCF     STATUS, RP0  ; Switch back to bank 0
        CLRF    B            ; init counter

loop:
        MOVF    PORTB, 0     ; Read PORTB
        MOVWF   A            ; Move the port value to A

        cmp_eq  A,0x81       ; Is A == 0x81? (sets Z
                             flags)
        jnz     skip_count   ; No, skip
        INCF    B,1          ; Pattern found -->
                             increment counter

skip_count:
        cmp_eq  B,10         ; Have we found 10
                             occurrences of pattern?
        BTFSS   STATUS,Z     ; skip goto if equal
        GOTO    loop

self:   GOTO    self

        end
```

The program consists of an entry point called `main`, that initializes PORTB and a counter, and repeatedly reads PORTB and compares the port value to a specific bit pattern. If the pattern is found the counter is updated. When the counter reaches a specified number, the program exits the loop.

Type in this listing and save it in a file `mac_ex.s39`. Alternatively, a copy of the source file is provided on the installation disk.

ASSEMBLING THE PROGRAM

Assembling the program using the Embedded Workbench
Close the **Tutorials** project, and create a new project, **Tutor2**, by choosing **New** from the **File** menu, and add the file `mac_ex.s39` to it.

Then assemble the file as before, by selecting it in the Project window and choosing **Compile** from the **Project** menu.

Assembling the program using the command line
To assemble the source program enter the command:

```
apic mac_ex -r -L ↵
```

Viewing the listing
The following output will be produced in the file `mac_ex.lst`. In this and subsequent listings the header information is omitted for clarity:

```
 1     000000         #define STATUS   3
 2     000000         #define Z        2
 3     000000         #define RP0      5
 4     000000         #define TEMP     0x0C
 5     000000         #define A        0x0D
 6    000000          #define B       0x0E
 7    000000          #define PORTB   0x06    ; PIC16C84 specific
                        ...
 8    000000          #define TRISB   0x06    ; address
 9    000000
10    000000
11    000000          ; macro to compare a register file and a literal
16    000000
17    000000
18    000000          ; macro that test the Z flag and jumps to <label> if
                           Z==1
19    000000
24    000000
25    000000          ; program that initializes PORTB and read and counts
                        some values
26    000000                  ORG     0
27    000000 2828             GOTO    main    ;reset vector
28    000002
29    000050                  ORG     0x50
30    000050
31    000050          main:
32    000050 1683             BSF     STATUS, RP0     ; Switch register
                                                        file bank
33    000052
34    000052 30FF             MOVLW   0xFF    ; Configure PORTB ...

35    000054 0086             MOVWF   TRISB   ; as input
36    000056
37    000056 1283             BCF     STATUS, RP0     ; Switch back to
                                                        bank 0
38    000058 018E             CLRF    B       ; init counter
39    00005A
40    00005A          loop:
41    00005A 0886             MOVF    PORTB, 0        ; Read PORTB
42    00005C 008D             MOVWF   A       ; Move the port value to
                                                A
43    00005E
44    00005E                  cmp_eq  A,0x81  ; Is A == 0x81? (sets Z
                        flags)
44.1  00005E 3081             MOVLW   0x81       ; load K
44.2  000060 020D             SUBWF   A, 0    ; Subtract K from register
```

```
file and store in W (and set Z accordingly)
44.3  000062                 ENDM
45    000062                 jnz     skip_count      ; No, skip
45.1  000062 1D03            BTFSS   STATUS, Z       ; check Z flag and
                                                           ...
45.2  000064 2834            GOTO    skip_count   ; branch if Z cleared,
                                                           else
                                                           skip
45.3  000066                 ENDM
46    000066 0A8E            INCF    B,1     ; Pattern found -->
                                              increment counter
47    000068
48    000068         skip_count:
49    000068                 cmp_eq  B,10    ; Have we found 10
                      occurrences of pattern?
49.1  000068 300A            MOVLW   10       ; load K
49.2  00006A 020E            SUBWF   B, 0    ; Subtract K from register
file and store in W (and set Z accordingly)
49.3  00006C                 ENDM
50    00006C 1D03            BTFSS   STATUS,Z        ; skip goto if
                                                      equal
51    00006E 282D            GOTO    loop
52    000070
53    000070 2838    self:   GOTO    self
54    000072
55    000072                 end
```

The macro-generated lines are identified with **.** (period) in the line number column.

LINKING THE PROGRAM

In order to be able to execute the program, the relocatable file produced by the assembler needs to be converted to an object code program with all the addresses resolved.

Linking the program using the Embedded Workbench
Link the file by choosing **Link** from the **Project** menu.

Linking the program using the command line
Run XLINK to produce code for debugging with the command:

```
xlink mac_ex -cPIC -r ↵
```

This generates a file `aout.d39`.

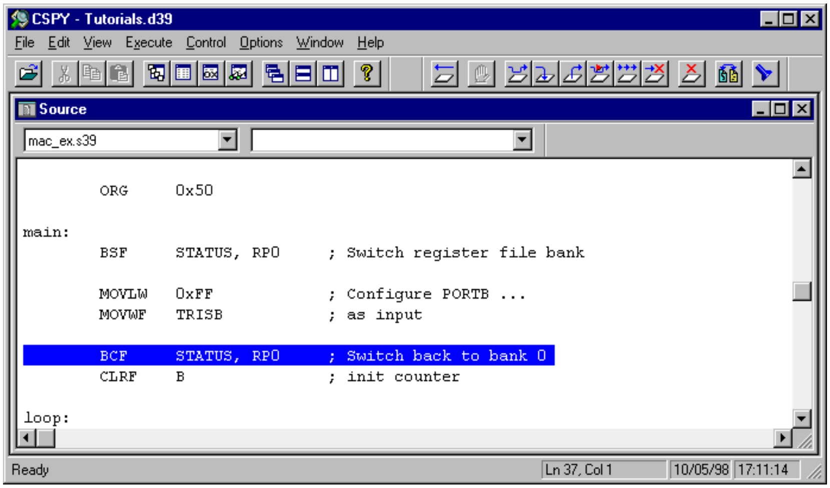
RUNNING THE PROGRAM

Running the program using the Embedded Workbench

To run the program using the IAR C-SPY Debugger choose **Debugger** from the **Project** menu and, as before, ignore the warning messages.

The C-SPY window will be displayed.

Choose **Step** from the **Execute** menu or click the **Step** button in the debug bar, to display the source program in the Source window. Repeatedly choose **Step**, or click the **Step** button to follow the execution of the program.



```
        ORG     0x50

main:
        BSF     STATUS, RP0     ; Switch register file bank

        MOVLW   0xFF            ; Configure PORTB ...
        MOVWF   TRISB           ; as input

        BCF     STATUS, RP0     ; Switch back to bank 0
        CLRF    B               ; init counter

loop:
```

Ready Ln 37, Col 1 10/05/98 17:11:14

Running the program using the command line

To run the example program, download `aout.d39` to the IAR C-SPY Debugger using the command line:

```
cwnn -d SPIC16 aout ↵
```

and follow the instructions above.

USING MODULES

The following example demonstrates how to create library modules and use the XLIB Librarian to maintain files of modules.

USING LIBRARIES

If you are working on a large project you will soon accumulate a collection of useful routines that are used by several of your programs.

To avoid having to assemble a routine each time the routine is needed, you can store such routines as object files; ie assembled but not linked.

A collection of routines in a single object file is referred to as a library. It is recommended that you use library files to create collections of related routines, such as graphical or math libraries.

You can use the XLIB Librarian to manipulate libraries; it allows you to:

- Change modules from `PROGRAM` to `LIBRARY` type, and vice versa.
- Add or remove modules from a library file.
- Change the names of entries.
- List module names, entry names, etc.

CREATING THE MAIN PROGRAM

The main program is as follows:

```
#define STATUS  3
#define Z       2
#define TEMP    0x0C
#define A0      0x0D
#define B0      0x0E
#define C0      0x0F


        NAME    main

        EXTERN  max

        ASEG    0
        GOTO    main

        RSEG    CODE
```

```
main:
        MOVLW   start        ; Move buffer start address
                             ...
        MOVWF   A0           ; to A0

        MOVLW   4            ; Move another constant ...
        MOVWF   B0           ; to B0

        CALL    max          ; return MAX(A0,B0) in C0

end_loop:
        GOTO    end_loop


        RSEG    DATA

start   DS 8                 ; Reserve space for a RAM
                             buffer

        END     main
```

This simply uses a routine called `max` to shift the contents of register C to the maximum value of the word registers A and B. The `EXTERN` directive declares `max` as an external symbol, to be resolved at link time.

Enter this program and save it as the file `main.s39`. Alternatively, a copy of the source file is provided on the installation disk.

CREATING THE LIBRARY ROUTINES

The following two library routines will form a separately assembled library. These consist of the `max` routine called by `main`, and a corresponding `min` routine, both of which operate on the contents of word registers A and B and return the result in C.

```
#define STATUS  3
#define Z       2
#define CARRY   0
#define TEMP    0x0C
#define A0      0x0D
#define B0      0x0E
#define C0      0x0F
```

```
        MODULE  max
        PUBLIC  max

        RSEG    CODE

max:
        MOVF    A0,0
        MOVWF   C0            ; Copy A0 to C0
        SUBWF   B0,0          ; WREG = B0 - A0
        BTFSS   STATUS, CARRY ; CARRY == 0 if B0<A0 -->
                              jump to end
        GOTO    end           ;

        MOVF    B0,0          ; Copy B0 ...
        MOVWF   C0            ; to C0

end:
        RETURN                ; C0 is now MAX(A0,B0)

        ENDMOD



        MODULE  MIN
        PUBLIC  min


        RSEG    CODE
min:
        MOVF    A0,0
        MOVWF   C0            ; Copy A0 to C0
        SUBWF   B0,0          ; WREG = B0 - A0
        BTFSC   STATUS, CARRY ; CARRY == 1 if B0<A0 -->
                              jump to end
        GOTO    end           ;

        MOVF    B0,0          ; Copy B0 ...
        MOVWF   C0            ; to C0
```

```
end:
        RETURN               ; C0 is now min(A0,B0)


        END
```

The routines are defined as library modules by the `MODULE` directives; these instruct the XLINK Linker to include them only if they are called by another module.

The `max` and `min` entry addresses are made public to other modules with a `PUBLIC` directive.

Save these modules in a source file called `maxmin.s39`. Alternatively, a copy of the source file is provided on the installation disk.

ASSEMBLING AND LINKING THE SOURCE FILES

Next you need to assemble both of the above source files.

Although it is possible to assemble both source files together, in a large project this would soon become very time-consuming. By assembling the library routines separately, changes to the main program only require reassembly of the main source file.

Assembling and linking using the Embedded Workbench

Create a project containing `main.s39` and `maxmin.s39`, as described for the previous tutorials.

To assemble and link both files choose **Make** from the **Project** menu.

Assembling and linking using the command line

To assemble the main program type:

```
apic main -r -L ↵
```

Similarly, to assemble the library routines type:

```
apic maxmin -r -L ↵
```

Assembling the files creates two relocatable files. You need to link these to produce a single executable object file containing the main program and the library routine it references, with all of the cross references resolved. In this case the only reference from one section to the other is the call of the `max` subroutine. The `min` routine is not used at all.

To link the files in a single step enter the following at the command line (on one line):

```
xlink main maxmin -cPIC -Z(CODE)CODE=50 -Z(DATA)DATA=20
-xsm -l main.map ↵
```

The following table explains the XLINK options which define the addresses for the code and data segments:

Parameter	*Description*
`-Z(CODE)CODE=50`	Specifies that the code segment is to be located at the hex address 50.
`-Z(DATA)DATA=20`	Specifies that the data segment is to be located at hex address 20.
`-xsm`	Requests a cross reference listing with segment and module maps in the optional list file.
`-l main.map`	Directs the listing output to `main.map`.

You can make the source visible from within C-SPY if you link with the `-r` option.

For more information about the XLINK options see the chapter *XLINK options reference*.

Viewing the listing

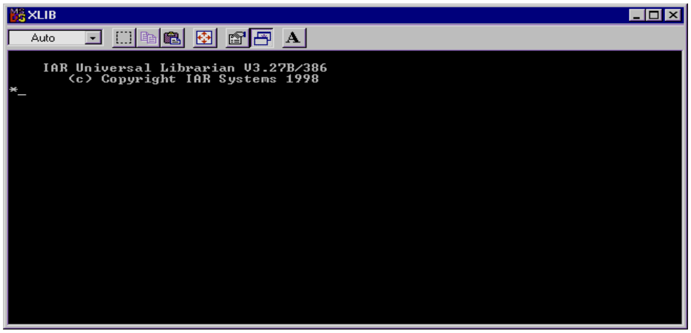
If you list the cross reference listing, `main.map`, you will see that the module created by XLINK includes the `main` program module and the `max` library module, but not the unused `min` library module.

USING THE XLIB LIBRARIAN

Once you have assembled and debugged a module intended for general use, like the `max` and `min` modules previously described, you can add them to a library using the XLIB Librarian.

Running the XLIB Librarian using the Embedded Workbench

Run the XLIB Librarian by choosing **Librarian** from the **Project** menu. The XLIB window will be displayed.



```
    IAR Universal Librarian V3.27B/386
       (c) Copyright IAR Systems 1998
*_
```

You can now enter XLIB commands at the * prompt.

Running the XLIB Librarian using the command line

Start the XLIB Librarian by typing:

```
XLIB ↵
```

XLIB runs in an interactive mode, and displays a * prompt for you to enter your command.

The first thing you need to do within XLIB is to define the CPU you are using:

```
DEFINE-CPU PIC ↵
```

Giving XLIB commands

Extract the modules you want from `maxmin.r39` into a library called `math.r39`. To do this enter the command:

```
FETCH-MODULES ↵
```

This prompts for the following arguments:

Prompt	*What you type*
Source file	`maxmin` ↵
Destination file	`math` ↵
Start module	↵ (uses the default, which is the first in the file).
End module	↵ (uses the default, which is the last in the file).

This creates the file `math.r39` which contains the code for the `max` and `min` routines.

You can confirm this by typing:

```
LIST-MODULES ↵
```

This prompts for the following arguments:

Prompt	*What you type*
Object file	`math`
List file	↵ (to use the screen).
Start module	↵ (to start from the first module).
End module	↵ (to end at the last module).

Finally, leave the librarian by typing:

```
EXIT ↵
```

or

```
QUIT ↵
```

You could use the same procedure to add further modules to the `math` library at any time.

ASSEMBLER OPTIONS SUMMARY

This chapter gives an alphabetical summary of the assembler 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 **APIC** and **General** options in the Embedded Workbench:

Code generation
List
Include
Command line
#define
#undef
Target

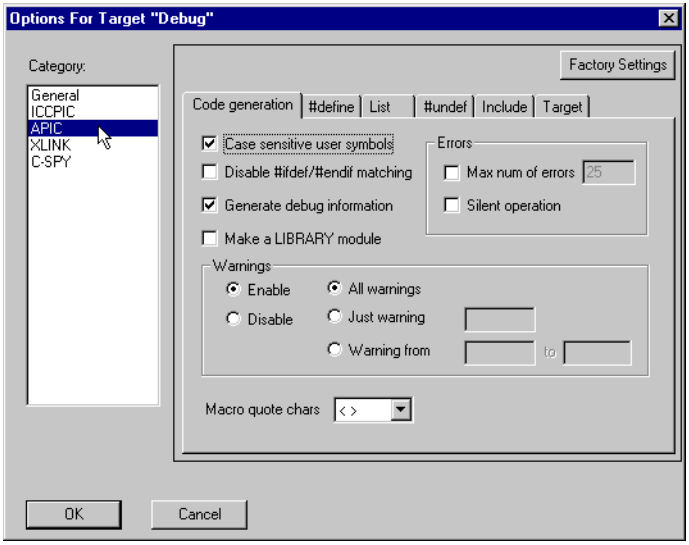
The section *Command line*, page 58, provides information about the options which are only available in the command line version.

For full reference about each of the assembler options, see the following chapter, *Assembler options reference*.

SETTING ASSEMBLER OPTIONS

Setting assembler options in the Embedded Workbench

To set assembler options in the Embedded Workbench choose **Options…** from the **Project** menu, and select **APIC** in the **Category** list to display the assembler options pages:



Options For Target "Debug"

Category:
- General
- ICCPIC
- APIC
- XLINK
- C-SPY

Factory Settings

Code generation | #define | List | #undef | Include | Target

☑ Case sensitive user symbols
☐ Disable #ifdef/#endif matching
☑ Generate debug information
☐ Make a LIBRARY module

Errors
☐ Max num of errors 25
☐ Silent operation

Warnings
● Enable ○ Disable
● All warnings
○ Just warning
○ Warning from to

Macro quote chars < >

OK Cancel

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

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

Setting assembler options from the command line

To set assembler options from the command line, you include them on the command line, after the `apic` command. For example, when assembling the source `power2`, to generate a listing to the default listing filename (`power2.lst`):

```
apic power2 -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`:

```
apic power2 -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:

```
apic power2 -Llist\ ↵
```

Specifying options using the ASMPIC environment variable
Options can also be specified using the `ASMPIC` environment variable. The assembler appends the value of this variable to every command line, so it provides a convenient method of specifying options that are required for every assembly.

For example, setting the following environment variable will always generate a listing to the file `temp.lst`:

```
ASMPIC=-l temp.lst
```

OPTIONS SUMMARY

The following is a summary of all the assembler options. For a full description of any option, see under the option's category name in the next chapter, *Assembler options reference*.

Command line	*Description*	*Section*
-B	Macro execution info	List
-b	Make a LIBRARY module	Code generation
-c{dmeao}	Conditional list	List
-D*symb*[=*xx*]	Define	#define
-d	Disable #ifdef/#endif matching	Code generation
-E*number*	Maximum number of errors	Command line
-f *filename*	Extend the command line	Command line
-G	Open standard input as source	Command line
-I*prefix*	Include paths	Include
-i	#included text	List
-L[*prefix*]	List to prefixed source name	List
-l *filename*	List to named file	List
-M*ab*	Macro quote chars	Code generation

Command line	*Description*	*Section*
-N	No header	List
-O*prefix*	Set object filename prefix	Command line
-o *filename*	Set object filename	Command line
-p*lines*	Lines/page	List
-r{en}	Generate debug information	Code generation
-S	Set silent operation	Command line
-s{+|-}	Case sensitive user symbols	Code generation
-T	Active lines only	List
-t*n*	Tab spacing	List
-U*symb*	Undefine symbol	#undef
-v{14|16|m|h}	Processor setup file	Target
-w[*string*][s]	Disable warnings	Code generation
-x{DI2}	Include cross reference	List

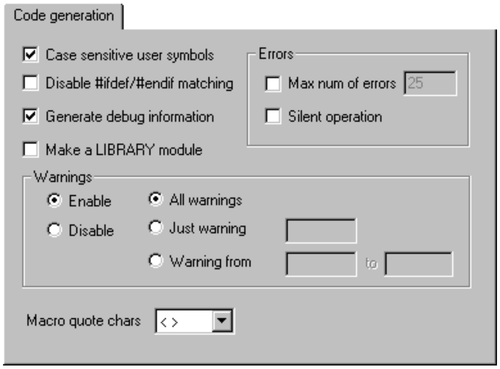
ASSEMBLER OPTIONS REFERENCE

This chapter gives detailed information on each of the PICmicro™ Assembler options, divided into functional categories.

CODE GENERATION

These options control the assembler's code generation.

Embedded Workbench



Code generation

☑ Case sensitive user symbols
□ Disable #ifdef/#endif matching
☑ Generate debug information
□ Make a LIBRARY module

Errors
□ Max num of errors 25
□ Silent operation

Warnings
● Enable
○ Disable
● All warnings
○ Just warning
○ Warning from to

Macro quote chars <>

Command line

`-s{+|-}`	Case sensitive user symbols.
`-d`	Disable `#ifdef/#endif` matching.
`-Mab`	Macro quote chars.
`-w[string][s]`	Disable warnings.
`-r{en}`	Generate debug information.
`-b`	Make a LIBRARY module.

CASE SENSITIVE USER SYMBOLS (-s)

Syntax: `-s{+|-}`

Sets whether the assembler is sensitive to the case of user symbols:

Workbench option	*Command line option*
Case sensitive user symbols	`-s+`
Case insensitive user symbols	`-s-`

By default, case sensitivity is on. This means that, for example, `LABEL` and `label` refer to different symbols. You can choose **Case insensitive user symbols** (`-s-`) to turn case sensitivity off, in which case `LABEL` and `label` will refer to the same symbol.

DISABLE #IFDEF/#ENDIF MATCHING (-d)

Syntax: `-d`

Allows unmatched `#ifdef` … `#endif` statements to be used without causing an error.

The checks for `#ifdef` … `#endif` matching are performed for each module, and a `#endif` outside modules will therefore normally generate an error message. Use this option to turn checking off.

This allows you to write constructs such as:

```
#ifdef Version1
   MODULE M1
   NOP
   ENDMOD
#endif
   MODULE M2
   .
   .
   .
   etc
```

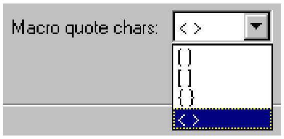
MACRO QUOTE CHARS (-M)

Syntax: `-M`*ab*

Sets the characters used for the left and right quotes of each macro argument to `a` and `b` respectively.

By default, the characters are < and >. The **Macro quote chars** (`-M`) option allows you to change the quote characters to suit an alternative convention or simply to allow a macro argument to contain < or > themselves.

You can select one of four types of brackets from the drop-down list as the macro quote characters:



Macro quote chars: < >

( )
[ ]
{ }
< >

For example, using the option:

```
-M[]
```

in the source you would write, for example:

```
print [>]
```

to call a macro print with > as the argument.

DISABLE WARNINGS (-w)

Syntax: `-w[`*`string`*`][s]`

Disables warnings.

By default, the assembler displays a warning message when it finds an element of the source which is legal, but probably due to a programming error (see *Assembler diagnostics* for details). The **Disable warnings** (`-w`) option with no range disables all warnings. The **Disable warnings** (`-w`) option with a range performs the following:

Range	*Effect*
+	Enables all warnings.
-	Disables all warnings.
+*n*	Enables just warning *n*.
-*n*	Disables just warning *n*.
+*m*-*n*	Enables warnings *m* to *n*.
-*m*-*n*	Disables warnings *m* to *n*.

By default, the assembler generates exit code 0 for warnings. Use the **Generate exit code 1 for warnings** (`-ws`) option to generate exit code 1 if a warning message is produced.

For example, to disable just warning 0 (unreferenced label), you might use:

```
apic prog -w-0 ↵
```

or to disable warnings 0 to 8:

```
apic prog -w-0-8 ↵
```

Only one **Disable warnings** (`-w`) option may be used on the command line.

GENERATE DEBUG INFORMATION (-r)

Syntax: `-r{en}`

Enables the inclusion of information that allows a debugger (such as C-SPY) to be used on the program.

By default, the assembler does not generate debug information, to reduce the size and link time of the object file. You must use the **Generate debug information** (`-r`) option if you want to use a debugger with the program.

Using the **Source files embedded into the object file** (`e`) modifier includes the full source file into the object file.

Using the **No source code option** (`n`) modifier will generate an object file without source information; symbol information will be available.

MAKE A LIBRARY MODULE (-b)

Syntax: `-b`

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

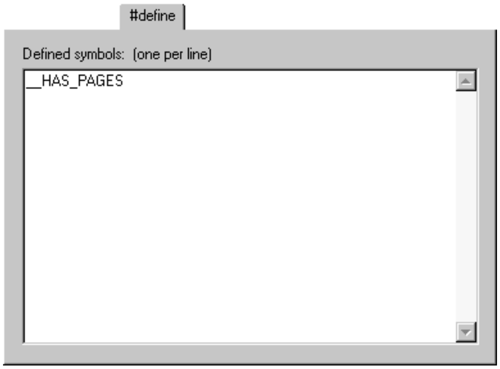
By default, the assembler produces a program module ready to be linked with XLIB. You use the **Make a LIBRARY module** (`-b`) option if you want it to make a library module for use with XLIB.

If the `NAME` directive is used in the source (to specify the name of the program module), the **Make a LIBRARY module** (`-b`) option is ignored, and the assembler produces a program module regardless of the **Make a LIBRARY module** (`-b`) option.

#define

This option allows you to define symbols.

Embedded Workbench



Defined symbols: (one per line)

```
__HAS_PAGES
```

Command line

-D*symb*[=*xx*] Define symbol.

#DEFINE (-D)

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

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

The **#define** (`-D`) option allows a value or choice that would otherwise have to be specified in the source file to be specified more conveniently on the command line. For example, you could arrange your source to produce either the test or production version of your program dependent 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:       apic prog
test version:             apic prog -Dtestver
```

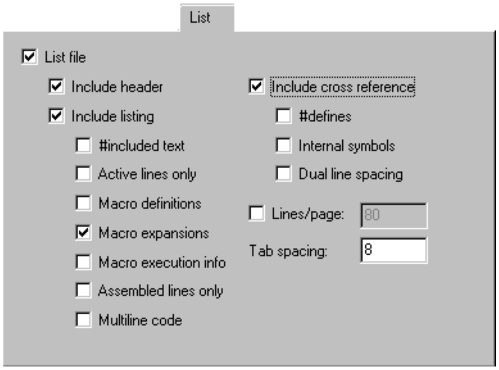
Alternatively, your source might use a variable that you need to change often. You would leave the variable undefined in the source, and use `-D` to specify the value on the command line; for example:

```
apic prog -Dframerate=3 ↵
```

LIST

The **List** options are used to cause the assembler to generate a listing, to select the contents of the listing, and to generate other listing-type output.

Embedded Workbench



List

☑ List file

- ☑ Include header
- ☑ Include listing
  - □ #included text
  - □ Active lines only
  - □ Macro definitions
  - ☑ Macro expansions
  - □ Macro execution info
  - □ Assembled lines only
  - □ Multiline code

☑ Include cross reference

- □ #defines
- □ Internal symbols
- □ Dual line spacing

□ Lines/page: 80

Tab spacing: 8

Command line

`-l` *`filename`*	List to named file.
`-L[`*`prefix`*`]`	List to prefixed source name.
`-N`	No header.
`-i`	#included text.
`-T`	Active lines only.
`-c{dmeao}`	Conditional list.
`-B`	Macro execution info.
`-x{DI2}`	Include cross reference.
`-p`*`lines`*	Lines/page.
`-t`*`n`*	Tab spacing.

LIST FILE

Causes the assembler to generate a listing and send it to the file `sourcename.lst`.

When **List file** is selected the following list options become available:

Option	*Description*
Include header	Includes a header in the listing.
Include listing	Includes the body of the listing.

Selecting **Include listing** makes the following options available:

Option	*Description*
#included text	Includes `#include` files in the listing.
Active lines only	Includes only active lines in the listing.
Macro definitions	Includes macro definitions in the listing.
Macro expansions	Includes macro expansions in the listing.
Macro execution info	Prints macro execution information on every call of a macro.
Assembled lines only	Lists only assembled lines.
Multiline code	Lists the code generated by directives on several lines if necessary.

List to named file (-l)

Syntax: `-l filename`

Causes the assembler to generate a listing and send it to the named file. If no extension is specified, `.lst` is used. Note that you must include a space before the filename.

By default, the assembler does not generate a listing. The `-l` option turns on listing, and directs it to a specific file. To just turn on listing to the default filename, use the `-L` option instead.

List to prefixed source name (-L)

Syntax: `-L[`*`prefix`*`]`

Causes the assembler to generate a listing and send it to the file *`prefixsourcename`*`.lst`. Note that you must not include a space before the prefix.

By default, the assembler does not generate a listing. To simply generate a listing, you use the `-L` option without a prefix. The listing is sent to the file with the same name as the source, but extension `.lst`.

The `-L` option lets you specify a prefix, for example to direct the list file to a subdirectory:

```
apic prog -Llist\ ↵
```

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

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

NO HEADER (-N)

Syntax: `-N`

Disables the header normally printed in the listing.

#INCLUDED TEXT (-i)

Syntax: `-i`

Includes `#include` files in the listing.

By default, the assembler does not list `#include` file lines since these are often from standard files that would waste space in the listing. The **#included text** (`-i`) option allows you to list `#include` files should you so require.

ACTIVE LINES ONLY (-T)

Syntax: `-T`

Includes only active lines, for example not those in false `#if` blocks. By default, all lines are listed.

This option is useful for reducing the size of listings by eliminating lines that do not generate or affect code.

CONDITIONAL LIST (-c)

Syntax: `-c{dmeao}`

Sets one or more of the following:

Option	*Command line*
Disable listing	`d`
Macro definitions	`m`
No macro expansions	`e`
Assembled lines only	`a`
Multiline code	`o`

MACRO EXECUTION INFO (-B)

Syntax: `-B`

Causes the assembler to print macro execution information to the standard output stream on every call of a macro. The information consists of:

- The name of the macro.
- The definition of the macro.
- The arguments to the macro.
- The expanded text of the macro.

INCLUDE CROSS-REFERENCE (-x)

Syntax: `-x{DI2}`

Causes the assembler to generate a cross-reference list at the end of the listing. See the chapter *Assembler file formats* for details.

The following options are available:

Option	*Command line*
#defines	D
Internal symbols	I
Dual line spacing	2

LINES/PAGE (-p)

Syntax: -p*lines*

Sets the number of lines per page to *lines*, which must be in the range 10 to 150.

TAB SPACING (-t)

Syntax: -t*n*

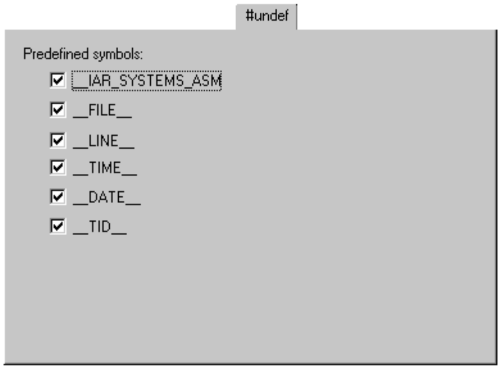
Sets the number of character positions per tab stop to *n*, which must be in the range 2 to 9.

By default, the assembler sets eight character positions per tab stop.

#undef

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

Embedded Workbench



#undef

Predefined symbols:

☑ __IAR_SYSTEMS_ASM
☑ __FILE__
☑ __LINE__
☑ __TIME__
☑ __DATE__
☑ __TID__

Command line

-U*symb*	Undefine symbol.

#UNDEF (-U)

Syntax: -U*symb*

Undefines the symbol *symb*.

By default, the assembler provides certain predefined symbols; see *Predefined symbols*, page 68. The **#undef** (-U) option allows you to undefine such a predefined symbol to make its name available for your own use through a subsequent **#define** (-D) option or source definition.

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

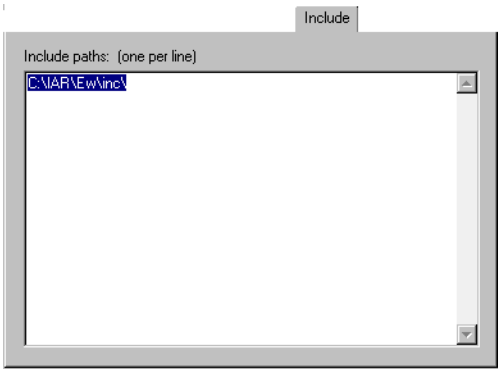
To use the name of the predefined symbol __TIME__ for your own purposes, you could undefine it with:

```
apic prog -U __TIME__ ↵
```

INCLUDE

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

Embedded Workbench



Include paths: (one per line)

C:\IAR\Ew\inc\

Command line

-I*prefix*	Include paths.

INCLUDE PATHS (-I)

Syntax: -I*prefix*

Adds the #include file search prefix *prefix*.

By default, the assembler searches for `#include` files only in the current working directory and in the paths specified in the `APIC_INC` environment variable. The **Include paths** (`-I`) option allows you to give the assembler the names of directories which it will also search if it fails to find the file in the current working directory.

For example, using the options:

```
-Ic:\global\ -Ic:\thisproj\headers\
```

and then writing:

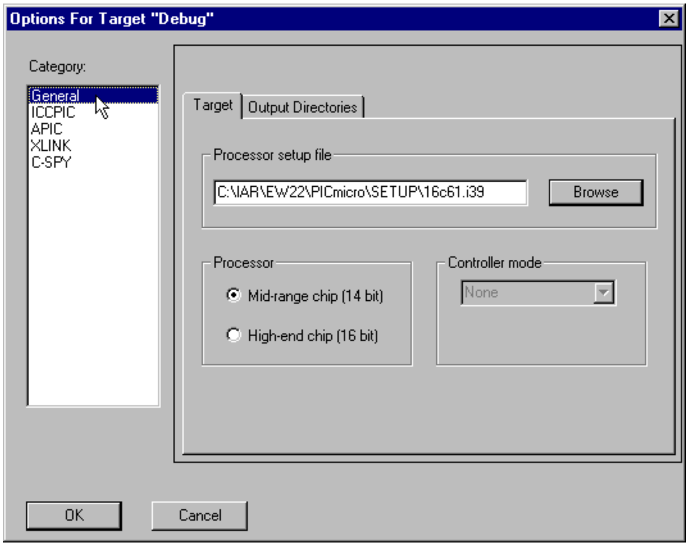
```
#include "asmlib.hdr" ↵
```

in the source, will make the assembler search first for the file `asmlib.hdr`, then for the file `c:\global\asmlib.hdr`, and finally for the file `c:\thisproj\headers\asmlib.hdr`.

TARGET

The **Target** option specifies the processor configuration. The target should normally be set under the **General** category but may be overridden by the **Target** options for **APIC.**

Embedded Workbench



Options For Target "Debug"

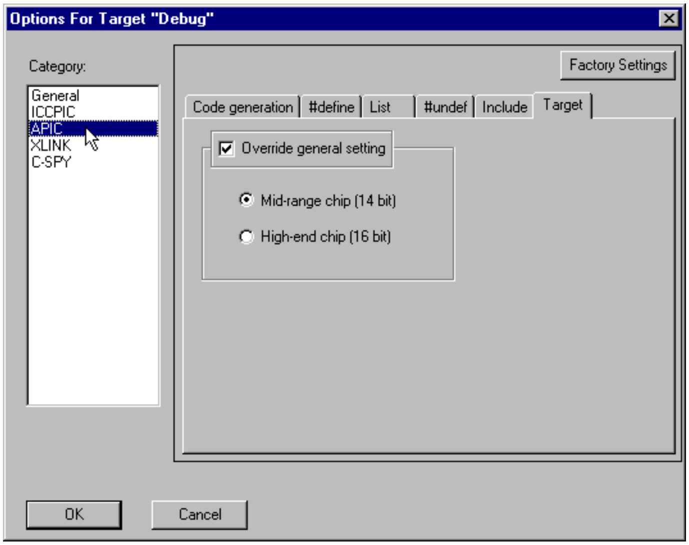
Category: General (selected), ICCPIC, APIC, XLINK, C-SPY

Target | Output Directories

Processor setup file: C:\IAR\EW22\PICmicro\SETUP\16c61.i39

Processor: ● Mid-range chip (14 bit) ○ High-end chip (16 bit)

Controller mode: None



☑ Override general setting

● Mid-range chip (14 bit)

○ High-end chip (16 bit)

Command line

`-v[14|16|m|h]` Processor.

PROCESSOR SETUP FILE (-v)

Syntax: `-v[14|16|m|h]`

Selects the assembler processor configuration from one of:

Processor option	*Command line*	*Processors supported*
Mid-range chip (14-bit)	`-v14` or `-vm`	16C61, 16C62A, 16C621, 16C622, 16C63, 16C64A, 16C641, 16C642, 16C65A, 16C66, 16C661, 16C662, 16C67, 16C71, 16C710, 16C711, 16C715, 16C72, 16C73, 16C74A, 16C76, 16C77, 16F84, 16C923, 16C924.
High-end chip (16-bit)	`-v16` or `-vh`	17C42A, 17C43, 17C44, 17C752, 17C756.

If no **Processor setup file** (`-v`) option is specified, the assembler uses `-v14` by default.

COMMAND LINE

The following additional options are available from the command line.

`-E`*`number`*	Maximum number of errors.
`-f` *`filename`*	Extend the command line.
`-G`	Open standard input as source.
`-O`*`prefix`*	Set object filename prefix.
`-o` *`filename`*	Set object filename.
`-S`	Set silent operation.

MAXIMUM NUMBER OF ERRORS (-E)

Syntax: `-E`*`number`*

Sets the maximum number of errors the assembler reports.

By default, the maximum number is 100. The (`-E`) option allows you to decrease or increase this number, for example, to see more errors in a single assembly.

EXTEND THE COMMAND LINE (-f)

Syntax: `-f` *`filename`*

Extends the command line with text read from the file *`filename`*`.xcl`. Note that there must be a space between the option itself and the filename.

The `-f` option is particularly useful where there are a large number of options which are more conveniently placed in a file than on the command line itself. For example, to run the assembler with further options taken from the file `asmopt.xcl`, you might use:

```
apic prog -f asmopt ↵
```

OPEN STANDARD INPUT AS SOURCE (-G)

Syntax: `-G`

Causes the assembler to read the source from the standard input stream, rather than a specified source file.

When `-G` is used, no source filename may be specified.

SET OBJECT FILENAME PREFIX (-O)

Syntax: `-O`*`prefix`*

Set the prefix to be used on the filename of the object. Note that you must not include a space before the prefix.

By default the prefix is null, so the object filename corresponds to the source filename (unless `-o` is used). The `-O` option lets you specify a prefix, for example to direct the object file to a subdirectory:

```
apic prog -Oobj\ ↵
```

This sends the object to `obj\prog.r39` rather than the default `prog.r39`.

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

SET OBJECT FILENAME (-o)

Syntax: `-o filename`

Sets the filename to be used for the object. Note that you must include a space before the filename. If no extension is specified, `.r39` is used.

By default the assembler uses the source filename with the extension changed to `.r39`. The `-o` option lets you use an alternative filename for the object.

For example, the following command puts the object to the file `obj.r39` instead of the default `prog.r39`:

```
apic prog -o obj ↵
```

Note that you must include a space between the option itself and the filename.

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

SET SILENT OPERATION (-S)

Syntax: `-S`

Causes the assembler to operate without sending any messages to the standard output stream.

By default, the assembler sends various inessential messages to the terminal via the standard output stream. You can use the `-S` option to prevent this, reducing the amount of screen clutter. The assembler sends error and warning messages to the error output stream, so they appear on the terminal regardless of this setting.

ENVIRONMENT VARIABLES

This chapter gives information about customizing your assembler configuration, using the command line options.

ASSEMBLER ENVIRONMENT VARIABLES

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

Environment variable	*Description*

`ASMPIC`	Specifies command line options; for example:
	`set ASMPIC=-v16`
	Note: In Windows 95 and DOS only one equal sign (=) is allowed on the same line. The # sign can be used instead, for example:
	`set ASMPIC=-DBUFSIZE#4`

`APIC_INC`	Specifies directories to search for include files; for example: `set APIC_INC=c:\myinc\`

XLINK ENVIRONMENT VARIABLES

The following environment variables can be used by XLINK:

Environment variable	*Description*
`XLINK_COLUMNS`	Sets the number of columns per line.
`XLINK_CPU`	Sets the target CPU type.
`XLINK_DFLTDIR`	Sets a path to a default directory for object files.
`XLINK_ENVPAR`	Creates a default XLINK command line.
`XLINK_FORMAT`	Sets the output format.
`XLINK_MEMORY`	Specifies whether XLINK is file-bound (0) or memory-bound (not 0).
`XLINK_PAGE`	Sets the number of lines per page.

Environment variable	*Description*
XLINK_TFILE	Specifies the temporary file.

XLIB ENVIRONMENT VARIABLES

The following environment variables can be used by XLIB:

Environment variable	*Description*
XLIB_COLUMNS	Sets the number of columns.
XLIB_CPU	Sets the CPU type.
XLIB_PAGE	Sets the number of lines per page.
XLIB_SCROLL_BREAK	Sets the scroll pause in number of lines.

ASSEMBLER FILE FORMATS

This chapter describes the source format for the PICmicro™ Assembler, and the format of assembler listings.

SOURCE FORMAT

The format of an assembler source line is as follows:

`[label [:]] operation [operands] [; comment] [\]`

where the components are as follows:

label	A label, which is assigned the value and type of the current location counter (PLC). The `:` (colon) is optional if the label starts in the first column.
operation	An assembler instruction or directive. This must not start in the first column.
operands	One or more operands, separated by commas.
comment	A comment, preceded by a `;` (semi-colon).
\	Line continuation character.

The fields can be separated by spaces or tabs.

A source line may not exceed 255 characters.

Tab characters (ASCII 09H), are expanded according to the most common practice; ie to columns 8, 16, 24 etc.

EXPRESSIONS AND OPERATORS

Expressions can consist of operands and operators.

The assembler will accept a wide range of expressions, including both arithmetic and logical operations. All operators use 32-bit two's complement integers, and range checking is only performed when a value is used to generate code.

Expressions are evaluated from left to right, unless this order is overridden by the priority of operators.

The valid operands in an expression are:

- User-defined symbols and labels.
- Constants, excluding floating point constants.
- The location counter symbol, `$`.

These are described in greater detail in the following sections.

The valid operators are described in the chapters *Assembler operator summary* and *Assembler operator reference*.

TRUE AND FALSE

In expressions a zero value is considered FALSE, and a non-zero value is considered TRUE.

Conditional expressions return the value 0 for FALSE and 1 for TRUE.

USING SYMBOLS IN RELOCATABLE EXPRESSIONS

Expressions that include symbols in relocatable segments cannot be resolved at assembly time, because they depend on where the segments are located by XLINK.

Such expressions are evaluated and resolved at link time, by XLINK. There are no restrictions on the expression; any operator can be used on symbols from any segment, or any combination of segments. For example, a program could define the segments `DATA` and `CODE` as follows:

```
          EXTERN    third
          RSEG      DATA
first     DS        5
second    DS        3
          ENDMOD
          RSEG      CODE
start     …
```

Then in segment `CODE` the following instructions are legal:

```
   INCF   first+7,1
   INCF   first-7,1
   INCF   7+first,1
   INCF   (first/second)*third,1
```

SYMBOLS

User-defined symbols can be up to 255 characters long, and all characters are significant.

Symbols must begin with a letter, a–z or A–Z, ? (question mark), or _ (underscore). Symbols can include the digits 0–9 and $ (dollar). For user-defined symbols case is significant. For built-in symbols like instructions, registers, operators, and directives case is insignificant.

LABELS

Symbols used for memory locations are referred to as labels.

Location counter

The location counter is called `$`. For example:

```
GOTO             A:$       ; Loop forever
```

INTEGER CONSTANTS

Since all IAR Systems Assemblers use 32-bit two's complement internal arithmetic, integers have a (signed) range from -2147483648 to 2147483647.

Constants are written as a sequence of digits with an optional `-` (minus) sign in front to indicate a negative number.

Commas and decimal points are not permitted.

The following number bases are supported:

Hexadecimal

Hexadecimal numbers can be written in any of the following formats:

Format	*Example*	*Value*
0x*hex-digits*	0x20	32 in decimal.
*hex-digits*H	20H	32 in decimal.
H'hex-digits'	H'20'	32 in decimal.

* Note that if the first digit is A-F, a leading zero must be included; for example, 0AH.

Octal

Octal numbers can be written as follows:

Format	*Example*	*Value*
*octal-digits*Q	10Q	8 in decimal.
O’octal-digits’	O’10’	8 in decimal.

Decimal

Decimal numbers can be written as follows:

Format	*Example*	*Value*
digits	123D	123 in decimal.
D’123’	D’123’	123 in decimal.

Binary

Binary numbers can be written as follows:

Format	*Example*	*Value*
*binary-digits*B	10B	2 in decimal.
B’binary-digits’	B’10’	2 in decimal.

ASCII CHARACTER CONSTANTS

ASCII constants can consist of between zero and more characters enclosed in single or double quotes. Only printable characters and spaces may be used in ASCII strings. If the quote character itself is to be accessed, two consecutive quotes must be used:

Format	*Value*
’ABCD’	ABCD (four characters).
"ABCD"	ABCD’\0’ (five characters the last ASCII null).
’A’’B’	A’B
’A’’’	A’
’’’’ (4 quotes)	’
’’ (2 quotes)	Empty string (no value).
""	Empty string (an ASCII null character).

Format	*Value*
`\'`	'
`\\`	\

REAL NUMBER CONSTANTS

The PICmicro™ Assembler will accept real numbers as constants and convert them into IEEE single-precision (signed 32-bit) real number format.

Floating point numbers can be written in the format:

`[+|-][`*`digits`*`].[`*`digits`*`][{E|e}[+|-]`*`digits`*`]`

Some valid examples are as follows:

Format	*Value*
10.23	1.023×10^{1}
1.23456E-24	1.23456×10^{-24}
1.0E3	1.0×10^{3}

No spaces or tabs are allowed in real constants.

Note that floating-point numbers will not give meaningful results when used in expressions.

PREDEFINED SYMBOLS

The PICmicro™ Assembler defines a set of symbols for use in assembler source files. The symbols provide information about the current assembly, allowing you to test them in preprocessor directives or include them in the assembled code.

<table>
<tr><th><em>Symbol</em></th><th><em>Value</em></th></tr>
<tr><td><code>__DATE__</code></td><td>Current date in <code>Mmm dd yyyy</code> format.</td></tr>
<tr><td><code>__FILE__</code></td><td>Current source filename.</td></tr>
<tr><td><code>__IAR_SYSTEMS_ASM__</code></td><td>IAR assembler identifier.</td></tr>
<tr><td><code>__LINE__</code></td><td>Current source line number.</td></tr>
<tr><td><code>__TID__</code></td><td>Target identity, consisting of two bytes. The high byte is the target identity, which is <code>0x27</code> for the PICmicro™. The low byte is the processor option. The possible values are therefore as follows:
<table>
<tr><th><em>Processor option</em></th><th><em>Value</em></th></tr>
<tr><td><code>-v14</code> and <code>-vm</code></td><td><code>0x2700</code></td></tr>
<tr><td><code>-v16</code> and <code>-vh</code></td><td><code>0x2710</code></td></tr>
</table></td></tr>
<tr><td><code>__TIME__</code></td><td>Current time in <code>hh:mm:ss</code> format.</td></tr>
<tr><td><code>__VER__</code></td><td>Version number in integer format; for example, version 4.17 is returned as 417.</td></tr>
</table>

Including symbol values in code

To include a symbol value in the code, you use the symbol in one of the data-definition directives.

For example, to include the time and date of assembly as a string for the program to display:

```
timdat: DT      __TIME__,",",__DATE__,0; time and date
```

Testing symbols for conditional assembly

To test a symbol at assembly-time, you use one of the conditional assembly directives.

For example, in a source file written for use on any one of the PICmicro™ family members, you may want to assemble appropriate code for a specific processor. You could do this using the __TID__ symbol as follows:

```
#define TARGET ((__TID__) >> 4) & 0x0F)
#if (TARGET==1)
.
.
.
#else
.
.
.
#endif
```

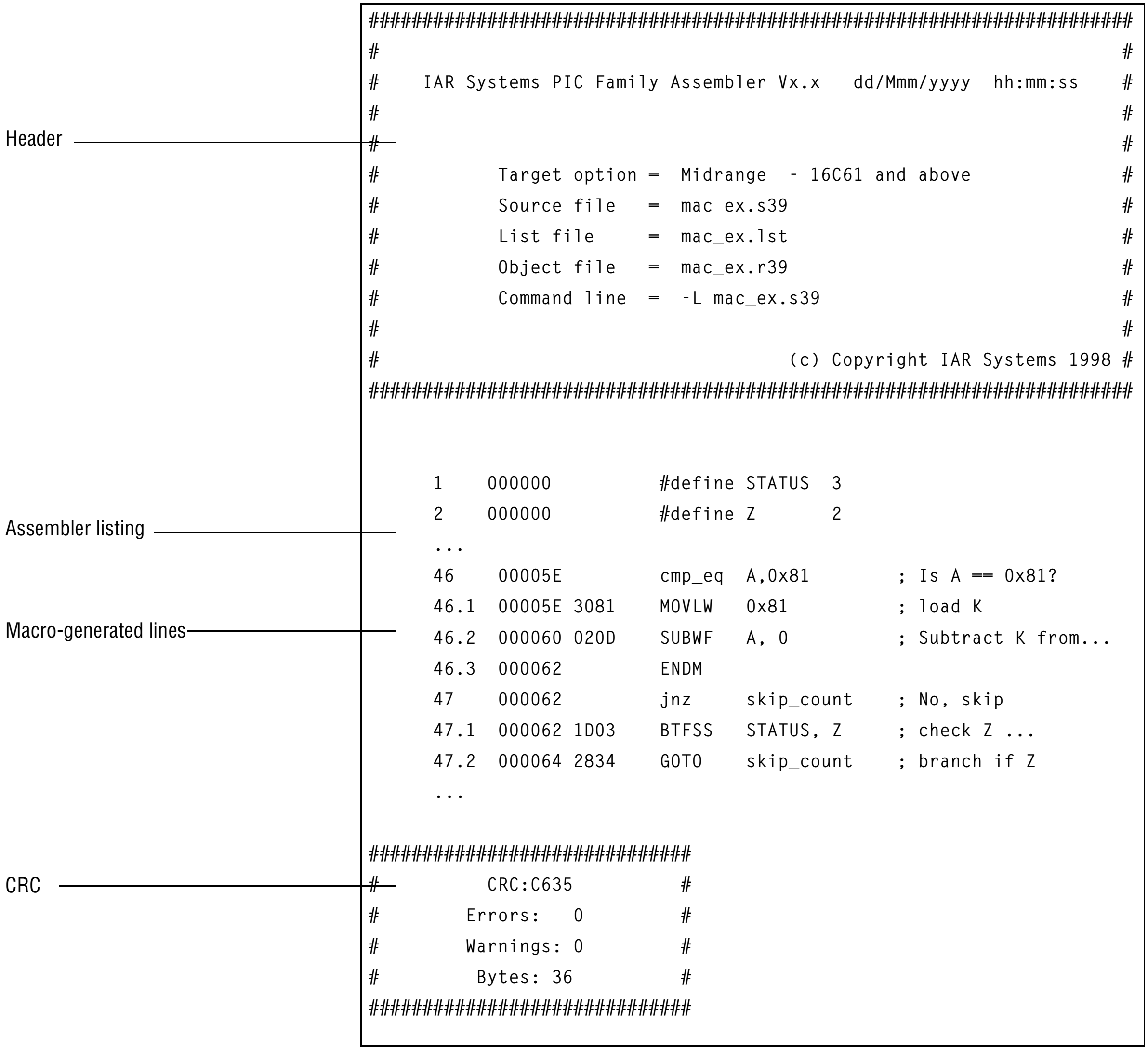
REGISTER SYMBOLS

Definitions of the symbols for registers, including standard SFRs, are supplied in the following files:

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

LISTING FORMAT

The format of the PICmicro™ Assembler listing is as follows:

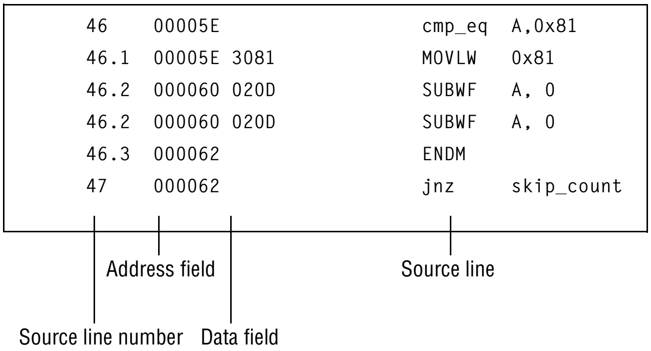

Header
Assembler listing
Macro-generated lines
CRC

```
##########################################################################
#                                                                        #
#    IAR Systems PIC Family Assembler Vx.x   dd/Mmm/yyyy  hh:mm:ss    #
#                                                                        #
#                                                                        #
#           Target option =  Midrange  - 16C61 and above                 #
#           Source file   =  mac_ex.s39                                  #
#           List file     =  mac_ex.lst                                  #
#           Object file   =  mac_ex.r39                                  #
#           Command line  =  -L mac_ex.s39                               #
#                                                                        #
#                                       (c) Copyright IAR Systems 1998 #
##########################################################################


      1    000000          #define STATUS  3
      2    000000          #define Z       2
      ...
      46    00005E         cmp_eq  A,0x81        ; Is A == 0x81?
      46.1  00005E 3081    MOVLW   0x81          ; load K
      46.2  000060 020D    SUBWF   A, 0          ; Subtract K from...
      46.3  000062         ENDM
      47    000062         jnz     skip_count    ; No, skip
      47.1  000062 1D03    BTFSS   STATUS, Z     ; check Z ...
      47.2  000064 2834    GOTO    skip_count    ; branch if Z
      ...

##############################
#        CRC:C635          #
#       Errors:   0        #
#       Warnings: 0        #
#        Bytes: 36         #
##############################
```

Assembly list information is put into four fields:



```
46    00005E               cmp_eq  A,0x81
46.1  00005E 3081          MOVLW   0x81
46.2  000060 020D          SUBWF   A, 0
46.2  000060 020D          SUBWF   A, 0
46.3  000062               ENDM
47    000062               jnz     skip_count
```

Source line number
The line number in the source file.

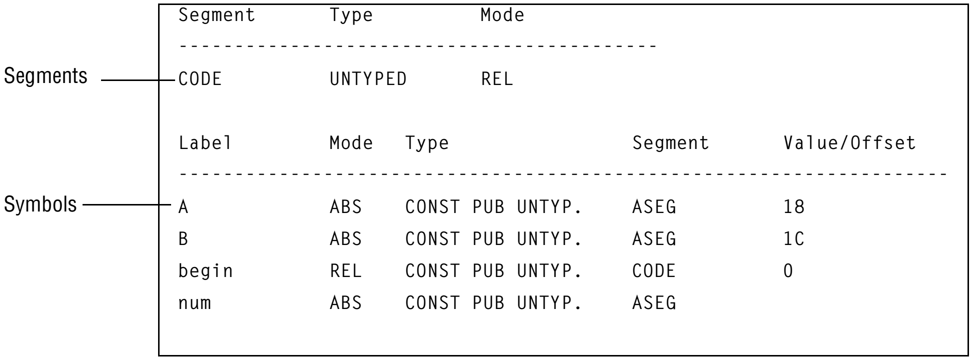
Lines generated by macros will, if listed, have a `.` (period) in the source line number field.

Address and data fields
These are always listed in hexadecimal notation.

Source line
Lists the source file line.

SYMBOL AND CROSS-REFERENCE TABLE

If the `LSTXRF+` directive has been included, or the `-x` command line option has been specified, the following symbol and cross-reference table is produced:



```
Segment       Type          Mode
------------------------------------------
CODE          UNTYPED       REL

Label         Mode   Type                Segment       Value/Offset
-------------------------------------------------------------------
A             ABS    CONST PUB UNTYP.    ASEG          18
B             ABS    CONST PUB UNTYP.    ASEG          1C
begin         REL    CONST PUB UNTYP.    CODE          0
num           ABS    CONST PUB UNTYP.    ASEG
```

Segments
Symbols

The following information is provided for each symbol in the table:

Information	*Description*
Label	The label's user-defined name.
Mode	`ABS` (Absolute), or `REL` (Relative).
Type	The label's type.
Segment	The name of the segment this label is defined relative to.
Value/Offset	The value (address) of the label within the current module, relative to the beginning of the current segment.

OUTPUT FORMATS

The relocatable and absolute output is in the same format for all assemblers, because object code is always meant to be processed by the IAR Systems XLINK Linker.

In absolute formats the output from XLINK is, however, normally compatible with the chip vendor's debugger programs (monitors), as well as with PROM programmers and stand-alone emulators from independent sources.

ASSEMBLER OPERATOR SUMMARY

This chapter summarizes the assembler operators, classified according to their precedence. A full alphabetical reference list of operators is given in the next chapter, *Assembler operator reference*.

PRECEDENCE OF OPERATORS

Each operator has a precedence number assigned to it which determines the order in which the operator and its operands are evaluated. The precedence numbers range from 1 (the highest precedence, ie first evaluated) to 7 (the lowest precedence, ie last evaluated).

The following rules determine how expressions are evaluated:

- The highest precedence (lowest number) operators are evaluated first, then the next highest precedence operators, and so on until the lowest precedence operators are evaluated.
- Operators of equal precedence are evaluated from left to right in the expression.
- Parentheses (and) can be used to group operators and operands and to control the order in which the expressions are evaluated. For example, the following expression evaluates to 1:

  ```
  7/(1+(2*3))
  ```

The following tables give a summary of the operators, in order of priority. Synonyms, where available, are shown in brackets after the operator name.

UNARY OPERATORS – 1

`+`	Unary plus.
`-`	Unary minus.
`NOT (!)`	Logical NOT.
`LOW`	Low byte.
`HIGH`	High byte.
`BYTE2`	Second byte.
`BYTE3`	Third byte.
`LWRD (LSW)`	Low word.
`HWRD (MSW)`	High word.
`DATE`	Current date/time.
`SFB`	Segment begin.
`SFE`	Segment end.
`SIZEOF`	Segment size.
`BINNOT (~)`	Bitwise NOT.

MULTIPLICATIVE ARITHMETIC OPERATORS – 2

`*`	Multiplication.
`/`	Division.
`MOD (%)`	Modulo.

ADDITIVE ARITHMETIC OPERATORS – 3

`+`	Addition.
`-`	Subtraction.

SHIFT OPERATORS – 4

`SHR (>>)`	Logical shift right.
`SHL (<<)`	Logical shift left.

AND OPERATORS – 5

`AND (&&)`	Logical AND.
`BINAND (&)`	Bitwise AND.

OR OPERATORS – 6

`OR (||)`	Logical OR.
`XOR`	Logical exclusive OR.
`BINOR (|)`	Bitwise OR.
`BINXOR (^)`	Bitwise exclusive OR.

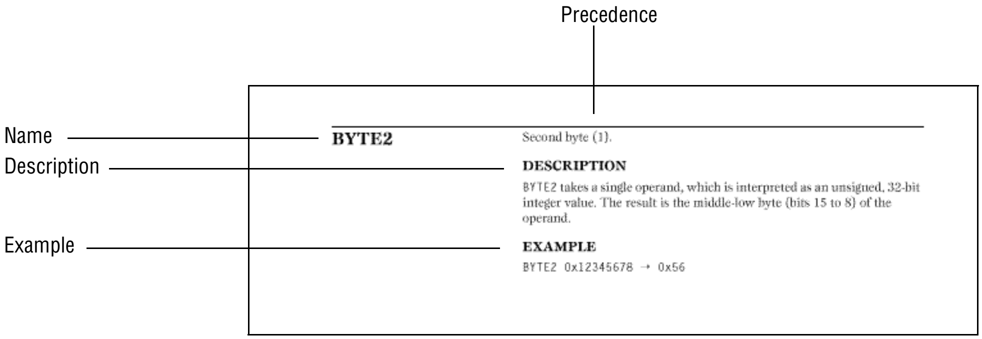
COMPARISON OPERATORS – 7

`EQ (=, ==)`	Equal.
`NE (<>, !=)`	Not equal.
`GT (>)`	Greater than.
`LT (<)`	Less than.
`UGT`	Unsigned greater than.
`ULT`	Unsigned less than.
`GE (>=)`	Greater than or equal.
`LE (<=)`	Less than or equal.

ASSEMBLER OPERATOR REFERENCE

This section gives an alphabetical list of the assembler operators with a full description of each one.

The format of each operator description is as follows:


Precedence
Name
Description
Example

BYTE2 Second byte (1).

DESCRIPTION

BYTE2 takes a single operand, which is interpreted as an unsigned, 32-bit integer value. The result is the middle-low byte (bits 15 to 8) of the operand.

EXAMPLE

```
BYTE2 0x12345678 → 0x56
```

NAME

The operator name, and where appropriate, any synonyms for the operator, and the operator precedence.

The operator name is followed by a description of the operator.

DESCRIPTION

A detailed description covering the operator's most general use.

EXAMPLES

Examples, illustrating typical applications of the operator and clarifying any special cases.

PRECEDENCE

The precedence of the operator is given in brackets directly after the name of the operator.

*

Multiplication (2).

DESCRIPTION

* produces the product of its two operands. The operands are taken as signed 32-bit integers and the result is also a signed 32-bit integer.

EXAMPLES

```
2*2 → 4
-2*2 → -4
```

+

Unary plus (1).

DESCRIPTION

Unary plus operator.

EXAMPLES

```
+3 → 3
3*+2 → 6
```

+

Addition (3).

DESCRIPTION

The + addition operator produces the sum of the two operands which surround it. The operands are taken as signed 32-bit integers and the result is also a signed 32-bit integer.

EXAMPLES

```
92+19 → 111
-2+2 → 0
-2+-2 → -4
```

–

Unary minus (1).

DESCRIPTION

The unary minus operator performs arithmetic negation on its operand.

The operand is interpreted as a 32-bit signed integer and the result of the operator is the two’s complement negation of that integer.

–

Subtraction (3).

DESCRIPTION

The subtraction operator produces the difference when the right operand is taken away from the left operand. The operands are taken as signed 32-bit integers and the result is also signed 32-bit integer.

EXAMPLES

```
92-19 → 73
-2-2 → -4
-2--2 → 0
```

/

Division (2).

DESCRIPTION

/ produces the integer quotient of the left operand divided by the right operator. The operands are taken as signed 32-bit integers and the result is also a signed 32-bit integer.

EXAMPLES

```
8/2 → 4
-12/3 → -4
```

AND (&&)

Logical AND (5).

DESCRIPTION

Use `AND` to perform logical AND between its two integer operands. If both operands are non-zero the result is 1; otherwise it is zero.

EXAMPLES

```
1010B AND 0011B → 1
1010B AND 0101B → 1
1010B AND 0000B → 0
```

BINAND(&)

Bitwise AND (5).

DESCRIPTION

Use `BINAND` to perform bitwise AND between the integer operands.

EXAMPLES

```
1010B BINAND 0011B → 0010B
1010B BINAND 0101B → 0000B
1010B BINAND 0000B → 0000B
```

BINNOT (~)

Bitwise NOT (1).

DESCRIPTION

Use `BINNOT` to perform bitwise NOT on its operand.

EXAMPLE

```
BINNOT 1010B → 11111111111111111111111111110101B
```

BINOR (|)

Bitwise OR (6).

DESCRIPTION

Use BINOR to perform bitwise OR on its operands.

EXAMPLES

```
1010B BINOR 0101B → 1111B
1010B BINOR 0000B → 1010B
```

BINXOR (^)

Bitwise exclusive OR (6).

DESCRIPTION

Use BINXOR to perform bitwise XOR on its operands.

EXAMPLES

```
1010B BINXOR 0101B → 1111B
1010B BINXOR 0011B → 1001B
```

BYTE2

Second byte (1).

DESCRIPTION

BYTE2 takes a single operand, which is interpreted as an unsigned, 32-bit integer value. The result is the middle-low byte (bits 15 to 8) of the operand.

EXAMPLE

```
BYTE2 0x12345678 → 0x56
```

BYTE3

Third byte (1).

DESCRIPTION

BYTE3 takes a single operand, which is interpreted as an unsigned, 32-bit integer value. The result is the middle-high byte (bits 23 to 16) of the operand.

EXAMPLE

```
BYTE3 0x12345678 → 0x34
```

DATE

Current date/time.

DESCRIPTION

Use the DATE operator to give the moment when the current assembly began.

The DATE operator takes an absolute argument (expression) and returns:

DATE 1	Current second (0–59).
DATE 2	Current minute (0–59).
DATE 3	Current hour (0–23).
DATE 4	Current day (1–31).
DATE 5	Current month (1–12).
DATE 6	Current year MOD 100 (1998 →98, 2002 →02).

EXAMPLE

To assemble the date of assembly:

```
today DCB DATE 5, DATE 4, DATE 3
```

EQ (=, ==)

Equal (7).

DESCRIPTION

EQ evaluates to 1 (true) if its two operands are identical in value, or to 0 (false) if its two operands are not identical in value.

EXAMPLES

```
1 EQ 2 → 0
2 EQ 2 → 1
'ABC' EQ 'ABCD' → 0
```

GE(>=)

Greater than or equal (7).

DESCRIPTION

`GE` evaluates to 1 (true) if the left operand is equal to or has a higher numeric value than the right operand.

EXAMPLES

```
1 GE 2 → 0
2 GE 1 → 1
1 GE 1 → 1
```

GT (>)

Greater than (7).

DESCRIPTION

`GT` evaluates to 1 (true) if the left operand has a higher numeric value than the right operand.

EXAMPLES

```
-1 GT 1 → 0
2 GT 1 → 1
1 GT 1 → 0
```

HIGH

Second byte (1).

DESCRIPTION

`HIGH` takes a single operand to its right which is interpreted as an unsigned, 16-bit integer value. The result is the unsigned 8-bit integer value of the higher order byte of the operand.

EXAMPLE

```
HIGH ABCDh → ABh
```

HWRD (MSW)

High word (1).

DESCRIPTION

`HWRD` takes a single operand, which is interpreted as an unsigned, 32-bit integer value. The result is the high word (bits 31 to 16) of the operand.

EXAMPLE

```
HWRD 0x12345678 → 0x1234
```

LE (<=)

Less than or equal (7).

DESCRIPTION

`LE` evaluates to 1 (true) if the left operand has a lower or equal numeric value to the right operand.

EXAMPLES

```
1 LE 2 → 1
2 LE 1 → 0
1 LE 1 → 1
```

LOW

Low byte (1).

DESCRIPTION

`LOW` takes a single operand, which is interpreted as an unsigned, 32-bit integer value. The result is the unsigned, 8-bit integer value of the lower order byte of the operand.

EXAMPLE

```
LOW ABCDh → CDh
```

LT (<)

Less than (7).

DESCRIPTION

LT evaluates to 1 (true) if the left operand has a lower numeric value than the right operand.

EXAMPLES

```
-1 LT 2 → 1
2 LT 1 → 0
2 LT 2 → 0
```

LWRD (LSW)

Low word (1).

DESCRIPTION

LWRD takes a single operand, which is interpreted as an unsigned, 32-bit integer value. The result is the low word (bits 15 to 0) of the operand.

EXAMPLE

```
LWRD 0x12345678 → 0x5678
```

MOD (%)

Modulo (2).

DESCRIPTION

MOD produces the remainder from the integer division of the left operand by the right operand. The operands are taken as signed, 32-bit integers and the result is also a signed, 32-bit integer.

`X MOD Y` is equivalent to `X-Y*(X/Y)` using integer division.

EXAMPLES

```
2 MOD 2 → 0
12 MOD 7 → 5
3 MOD 2 → 1
```

NE (<> , ! =)

Not equal (7).

DESCRIPTION

NE evaluates to 0 (false) if its two operands are identical in value or to 1 (true) if its two operands are not identical in value.

EXAMPLES

```
1 NE 2 → 1
2 NE 2 → 0
'A' NE 'B' → 1
```

NOT (!)

Logical NOT (1).

DESCRIPTION

Use NOT to negate a logical argument.

EXAMPLES

```
NOT 0101B → 0
NOT 0000B → 1
```

OR (||)

Logical OR (6).

DESCRIPTION

Use OR to perform a logical OR between two integer operands.

EXAMPLES

```
1010B OR 0000B → 1
0000B OR 0000B → 0
```

SFB

Segment begin (1).

SYNTAX

SFB(*segment* [{+ | -} *offset*])

PARAMETERS

segment The name of a relocatable segment, which must be defined before SFB is used.

offset An optional offset from the start address. The parentheses are optional if *offset* is omitted.

DESCRIPTION

SFB accepts a single operand to its right. The operand must be the name of a relocatable segment. The operator evaluates to the absolute address of the first byte of that segment. This evaluation takes place at linking time.

EXAMPLES

```
       NAME  demo
       RSEG  CODE
start  DCW   SFB(CODE)
```

Even if the above code is linked with many other modules, `start` will still be set to the address of the first byte of the segment.

SFE

Segment end (1).

SYNTAX

SFE (*segment* [{+ | -} *offset*])

PARAMETERS

segment The name of a relocatable segment, which must be defined before SFE is used.

offset An optional offset from the start address. The parentheses are optional if *offset* is omitted.

DESCRIPTION

SFE accepts a single operand to its right. The operand must be the name of a relocatable segment. The operator evaluates to the segment start address plus the segment size. This evaluation takes place at linking time.

EXAMPLES

```
        NAME  demo
        RSEG  CODE
end     DCW   SFE(CODE)
```

Even if the above code is linked with many other modules, end will still be set to the address of the last byte of the segment.

SHL (<<)

Logical shift left (4).

DESCRIPTION

Use SHL to shift the left operand to the left. The number of bits to shift is specified by the right operand, interpreted as an integer value between 0 and 32.

EXAMPLES

```
00011100B SHL 3 → 11100000B
0000011111111111B SHL 5 → 1111111111100000B
14 SHL 1 → 28
```

SHR (>>)

Logical shift right (4).

DESCRIPTION

Use SHR to shift the left operand to the right. The number of bits to shift is specified by the right operand, interpreted as an integer value between 0 and 32.

EXAMPLES

```
01110000B SHR 3 → 00001110B
1111111111111111B SHR 20 → 0
14 SHR 1 → 7
```

SIZEOF

Segment size (1).

SYNTAX

`SIZEOF` *`segment`*

PARAMETERS

`segment`	The name of a relocatable segment, which must be defined before `SIZEOF` is used.

DESCRIPTION

`SIZEOF` generates `SFE-SFB` for its argument, which should be the name of a relocatable segment; ie it calculates the size in bytes of a segment. This is done when modules are linked together.

EXAMPLES

```
      NAME   demo
      RSEG   CODE
size  DCW    SIZEOF CODE
```

sets `size` to the size of segment `CODE`.

UGT

Unsigned greater than (7).

DESCRIPTION

`UGT` evaluates to 1 (true) if the left operand has a larger absolute value than the right operand.

EXAMPLES

```
2 UGT 1 → 1
-1 UGT 1 → 1
```

ULT

Unsigned less than (7).

DESCRIPTION

`ULT` evaluates to 1 (true) if the left operand has a smaller absolute value than the right operand.

EXAMPLES

```
1 ULT 2 → 1
-1 ULT 2 → 0
```

XOR

Logical exclusive OR (6).

DESCRIPTION

Use `XOR` to perform logical XOR on its two operands.

EXAMPLES

```
0101B XOR 1010B → 0
0101B XOR 0000B → 1
```

ASSEMBLER DIRECTIVES SUMMARY

This chapter gives an alphabetical summary of the assembler directives.

The directives are divided into the following sections:

Module control
Symbol control
Segment control
Value assignment
Conditional assembly
Macro processing
Listing control
C-style preprocessor
Data definition or allocation
Assembler control

For a full description of any directive, see under the directive's category name in the next chapter, *Assembler directives reference*.

DIRECTIVES SUMMARY

The following table gives a summary of all the assembler directives.

Directive	*Description*	*Section*
`$`	Includes a file.	Assembler control
`#define`	Assigns a value to a label.	C-style preprocessor
`#elif`	Introduces a new condition in an `#if`…`#endif` block.	C-style preprocessor
`#else`	Assembles instructions if a condition is false.	C-style preprocessor
`#endif`	Ends a `#if`, `#ifdef`, or `#ifndef` block.	C-style preprocessor
`#error`	Generates an error.	C-style preprocessor
`#if`	Assembles instructions if a condition is true.	C-style preprocessor
`#ifdef`	Assembles instructions if a symbol is defined.	C-style preprocessor
`#ifndef`	Assembles instructions if a symbol is undefined.	C-style preprocessor
`#include`	Includes a file.	C-style preprocessor
`#message`	Generates a message on standard output.	C-style preprocessor

Directive	*Description*	*Section*
#undef	Undefines a label.	C-style preprocessor
/*comment*/	C-style comment delimiter.	Assembler control
//	C++ style comment delimiter.	Assembler control
=	Assigns a permanent value local to a module.	Value assignment
ALIGN	Aligns the location counter by inserting zero-filled bytes.	Segment control
ASEG	Begins an absolute segment.	Segment control
ASSIGN	Assigns a temporary value.	Value assignment
CASEOFF	Disables case sensitivity.	Assembler control
CASEON	Enables case sensitivity.	Assembler control
COL	Sets the number of columns per page.	Listing control
COMMON	Begins a common segment.	Segment control
const	Specifies an SFR label as read-only.	Value assignment
DB	Generates 8-bit byte constants.	Data definition or allocation
DD	Generates 32-bit double word constants.	Data definition or allocation
DEFINE	Defines a file-wide value.	Value assignment
DF	Generates 32-bit double word constants.	Data definition or allocation
DR	Generates 32-bit double word constants.	Data definition or allocation
DS	Allocates space for 8-bit bytes.	Data definition or allocation
DT	Generates a RETLW instruction with the argument as return value.	Data definition or allocation
DW	Generates 16-bit word constants.	Data definition or allocation
ELSE	Assembles instructions if a condition is false.	Conditional assembly

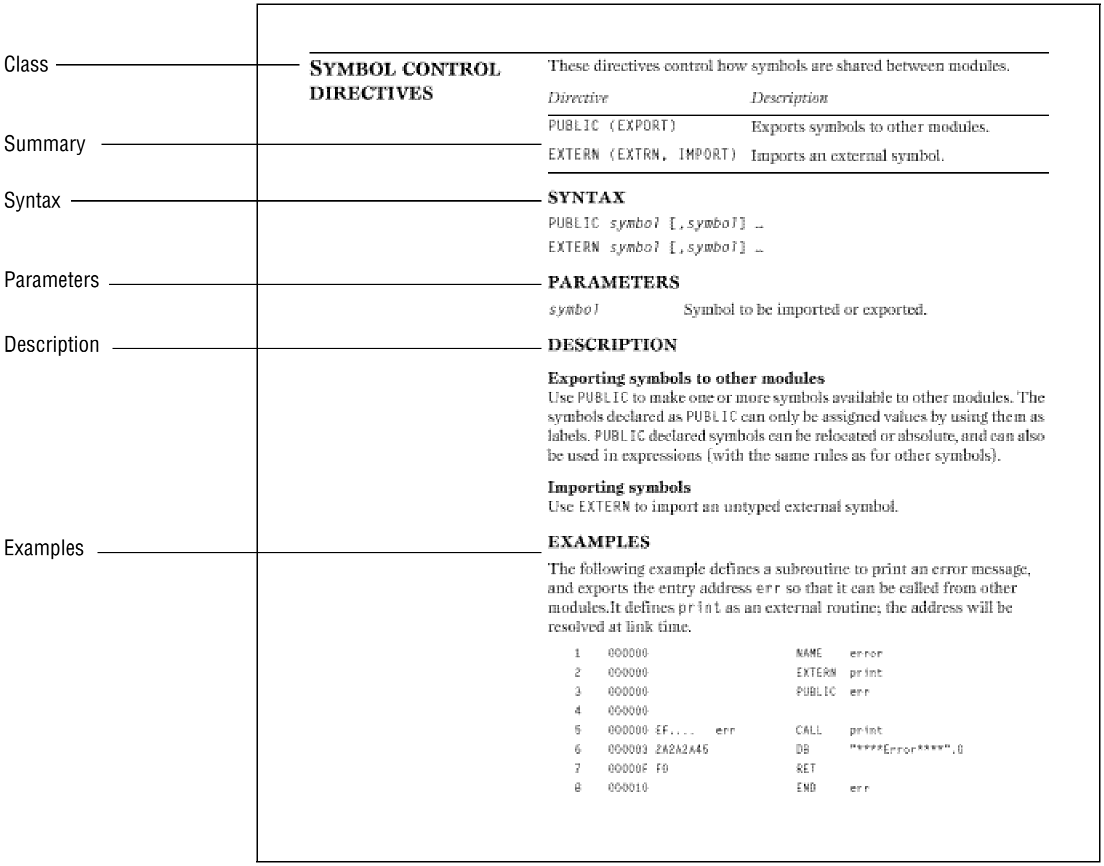
Directive	*Description*	*Section*
ELSEIF	Specifies a new condition in an IF…ENDIF block.	Conditional assembly
END	Terminates the assembly of the last module in a file.	Module control
ENDIF	Ends an IF block.	Conditional assembly
ENDM	Ends a macro definition.	Macro processing
ENDMOD	Terminates the assembly of the current module.	Module control
ENDR	Ends a repeat structure.	Macro processing
EQU	Assigns a permanent value local to a module.	Value assignment
EVEN	Aligns the program counter to an even address.	Segment control
EXITM	Exits prematurely from a macro.	Macro processing
EXTERN	Imports an external symbol.	Symbol control
GLOBAL	Exports symbols to other modules.	Symbol control
IF	Assembles instructions if a condition is true.	Conditional assembly
LIBRARY	Begins a library module.	Module control
LOCAL	Creates symbols local to a macro.	Macro processing
LSTCND	Controls conditional assembly listing.	Listing control
LSTCOD	Controls multi-line code listing.	Listing control
LSTEXP	Controls the listing of macro generated lines.	Listing control
LSTMAC	Controls the listing of macro definitions.	Listing control
LSTOUT	Controls assembly listing output.	Listing control
LSTPAG	Controls the formatting of output into pages.	Listing control
LSTREP	Controls the listing of lines generated by repeat directives.	Listing control
LSTXRF	Generates a cross-reference table.	Listing control
MACRO	Defines a macro.	Macro processing
MODULE	Begins a library module.	Module control
NAME	Begins a program module.	Module control

Directive	*Description*	*Section*
`ORG`	Sets the location counter.	Segment control
`PAGE`	Generates a new page.	Listing control
`PAGSIZ`	Sets the number of lines per page.	Listing control
`PROGRAM`	Begins a program module.	Module control
`PUBLIC`	Exports symbols to other modules.	Symbol control
`RADIX`	Sets the default base.	Assembler control
`REPT`	Assembles instructions a specified number of times.	Macro processing
`REPTC`	Repeats and substitutes characters.	Macro processing
`REPTI`	Repeats and substitutes strings.	Macro processing
`REQUIRE`	Marks a symbol as required.	Symbol control
`RES`	Generates 16-bit word constants.	Data definition or allocation
`RSEG`	Begins a relocatable segment.	Segment control
`RTMODEL`	Declares run-time model attributes.	Module control
`SET`	Assigns a temporary value.	Value assignment
`STACK`	Begins a stack segment.	Segment control
`VAR`	Assigns a temporary value.	Value assignment

ASSEMBLER DIRECTIVES REFERENCE

This chapter gives a list of the PICmicro™ directives, classified according to their function, with a full description of their operation, and the options available for each one.

The format of each section is as follows:



SYMBOL CONTROL DIRECTIVES

These directives control how symbols are shared between modules.

Directive	*Description*
`PUBLIC (EXPORT)`	Exports symbols to other modules.
`EXTERN (EXTRN, IMPORT)`	Imports an external symbol.

SYNTAX

```
PUBLIC symbol [,symbol] …
EXTERN symbol [,symbol] …
```

PARAMETERS

symbol	Symbol to be imported or exported.

DESCRIPTION

Exporting symbols to other modules
Use `PUBLIC` to make one or more symbols available to other modules. The symbols declared as `PUBLIC` can only be assigned values by using them as labels. `PUBLIC` declared symbols can be relocated or absolute, and can also be used in expressions (with the same rules as for other symbols).

Importing symbols
Use `EXTERN` to import an untyped external symbol.

EXAMPLES

The following example defines a subroutine to print an error message, and exports the entry address `err` so that it can be called from other modules.It defines `print` as an external routine; the address will be resolved at link time.

```
1    000000                        NAME    error
2    000000                        EXTERN  print
3    000000                        PUBLIC  err
4    000000
5    000000 EF....    err          CALL    print
6    000003 2A2A2A45               DB      "****Error****",0
7    00000F F0                     RET
8    000010                        END     err
```

CLASS

The class of directives.

SUMMARY

The class is followed by a summary of the class, and a description of each directive in the class.

SYNTAX

A full syntax definition of each directive.

PARAMETERS

Details of each parameter in the syntax definitions.

DESCRIPTION

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

EXAMPLES

Examples, illustrating typical applications of the directives and clarifying any special cases.

SYNTAX CONVENTIONS

In the syntax definitions the following conventions are used:

Parameters, representing what you would type, are shown in italics. So, for example, in:

```
ORG expr
```

expr represents an arbitrary expression.

Optional parameters are shown in square brackets. So, for example, in:

```
END [expr]
```

the *expr* parameter is optional. An ellipsis indicates that the previous item can be repeated an arbitrary number of times. For example:

```
LOCAL symbol [,symbol] …
```

indicates that `LOCAL` can be followed by one or more symbols, separated by commas.

Alternatives are enclosed in { and } brackets, separated by a vertical bar. For example:

```
LSTOUT{+ | -}
```

indicates that the directive must be followed by either + or `-`.

LABELS AND COMMENTS

Where a directive must be preceded by a label, this is indicated in the syntax, as in:

```
label SET expr
```

All other directives can be preceded by an optional label, which will assume the value and type of the current location counter (PLC), and for clarity this is not included in each syntax definition.

In addition, unless explicitly specified, all directives can be followed by a comment, preceded by **;** (semi-colon).

PARAMETERS

The following table shows the correct form of the most commonly-used types of parameter:

Parameter	*What it consists of*
symbol	An assembler symbol.
label	A symbolic label.
expr	An expression; see *Expressions and operators*, page 63.

SYMBOL PREFIX

Since the IAR assembler and linker utilizes a very versatile and general module/segment concept, it is not always possible for the assembler to determine if a symbol is intended to refer to a place in CODE memory (i.e. word-accessible) or in DATA memory (i.e. byte-accessible). This is a consequence of the fact that it is not until link time the user determines what kind of memory a segment belongs to. However, the assembler has a default interpretation as follows:

- Arguments to `GOTO`/`LGOTO`/`CALL`/`LCALL` and references to `EXTERNAL` symbols are always interpreted as CODE addresses.

- All other uses are interpreted as referring to DATA (byte) memory. In some situations these assumptions are wrong and the addresses will come out scaled in a factor of 2 up or down as compared to the expected value.

To force a certain interpretation, there are two prefixes that can be used:

`A:` Interpret symbol as referring to CODE memory.

`L:` Interpret symbol as referring to DATA memory.

As a convention, it is recommended to always use these prefixes so the intended meaning is made explicit.

Example

`A:`*`symbol`*

Treats the *`symbol`* as word address.

MODULE CONTROL DIRECTIVES

Module control directives are used to mark the beginning and end of source program modules, and to assign names and types to them.

Directive	*Description*
`NAME (PROGRAM)`	Begins a program module.
`MODULE (LIBRARY)`	Begins a library module.
`ENDMOD`	Terminates the assembly of the current module.
`END`	Terminates the assembly of the last module in a file.
`RTMODEL`	Declares run-time model attibutes.

SYNTAX

```
NAME symbol [(expr)]
MODULE symbol [(expr)]
ENDMOD [label]
END [label]
RTMODEL key, value
```

PARAMETERS

symbol	Name assigned to module, used by XLIB when referencing the module.
expr	Optional expression (0–255) used by the IAR C Compiler.
label	An expression or label which can be resolved at assembly time. It is output in the object code as a program entry address.
key	A text string specifying the key.
value	A text string specifying the value.

DESCRIPTION

Beginning a program module

Use NAME to begin a program module, and assign a name for future reference by XLINK and XLIB.

Program modules are unconditionally linked by XLINK, even if they are not referenced by other modules.

Beginning a library module

Use MODULE to create libraries containing lots of small modules (like run-time systems for high level languages), where each module also often represent a single routine. With the multi-module facility you can significantly reduce the number of source and object files needed.

Library modules are only copied into the linked code if a public symbol in the module is referenced by other modules.

Terminating a module

Use ENDMOD to define the end of a module.

Terminating the last module

Use END to indicate the end of the source file. Any lines after the END directive are ignored.

Program entries must be either relocatable or absolute (no externals allowed), and will show up in XLINK load maps, as well as in some of the hexadecimal absolute output formats.

The following rules apply to multi-module assemblies:

- At the beginning of a new module all user symbols are deleted, except for those created by DEFINE, #define, or MACRO, the location counters are cleared, and the mode is set to absolute.
- List control directives remain in effect throughout the assembly.

Note that END must always be used in the *last* module, and that there must not be any source lines (except for comments and list control directives) between an ENDMOD and a MODULE directive.

If the NAME or MODULE directive is missing, the module will be assigned the name of the source file and the attribute program.

Declaring run-time model attributes
Use RTMODEL to enforce compatibility between modules. If a module defines a run-time model attribute, all other modules must have the same *value* for that key, or the special *value**.

EXAMPLES

The following example defines three modules:

```
MODULE
.
. Module #1
.
ENDMOD
MODULE
.
. Module #2

.
ENDMOD
MODULE
.
. Last module
.
END
```

SYMBOL CONTROL DIRECTIVES

These directives control how symbols are shared between modules.

Directive	*Description*
`PUBLIC (GLOBAL)`	Exports symbols to other modules.
`EXTERN`	Imports an external symbol.
`REQUIRE`	Marks a symbol as required.

SYNTAX

```
PUBLIC symbol [,symbol] …
EXTERN symbol [,symbol] …
REQUIRE symbol [,symbol] …
```

PARAMETERS

symbol	Symbol to be imported, exported or set as required.

DESCRIPTION

Exporting symbols to other modules
Use `PUBLIC` to make one or more symbols available to other modules. The symbols declared as `PUBLIC` can only be assigned values by using them as labels. `PUBLIC` declared symbols can be relocated or absolute, and can also be used in expressions (with the same rules as for other symbols).

The `PUBLIC` directive always exports full 32-bit values, which makes it feasible to use global 32-bit constants also in assemblers for 8 and 16-bit processors. With the `LOW`, `HIGH`, `>>`, and `<<` operators any part of such a constant can be loaded in a 8 or 16-bit register or word.

There are no restrictions on the number of `PUBLIC` declared symbols in a module.

Importing symbols
Use `EXTERN` to import an untyped external symbol.

Marking a symbol as required
Use `REQUIRE` to mark a symbol as required.

EXAMPLES

The following example defines a subroutine to print an error message, and exports the entry address err so that it can be called from other modules.

It defines print as an external routine; the address will be resolved at link time.

```
1    000000                       NAME    error
2    000000                       EXTERN  print
3    000000                       PUBLIC  err
4    000000
5    000000 EF....   err          CALL    print
6    000003 2A2A2A45              DB      "****Error****".0
7    00000F F0                    RET
8    000010                       END     err
```

SEGMENT CONTROL DIRECTIVES

The segment directives control how code and data are generated.

Directive	*Description*
`ASEG`	Begins an absolute segment.
`RSEG`	Begins a relocatable segment.
`STACK`	Begins a stack segment.
`COMMON`	Begins a common segment.
`ORG`	Sets the location counter.
`ALIGN`	Aligns the location counter by inserting zero-filled bytes.

SYNTAX

```
ASEG [start [(align)]]
RSEG segment [:type] [(align)]
STACK segment [:type] [(align)]
COMMON segment [:type] [(align)]
ORG expr
ALIGN align [,value]
```

PARAMETERS

start	A start address which has the same effect as using an `ORG` directive at the beginning of the absolute segment.
segment	The name of the segment.
type	The memory type; one of: `UNTYPED` (the default), `CODE`, or `DATA`. In addition, the following types are provided for compatibility with the IAR C Compilers: `XDATA`, `IDATA`, `BIT`, `REGISTER`, `CONST`, `NEARDATA`, `FARDATA`, `HUGEDATA`, `NEARCONST`, `FARCONST`, `HUGECONST`, `NEARCODE`, `FARCODE` and `HUGECODE`.
expr	Address to set location counter to.
align	Power of two to which the address should be aligned, in the range 0 to 30.
value	Byte value used for padding, default is zero.

DESCRIPTION

Beginning an absolute segment
Use `ASEG` to set the absolute mode of assembly, which is the default at the beginning of a module.

If the parameter is omitted, the start address of the first segment is 0, and subsequent segments continue after the last address of the previous segment.

Beginning a relocatable segment
Use `RSEG` to set the current mode of the assembly to relocatable assembly mode. The assembler maintains separate location counters (initially set to zero) for all segments, which makes it possible to switch segments and mode anytime without the need to save the current segment location counter.

Up to 256 unique, relocatable segments may be defined in a single module.

Beginning a stack segment
Use `STACK` to allocate code or data allocated from high to low addresses (vs. the `RSEG` directive which causes low-to-high allocation).

Note that the contents of the segment are not generated in reverse order.

Beginning a common segment
Use `COMMON` to place data in memory at the same location as `COMMON` segments from other modules that have the same name. In other words, all `COMMON` segments of the same name will start at the same location in memory and overlay each other.

Obviously, the `COMMON` segment type should not be used for overlaid executable code. A typical application would be when you want a number of different routines to share a reusable, common area of memory for data.

It can be practical to have the interrupt vector table in a `COMMON` segment, thereby allowing access from several routines.

The final size of the `COMMON` segment is determined by the size of largest occurrence of this segment. The location in memory is determined by the XLINK `-Z` command; see *Segment control*, page 184.

Setting the location counter

Use ORG to set the location counter of the current segment to the value of an expression. The optional label will assume the value and type of the new location counter.

The result of the expression must be of the same type as the current segment, that is, it is not valid to use ORG 10 during RSEG, since the expression is absolute; instead use ORG $+10. The expression must not contain any forward or external references.

All location counters are set to zero at the beginning of an assembly module.

Aligning a segment

Use ALIGN to align the location counter to a specified address boundary. The expression gives the power of two to which the program counter should be aligned.

EXAMPLES

Beginning an absolute segment

The following example assembles the jump to the function main in address 0. On RESET the chip sets PC to address 0.

```
        MODULE  reset
        EXTERN  main

        ASEG
        ORG     0          ; RESET vector address
reset:  GOTO    main       ; Instruction that
                           ; executes on startup
        end
```

Beginning a relocatable segment

In the following example the data following the first RSEG directive is placed in a relocatable segment called table:

The code following the second RSEG directive is placed in a relocatable segment called code:

```
        EXTERN  subrtn, divrtn

        RSEG    table
```

```
functable:
        DW      subrtn
        DW      divrtn

        RSEG    code
main:
        MOWLW   0x12
        ADDWF   0x20,0
        RETURN


        END
```

Beginning a stack segment

The following example defines two 10-byte stacks in a relocatable segment called `rpnstack`:

```
        STACK   rpnstack
parms   DS      10
opers   DS      10
        END
```

The data is allocated from high to low addresses.

Beginning a common segment

The following example defines two common segments containing variables:

```
        NAME    common1
        COMMON  data
count   DD      1
        ENDMOD
        NAME    common2
        COMMON  data
up      DS      1
        DS      2
down    DS      1
        END
```

Because the common segments have the same name, `data`, the variables `up` and `down` refer to the same locations in memory as the first and last bytes of the 4-byte variable `count`.

VALUE ASSIGNMENT DIRECTIVES

These directives are used to assign values to symbols.

Directive	*Description*
`SET (VAR, ASSIGN)`	Assigns a temporary value.
`EQU (=)`	Assigns a permanent value local to a module.
`DEFINE`	Defines a file-wide value.

SYNTAX

```
symbol SET expr
symbol EQU expr
symbol = expr
symbol DEFINE expr
```

PARAMETERS

symbol	Symbol to be defined.
expr	Value assigned to symbol.

DESCRIPTION

Defining a temporary value

Use `SET` to define a symbol which may be redefined, such as for use with macro variables. Symbols defined with `SET` cannot be declared `PUBLIC`.

Defining a permanent local value

Use `EQU` or = to assign a value to a symbol.

Use `EQU` to create a local symbol that denotes a number or offset.

The symbol is only valid in the module in which it was defined, but can be made available to other modules with a `PUBLIC` directive.

To import symbols from other modules use `EXTERN`.

Defining a permanent global value

Use `DEFINE` to define symbols that should be known to all modules in the source file.

A symbol which has been given a value with `DEFINE` can be made available to modules in files with the `PUBLIC` directive.

Symbols defined with `DEFINE` cannot be redefined.

EXAMPLES

Redefining a symbol

The following example uses SET to redefine the symbol cons in a REPT loop to generate a table of the first 8 powers of 3:

```
        NAME    table

; Generate table of powers of 3

cons    SET     1

cr_tabl MACRO   times
        DW      cons
cons    SET     cons*3
        IF      times>1
        cr_tabl times-1
        ENDIF
        ENDM

table:
        cr_tabl 4
        END     table
```

It generates the following code:

```
  1    000000
  2    000000
  3    000000                 NAME table
  4    000000
  5    000000          ; Generate table of powers of 3
  6    000000
  7    000001          cons    SET 1
  8    000000
 16    000000
 17    000000          table:
 18    000000                  cr_tabl 4
 18.1  000000 0001             DW      cons
 18.2  000003          cons    SET     cons*3
 18.3  000002                  IF      4>1
 18    000002                  cr_tabl 4-1
 18.1  000002 0003             DW      cons
 18.2  000009          cons    SET     cons*3
```

```
18.3  000004                 IF      4-1>1
18    000004                 cr_tabl 4-1-1
18.1  000004 0009            DW      cons
18.2  00001B         cons    SET     cons*3
18.3  000006                 IF      4-1-1>1
18    000006                 cr_tabl 4-1-1-1
18.1  000006 001B            DW      cons
18.2  000051         cons    SET     cons*3
18.3  000008                 IF      4-1-1-1>1
18.4  000008                 cr_tabl 4-1-1-1-1
18.5  000008                 ENDIF
18.6  000008                 ENDM
18.7  000008                 ENDIF
18.8  000008                 ENDM
18.9  000008                 ENDIF
18.10 000008                 ENDM
18.11 000008                 ENDIF
18.12 000008                 ENDM
19    000008                 END table
```

Using local and global symbols

In the following example the symbol `value` defined in module `add1` is local to that module; a distinct symbol of the same name is defined in module `add2`. The `DEFINE` directive is used to declare `R0` for use anywhere in the file:

```
        NAME    add1
        PUBLIC  add12
R0      DEFINE  0x20
value   EQU     12

add12:
        MOVLW   value
        ADDWF   R0,1
        RETURN
        ENDMOD

        NAME    add2
        PUBLIC  add20
value   EQU     20
```

```
add20:
        MOVLW   value
        ADDWF   R0,1
        RETURN

        END
```

CONDITIONAL ASSEMBLY DIRECTIVES

These directives provide logical control over the selective assembly of source code.

Directive	*Description*
`IF`	Assembles instructions if a condition is true.
`ELSE`	Assembles instructions if a condition is false.
`ELSEIF`	Specifies a new condition in an `IF...ENDIF` block.
`ENDIF`	Ends an `IF` block.

SYNTAX

`IF` *`condition`*

`ELSE`

`ELSEIF`

`ENDIF`

PARAMETERS

`condition`	One of the following:	
	An absolute expression	The expression must not contain forward or external references, and any non-zero value is considered as true.
	`string1=string2`	The condition is true if *`string1`* and *`string2`* have the same length and contents.

`string1<>string2`	The condition is true if *string1* and *string2* have different length or contents.

DESCRIPTION

Use the `IF … ELSE … ELSEIF … ENDIF` directives to control the assembly process at assembly time. If the condition following the `IF` directive is not true, the subsequent instructions will not generate any code (i.e. it will not be assembled or syntax checked) until an `ELSE`, `ELSEIF` or `ENDIF` directive is found. `ELSEIF` is used to introduce a new condition in the `IF … ENDIF` block.

Conditional assembler directives may be used anywhere in an assembly, but have their greatest use in conjunction with macro processing.

All assembler directives (except for `END`), and file inclusion, may be disabled by the conditional directives. Each `IF`*xx* directive must be terminated by an `ENDIF` directive. The `ELSE` and `ELSEIF` directives are optional, and if used, they must be inside an `IF … ENDIF` block.

`IF … ENDIF` and `IF … ELSE … ENDIF` blocks may be nested to any level.

EXAMPLES

```
add     MACRO   a,b        ; A should be a register file,
                           ; b a literal
        IF      b=1
        INCF    a,1
        ELSE
        MOVLW   b
        ADDWF   a,1
        ENDIF
        ENDM
```

If the argument to the macro is `1`, it generates an `INC` instruction; otherwise it generates an `ADD` instruction.

It could be tested with the following program:

```
R0      DEFINE  0x20
R1      DEFINE  0x21
main:
        MOVLW   0x0F
        MOVWF   R0
```

```
add     R0, 0x12
add     R1, 1
RETURN

END
```

MACRO PROCESSING DIRECTIVES

These directives allow user macros to be defined.

Directive	*Description*
`MACRO`	Defines a macro.
`ENDM`	Ends a macro definition.
`EXITM`	Exits prematurely from a macro.
`LOCAL`	Creates symbols local to a macro.
`REPT`	Assembles instructions a specified number of times.
`REPTC`	Repeats and substitutes characters.
`REPTI`	Repeats and substitutes strings.
`ENDR`	Ends a repeat structure.

SYNTAX

```
name MACRO [argument] …
ENDM
EXITM
LOCAL symbol [,symbol] …
REPT expr
REPTC formal,actual
REPTI formal,actual [,actual] …
ENDR
```

PARAMETERS

name	The name of the macro.
argument	A symbolic argument name.
symbol	Symbol to be local to the macro.

expr An expression.

formal Argument into which each character of *actual* (REPTC) or each *actual* (REPTI) is substituted.

actual String to be substituted.

DESCRIPTION

A macro is a user-defined symbol that represents a block of one or more assembler source lines. Once you have defined a macro you can use it in your program like an assembler directive or assembler mnemonic.

When the assembler encounters a macro, it looks up the macro's definition, and inserts the lines that the macro represents as if they were included in the source file at that position.

Although macros effectively perform simple text substitution, you can control what they substitute by supplying parameters to them.

Defining a macro

You define a macro with the statement:

macroname MACRO [*arg*] [*arg*] …

Here *macroname* is the name you are going to use for the macro, and *arg* is an argument for values you want to pass to the macro when it is expanded.

For example, you could define a macro `errmac` as follows:

```
errmac  MACRO   errno
        MOVLW   errno
        CALL    abort
        ENDM
```

This uses a parameter `errno` to set up an error number for a routine abort. You would call the macro with a statement such as:

```
        errmac  2
```

This will be expanded by the assembler to:

```
        MOVLW   2
        CALL    abort
```

If you omit a list of one or more arguments, the arguments you supply when calling the macro are called `\1` to `\9` and `\A` to `\Z`.

The previous example could therefore be written as follows:

```
errmac  MACRO
        MOVLW   \1
        CALL    abort
        ENDM
```

Use the `EXITM` directive to generate a premature exit from a macro.

`EXITM` is not allowed inside `REPT … ENDR`, `REPTC … ENDR`, or `REPTI … ENDR`.

Use `LOCAL` to create symbols local to a macro. The `LOCAL` directive must be used before the symbol is used.

Each time a macro is expanded new instances of local symbols are created by the `LOCAL` directive, so it is legal to use local symbols in recursive macros.

It is illegal to *redefine* a macro.

Passing special characters

Macro arguments that include commas or white space can be forced to be interpreted as one argument by using the matching quote characters `<` and `>` in the macro call.

For example:

```
macld   MACRO   op
        MOVFP   op
        ENDM
```

It could be called using:

```
        macld   <R0, R1>
        END
```

You can redefine the macro quote characters with the **Macro quote chars (**`-M`**)** option; see *Macro quote chars (-M)*, page 46.

Predefined macro symbols

The symbol `_args` is set to the number of arguments passed to the macro.

How macros are processed

There are three distinct phases in the macro process:

- Scanning and saving of macro definitions is performed by the assembler. The text between `MACRO` and `ENDM` is saved but not syntax-checked. Include file references `$`*`file`* are recorded and will be included during macro *expansion*.
- A macro call forces the assembler to invoke the macro processor (expander) which switches (if not already in a macro) the assembler input stream from a source file to the output from the macro expander (which takes its input from the requested macro definition).

  The macro expander has no knowledge of assembler symbols since it only deals with text substitutions at source level. Before a line from the called macro definition is handed over to the assembler, the expander scans the line for all occurrences of symbolic macro arguments, and replaces them with their expansion arguments.
- The expanded line is then processed as any other assembler source line. The input stream to the assembler will continue to be the output from the macro processor, until all lines of the current macro definition have been read.

Repeating statements

Use the `REPT` … `ENDR` structure to assemble the same block of instructions a number of times. If *`expr`* evaluates to 0 nothing will be generated.

Use `REPTC` to assemble a block of instructions once for each character in a string. If the string contains a comma it should be enclosed in quotation marks.

Use `REPTI` to assemble a block of instructions once for each string in a series of strings. Strings containing commas should be enclosed in quotation marks.

EXAMPLES

This section gives examples of the different ways in which macros can make assembler programming easier.

Coding in-line for efficiency

In time-critical code it is often desirable to code routines in-line to avoid the overhead of a subroutine call and return. Macros provide a convenient way of doing this.

The following subroutine adds two 16-bit constants found in R1 and R2, and returns the result in R1. The program does not handle the case where overflow occurs in the high byte when propagating the carry from the low byte addition:

```
#define R1      0x20
#define R2      0x22
#define STATUS  3
#define CARRY   0

        ORG     0
start:  GOTO    A:main

        RSEG    CODE
add16:
        MOVF    R2+1,0        ; Read R2LOW to w
        ADDWF   R1+1,1        ; ADD and store in R1LOW
        MOVF    R2,0          ; Load high part
        BTFSS   STATUS, CARRY
        GOTO    no_carry
        ADDLW   1             ; Take care of carry

no_carry:
        ADDWF   R1            ; Store result
        RETURN


        RSEG    CODE
        PUBLIC main
main:   MOVLW  0xF0
        MOVWF  R1 + 1
        MOVLW  0x77
        MOVWF  R2 + 1
        MOVLW  0x10
        MOVWF  R1
        MOVLW  0x10
        MOVWF  R2
        CALL   add16

loop:   GOTO   loop

        end    main
```

The main program calls this routine as follows:

```
        CALL   add16
```

For efficiency we can recode this as the following macro:

```
#define R1     0x20
#define R2     0x22
#define STATUS 3
#define CARRY  0

        ORG    0
start:  GOTO   A:main

        RSEG   CODE
add16   MACRO
        MOVF   R2+1,0        ; Read R2LOW to w
        ADDWF  R1+1,1        ; ADD and store in R1LOW
        MOVF   R2,0          ; Load high part
        BTFSS  STATUS, CARRY
        GOTO   no_carry
        ADDLW  1             ; Take care of carry

no_carry:
        ADDWF  R1            ; Store result
        ENDM

        RSEG   CODE
        PUBLIC main
main:   MOVLW  0xF0
        MOVWF  R1 + 1
        MOVLW  0x77
        MOVWF  R2 + 1
        MOVLW  0x10
        MOVWF  R1
        MOVLW  0x10
        MOVWF  R2
        add16

loop:   GOTO   loop

        end    main
```

To use in-line code the main program is then simply altered to:

```
add16
```

Using REPTC and REPTI

The following example assembles a series of calls to a subroutine `plot` to plot each character in a string:

```
        NAME   reptc

        EXTERN plotc
R0      DEFINE 0x20
banner  REPTC  chr, "Welcome"

        MOVLW  ’chr’
        MOVWF  R0           ; Pass char in R0 as parameter
        CALL   plotc
        ENDR

        END
```

This produces the following code:

```
 1    000000
 2    000000
 3    000000                 NAME    reptc
 4    000000
 5    000000
 6    000000                 EXTERN plotc
 7    000020          R0     DEFINE 0x20
 8    000000          banner REPTC  chr, "Welcome"
 9    000000
10    000000                 MOVLW   ’chr’
11    000000                 MOVWF   R0      ; Pass char in R0 as
```

```
                     parameter
12    000000                  CALL    plotc
13    000000                  ENDR
13.1  000000
13.2  000000 3057             MOVLW   'W'
13.3  000002 00A0             MOVWF   R0      ; Pass char in R0 as
                                               parameter
13.4  000004 ....             CALL    plotc
13.5  000006
13.6  000006 3065             MOVLW   'e'
13.7  000008 00A0             MOVWF   R0      ; Pass char in R0 as
                                               parameter
13.8  00000A ....             CALL    plotc
13.9  00000C
13.10 00000C 306C             MOVLW   'l'
13.11 00000E 00A0             MOVWF   R0      ; Pass char in R0 as
                                               parameter
13.12 000010 ....             CALL    plotc
13.13 000012
13.14 000012 3063             MOVLW   'c'
13.15 000014 00A0             MOVWF   R0      ; Pass char in R0 as
                                               parameter
13.16 000016 ....             CALL    plotc
13.17 000018
13.18 000018 306F             MOVLW   'o'
13.19 00001A 00A0             MOVWF   R0      ; Pass char in R0 as
                                               parameter
13.20 00001C ....             CALL    plotc
13.21 00001E
13.22 00001E 306D             MOVLW   'm'
13.23 000020 00A0             MOVWF   R0      ; Pass char in R0 as
                                               parameter
13.24 000022 ....             CALL    plotc
13.25 000024
13.26 000024 3065             MOVLW   'e'
13.27 000026 00A0             MOVWF   R0      ; Pass char in R0 as
                                               parameter
13.28 000028 ....             CALL    plotc
14    00002A
15    00002A
16    00002A                  END
```

The following example uses REPTI to clear a number of memory locations:

```
         NAME   repti

        EXTERN base, count, init

banner  REPTI  adds, base, count, init

        CLRF   adds
        ENDR

        END
```

This produces the following code:

```
 1    000000
 2    000000
 3    000000                NAME    repti
 4    000000
 5    000000                EXTERN  base, count, init
 6    000000
 7    000000
 8    000000         banner REPTI   adds, base, count, init
 9    000000
10    000000                CLRF    adds
11    000000                ENDR
11.1  000000
11.2  000000 01..           CLRF     base
11.3  000002
11.4  000002 01..           CLRF     count
11.5  000004
11.6  000004 01..           CLRF     init
12    000006
13    000006
14    000006                END
```

LISTING CONTROL DIRECTIVES

These directives provide control over the assembler listing.

Directive	*Description*
`LSTCND`	Controls conditional assembly listing.
`LSTCOD`	Controls multi-line code listing.
`LSTEXP`	Controls the listing of macro generated lines.
`LSTMAC`	Controls the listing of macro definitions.
`LSTOUT`	Controls assembly listing output.
`LSTPAG`	Controls the formatting of output into pages.
`LSTREP`	Controls the listing of lines generated by repeat directives.
`LSTXRF`	Generates a cross reference table.
`PAGSIZ`	Sets the number of lines per page.
`COL`	Sets the number of columns per page.
`PAGE`	Generates a new page.

The following directives are provided for backward compatibility only, and are ignored:

`LSTFOR, LSTWID`, `TITL`, `STITL`, and `PTITL`.

SYNTAX

```
LSTCND{+ | -}
LSTCOD{+ | -}
LSTEXP{+ | -}
LSTMAC{+ | -}
LSTOUT{+ | -}
LSTPAG{+ | -}
LSTREP{+ | -}
LSTXRF{+ | -}
COL columns
PAGSIZ lines
PAGE
```

PARAMETERS

columns	An absolute expression in the range 80 to 132, default 80.
lines	An absolute expression in the range 10 to 150.

DESCRIPTION

Turning the listing on or off

Use `LSTOUT-` to disable all list output except error messages. This directive overrides all other list control directives.

The default is `LSTOUT+`, which lists the output (if a list file was specified).

Listing conditional code and strings

Use `LSTCND+` to force the assembler to list source code only for the parts of the assembly that are not disabled by previous conditional `IF` statements, `ELSE`, or `END`.

The default setting is `LSTCND-`, which lists all source lines.

Use `LSTCOD-` to restrict the listing of output code to just the first line of code for a source line.

The default setting is `LSTCOD+`, which lists more than one line of code for a source line, if needed; ie long ASCII strings will produce several lines of output. Code generation is *not* affected.

Controlling the listing of macros

Use `LSTEXP-` to disable the listing of macro generated lines. The default is `LSTEXP+`, which lists all macro generated lines.

Use `LSTMAC+` to list macro definitions. The default is `LSTMAC-`, which disables the listing of macro definitions.

Controlling the listing of generated lines

Use `LSTREP-` to turn off the listing of lines generated by `REPT`, `REPTC`, and `REPTI` directives.

The default is `LSTREP+`, which lists the generated lines.

Generating a cross reference table

Use `LSTXRF+` to generate a cross reference table at the end of the assembly list for the current module. The table shows values and line numbers, and the type of the symbol.

The default is `LSTXRF-`, which does not give a cross reference table.

Formatting listed output

Use COL to set the number of columns per page of the assembly list. The default number of columns is 80.

Use PAGSIZ to set the number of printed lines per page of the assembly list. The default number of lines per page is 44.

Use LSTPAG+ to format the assembly output list into pages.

The default is LSTPAG-, which gives a continuous listing.

Use PAGE to generate a new page in the assembly listing if paging is active.

EXAMPLES

Turning the listing on or off

To disable the listing of a debugged section of program:

```
LSTOUT-
; Debugged section
LSTOUT+
; Not yet debugged
```

Listing conditional code and strings

The following example shows how LSTCND+ hides a call to a subroutine that is disabled by an IF directive:

```
         NAME   lstcndtst
         EXTERN print

         RSEG   prom

debug    SET    0
begin    IF     debug
         CALL   print
         ENDIF

         LSTCND+
begin2   IF     debug
         CALL   print
         ENDIF

         END
```

This will generate the following listing:

```
    1    000000
    2    000000
    3    000000                 NAME lstcndtst
    4    000000                 EXTERN  print
    5    000000
    6    000000                 RSEG prom
    7    000000
    8    000000         debug   SET     0
    9    000000         begin   IF      debug
   10    000000                 CALL print
   11    000000                 ENDIF
   12    000000
   13    000000                 LSTCND+
   14    000000         begin2  IF      debug
   16    000000                 ENDIF
   17    000000
   18    000000                 END
```

The following example shows the effect of `LSTCOD-` on the code generated by a `DB` directive:

```
    1    000000
    2    000000
    3    000000 54686973*       DB      "This is a long long long long
long
                                         long long long long long long
long
                                         long long long long long line"
    4    000064
    5    000064                 LSTCND-
    6    000064 54686973*       DB      "This is a long long long long
long
                                         long long long long long long
long
                                         long long long long long line"
    7    0000C8
    8    0000C8
    9    0000C8                 END
```

Controlling the listing of macros

The following example shows the effect of LSTMAC and LSTEXP:

```
dec2     MACRO  arg
         DECF   arg,1
         DECF   arg,1
         ENDM

         LSTMAC-
inc2     MACRO  arg
         INCF   arg,1
         INCF   arg,1
         ENDM

         EXTERN memlock
begin    dec2   memlock
         LSTEXP-
         inc2   memlock
         RETURN

         END    begin
```

This will produce the following output:

```
 1    000000
 6    000000
 7    000000
 8    000000                  LSTMAC-
13    000000
14    000000
15    000000                  EXTERN memlock
16    000000
17    000000          begin   dec2 memlock
17    000000          begin   dec2 memlock
17.1  000000 03..             DECF    memlock,1
17.2  000002 03..             DECF    memlock,1
17.3  000004                  ENDM
18    000004
19    000004                  LSTEXP-
20    000004                  inc2    memlock
21    000008 0008             RETURN
22    00000A
23    00000A
24    00000A                  END     begin
```

Formatting listed output

The following example formats the output into pages of 66 lines each with 132 columns. The LSTPAG directive organizes the listing into pages, starting each module on a new page. The PAGE directive inserts additional page breaks.

```
PAGSIZ 66  ; Page size
COL 132
LSTPAG+
…
ENDMOD
MODULE
…
PAGE
…
```

C-STYLE PREPROCESSOR DIRECTIVES

The following C-language preprocessor directives are available:

Directive	*Description*
`#define`	Assigns a value to a label.
`#undef`	Undefines a label.
`#if`	Assembles instructions if a condition is true.
`#ifdef`	Assembles instructions if a symbol is defined.
`#ifndef`	Assembles instructions if a symbol is undefined.
`#elif`	Introduces a new condition in a `#if`...`#endif` block.
`#else`	Assembles instructions if a condition is false.
`#endif`	Ends a `#if`, `#ifdef`, or `#ifndef` block.
`#include`	Includes a file.
`#message`	Generates a message on standard output.
`#error`	Generates an error.

SYNTAX

```
#define label text
#undef label
#if condition
#ifdef label
#ifndef label
#elif condition
#else
#endif
#include {"filename" | <filename>}
#error "message"
#message "message"
```

PARAMETERS

label	Symbol to be defined, undefined, or tested.	
text	Value to be assigned.	
condition	One of the following:	
	An absolute expression	The expression must not contain forward or external references, and any non-zero value is considered as true.
	string1=string	The condition is true if *string1* and *string2* have the same length and contents.
	string1<>string2	The condition is true if *string1* and *string2* have different length or contents.
filename	Name of file to be included.	
message	Text to be displayed.	

DESCRIPTION

The preprocessor directives are processed before other directives. As an example avoid constructs like

```
redef macro
#define \1 \2
      endm
```

since the `\1` and `\2` macro arguments will not be available during the preprocess.

Also be careful with comments; the preprocessor understands `/* */` and `//`. The following expression will evaluate to 3 since the comment `char` will be preserved by `#define`:

```
#define x 3; comment
exp EQU x*8+5
```

Defining and undefining labels

Use #define to define a temporary label.

```
#define label value
```

is similar to:

```
label VAR value
```

Use #undef to undefine a label; the effect is as if it had not been defined.

Conditional directives

Use the #if … #else … #endif directives to control the assembly process at assembly time. If the condition following the #if directive is not true, the subsequent instructions will not generate any code (ie it will not be assembled or syntax checked) until a #endif or #else directive is found.

All assembler directives (except for END), and file inclusion, may be disabled by the conditional directives. Each #if directive must be terminated by a #endif directive. The #else directive is optional, and if used, it must be inside a #if … #endif block.

#if … #endif and #if … #else … #endif blocks may be nested to any level.

Use #ifdef to assemble instructions up to the next #else or #endif directive only if a symbol is defined.

Use #ifndef to assemble instructions up to the next #else or #endif directive only if a symbol is undefined.

Including source files

Use #include to insert the contents of a file into the source file at a specified point.

Displaying errors

Use #error to force the assembler to generate an error, such as in a user defined test.

EXAMPLES

Using conditional directives

The following example uses #ifdef to check that a certain symbol is defined and in that case uses two internally defined symbols. Otherwise, the same symbols are declared EXTERNAL and a message is displayed by #message. The STAND_ALONE symbol can, for example, be defined on the command line via the -D option, see *#define (-D)*, page 49.

```
         PROGRAM target
         PUBLIC  main

#ifdef STAND_ALONE

alpha    EQU     0x20
beta     EQU     0x22

#else
         EXTERN  alpha, beta
#message "Program depends on additional information"
#endif

main:
         MOVF    alpha, 0
         ADDWF   beta, 0
         XORWF   alpha,1         ; alpha = (alpha XOR
                                 ; (alpha + beta))
         RETURN
         END     main
```

Including a source file

The following example uses `#include` to include a file defining macros into the source file. For example, the following macros could be defined in `macros.s39`:

```
; exchange a and b using c as temporary
xch      MACRO   a,b, c
         MOVF    a,0
         MOVWF   c
         MOVF    b,0
         MOVWF   a
         MOVF    c,0
         MOVWF   b
         ENDMAC
```

The macro definitions can then be included, using `#include`, as in the following example.

```
         NAME    include

R0 DEFINE 0x20
R1 DEFINE 0x21
R2 DEFINE 0x22

; standard macro definitions

#include "macros.s39"

; program

main:
         xch     R0,R1,R2
         RETURN

         END     main
```

DATA DEFINITION OR ALLOCATION DIRECTIVES

These directives define temporary values or reserve memory.

Directive	*Description*
`DB`	Generates 8-bit byte constants, including strings.
`DW`	Generates 16-bit word constanst, including strings.
`DL`	Generates 32-bit long word constants and IEEE floats.
`DF`	Generates 32-bit long word constants and IEEE floats.
`DS`	Allocates space for 8-bit bytes.
`RES`	Allocates space for 16-bit words.
`DT`	Generates 8-bit table data with RETLW instruction.

SYNTAX

```
DB expr [,expr] ...
DW expr [,expr] ...
DD expr [,expr] ...
DS expr
RES expr
DT expr [,expr] ...
```

PARAMETERS

expr	A valid absolute, relocatable, or external expression, or an ASCII string. ASCII strings will be zero filled to a multiple of the size. Double-quoted strings will be zero-terminated.

DESCRIPTION

Use `DB`, `DW`, `DF` and `DD` to reserve and initialize and reserve memory space.

Use `DS` and `RES` to reserve uninitialized memory.

Use `DT` to create 8-bit table data in program memory, suitable to access with `CALL` instruction.

EXAMPLES

Generating lookup table

The following example generates a constant table of 8-bit data that is accessed via the `call` instruction and added up to a sum.

```
        NAME    table

        RSEG    CODE
table:  DT      12
        DT      15
        DT      17
        DT      16
        DT      14
        DT      11
        DT      9

        RSEG    CODE
sum     DEFINE  0x20
        COUNT   SET 0

fsum:
        REPT    7
        IF      COUNT == 7
        EXITM
        ENDIF
        CALL    table+COUNT         ; load table data in
                                    ; WREG
        ADDWF   sum,1               ; ADD up
COUNT   SET     COUNT+1
        ENDR
        MOVF    sum,0               ; Get sum into WREG
        RETURN

        END
```

Defining strings

To define a string:

```
mymess  DT      'Please enter your name'
```

To define a string which includes a trailing zero:

```
myCstr  DT      "This is a string."
```

To include a single quote in a string, enter it twice; for example:

```
errmess  DT      'Don''t understand!'
```

Reserving space
To reserve space for 0xA bytes:

```
table    DS      0xA
```

ASSEMBLER CONTROL DIRECTIVES

These directives provide control over the operation of the assembler.

Directive	*Description*
`$`	Includes a file.
`/*comment*/`	C-style comment delimiter.
`//`	C++ style comment delimiter.
`RADIX`	Sets the default base.
`CASEON`	Enables case sensitivity.
`CASEOFF`	Disables case sensitivity.

SYNTAX

```
$filename
/*comment*/
//comment
RADIX expr
CASEON
CASEOFF
```

PARAMETERS

filename	Name of file to be included. The `$` character must be the first character on the line.
comment	Comment ignored by the assembler.
expr	Default base; default 10 (decimal).

DESCRIPTION

Use `$` to insert the contents of a file into the source file at a specified point.

Use `/* … */` to comment sections of the assembler listing.

Use `//` to mark the rest of the line as comment.

Use `RADIX` to set the default base for use in conversion of constants from ASCII source to the internal binary format.

To reset the base from 16 to 10 *`expr`* must be written in hexadecimal. For example:

```
RADIX 0x0A
```

Controlling case sensitivity

Use `CASEON` or `CASEOFF` to turn on or off case sensitivity for user-defined symbols. By default case sensitivity is off.

When `CASEOFF` is active all symbols are stored in upper case, and all symbols used by XLINK should be written in upper case in the XLINK definition file.

EXAMPLES

Including a source file

The following example uses `$` to include a file defining macros into the source file. For example, the following macros could be defined in `macros.s39`:

```
table    DS      0xA

; exchange a and b using c as temporary
xch      MACRO   a,b, c
         MOVF    a,0
         MOVWF   c
         MOVF    b,0
         MOVWF   a
         MOVF    c,0
         MOVWF   b
         ENDMAC
```

The macro definitions can be included with a `$` directive, as in:

```
         NAME    include

R0 DEFINE 0x20
R1 DEFINE 0x21
R2 DEFINE 0x22

; standard macro definitions

#include "macros.s39"

; program

main:
         xch     R0,R1,R2
         RETURN

         END     main
```

Defining comments

The following example shows how `/* … */` can be used for a multi-line comment:

```
/*
Program to read serial input.
Version 2: 19.1.98
Author: mjp
*/
```

Controlling case sensitivity

When `CASEOFF` is set, in the following example `label` and `LABEL` are identical:

```
label    NOP         ; Stored as "LABEL"
         GOTO LABEL
```

The following will generate a duplicate label error:

```
label    NOP
LABEL    NOP         ; Error, "LABEL" already defined
```

STATIC OVERLAY DIRECTIVES

The static overlay directives are used to ease coexistence of routines written in C and assembler. For information on how these directives can be utilized, see the *PICmicro™ C Compiler Programming Guide.*

The directives are: `LOC`, `PRM`, `FUNCTION`, `LOCFRAME`, `ARGFRAME`.

COMPATIBILITY WITH MPASM DIRECTIVES

One of the primary goals of IAR's unified toolsets for different targets, is to minimize learning time and increase productivity. This is accomplished in the assemblers by a high degree of uniformity in the set of operators and directives accepted.

To help you, as a developer, to make the transition from MPASM to IAR, we have aimed to support all of the MPASM syntax regarding instructions, operands, arithmetic operators and labels.

In the case we do not fully support a specific directive, the assembler issues a warning that states the amount of support. However, we do not support the full set of directives found in MPASM.

LEVELS OF SUPPORT

There are three informal levels of support:

Full support
There is a generic IAR directive with the same name that performs more or less the same task but is probably slightly more general.

Example:

`IF` - conditional assembly,

`DB` - data allocation.

Limited support
The directive is transformed to an alternative form that performs some minimum action, which could be enough but is probably not.

Example:

`UDATA` - transformed into an `RSEG` directive with a fixed segment name.

`LIST` - some options are handled fully while some are ignored and will only invoke a warning message.

No support at all

The directive is recognized and ignored. It will only invoke a warning message.

Example:

CBLOCK - The assembler just reads up to the corresponding ENDC with the consequence that all labels appearing in the block are undefined.

__CONFIG - Ignored. The same functionality can be achieved by utilizing the segment concept, the data definition directives and the linker.

DIRECTIVES WITH LIMITED SUPPORT

The following directives have limited support:

MPASM directive	*Handling*
EXPAND NOEXPAND	Transformed to the corresponding LIST option.
UDATA	The directive is transformed to an RSEG directive and the segment is given the name UNDATA. This means that your code probably assembles without problems. If the original code contained several segment names in order to enable linking to different memory areas, the meaning will, however, be lost. It is therefore suggested that you change your code to explicitly use RSEG.
UDATA_SHR	Transformed to the RSEG directive. The hard-coded name of the segment is SHR_DATA. Substitute an RSEG <*name*> directive and use XLINK to place the segment in memory.
UDATA_OVR	Transformed to the COMMON directive with segment name OVR_DATA. The function is the same, but if you want different overlayed segments, rewrite your code to utilize COMMON.
VARIABLE CONSTANT	CONSTANT can only handle one expression at a time. VARIABLE is treated as CONSTANT; use SET instead.
DE	The assembler transforms this to a DW directive without range check. If the expression is within 8 bits, the same effect is accomplished.

MPASM directive	*Handling*
DATA	The assembler transforms this to a DW directive. The function is the same.
LIST	b, f, free, fixed, mm, p, t, st, and w options are ignored. c, n, r, and x options are handled in IAR manner by transforming each option to the corresponding IAR directive. c --> COL n --> PAGSIZ r --> RADIX x --> LISTEXP and LSTMAC

NON-SUPPORTED DIRECTIVES

The following directives are not supported:

__BADRAM, __MAXRAM, BANKISEL, BANKSEL, PROCESSOR, PAGESEL, CODE, ERRORLEVEL, MESSG, __CONFIG, __IDLOCS, CBLOCK.

ERROR, IFDEF, IFNDEF, INCLUDE.

NOLIST, TITLE, SUBTITLE, SPACE.

MPASM directive	*Handling*
CODE	Use directive RSEG instead to be compliant with the segment/module model.
__CONFIG __IDLOCS	Create a segment with RSEG that uses DW to initialize a word of memory with the desired bit pattern. Then use the linker to place this segment at the right place. Check the standard link file for a predefined entry with segment name and link address.
BANKSEL BANKISEL PAGESEL	These are incompatible with the IAR assembler structure and thus not supported.
ERROR IFDEF IFNDEF INCLUDE	Use the proper CPP preprocessor directives instead, i.e. #ERROR, #IFDEF, #IFNDEF, #INCLUDE.

MPASM directive	*Handling*

`NOLIST`
`TITLE`
`SUBTITLE`
`SPACE`
	Use IAR list control directives instead.

ASSEMBLER INSTRUCTIONS

This chapter lists the mnemonics of the PICmicro™ microcontroller versions.

PIC16CXX INSTRUCTION SET

The PIC16CXX microcontroller uses a 14-bit wide instruction set. The PIC16CXX instruction set consists of 36 instructions, each a single 14-bit wide word. Most instructions operate on a file register, f, and the working register, W (accumulator), while some instructions operate on constants (k). The result can be directed (d) either to the file register (f) or the working register (W) or, for some instruction, to both. A few instructions operate solely on a file register, for example BSF which specifies the bit set (b).

LITERAL AND CONTROL OPERATIONS

Instruction	*Operands*	*Description*
ADDLW	k	Add literal to W.
ANDLW	k	AND literal and W.
CALL	k	Call subroutine.
CLRWDT		Clear watchdog timer.
GOTO	k	Goto address (k is nine bits).
IORLW	k	Incl. OR literal and W.
MOVLW	k	Move literal to W.
OPTION		Load OPTION register.
RETFIE		Return from interrupt.
RETLW	k	Return with literal in W.
RETURN		Return from subroutine.
SLEEP		Go into stand-by mode.
SUBLW	k	Subtract W from literal.

Instruction	*Operands*	*Description*
`TRIS`	`f`	Tristate port `f`.
`XORLW`	`k`	Exclusive OR literal and `W`.

BYTE-ORIENTED FILE REGISTER OPERATIONS

Instruction	*Operands*	*Description*
`ADDWF`	`f,d`	Add `W` and `f`.
`ANDWF`	`f,d`	AND `W` and `f`.
`CLRF`	`f`	Clear `f`.
`CLRW`		Clear `W`.
`COMF`	`f,d`	Compliment `f`.
`DECF`	`f,d`	Decrement `f`.
`DECFSZ`	`f,d`	Decrement `f`, skip if zero.
`INCF`	`f,d`	Increment `f`.
`INCFSZ`	`f,d`	Increment `f`, skip if zero.
`IORWF`	`f,d`	Inclusive OR `W` and `f`.
`MOVF`	`f,d`	Move `f`.
`MOVWF`	`f`	Move `W` to `f`.
`NOP`		No operation.
`RLF`	`f,d`	Rotate left `f`.
`RRF`	`f,d`	Rotate right `f`.
`SUBWF`	`f,d`	Subtract W from `f`.
`SWAPF`	`f,d`	Swap halves `f`.
`XORWF`	`f,d`	Exclusive OR W and `f`.

BIT-ORIENTED FILE REGISTER OPERATIONS

Instruction	*Operands*	*Description*
BCF	f,b	Bit clear f.
BSF	f,b	Bit set f.
BTFSC	f,b	Bit test, skip if clear.
BTFSS	f,b	Bit test, skip if set.

SPECIAL INSTRUCTION MNEMONICS

			Equivalent	
Instruction	*Operands*	*Description*	*operation(s)*	
ADDCF	f,d	Add carry to file.	BTFSC INCF	3,0 f,d
ADDDCF	f,d	Add digit carry to file.	BTFSC INCF	3,1 f,d
B	k	Branch.	GOTO	k
BC	k	Branch on carry.	BTFSC GOTO	3,0 k
BDC	k	Branch on digit carry.	BTFSC GOTO	3,1 k
BNC	k	Branch on no carry.	BTFSS GOTO	3,0 k
BNDC	k	Branch on no digit carry.	BTFSS GOTO	3,1 k
BZ	k	Branch on zero.	BTFSC GOTO	3,2 k
CLRC		Clear carry.	BCF	3,0
CLRDC		Clear digit carry.	BCF	3,1
CLRZ		Clear zero.	BCF	3.2
LCALL	k			
LGOTO	k			
MOVFW	f	Move file to W.	MOVF	f,0

Instruction	*Operands*	*Description*	*Equivalent operation(s)*	
NEGF	f,d	Negate file.	COMF INCF	f,1 f,d
SETC		Set carry.	BSF	3,0
SETDC		Set digit carry.	BSF	3,1
SETZ		Set zero.	BSF	3,2
SKPC		Skip on carry.	BTFSS	3.0
SKPDC		Skip on digit carry	BTFSS	3,1
SKPNC		Skip on no carry.	BTFSC	3.0
SKPNDC		Skip on no digit carry.	BTFSC	3,1
SKPNZ		Skip on non zero.	BTFSC	3,2
SKPZ		Skip on zero.	BTFSS	3,2
SUBCF	f,d	Subtract carry from file.	BTFSC DECF	3,0 f,d
SUBDCF	f,d	Subtract digit carry from file.	BTFSC DECF	3,1 f,d
TSTF	f	Test file.	MOVF	f,1

PIC17CXX INSTRUCTION SET

The PIC17CXX microcontroller uses a 16-bit wide instruction set. The PIC17CXX instruction set consists of 55 instructions, each a single 16-bit wide word. Most instructions operate on a file register, f, and the working register, W (accumulator), while some instructions operate on constants (k). The result can be directed (d) either to the file register (f) or the working register (W) or, for some instruction, to both. Some devices in this family also includes hardware multiply instructions. A few instructions operate solely on a file register, for example BSF. In addition there are instructions for table read/write operations which specify high or low byte access (t) and increment the table address (i).

DATA MOVEMENT INSTRUCTIONS

Instruction	*Operands*	*Description*
MOVFP	f,p	Move f to p.
MOVLB	k	Move literal to BSR.
MOVLR	k	Move literal to RAM page select.
MOVFP	p,f	Move p to f.
MOVWF	f	Move W to f.
TABLRD	t,i,f	Read data from table latch into file f, then update table latch with 16-bit contents of memory location addressed by table pointer.
TABLWT	t,i,f	Writes data from file f to table latch and then write 16-bit table latch to program memory location addressed by table pointer.
TLRD	t,f	Read data from table latch into file f. Table latch unchanged.
TLWT	t,f	Write data from file f into table latch.

ARITHMETIC AND LOGICAL INSTRUCTIONS

Instruction	*Operands*	*Description*
ADDLW	k	Add literal to W.
ADDWF	f,d	Add W to f.
ADDWFC	f,d	Add W and carry to f.

Instruction	*Operands*	*Description*
ANDLW	k	AND literal and W.
ANDWF	f,d	AND W with f.
CLRF	f,d	Clear f and clear d.
COMF	f,d	Complement f.
DAW	f,d	Dec. adjust W, store in f, d.
DECF	f,d	Decrement f.
INCF	f,d	Increment f.
IORLW	k	Inclusive OR literal with W.
IORWF	f,d	Inclusive or W with f.
MOVLW	k	Move literal to W.
MULLW	k	Multiply literal and W.
MULWF	f	Multiply W and f.
NEGW	f,d	Negate W, store in f and d.
RLCF	f,d	Rotate left through carry.
RLNCF	f,d	Rotate left (no carry).
RRCF	f,d	Rotate right through carry.
RRNCF	f,d	Rotate right (no carry).
SETF	f.d	Set f and set d.
SUBLW	k	Subtract W from literal.
SUBWF	f,d	Subtract W from f.
SUBWFB	f,d	Subtract from f with borrow.
SWAPF	f,d	Swap f.
XORLW	k	Exclusive OR literal with W.
XORWF	f,d	Exclusive OR W with f.

BIT HANDLING INSTRUCTIONS

Instruction	*Operands*	*Description*
BCF	f,b	Bit clear f.
BSF	f,b	Bit set f.
BTFSC	f,b	Bit test, skip if clear.
BTFSS	f,b	Bit test, skip if set.
BTG	f,b	Bit toggle f.
CALL	k	Subroutine call (within 8k page).
CPFSEQ	f	Compare f/W, skip if f=W.
CPFSGT	f	Compare f/W, skip if f>W.
CPFSLT	f	Compare f/W, skip if f<W.
DECFSZ	f,d	Decrement f, skip if 0.
DCFSNZ	f,d	Decrement f, skip if not 0.
GOTO	k	Unconditional branch (within 8k page).
INCFSZ	f,d	Increment f, skip if zero.
INFSNZ	f,d	Increment f, skip if not zero.
LCALL	k	Long call (within 64k).
RETFIE		Return from interrupt, enable interrupt.
RETLW	k	Return with literal in W.
RETURN		Return from subroutine.
TSTFSZ	f	Test f, skip if zero.

SPECIAL CONTROL INSTRUCTIONS

Instruction	*Operands*	*Description*
CLRWDT		Clear watchdog timer.
SLEEP		Enter sleep mode.

XLINK LINKER

The following chapter describes the IAR Systems XLINK Linker, and gives examples of how it can be used.

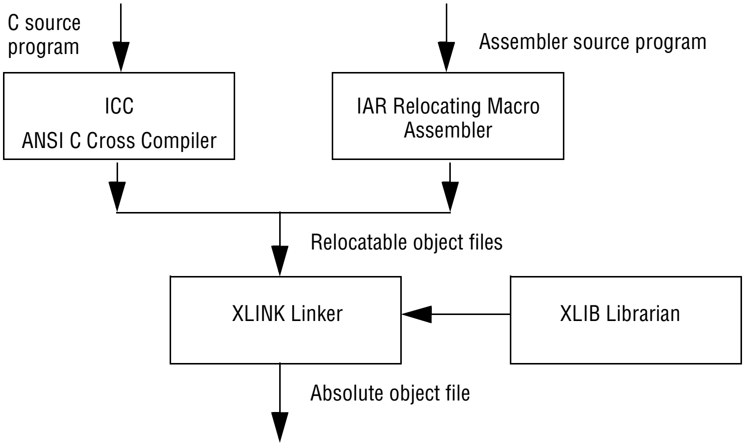
Note that some of the options described in the following chapters may not be available for your specific assembler.

INTRODUCTION

The XLINK Linker is a powerful, flexible software tool for use in the development of embedded-controller applications. XLINK reads one or more relocatable object files produced by the IAR Systems Assembler or C Compiler and produces absolute, machine-code programs as output.

It is equally well suited for linking small, single-file, absolute assembler programs as it is for linking large, relocatable, multi-module, C, or mixed C and assembler programs.

The following diagram illustrates the linking process:



OBJECT FORMAT

The object files produced by the IAR Systems Assembler and C Compiler use a proprietary format called UBROF, which stands for Universal Binary Relocatable Object Format. An application can be made up of any number of UBROF relocatable files, in any combination of assembler and C programs.

XLINK FUNCTIONS

XLINK performs three distinct functions when you link a program:

- It loads modules containing executable code or data from the input file(s).
- It links the various modules together by resolving all global (ie non-local, program-wide) symbols that could not be resolved by the assembler or compiler.
- It loads modules needed by the program from user-defined or IAR-supplied libraries.
- It locates each segment of code or data at a user-specified address.

LIBRARIES

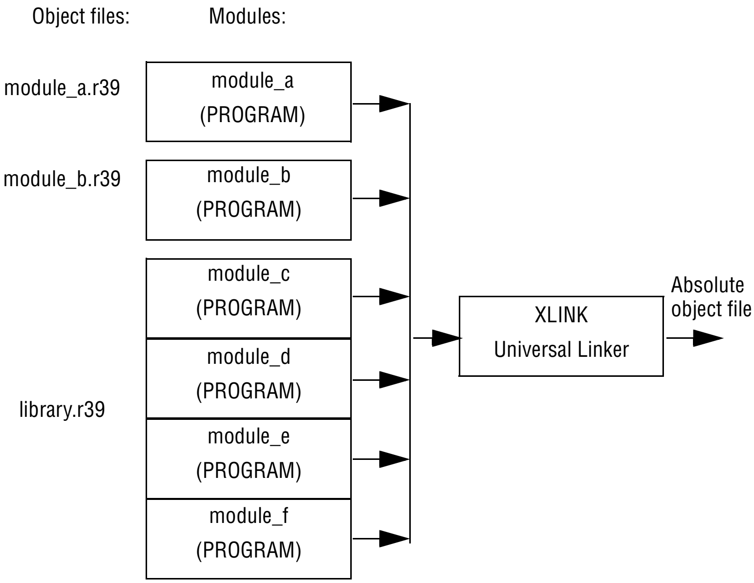
When XLINK reads a library file (which can contain multiple C or assembler modules) it will only load those modules which are actually needed by the program you are linking. This avoids having to load all the modules in a library file when you only need one routine. The XLIB Librarian is used to manage these library files.

OUTPUT FORMAT

The final output produced by XLINK is an absolute, executable object file that can be put into an EPROM, downloaded to a hardware emulator, or executed on the PC or workstation using the IAR Systems C-SPY debugger.

INPUT FILES AND MODULES

The following diagram shows how XLINK processes input files and load modules for a typical assembler or C program:



The main program has been assembled from two source files, `module_a.s39` and `module_b.s39`, to produce two relocatable files. Each of these files consists of a single module `module_a` and `module_b`. By default, the assembler assigns the `PROGRAM` attribute to both `module_a` and `module_b`. This means that they will always be loaded and linked whenever the files they are contained in are processed by XLINK.

The code and data from a single C source file ends up as a single module in the file produced by the compiler. In other words, there is a one-to-one relationship between C source files and C modules. By default, the compiler gives this module the same name as the original C source file. Libraries of multiple C modules can only be created using XLIB.

Assembler programs can be constructed so that a single source file contains multiple modules, each of which can be a program module or a library module.

LIBRARIES

In the previous diagram, the file library.r39 consists of multiple modules, each of which could have been produced by the assembler or the C compiler.

The module module_c, which has the PROGRAM attribute will *always* be loaded whenever the library.r39 file is listed among the input files for the linker. In the run-time libraries, the startup module cstartup (which is a required module in all C programs) has the PROGRAM attribute so that it will always get included when you link a C project.

The other modules in the library.r39 file have the LIBRARY attribute. Library modules are only loaded if they contain an entry (a function, variable, or other symbol declared as PUBLIC) that is referenced in some way by another module that is loaded. This way, XLINK only gets the modules from the library file that it needs to build the program. For example, if the entries in module_e are not referenced by any loaded module, module_e will not be loaded.

This works as follows:

If module_a makes a reference to an external symbol, XLINK will search the other input files for a module containing that symbol as a PUBLIC entry; ie a module where the entry itself is located. If it finds the symbol declared as PUBLIC in module_c, it will then load that module (if it has not already been loaded). This procedure is iterative, so if module_c makes a reference to an external symbol the same thing happens.

It is important to understand that a library file is just like any other relocatable object file. There is really no distinct type of file called a library (modules have a LIBRARY or PROGRAM attribute). What makes a file a library is what it contains and how it is used. Put simply, a library is a .r39 file that contains a group of related, often-used modules, most of which have a LIBRARY attribute so that they can be loaded on a demand-only basis.

CREATING LIBRARIES

You can create your own libraries, or add to existing libraries, using C or assembler modules. The C compiler -b option can be used to make a C module have a LIBRARY attribute instead of the default PROGRAM attribute. In assembler programs, the MODULE directive is used to give a module the LIBRARY attribute, and the NAME directive is used to give a module the PROGRAM attribute.

The XLIB Librarian is used to create and manage libraries. Among other tasks, it can be used to alter the attribute (`PROGRAM/LIBRARY`) of any other module after it has been compiled or assembled.

SEGMENT LOCATION

Once XLINK has identified the modules to be loaded for a program, one of its most important functions is to assign load addresses to the various code and data segments that are being used by the program.

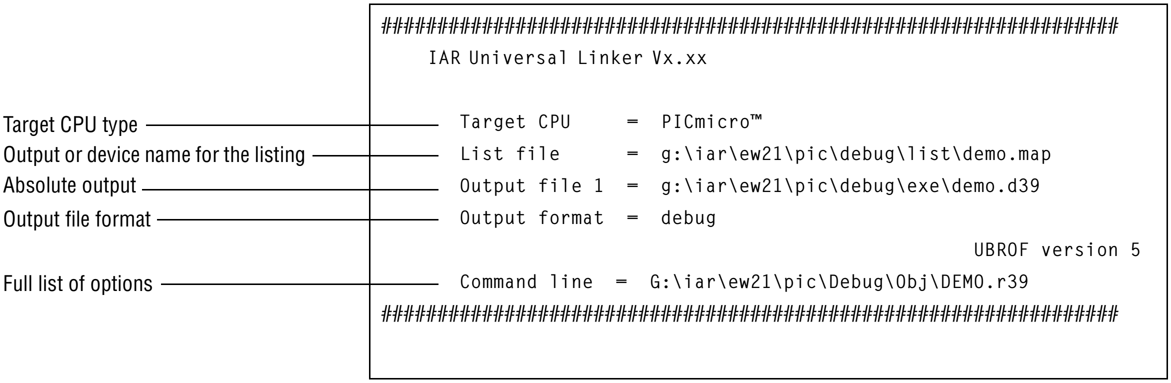
In assembly language programs the programmer is responsible for declaring and naming relocatable segments and determining how they are used. In C programs the compiler creates and uses a set of pre-defined code and data segments, and the programmer has only limited control over segment naming and usage.

LISTING FORMAT

The default XLINK listing consists of the following sections:

HEADER

Shows the command line options selected for the XLINK command:



```
##################################################################
    IAR Universal Linker Vx.xx


       Target CPU     =  PICmicro™
       List file      =  g:\iar\ew21\pic\debug\list\demo.map
       Output file 1  =  g:\iar\ew21\pic\debug\exe\demo.d39
       Output format  =  debug
                                                     UBROF version 5
       Command line  =  G:\iar\ew21\pic\Debug\Obj\DEMO.r39
##################################################################
```

Target CPU type
Output or device name for the listing
Absolute output
Output file format
Full list of options

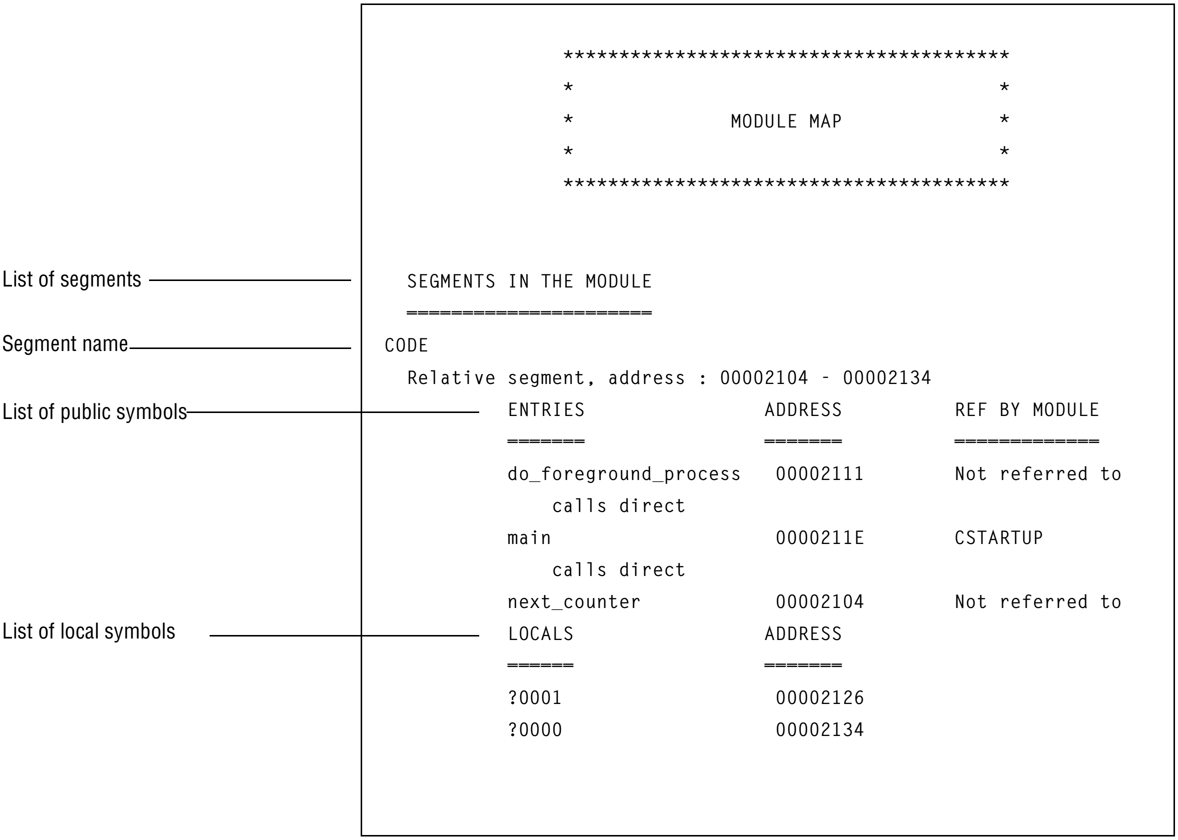
The full list of options shows the options specified on the command line. Options in command files specified with the `-f` option are also shown, in brackets.

CROSS REFERENCE

The cross reference consists of the entry list, module map and/or the segment map. It includes the program entry point, used in some output formats for hardware emulator support; see the assembler `END` directive in *Module control directives*, page 100.

Module map (-xm)

The module map consists of a subsection for each module that was loaded as part of the program. Each subsection shows the following information:


List of segments
Segment name
List of public symbols
List of local symbols

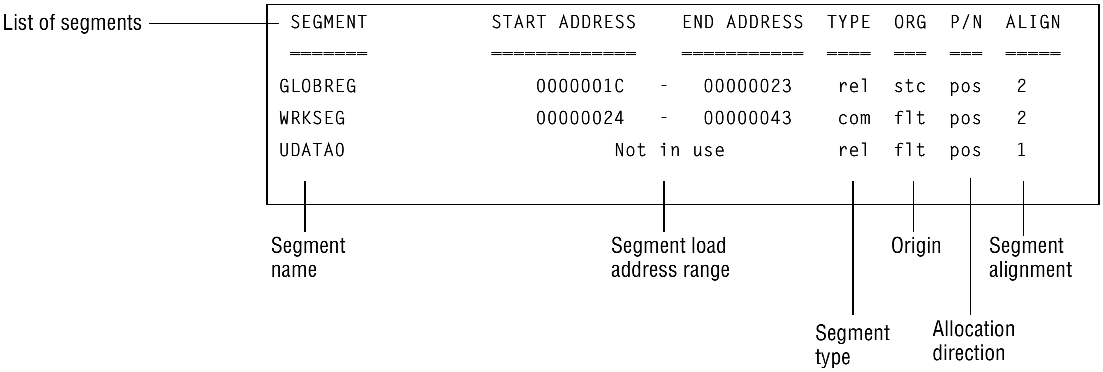
```
                ****************************************
                *                                      *
                *              MODULE MAP              *
                *                                      *
                ****************************************


  SEGMENTS IN THE MODULE
  ======================
CODE
  Relative segment, address : 00002104 - 00002134
           ENTRIES                ADDRESS          REF BY MODULE
           =======                =======          =============
           do_foreground_process   00002111        Not referred to
               calls direct
           main                    0000211E        CSTARTUP
               calls direct
           next_counter            00002104        Not referred to
           LOCALS                 ADDRESS
           ======                 =======
           ?0001                   00002126
           ?0000                   00002134
```

If the module contains any non-relocatable parts, they are listed before the segments.

Segment list (-xs)

The segment list gives the segments in increasing address order:



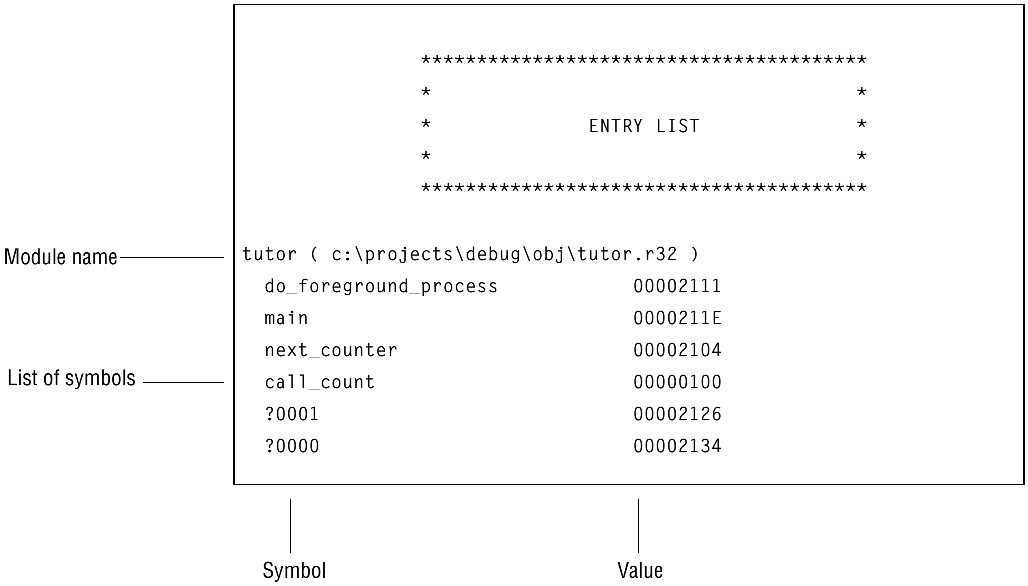
```
 SEGMENT           START ADDRESS    END ADDRESS  TYPE  ORG  P/N  ALIGN
 =======           =============    ===========  ====  ===  ===  =====
GLOBREG                0000001C   -   00000023    rel  stc  pos    2
WRKSEG                 00000024   -   00000043    com  flt  pos    2
UDATA0                        Not in use          rel  flt  pos    1
```

This lists the start and end address for each segment, and the following parameters:

Parameter	*Description*
`TYPE`	The type of segment: `rel` Relative `stc` Stack. `bnk` Banked. `com` Common. `dse` Defined but not used.
`ORG`	The origin; the type of segment start address: `stc` Absolute, for `ASEG` segments. `flt` Floating, for `RSEG`, `COMMON`, or `STACK` segments.
`P/N`	Positive/Negative; how the segment is allocated: `pos` Upwards, for `ASEG`, `RSEG`, or `COMMON` segments. `neg` Downwards, for `STACK` segment.
`ALIGN`	The segment is aligned to the next `2^ALIGN` address boundary.

Symbol listing (-xe)

Shows the entry name and address for each module and filename.



```
                ****************************************
                *                                      *
                *              ENTRY LIST              *
                *                                      *
                ****************************************

tutor ( c:\projects\debug\obj\tutor.r32 )
  do_foreground_process            00002111
  main                             0000211E
  next_counter                     00002104
  call_count                       00000100
  ?0001                            00002126
  ?0000                            00002134
```

Module name
List of symbols
Symbol
Value

CHECKSUMMED AREAS AND MEMORY USAGE

If the **Generate checksum** (`-J`) and **Fill unused code memory** (`-H`) options have been specified, the listing includes a list of the checksummed areas, in order:

```
         *********************************
            CHECKSUMMED AREAS, IN ORDER
         *********************************


   00000000   -   00007FFFin CODE memory
   0000D414   -   0000D41Fin CODE memory
Checksum = 32e19
         *********************************
               END OF CROSS REFERENCE
         *********************************
   2068 bytes of CODE memory   (30700 range fill)
   2064 bytes of DATA memory   (12 range fill)
Errors: none
Warnings: none
```

This is followed, irrespective of the options selected, by the memory usage, and the number of errors and warnings.

XLINK OPTIONS SUMMARY

XLINK options allow you to control the operation of XLINK from the command line or from the Embedded Workbench.

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

Output
#define
Diagnostics
List
Include
Input
Processing
Command line
Segment control

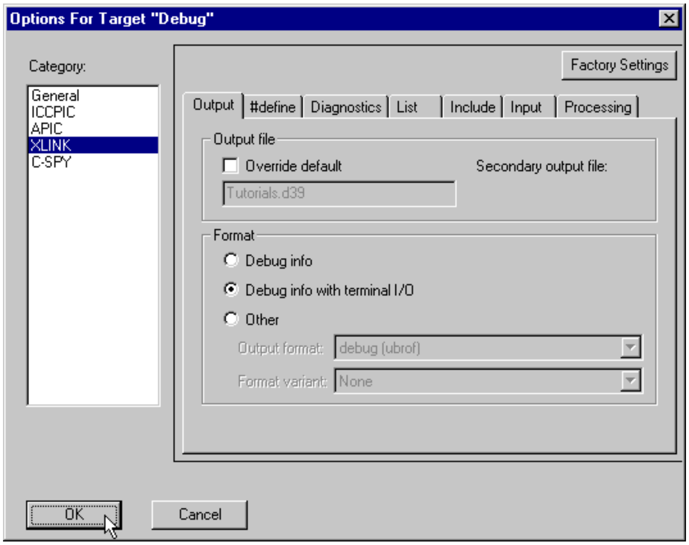
The *Command line* and *Segment control* sections provide information about additional options which are only available in the command line version, or in an extended command line, XCL, file.

For full reference about each option, see the following chapter, *XLINK options reference*.

SETTING XLINK OPTIONS

Setting XLINK options in the Embedded Workbench

To set XLINK options in the Embedded Workbench choose **Options…** from the **Project** menu, and select **XLINK** in the **Category** list to display the XLINK options pages:



Category: General, ICCPIC, APIC, XLINK, C-SPY

Output file

□ Override default

Tutorials.d39

Secondary output file:

Format

○ Debug info

● Debug info with terminal I/O

○ Other

Output format: debug (ubrof)

Format variant: None

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

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

Setting XLINK options from the command line

To set options from the command line, either:

- Specify the options on the command line, after the `xlink` command.
- Specify the options in the `XLINK_ENVPAR` environment variable; see the *Environment variables* chapter.
- Specify the options in an extended command line (XCL) file, and include this on the command line with the `-f` *`file`* command.

  Note that you can include C-style `/*...*/` or `//` comments in XCL files.

SUMMARY OF OPTIONS

The following is a summary of all the XLINK options. For a full description of any option, see under the option's category name in the next chapter, XLINK options reference.

Option	*Description*	*Section*
`-!`	Comment delimeter	Command line
`-A` *`file,…`*	Load as PROGRAM	Input
`-a`	Disable static overlay	Command line
`-B`	Always generate output	Diagnostics
`-b`*`bank_def`*	Define banked segments	Segment control
`-C` *`file, …`*	Load as LIBRARY	Input
`-c`*`cpu`*	Processor type	Command line
`-D`*`symbol=value`*	Define symbol	#define
`-d`	Disable code generation	Command line
`-E` *`file,…`*	Inherent, no object code	Input
`-e`*`new=old`*`[,`*`old`*`] …`	Rename external symbols	Command line
`-F`*`forma`*`t`	Output format	Output
`-f` *`file`*	XCL filename	Include
`-G`	No global type checking	Diagnostics
`-H`*`hexvalue`*	Fill unused code memory	Processing
`-I`*`pathname`*	Include paths	Include
`-J`*`size,method`*`[,`*`comp`*`]`	Generate checksum	Processing
`-K`*`segs=inc,count`*	Duplicate code	Segment control
`-L[`*`directory]`*	List to directory	List
`-l` *`file`*	List to named file	List
`-m`	Use less host memory	Command line
`-n[c]`	Ignore local symbols	Command line
`-o` *`file`*	Output file	Output
`-P`*`pack_def`*	Define packed segments	Segment control

Option	*Description*	*Section*
`-p`*`lines`*	Lines/page	List
`-R`	Disable range check	Diagnostics
`-r`	Debug info	Output
`-rt`	Debug info with terminal I/O	Output
`-S`	Silent operation	Command line
`-t`	Temporary file	Command line
`-w[`*`n`*`|`*`s`*`|`*`t`*`]`	Disable warnings	Diagnostics
`-x[e][m][s]`	Cross reference	List
`-Y[`*`char`*`]`	Format variant	Output
`-y[`*`chars`*`]`	Format variant	Output
`-Z`*`seg_def`*	Define segments	Segment control
`-z`	Segment overlap warnings	Diagnostics

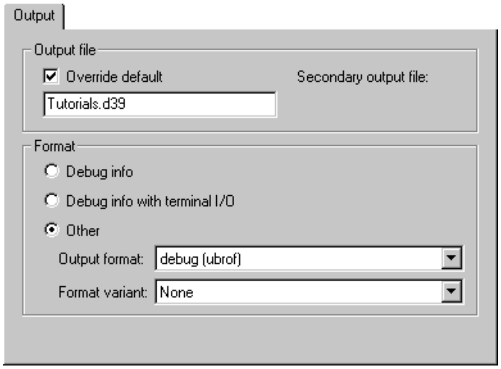
XLINK OPTIONS REFERENCE

This section gives details of the XLINK options classified according to their function.

OUTPUT

The output options are used to specify the output format and the level of debugging information.

Embedded Workbench



Output file

☑ Override default

Tutorials.d39

Secondary output file:

Format

○ Debug info

○ Debug info with terminal I/O

● Other

Output format: debug (ubrof)

Format variant: None

Command line

`-o` *`file`*	Output file.
`-r`	Debug info.
`-rt`	Debug info with terminal I/O.
`-F`*`format`*	Output format.
`-Y[`*`char`*`]`	Format variant.
`-y[`*`chars`*`]`	Format variant.

OUTPUT FILE (-o)

Syntax: `-o *file*`

Use **Output file** (`-o`) to specify the name of the XLINK output file. If a name is not specified the linker will use the name `aout.hex`. If a name is supplied without a file type, the default file type for the selected output format **Output format** (`-F`) option) will be used.

If a format is selected that generates two output files, the user-specified file type will only affect the primary output file (first format).

DEBUG INFO (-r)

Syntax: `-r`

Use **Debug info** (`-r`) to output a file in DEBUG (UBROF) format, with a `.d39` extension, to be used with the C-SPY debugger. For emulators that support the IAR Systems DEBUG format, use `-F ubrof`.

Specifying **Debug info** (`-r`) overrides any **Output format** (`-F`) option.

DEBUG INFO WITH TERMINAL I/O (-rt)

Syntax: `-rt`

Use **Debug info with terminal I/O** (`-rt`) to use the output file with the C-SPY debugger and emulate terminal I/O.

OUTPUT FORMAT (-F)

Syntax: `-F*format*`

Use **Output format** (`-F`) to select the output format.

The environment variable `XLINK_FORMAT` can be set to install an alternate default format on your system; see *XLINK_FORMAT* in the *Environment variables* chapter.

The parameter should be one of the supported XLINK output formats; for details of the formats see the chapter *XLINK output formats*.

If not specified, the default `INTEL-EXTENDED` format will be used.

Note that specifying the **Output format** (`-F`) option as `DEBUG` does not include C-SPY debug support. Use the **Debug info** (`-r`) option instead.

FORMAT VARIANT (-Y)

Syntax: `-Y[`*`char`*`]`

Use **Format variant** (`-Y`) to select enhancements available for some output formats. For more information, see the chapter *XLINK output formats*.

In the Embedded Workbench the **Format variant** options depend on the output format chosen.

FORMAT VARIANT (-y)

Syntax: `-y[`*`chars`*`]`

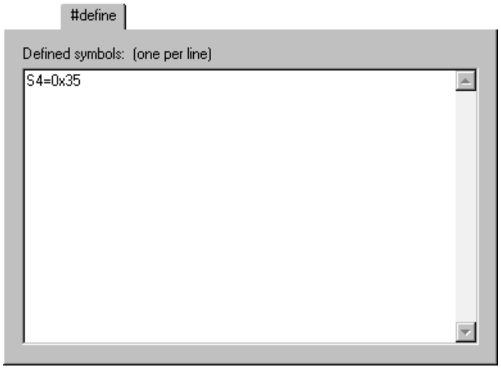
Use **Format variant** (`-y`) to specify output format variants for some formats. A sequence of flag characters can be specified after the new option `-y`. The affected formats are `IEEE695` and `XCOFF78K`.

For more information, see the chapter *XLINK output formats*.

#define

The **#define** option allows you to define symbols.

Embedded Workbench



Defined symbols: (one per line)

S4=0x35

Command line

`-Dsymbol=value` Define symbol.

DEFINE SYMBOL (-D)

Syntax: `-Dsymbol=value`

where *symbol* is any external (`EXTERN`) symbol in the program that is not defined elsewhere, and *value* the value to be assigned to *symbol*.

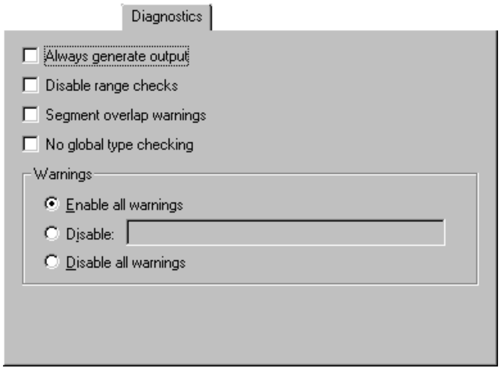
Use **Define symbol** (`-D`) to define absolute symbols at link time. This is especially useful for configuration purposes. Any number of symbols can be defined using the XCL file mode of XLINK operation. The symbol(s) defined in this manner will belong to a special module generated by the linker called `?ABS_ENTRY_MOD`.

XLINK will display an error message if you attempt to redefine an existing symbol.

DIAGNOSTICS

The **Diagnostics** options determine the error and warning messages generated by the XLINK Linker.

Embedded Workbench



Diagnostics

□ Always generate output

□ Disable range checks

□ Segment overlap warnings

□ No global type checking

Warnings

● Enable all warnings

○ Disable:

○ Disable all warnings

Command line

`-B`	Always generate output.
`-R`	Disable range check.
`-w[`*n*`|s|t]`	Disable warnings.
`-z`	Segment overlap warnings.
`-G`	No global type checking.

ALWAYS GENERATE OUTPUT (-B)

Syntax: `-B`

Use **Always generate output** (`-B`) to generate an output file even if a non-fatal error was encountered during the linking process, such as a missing global entry or a duplicate declaration. Normally, XLINK will not generate an output file if an error is encountered. Note that XLINK always aborts on fatal errors, even with the **Always generate output** (`-B`) option.

The **Always generate output** (`-B`) option allows missing entries to be patched in later in the absolute output image.

DISABLE RANGE CHECK (-R)

Syntax: `-R`

Use **Disable range check** (`-R`) to disable the address range check.

If an address is relocated out of the target CPU's address range (code, external data, or internal data address) an error message is generated. This usually indicates an error in an assembly language module or in the segment placement.

DISABLE WARNINGS (-w)

Syntax: `-w[`*n*`|s|t]`

Use **Disable warnings** (`-w`) to suppress warning messages.

The optional argument specifies which warning to disable; for example, to disable warnings 3 and 7:

```
-w3 -w7
```

Specifying `-ws` changes the return status of XLINK as follows:

Condition	*Default*	*-ws*
No errors or warnings	0	0
Warnings but no errors	0	1
One or more errors	2	2

Specifying `-wt` suppresses the detailed type information given for warnings 6 (type conflict) and 35 (multiple structs with the same tag).

If the argument is omitted all warnings are disabled.

SEGMENT OVERLAP WARNINGS (-z)

Syntax: `-z`

Use **Segment overlap warnings** (`-z`) to reduce segment overlap errors to warnings, making it possible to produce cross-reference maps, etc.

NO GLOBAL TYPE CHECKING (-G)

Syntax: `-G`

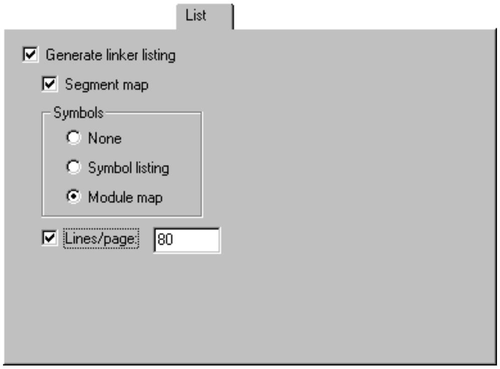
Use **No global type checking** (`-G`) to disable type checking at link time. While a well-written program should not need this option, there may be occasions where it is helpful.

By default, XLINK performs link-time type checking between modules by comparing the external references to an entry with the `PUBLIC` entry (if the information exists in the object modules involved). A warning is printed if there are mismatches.

LIST

The **List** options determine the generation of an XLINK cross-reference listing.

Embedded Workbench



List

☑ Generate linker listing

☑ Segment map

Symbols

○ None

○ Symbol listing

● Module map

☑ Lines/page: 80

Command line

`-l` *`file`*	List to named file.
`-L[`*`directory`*`]`	List to directory.
`-x[e][m][s]`	Cross reference.
`-p`*`lines`*	Lines/page.

GENERATE LINKER LISTING

Causes the linker to generate a listing and send it to the file `*project*.lst`.

List to named file (-l)

Syntax: `-l *file*`

Causes the linker to generate a listing and send it to the named file. If no extension is specified, `.lst` is used by default. However, an extension of `.map` is recommended to avoid confusing linker list files with assembler or compiler list files.

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

List to directory (-L)

Syntax: `-L[*directory*]`

Causes the linker to generate a listing and send it to the file `*directory*\*outputname*.lst`. Note that you must not include a space before the prefix.

By default, the linker does not generate a listing. To simply generate a listing, you use the `-L` option without a directory. The listing is sent to the file with the same name as the output file, but extension `.lst`.

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

CROSS REFERENCE (-x)

Syntax: `-x[e][m][s]`

Use **Cross reference** (`-x`) to include a segment map in the XLINK listing file.

The following options are available:

Workbench option	*Command line*	*Description*
Segment map	`s`	A list of all the segments in dump order.
Symbol listing	`e`	An abbreviated list of every entry (global symbol) in every module. This entry map is useful for quickly finding the address of a routine or data element.
Module map	`m`	A list of all segments, local symbols, and entries (public symbols) for every module in the program.

When the `-x` option is specified without any of the optional parameters, a default cross-reference listing will be generated which is equivalent to `-xms`. This includes:

- A header section with basic program information.
- A module load map with symbol cross-reference information.
- A segment load map in dump order.

Cross-reference information is listed to the screen if neither of the `-l` or `-L` options has been specified.

LINES/PAGE (-p)

Syntax: `-p`*`lines`*

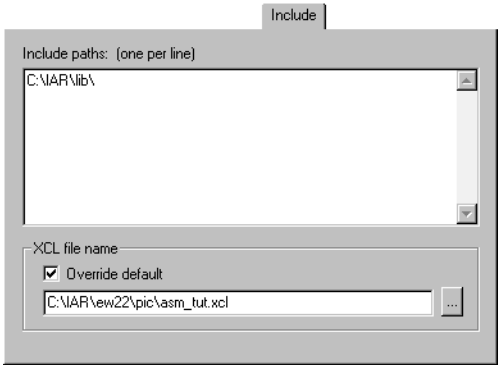
Sets the number of lines per page for the XLINK listings to *`lines`*, which must be in the range 10 to 150.

The environment variable `XLINK_PAGE` can be set to install a default page length on your system; see *XLINK_PAGE* in the *Environment variables* chapter.

INCLUDE

The **Include** option allows you to set the include path for linker command files, and specify the linker command file.

Embedded Workbench



Include

Include paths: (one per line)

C:\IAR\lib\

XCL file name

☑ Override default

C:\IAR\ew22\pic\asm_tut.xcl

Command line

`-I`*`pathname`*	Include paths.
`-f` *`file`*	XCL filename.

INCLUDE PATHS (-I)

Syntax: `-I`*`pathname`*

Specifies a pathname to be searched for object files.

By default, XLINK searches for object files only in the current working directory. The **Include paths** (`-I`) option allows you to specify the names of the directories which it will also search if it fails to find the file in the current working directory.

This is equivalent to the `XLINK_DFLTDIR` command line option; see the *Environment variables* chapter.

XCL FILENAME (-f)

Syntax: `-f` *`file`*

Use `-f` to extend the XLINK command line by reading arguments from a command file, just as if they were typed in on the command line. If not specified an extension of `.xcl` is assumed.

Arguments are entered into the XCL file with a text editor using the same syntax as on the command line. However, in addition to spaces and tabs, the end-of-line CR is also treated as a valid delimiter between arguments. A command line may be extended by the `\`↵ sequence.

Note that you can include C-style `/*...*/` or `//` comments in XCL files.

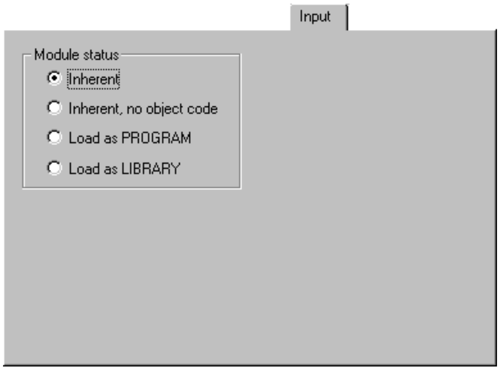
A default XCL file is selected automatically for the **General Target** memory model and processor configuration selected. You can override this by selecting **Override default**, and then specifying an alternative file.

Note that you can include C-style `/*...*/` or `//` comments in XCL files.

INPUT

The **Input** options define the status of input modules.

Embedded Workbench



Input

Module status

● Inherent
○ Inherent, no object code
○ Load as PROGRAM
○ Load as LIBRARY

Command line

`file,…`	Inherent.
`-E file,…`	Inherent, no object code.
`-A file,…`	Load as PROGRAM.
`-C file,…`	Load as LIBRARY.

INHERENT

Syntax: `file,…`

Use **Inherent** to link files normally, and generate output code.

INHERENT, NO OBJECT CODE (-E)

Syntax: `-E` *`file`*`,…`

Use **Inherent, no object code** (`-E`) to empty load specified input files; they will be processed normally in all regards by the linker but output code will not be generated for these files.

One potential use for this feature is in creating separate output files for programming multiple EPROMs. This is done by empty loading all input files except the ones you want to appear in the output file.

In the following example a project consists of four files, `file1` to `file4`, but we only want object code generated for `file4` to be put into an EPROM:

```
-E file1,file2,file3
file4
-o project.hex
```

To read object files from `v:\general\lib` and `c:\project\lib`:

```
-Iv:\general\lib;c:\project\lib
```

LOAD AS PROGRAM (-A)

Syntax: `-A file,…`

Use **Load as PROGRAM** (`-A`) to temporarily force all of the modules within the specified input files to be loaded as if they were all program modules, even if some of the modules have the `LIBRARY` attribute.

This option is particularly suited for testing library modules before they are installed in a library file, since the `-A` option will override an existing library module with the same entries. In other words, XLINK will load the module from the *input file* specified in the `-A` argument instead of one with an entry with the same name in a library module.

For example, to load the user-written library module `putchar.r39` instead of the standard one in the `CLIB` library:

```
-! these lines are in an XCL file … -!
-A putchar
CLIB
```

This assumes that the `putchar` file contains the same global entry as one of the modules in `CLIB`.

LOAD AS LIBRARY (-C)

Syntax: `-C file,…`

Use `-C` to temporarily cause all of the modules within the specified input files to be treated as if they were all library modules, even if some of the modules have the `PROGRAM` attribute. This means that the modules in the input files will be loaded only if they contain an entry that is referenced by another loaded module.

For example, to load the user-defined `CSTARTUP` module from the file `cstartup` instead of the program module of the same name in `CLIB`:

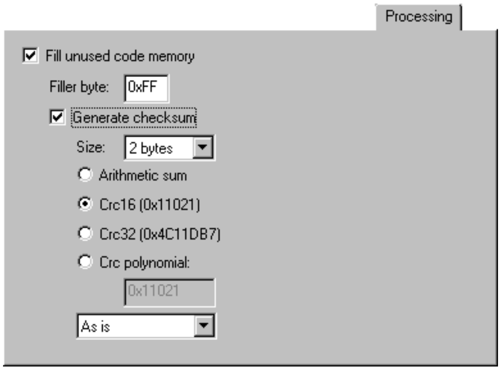
```
-! these lines are in an XCL file -!
cstartup
-C CLIB
```

This allows you to test the `CSTARTUP` module before installing it in the library.

PROCESSING

The **Processing** options allow you to specify additional options determining how the code is generated.

Embedded Workbench



Processing

☑ Fill unused code memory

Filler byte: 0xFF

☑ Generate checksum

Size: 2 bytes

○ Arithmetic sum

● Crc16 (0x11021)

○ Crc32 (0x4C11DB7)

○ Crc polynomial:

0x11021

As is

Command line

-H*hexvalue*	Fill unused code memory.
-J*size*,*method*[,*comp*]	Generate checksum.

FILL UNUSED CODE MEMORY (-H)

Syntax: `-H`*`hexvalue`*

Use **Fill unused code memory** (`-H`) to fill all gaps between segment parts introduced by the linker with the value *`hexvalue`*. The linker can introduce gaps either because of alignment restriction, or at the end of ranges given in segment placement options.

The normal behavior, when no `-H` option is given, is that these gaps are not given a value in the output file.

For example, specifying:

```
-HFF
```

fills all the gaps with the value `0xFF`.

GENERATE CHECKSUM (-J)

Syntax: `-J`*`size,method`*`[,`*`comp`*`]`

Use **Generate checksum** (`-J`) to checksum all generated raw data bytes. This option can only be used if the **Fill unused code memory** (`-H`) option has been specified.

`size` specifies the number of bytes in the checksum, and can be 1, 2, or 4.

`method` specifies the algorithm used, and can be one of the following:

Method	*Description*
`sum`	Simple arithmetic sum.
`crc16`	CRC16 (generating polynomial `0x11021`).
`crc32`	CRC32 (generating polynomial `0x104C11DB7`).
`crc=`*`n`*	CRC with a generating polynomial of *`n`*.

`comp` can be 1 to specify one's complement, or 2 to specify two's complement.

In all cases it is the least significant 1,2, or 4 bytes of the result that will be output, in the natural byte order for the processor. The CRC checksum is calculated as if the following code was called for each bit in the input, starting with a CRC of 0:

```
unsigned long
crc(int bit, unsigned long oldcrc)
{
   unsigned long newcrc = (oldcrc << 1) ^ bit;
   if (oldcrc & 0x80000000)
      newcrc ^= POLY;
   return newcrc;
}
```

POLY is the generating polynomial. The checksum is the result of the final call to this routine. If *comp* is specified, the checksum is the one's or two's compliment of the result.

The linker will place the checksum byte(s) at the label __checksum in the segment CHECKSUM. This segment must be placed using the segment placement options like any other segment.

For example, to calculate a 4-byte checksum using the generating polynomial 0x104C11DB7 and output the one's complement of the calculated value, specify:

```
-J4,crc32,1
```

COMMAND LINE

The following additional options can be set from the command line or in XCL files:

`-!` *comment* `-!`	Comment delimiter.
`-a`	Disable static overlay.
`-cPIC`	Processor type.
`-d`	Disable code generation.
`-e`*new*=*old*[,*old*] …	Rename external symbols.
`-m`	Use less host memory.
`-n[c]`	Ignore local symbols.
`-S`	Silent operation.
`-t`	Temporary file.

The C compiler includes default XCL files for each chip option and memory model.

COMMENT DELIMITER (-!)

Syntax: `-! comment -!`

A `-!` can be used to bracket off comments in an XLINK `.xcl` file. Unless the `-!` is at the beginning of a line, it must be preceded by a space or tab.

Note that you can include C-style and C++ style comments in your files; the use of these is recommended since they are less error-prone than `-!`.

Example:

```
-Fubrof7    /* UBROF7 output. */
```

DISABLE STATIC OVERLAY (-a)

Syntax: `-a{i|w}[function-lists]`

Use `-a` to control the static memory allocation of variables. The options are as follows:

Option	*Description*
`-a`	Disables overlaying totally, for debugging purposes.
`-ai`	Disables indirect tree overlaying.

Option	*Description*
`-aw`	Disables warning 16, `Function is called from two function trees`. Do this only if you are sure the code is correct.

In addition, the `-a` option can specify one or more function lists, to specify additional options for specified functions. Each function list can have the following form, where *`function`* specifies a public function or a *`module:function`* combination:

Function list	*Description*
`(`*`function,function…`*`)`	Function trees will not be overlayed with another function.
`[`*`function,function…`*`]`	Function trees will not be allocated unless they are called by another function.
`{`*`function,function…`*`}`	Indicates that the specified functions are interrupt functions.

Several `-a` options may be specified, and each `-a` option may include several suboptions, in any order.

PROCESSOR TYPE (-c)

Syntax: `-cPIC`

Use `-c` to set the CPU type to PICmicro™.

The environment variable `XLINK_CPU` can be set to install a default for the `-c` option so that it does not have to be specified on the command line; see *XLINK_CPU* in the *Environment variables* chapter.

DISABLE CODE GENERATION (-d)

Syntax: `-d`

Use `-d` to disable the generation of output code from XLINK. This option is useful for the trial linking of programs; eg checking for syntax errors, missing symbol definitions, etc. XLINK will run slightly faster for larger programs when this option is used.

RENAME EXTERNAL SYMBOLS (-e)

Syntax: `-enew=old [,old] …`

Use `-e` to configure a program at link time by redirecting a function call from one function to another.

This can also be used for creating stub functions; ie when a system is not yet complete, undefined function calls can be directed to a dummy routine until the real function has been written.

USE LESS HOST MEMORY (-m)

Syntax: `-m`

Use `-m` to reduce the amount of host system memory needed by using file pointers to all segments and modules, instead of reading all input files into RAM. If XLINK runs out of host memory during a link, this option will often help. However, XLINK will run more slowly if the `-m` option is used.

The `-m` option is equivalent to:

```
set XLINK_MEMORY=0
```

See *XLINK_MEMORY* in the *Environment variables* chapter.

IGNORE LOCAL SYMBOLS (-n)

Syntax: `-n[c]`

Use `-n` to ignore all local (non-public) symbols in the input modules. This option speeds up the linking process and can also reduce the amount of host memory needed to complete a link. If `-n` is used, locals will not appear in the listing cross-reference and will not be passed on to the output file.

Use `-nc` to ignore just compiler-generated local symbols, such as jump or constant labels. These are usually only of interest when debugging at assembler level.

Note that local symbols are only included in files if they were compiled or assembled with the appropriate option to specify this.

SILENT OPERATION (-S)

Syntax: `-S`

Use `-S` to turn off the XLINK sign-on message and final statistics report so that nothing appears on your screen while it runs. However, it does not disable error and warning messages or the listing output.

TEMPORARY FILE (-t)

Syntax: `-t`

This option is provided mainly for backward compatibility, and design improvements to XLINK make it unlikely to be needed.

Use `-t` to force XLINK to use a temporary file, with the default name `xlink.tmp` in the current directory, to store a large part of the linker symbol tables. This can significantly reduce the amount of host system memory needed to link a program with a large number of symbols; eg more than 1500. In some cases, it may be necessary to use this option to complete a link process.

Note that the `-t` option can significantly increase the time it takes to link a program. The `-m` (**Use less host memory**) file-bound processing option will also be enabled automatically when `-t` is used.

The environment variable `XLINK_TFILE` can be set to an alternate filename (with drive and directory path) to use for the temporary file; see *XLINK_TFILE* in the *Environment variables* chapter.

SEGMENT CONTROL

These options control the allocation of segments.

`-b`*`bank_def`*	Define banked segments.
`-K`*`segs=inc,count`*	Duplicate code.
`-P`*`pack_def`*	Define packed segments.
`-Z`*`seg_def`*	Define segments.

DEFINE BANKED SEGMENTS (-b)

Syntax: `-b [addrtype] [(type)] segments=first, length,increment[, count]`

where the parameters are as follows:

`addrtype` The type of load addresses used when dumping the code:

`omitted`	Logical addresses with bank number.
`#`	Linear physical addresses.
`@`	64180-type physical addresses.

`type` Specifies the memory type for all segments if applicable for the target processor. If omitted it defaults to `UNTYPED`.

`segments` The list of banked segments to be linked.

The delimiter between segments in the list determines how they are packed:

`:` (colon) The next segment will be placed in a new bank.

`,` (comma) The next segment will be placed in the same bank as the previous one.

`first` The start address of the first segment in the banked segment list. This is a 32-bit value: the high-order 16 bits represent the starting bank number while the low-order 16 bits represent the start address for the banks in the logical address area.

`length` The length of each bank, in bytes. This is a 16-bit value.

`increment` The incremental factor between banks, ie the number that will be added to *`first`* to get to the next bank. This is a 32-bit value: the high-order 16 bits are the bank increment, and the low-order 16 bits are the increment from the start address in the logical address area.

`count` Number of banks available, in decimal.

Use `-b` to allocate banked segments for a program that is designed for bank-switched operation. It also enables the banking mode of linker operation.

There can be more than one `-b` definition.

Logical addresses are the addresses as seen by the program. In most bank-switching schemes this means that a logical address contains a bank number in the most significant 16 bits and an offset in the least significant 16 bits.

Linear physical addresses are calculated by taking the bank number (the most significant 16 bits of the address) times the bank length and adding the offset (the least significant 16 bits of the address). Specifying linear physical addresses affects the load addresses of bytes output by XLINK, not the addresses seen by the program.

64180-type physical addresses are calculated by taking the least significant 8 bits of the bank number, shifting it left 12 bits and then adding the offset.

Using either of these simple translations is only useful for some rather simple memory layouts. Linear physical addressing as calculated by XLINK is useful for a bank memory at the very end of the address space. Anything more complicated will need some post-processing of XLINK output, either by a PROM programmer or a special program. See the `simple` subdirectory for source code for the start of such a program.

For example, to specify that the three code segments `BSEG1`, `BSEG2`, and `BSEG3` should be linked into banks starting at 8000, each with a length of 4000, with an increment between banks of 10000:

```
-b(CODE)BSEG1,BSEG2,BSEG3=8000,4000,10000
```

DUPLICATE CODE (-K)

Syntax: `-K`*`segs`*`=`*`inc`*`,`*`count`*

Use **Duplicate code** (`-K`) to duplicate any raw data bytes from the segments in *`segs count`* times, adding *`inc`* to the addresses each time. This will typically be used for segments mentioned in a **Define segments** (`-Z`) option.

This can be used to make part of a PROM be non-banked even though the entire PROM is physically banked. Use the **Define banked segments** (`-b`) or **Define packed segments** (`-P`) options to place the banked segments into the rest of the PROM.

For example, to duplicate the contents of the RCODE0 and RCODE1 segments 4 times, using addresses `0x20000` higher each time, specify:

```
-KRCODE0,RCODE1=20000,4
```

This will place 5 copies of each of the bytes from the segments into the output file, at the addresses `x`, `x+0x20000`, `x+0x60000`, and `x+0x80000`.

DEFINE PACKED SEGMENTS (-P)

Syntax: `-P [(`*`type`*`)]` *`segments`*`=`*`start`*`-`*`end`*`[*`*`nnn`*`[+`*`xxx`*`]][/`*`ppp`*`]` `[,`*`start`*`-`*`end`*`] …`

where the parameters are as follows:

`type` Specifies the memory type for all segments if applicable for the target processor. If omitted it defaults to `UNTYPED`.

`segments` A list of one or more segments to be linked, separated by commas.

`start`, *`end`* Addresses defining a range within which the listed *`segments`* should be placed.

`*`*`nnn`* Repeats the *`start`*`-`*`end`* range *`nnn`* times.

`+`*`xxx`* Add *`xxx`* to the range for each repetition.

`/`*`ppp`* Splits the entire *`start`*`-`*`end`* range into pages of size and alignment *`ppp`*.

Use `-P` to pack the segment parts from the specified segments into the specified ranges, where a segment part is defined as that part of a segment that originates from one module. The linker splits each segment into its segment parts and forms new segments for each of the ranges. All the ranges must be closed; ie both *start* and *end* must be specified. The segment parts will not be placed in any specific order into the ranges.

Use `*`*nnn*+*ppp* to repeat the *start-end* range *nnn* times adding *xxx* each repetition.

For page ranges you can use /*ppp* to split the entire *start-end* range into pages of *ppp* size and alignment. When using the page range the *start-end* do not have to coincide with a page boundary.

Examples
Ranges with a repeat count:

```
1000-1FFF*3+2000
```

is the same as

```
1000-1FFF,3000-3FFF,5000-5FFF
```

Page ranges:

```
50-77F/200
```

is the same as

```
50-1FF,200-3FF,400-5FF,600-77F
```

DEFINE SEGMENTS (-Z)

Syntax: `-Z [(`*type*`)]` *segments* `[=|#]`
`[`*start-end*`,][*`*nnn*`[+`*xxx*`]][/`*ppp*`] … [`*address*`]`

where the parameters are as follows:

type Specifies the memory type for all segments if applicable for the target processor. If omitted it defaults to `UNTYPED`.

segments A list of one or more segments to be linked, separated by commas.

The segments are allocated in memory in the same order as they are listed. Appending +*nnnn* to a segment name increases the amount of memory that XLINK will allocate for that segment by *nnnn* bytes.

= or # Specifies how segments are allocated.

= Allocates the segments so they begin at the start of the specified range (upwards allocation).

# Allocates the segments so they finish at the end of the specified range (downwards allocation).

If an allocation operator (and range) is not specified, the segments will be allocated upwards from the last segment that was linked, or from address 0 if no segments have been linked.

start, end Addresses defining a range within which the listed *segments* should be placed.

**nnn* Repeats the *start-end* range *nnn* times.

+xxx Add *xxx* to the range for each repetition.

/ppp Splits the entire *start-end* range into pages of size and alignment *ppp*.

address Start address for placing any remaining segments to be allocated.

Use `-Z` to specify how and where segments will be allocated in the memory map.

If the linker finds a segment in an input file that is not defined either with `-Z, -b,` or `-P,` an error is reported. There can be more than one `-Z` definition.

Additional related topics and optional forms for `-Z` are described in the following section.

Allocation segment types

The following table lists the different types of segments that can be processed by XLINK:

Segment type	*Description*
`STACK`	Allocated from high to low addresses by default. The aligned segment size is subtracted from the load address before allocation, and successive segments are placed below the preceding segment.

Segment type	*Description*
`RELATIVE` `COMMON`	Allocated from low to high addresses by default.

If stack segments are mixed with relative or common segments in a segment definition, the linker will produce a warning message but will allocate the segments according to the default allocation set by the first segment in the segment list.

Common segments have a size equal to the largest declaration found for the particular segment. That is, if module `A` declares a common segment `COMSEG` with size 4, while module `B` declares this segment with size 5, the latter size will be allocated for the segment.

Be careful not to overlay common segments containing code or initializers.

Relative and stack segments have a size equal to the sum of the different (aligned) declarations.

Memory types of segments

The optional *`type`* parameter is used to assign a type to all of the segments in the list. The *`type`* parameter affects how XLINK processes the segment overlaps. Additionally, it generates information in some of the output formats that are used by some hardware emulators and by C-SPY.

Segment type	*Description*
`CODE`	Code memory.
`CONST`	Initializer data and constants located in internal/external memory.
`DATA`	Data in bank 0-7.
`IDATA`	EEPROM memory.
`CODE`	TABLE data in internal/external memory.
`UNTYPED`	(Default) Maps to code memory.

Note: All segments located in `CODE` memory are specified using byte adressing (`CODE` and `CONST`).

Range errors

If the ranges specified in the `-Z` command are too short, it will cause either error `24 Segment` *segment* `overlaps segment` *segment*, if any segment overlaps another, or error `26 Segment` *segment* `is too long`, if the ranges are too small.

By default, XLINK checks to be sure that the various segments that have been defined (by the segment placement command and absolute segments) do not overlap in memory.

Repeat counts

Use `*`*nnn*+*ppp* to repeat the `start`-*end* range *nnn* times adding *xxx* each repetition.

Paged ranges

For page ranges you can use `/`*ppp* to split the entire `start`-*end* range into pages of *ppp* size and alignment. When using the page range the `start`-*end* do not have to coincide with a page boundary.

Examples

To locate `SEGA` at address `0`, followed immediately by `SEGB`:

```
-Z(CODE)SEGA,SEGB=0
```

To allocate `SEGA` downwards from `FFFH`, followed by `SEGB` below it:

```
-Z(CODE)SEGA,SEGB#FFF
```

To allocate specific areas of memory to `SEGA` and `SEGB`:

```
-Z(CODE)SEGA,SEGB=100-1FF,400-6FF,1000
```

In this example `SEGA` will be placed between address `100` and `1FF`, if it fits in that amount of space. If it does not, XLINK will try the range `400-6FF`. If none of these ranges are large enough to hold `SEGA`, it will start at `1000`.

`SEGB` will be placed, according to the same rules, after segment `SEGA`. If `SEGA` fits the `100–1FF` range then XLINK will try to put `SEGB` there as well (following `SEGA`). Otherwise, `SEGB` will go into the `400` to `6FF` range if it is not too large, or else it will start at `1000`.

```
-Z(NEAR)SEGA,SEGB=19000-1FFFF
```

Segments `SEGA` and `SEGB` will be dumped at addresses `19000` to `1FFFF` but the default 16-bit addressing mode will be used to access the data (ie `9000` to `FFFF`).

XLINK OUTPUT FORMATS

This chapter gives a summary of the XLINK output formats.

SINGLE OUTPUT FILE

The following formats result in the generation of a single output file:

Format	*Type*	*Extension*	*Address type*
AOMF8051†	binary	from CPU	N
AOMF8096†	binary	from CPU	N
AOMF80196†	binary	from CPU	N
AOMF251	binary	from CPU	N
ASHLING	binary	none	N
ASHLING-6301†	binary	from CPU	N
ASHLING-64180†	binary	from CPU	NS
ASHLING-6801†	binary	from CPU	N
ASHLING-8080†	binary	from CPU	NS
ASHLING-8085†	binary	from CPU	NS
ASHLING-Z80†	binary	from CPU	NS
DEBUG (UBROF)†§	binary	.dbg	NL
EXTENDED-TEKHEX†	ASCII	from CPU	NLPS
HP-CODE	binary	.x	NLPS
HP-SYMB	binary	.l	NLPS
IEEE695†**			
INTEL-EXTENDED	ASCII	from CPU	NLPS
INTEL-STANDARD	ASCII	from CPU	N
MILLENIUM (Tektronix)	ASCII	from CPU	N
MOTOROLA	ASCII	from CPU	NLPS
MPDS-CODE	binary	.tsk	N

Format	*Type*	*Extension*	*Address type*
MPDS-SYMB	binary	.sym	NLPS
MSD	ASCII	.sym	N
MSP430_TXT	ASCII	.txt	NLPS
NEC-SYMBOLIC†	ASCII	.sym	N
NEC2-SYMBOLIC†	ASCII	.sym	N
NEC78K-SYMBOLIC†	ASCII	.sym	N
PENTICA-A	ASCII	.sym	NLPS
PENTICA-B	ASCII	.sym	NLPS
PENTICA-C	ASCII	.sym	NLPS
PENTICA-D	ASCII	.sym	NLPS
RCA	ASCII	from CPU	N
SIMPLE	binary	.raw	NLPS
SYMBOLIC	ASCII	from CPU	NLPS
SYSROF†	binary	.abs	NLPS
TEKTRONIX (Millenium)	ASCII	.hex	N
TI7000 (TMS7000)	ASCII	from CPU	N
TYPED	ASCII	from CPU	NLPS
UBROF†	binary	.dbg	NL
UBROF5†	binary	.dbg	NL
UBROF6†	binary	.dbg	NL
XCOFF78k	binary	.lnk	NL
ZAX	ASCII	from CPU	NLPS

† the format depends on the typing of the segments; ie the *type* field specified in the XLINK -Z option is important.

** only supported for certain combinations of CPU and debugger; see XLINK.TXT and XMAN.TXT for more information.

§ Using -FUBROF (or -FDEBUG) will generate UBROF output matching the latest UBROF format version in the input. Using -FUBROF5 (or -FUBROF6) will force output of the specified version of the format, irrespective of the input.

Address type
The address type is one of the following:

N = Non-banked address.

L = Banked logical address.

P = Banked physical address.

S = Banked 64180 physical address.

TWO OUTPUT FILES

The following formats result in the generation of two output files:

Format	*Code format*	*Ext.*	*Symbolic format*	*Ext.*
DEBUG-MOTOROLA	DEBUG	.axx	MOTOROLA	.obj
DEBUG-INTEL-EXT	DEBUG	.axx	INTEL-EXT	.hex
DEBUG-INTEL-STD	DEBUG	.axx	INTEL-STD	.hex
HP	HP-CODE	.x	HP-SYMB	.l
MPDS	MPDS-CODE	.tsk	MPDS-SYMB	.sym
MPDS-I	INTEL-STANDARD	.hex	MPDS-SYMB	.sym
MPDS-M	Motorola	.s19	MPDS-SYMB	.sym
MSD-I	INTEL-STANDARD	.hex	MSD	.sym
MSD-M	Motorola	.hex	MSD	.sym
MSD-T	MILLENIUM	.hex	MSD	.sym
NEC	INTEL-STANDARD	.hex	NEC-SYMB	.sym
NEC2	INTEL-STANDARD	.hex	NEC2-SYMB	.sym
NEC78K	INTEL-STANDARD	.hex	NEC2-SYMB	.sym
PENTICA-AI	INTEL-STANDARD	.obj	Pentica-a	.sym
PENTICA-AM	Motorola	.obj	Pentica-a	.sym
PENTICA-BI	INTEL-STANDARD	.obj	Pentica-b	.sym
PENTICA-BM	Motorola	.obj	Pentica-b	.sym

Format	*Code format*	*Ext.*	*Symbolic format*	*Ext.*
PENTICA-CI	INTEL-STANDARD	.obj	Pentica-c	.sym
PENTICA-CM	Motorola	.obj	Pentica-c	.sym
PENTICA-DI	INTEL-STANDARD	.obj	Pentica-d	.sym
PENTICA-DM	Motorola	.obj	Pentica-d	.sym
ZAX-I	INTEL-STANDARD	.hex	ZAX	.sym
ZAX-M	Motorola	.hex	ZAX	.sym

OUTPUT FORMAT VARIANTS

The following enhancements can be selected for the specified output formats, using the **Format variant** (-Y) option:

Output format	*Option*	*Description*
PENTICA-A,B,C,D and MPDS-SYMB	Y0	Symbols as modules:symbolname.
	Y1	Labels and lines as *module:symbolname*.
	Y2	Lines as *module:symbolname*.
AOMF8051	Y0	Extra type of information for Hitex.
INTEL-STANDARD	Y0	End only with :00000001FF.
	Y1	End with PGMENTRY, else :0000001FF.
MPDS-CODE	Y0	Fill with 0xFF instead.
DEBUG, -r	Y#	Old UBROF version.
INTEL-EXTENDED	Y0	Segmented variant.
	Y1	32-bit linear variant.

Refer to the file *XLINK.TXT* for additional options that have become available since this guide was published.

Use **Format variant** (-y) to specify output format variants for some formats. A sequence of flag characters can be specified after the new option -y. The affected formats are IEEE695 and XCOFF78K.

IEEE695

For IEEE695 the available format modifier flags are:

-yg	Output global types globally.
-yl	Output global types in each module.
-yb	Treat bit sections as byte sections.
-ym	Adjust output for Mitsubishi PDB30 debugger.
-ye	No block-local constants.
-yv	Handle variable life times.
-ys	Output stack adjust records.
-ya	Output module locals in BB10 block.

The recommended format variant modifiers for specific debuggers are given below:

Debugger	*Format variant modifier*
6812 Noral debugger	-ygvs
68HC16 Microtek debugger	-ylb
740 Mitsubishi debugger	-ylmba
7700 HP RTC debugger	-ygb
7700 Toshiba RTE900 m25	-ygbe
H8300 HP RTC debugger	-ygb
H8300H HP RTC debugger	-ygb
H8S HP RTC debugger	-ygb
M16C HP RTC debugger	-ygb
M16C Mitsubishi PDB30	-ylbm
T900 Toshiba RTE900 m25	-ygbe

XCOFF78K

For `XCOFF78K` the available format modifier flags are:

`-ys`	Truncates symbols to 31 characters.
`-yp`	Strips source file paths.
`-ye`	Includes module enums.
`-yl`	Hobbles line number info.

To specify more than one flag, they must all be specified after the same `-y` option. For example, to use both the `s` and the `p` flag, use `-ysp`.

XLIB LIBRARIAN

This chapter describes the XLIB Librarian, which is designed to allow you to create and maintain relocatable libraries of routines.

INTRODUCTION

Like the XLINK Linker, the XLIB Librarian uses the UBROF standard object format (Universal Binary Relocatable Object Format) to allow it to support a wide range of 32-bit byte-oriented processors (applies to almost all current major microprocessors).

LIBRARIES

A library is a single file that contains a number of relocatable object modules, each of which can be loaded independently from other modules in the file as it is needed.

Normally, the modules in a library file all have the `LIBRARY` attribute, which means that they will only be loaded by the linker if they are actually needed in the program. This is referred to as *demand loading* of modules.

On the other hand, a module with the `PROGRAM` attribute is *always* loaded when the file in which it is contained is processed by the linker.

A library file is no different from any other relocatable object file produced by the assembler or C compiler, except that it includes a number of modules of the `LIBRARY` type.

USING LIBRARIES WITH C PROGRAMS

All C programs make use of libraries, and the IAR Systems C compilers are supplied with a number of standard library files.

Most C programmers will use the XLIB Librarian at some point, for one of the following reasons:

- To replace or modify a module in one of the standard libraries. For example, the librarian can be used to replace the distribution versions of the `CSTARTUP` and/or `putchar` modules with ones that you have customized.
- To add C or assembler modules to the standard library file so they will always be available whenever a C program is linked.

- To create custom library files that can be linked into their programs, as needed, along with the standard C library.

USING LIBRARIES WITH ASSEMBLER PROGRAMS

If you are only using assembler there is no need to use libraries. However, libraries provide the following advantages, especially when writing medium- and large-sized assembler applications:

- They allow you to combine utility modules used in more than one project into a simple library file. This simplifies the linking process by eliminating the need to include a list of input files for all the modules you need. Only the library module(s) needed for the program will be included in the output file.
- They simplify program maintenance by allowing multiple modules to be placed in a single assembler source file. Each of the modules can be loaded independently as a library module.
- They reduce the number of object files that make up an application, maintenance, and documentation.

You can create your assembly language library files using one of two basic methods:

- A library file can be created by assembling a single assembler source file which contains multiple library-type modules. The resulting library file can then be modified using XLIB.
- A library file can be produced by using the XLIB Librarian to merge any number of existing modules together to form a user-created library.

The `NAME` and `MODULE` assembler directives are used to declare modules as being of `PROGRAM` or `LIBRARY` type, respectively.

XLIB COMMAND SUMMARY

This chapter summarizes the librarian commands, classified according to their function.

A full alphabetical reference list of commands is given in the next chapter.

LIBRARY LISTING COMMANDS

`LIST-ALL-SYMBOLS`	Lists every symbol in modules.
`LIST-CRC`	Lists CRC values of modules.
`LIST-DATE-STAMPS`	Lists dates of modules.
`LIST-ENTRIES`	Lists `PUBLIC` symbols in modules.
`LIST-EXTERNALS`	Lists `EXTERN` symbols in modules.
`LIST-MODULES`	Lists modules.
`LIST-OBJECT-CODE`	Lists low-level relocatable code.
`LIST-SEGMENTS`	Lists segments in modules.

LIBRARY EDITING COMMANDS

`DELETE-MODULES`	Removes modules from a library.
`FETCH-MODULES`	Adds modules to a library.
`INSERT-MODULES`	Moves modules in a library.
`MAKE-LIBRARY`	Changes a module to library type.
`MAKE-PROGRAM`	Changes a module to program type.
`RENAME-ENTRY`	Renames `PUBLIC` symbols.
`RENAME-EXTERNAL`	Renames `EXTERN` symbols.
`RENAME-GLOBAL`	Renames `EXTERN` and `PUBLIC` symbols.
`RENAME-MODULE`	Renames one or more modules.
`RENAME-SEGMENT`	Renames one or more segments.
`REPLACE-MODULES`	Updates executable code.

MISCELLANEOUS LIBRARY COMMANDS

`COMPACT-FILE`	Shrinks library file size.
`DEFINE-CPU`	Specifies CPU type.
`DIRECTORY`	Displays available object files.
`DISPLAY-OPTIONS`	Displays XLIB options.
`ECHO-INPUT`	Command file diagnostic tool.
`EXIT`	Returns to operating system.
`HELP`	Displays help information.
`ON-ERROR-EXIT`	Quits on a batch error.
`QUIT`	Returns to operating system.
`REMARK`	Comment in command file.

XLIB COMMAND REFERENCE

This chapter gives a full syntactic and functional description of all librarian commands.

The individual words of an identifier can be abbreviated to the limit of ambiguity. For example, `LIST-MODULES` can be abbreviated to `L-M`.

When running XLIB you can press ↵ at any time to prompt for information, or display a list of the possible options.

Giving XLIB commands from the command line

The `-c` command line option allows you to run XLIB commands from the command line. Each argument specified after the `-c` option is treated as one XLIB command.

For example, specifying:

```
xlib -c "LIST-MOD math.r39" "LIST-MOD mod.r39 m.txt" ↵
```

is equivalent to entering the following commands in XLIB:

```
*LIST-MOD math.r39
*LIST-MOD mod.r39 m.txt
*QUIT
```

Note that each command line argument must be enclosed in double quotes if it includes spaces.

XLIB BATCH FILES

Running XLIB with a single command-line parameter specifying a file causes XLIB to read commands from that file instead of from the console.

PARAMETERS

The following parameters are common to many of the XLIB commands.

Parameter	*What it means*	
objectfile	File containing object modules.	
start, end	The first and last modules to be processed, in one of the following forms:	
	n	The *n*th module.
	$	The last module.
	name	Module *name*.
	name+*n*	The module *n* modules after *name*.
	$-*n*	The module *n* modules before the last.
listfile	File to which a listing will be sent.	
source	A file from which modules will be read.	
destination	The file to which modules will be sent.	

MODULE EXPRESSIONS

In most of the XLIB commands you can or must specify a source module (like *oldname* in RENAME-MODULE), or a range of modules (*startmodule, endmodule*).

Internally in all XLIB operations modules are numbered upwards from one. Modules may be referred to by the actual name of the module, by the name plus or minus a relative expression, or by an absolute number. The latter is very useful when a module name is very long, unknown, or contains unusual characters (like space or comma).

Below is a list of the available variations on module expressions:

Name	*Description*
`3`	The third module.
`$`	The last module.
`name``+4`	The module 4 modules after *`name`*.
`name``-12`	The module 12 modules before *`name`*.
`$-2`	The module 2 modules before the last module.

The command `LIST-MOD FILE,,$-2` will thus list the three last modules in `FILE` on the terminal.

LIST FORMAT

The `LIST` commands give a list of symbols, where each symbol has one of the following prefixes:

Prefix	*Description*
`nn``.Pgm`	A program module with relative number *`nn`*.
`nn``.Lib`	A library module with relative number *`nn`*.
`Ext`	An external in the current module.
`Ent`	An entry in the current module.
`Loc`	A local in the current module.
`Rel`	A standard segment in the current module.
`Stk`	A stack segment in the current module.
`Com`	A common segment in the current module.

COMPACT-FILE

Shrinks library file size.

SYNTAX

```
COMPACT-FILE objectfile
```

DESCRIPTION

Use `COMPACT-FILE` to concatenate short, absolute records into longer records of variable length. This will decrease the size of a library file by about 5 %, in order to give library files which take up less time during the loader/linker process.

EXAMPLE

The following command compacts the file `maxmin.r39`:

```
COMPACT-FILE maxmin ↵
```

This displays:

```
20 byte(s) deleted
```

DEFINE-CPU

Specifies CPU type.

SYNTAX

```
DEFINE-CPU cpu
```

PARAMETERS

cpu	The target processor.

DESCRIPTION

This command must be issued before any operations on object files can be done.

EXAMPLES

The following command defines the CPU as PICmicro™:

```
DEF-CPU PIC ↵
```

DELETE-MODULES

Removes modules from a library.

SYNTAX

```
DELETE-MODULES objectfile start end
```

DESCRIPTION

Use `DELETE-MODULES` to delete the specified modules.

EXAMPLES

The following command deletes module 2 from the file `math.r39`:

```
DEL-MOD math 2 2 ↵
```

DIRECTORY

Displays available object files.

SYNTAX

```
DIRECTORY [specifier]
```

DESCRIPTION

Use `DIRECTORY` to display on the terminal all files of the type that applies to the target processor. If no *`specifier`* is given, the current directory is listed.

EXAMPLES

The following command lists object files in the current directory:

```
DIR ↵
```

It displays:

```
general      770
math         502
maxmin       375
```

DISPLAY-OPTIONS

Displays XLIB options.

SYNTAX

```
DISPLAY-OPTIONS [listfile]
```

DESCRIPTION

Use `DISPLAY-OPTIONS` to list on the *`listfile`* the names of all the CPUs which are recognized by this version of XLIB. The default file types of object files for the different CPUs are also listed. After that a list of all UBROF tags is output.

EXAMPLES

To list the options to the file `opts.lst`:

```
DISPLAY-OPTIONS opts ↵
```

ECHO-INPUT

Command file diagnostic tool.

SYNTAX

```
ECHO-INPUT
```

DESCRIPTION

`ECHO-INPUT` is useful when debugging command files in batch mode because it makes all command input visible on the terminal. In the interactive mode it has no effect.

EXAMPLES

In a batch file

```
ECHO-INPUT
```

echoes all subsequent XLIB commands.

EXIT

Returns to operating system.

SYNTAX

```
EXIT
```

DESCRIPTION

Use `EXIT` to exit from XLIB after an interactive session.

EXAMPLES

To exit from XLIB:

```
EXIT ↵
```

EXTENSION

Sets the default extension.

SYNTAX

```
EXTENSION
```

DESCRIPTION

Use `EXTENSION` to set the default extension.

FETCH-MODULES

Adds modules to a library.

SYNTAX

```
FETCH-MODULES source destination [start] [end]
```

DESCRIPTION

Use `FETCH-MODULES` to append the specified modules to the *`destination`* file. If *`destination`* already exists, it must be empty or contain valid object modules; otherwise it will be created.

EXAMPLES

The following command copies the module `mean` from `math.r39` to `general.r39`:

```
FETCH-MOD math general mean ↵
```

HELP

Displays help information.

SYNTAX

```
HELP [command] [listfile]
```

PARAMETERS

command	Command for which help is displayed.

DESCRIPTION

If the `HELP` command is given with no parameters, a list of the available commands will be displayed on the terminal. If a parameter is specified, all commands which match the parameter will be displayed with a brief explanation of their syntax and function. A * matches all commands. `HELP` output can be directed to any file.

EXAMPLES

For example, the command:

```
HELP LIST-MOD ↵
```

displays:

```
LIST-MODULES <Object file> [<List file>] [<Start module>]
[<End module>]
    List the module names from [<Start module>] to
    [<End module>].
```

INSERT-MODULES

Moves modules in a library.

SYNTAX

```
INSERT-MODULES objectfile start end {BEFORE | AFTER} dest
```

DESCRIPTION

Use `INSERT-MODULES` to move the specified modules before or after the `dest`.

EXAMPLES

The following command moves the module `mean` before the module `min` in the file `math.r39`:

```
INSERT-MOD math mean mean BEFORE min ↵
```

LIST-ALL-SYMBOLS

Lists every symbol in modules.

SYNTAX

LIST-ALL-SYMBOLS *objectfile* [*listfile*] [*start*] [*end*]

DESCRIPTION

Use `LIST-ALL-SYMBOLS` to list all symbols (module names, segments, externals, entries, and locals) for the specified modules in the *objectfile*. They are listed to the *listfile*.

Each symbol is identified with a prefix; see *List Format*, page 203.

EXAMPLES

The following command lists all the symbols in `math.r39`:

```
LIST-ALL-SYMBOLS math ↵
```

This displays:

```
1.  Lib  max
      Rel   CODE
      Ent   max
      Loc   A
      Loc   B
      Loc   C
      Loc   ncarry
2.  Lib  mean
      Rel   DATA
      Rel   CODE
      Ext   max
      Loc   A
      Loc   B
      Loc   C
      Loc   main
      Loc   start
```

```
3.  Lib  min
      Rel   CODE
      Ent   min
      Loc   carry
```

LIST-CRC

Lists CRC values of modules.

SYNTAX

`LIST-CRC` *`objectfile`* `[`*`listfile`*`] [`*`start`*`] [`*`end`*`]`

DESCRIPTION

Use `LIST-CRC` to list the module names and their associated CRCs for the specified modules.

Each symbol is identified with a prefix; see *List Format*, page 203.

EXAMPLES

The following command lists the CRCs for all modules in `math.r39`:

```
LIST-CRC math ↵
```

This displays:

```
EC41            1.  Lib  max
ED72            2.  Lib  mean
9A73            3.  Lib  min
```

LIST-DATE-STAMPS

Lists dates of modules.

SYNTAX

`LIST-DATE-STAMPS` *`objectfile`* `[`*`listfile`*`] [`*`start`*`] [`*`end`*`]`

DESCRIPTION

Use `LIST-DATE-STAMPS` to list the module names and their associated generation dates for the specified modules.

Each symbol is identified with a prefix; see *List Format*, page 203.

EXAMPLES

The following command lists the date stamps for all the modules in `math.r39`:

```
LIST-DATE-STAMPS math ↵
```

This displays:

```
15/Feb/98       1.  Lib  max
15/Feb/98       2.  Lib  mean
15/Feb/98       3.  Lib  min
```

LIST-ENTRIES

Lists `PUBLIC` symbols in modules.

SYNTAX

```
LIST-ENTRIES objectfile [listfile] [start] [end]
```

DESCRIPTION

Use `LIST-ENTRIES` to list the names and associated entries for the specified modules.

Each symbol is identified with a prefix; see *List Format*, page 203.

EXAMPLES

The following command lists the entries for all the modules in `math.r39`:

```
LIST-ENTRIES math ↵
```

This displays:

```
1.  Lib  max
      Ent   max
2.  Lib  mean
3.  Lib  min
      Ent   min
```

LIST-EXTERNALS

Lists EXTERN symbols in modules.

SYNTAX

```
LIST-EXTERNALS objectfile [listfile] [start] [end]
```

DESCRIPTION

Use LIST-EXTERNALS to list the module names and associated externals for the specified modules.

Each symbol is identified with a prefix; see *List Format*, page 203.

EXAMPLES

The following command lists the externals for all the modules in math.r39:

```
LIST-EXT math ↵
```

This displays:

```
          1.  Lib  max
          2.  Lib  mean
                Ext   max
          3.  Lib  min
```

LIST-MODULES

Lists modules.

SYNTAX

```
LIST-MODULES objectfile [listfile] [start] [end]
```

DESCRIPTION

Use LIST-MODULES to list the module names for the specified modules.

Each symbol is identified with a prefix; see *List Format*, page 203.

EXAMPLES

The following command lists all the modules in math.r39:

```
LIST-MOD math ↵
```

It produces the following output:

```
1.  Lib  max
2.  Lib  min
3.  Lib  mean
```

LIST-OBJECT-CODE

Lists low-level relocatable code.

SYNTAX

`LIST-OBJECT-CODE` *`objectfile`* `[`*`listfile`*`]`

DESCRIPTION

Use `LIST-OBJECT-CODE` to list the contents of the object file on the list file in an ASCII format.

Each symbol is identified with a prefix; see *List Format*, page 203.

EXAMPLES

The following command lists the object code of `math.r39` to `object.lst`:

```
LIST-OBJECT-CODE math object ↵
```

LIST-SEGMENTS

Lists segments in modules.

SYNTAX

`LIST-SEGMENTS` *`objectfile`* `[`*`listfile`*`] [`*`start`*`] [`*`end`*`]`

DESCRIPTION

Use `LIST-SEGMENTS` to list the module names and associated segments for the specified modules.

Each symbol is identified with a prefix; see *List Format*, page 203.

EXAMPLES

The following command lists the segments in the module `mean` in the file `math.r39`:

```
LIST-SEG math,,mean mean ↵
```

Note the use of two commas to skip the *listfile* parameter.

This produces the following output:

```
2.  Lib  mean
       Rel   DATA
       Rel   CODE
```

MAKE-LIBRARY

Changes a module to library type.

SYNTAX

MAKE-LIBRARY *objectfile* [*start*] [*end*]

DESCRIPTION

Use MAKE-LIBRARY to change the module header attributes to conditionally loaded for the specified modules.

EXAMPLES

The following command converts all the modules in main.r39 to library modules:

```
MAKE-LIB main ↵
```

MAKE-PROGRAM

Changes a module to program type.

SYNTAX

MAKE-PROGRAM *objectfile* [*start*] [*end*]

DESCRIPTION

Use MAKE-PROGRAM to change the module header attributes to unconditionally loaded for the specified modules.

EXAMPLES

The following command converts module start in main.r39 into a program module:

```
MAKE-PROG main start ↵
```

ON-ERROR-EXIT

Quits on a batch error.

SYNTAX

```
ON-ERROR-EXIT
```

DESCRIPTION

Use `ON-ERROR-EXIT` to make the librarian abort if an error is found. Most suited for use in batch mode.

EXAMPLES

The following batch file aborts if the `FETCH-MODULES` command fails:

```
ON-ERROR-EXIT
FETCH-MODULES math new
```

QUIT

Returns to the operating system.

SYNTAX

```
QUIT
```

DESCRIPTION

Use `QUIT` to exit and return to the operating system.

EXAMPLES

To quit from XLIB:

```
QUIT ↵
```

REMARK

Comment in command file.

SYNTAX

```
REMARK text
```

DESCRIPTION

Use `REMARK` to include a comment.

EXAMPLES

The following example illustrates the use of a comment in an XLIB command file:

```
REM Now compact file
COMPACT-FILE math
```

RENAME-ENTRY

Renames `PUBLIC` symbols.

SYNTAX

`RENAME-ENTRY` *`objectfile old new`* `[`*`start`*`] [`*`end`*`]`

DESCRIPTION

Use `RENAME-ENTRY` to rename all occurrences of an entry from *`old`* to *`new`* in the specified modules.

EXAMPLES

The following command renames the entry for modules `2` to `4` in `math.r39` from `mean` to `average`:

```
RENAME-ENTRY math mean average 2 4 ↵
```

RENAME-EXTERNAL

Renames `EXTERN` symbols.

SYNTAX

`RENAME-EXTERN` *`objectfile old new`* `[`*`start`*`] [`*`end`*`]`

DESCRIPTION

Use `RENAME-EXTERN` to rename all occurrences of an external symbol from *`old`* to *`new`* in the specified modules.

EXAMPLES

The following command renames all external symbols in `math.r39` from `error` to `err`:

```
RENAME-EXT math error err ↵
```

RENAME-GLOBAL

Renames `EXTERN` and `PUBLIC` symbols.

SYNTAX

```
RENAME-GLOBAL objectfile old new [start] [end]
```

DESCRIPTION

Use `RENAME-GLOBAL` to rename all occurrences of an external symbol or entry from *`old`* to *`new`* in the specified modules.

EXAMPLES

The following command renames all occurrences of `mean` to `average` in `math.r39`:

```
RENAME-GLOBAL math mean average ↵
```

RENAME-MODULE

Renames one or more modules.

SYNTAX

```
RENAME-MODULE objectfile old new
```

DESCRIPTION

Use `RENAME-MODULE` to rename a module. Note that if there is more than one module with the name *`old`*, only the first one encountered is changed.

EXAMPLES

The following example renames the module `average` to `mean` in the file `math.r39:`

```
RENAME-MOD math average mean ↵
```

RENAME-SEGMENT

Renames one or more segments.

SYNTAX

```
RENAME-SEGMENT objectfile old new [start] [end]
```

DESCRIPTION

Use `RENAME-SEGMENT` to rename all occurrences of a segment from the name *`old`* to *`new`* in the specified modules.

EXAMPLES

The following example renames all `CODE` segments to `ROM` in the file `math.r39`:

```
RENAME-SEG math CODE ROM ↵
```

REPLACE-MODULES

Updates executable code.

SYNTAX

```
REPLACE-MODULES source destination
```

DESCRIPTION

Use `REPLACE-MODULES` to replace modules with the same name from *`source`* to *`destination`*. All replacements are logged on the terminal. The main application for this command is to update large run-time libraries etc.

EXAMPLES

The following example replaces modules in `math.r39` with modules from `newmath.r39`:

```
REPLACE-MOD math newmath ↵
```

This displays:

```
   Replacing module 'max'
   Replacing module 'mean'
   Replacing module 'min'
```

ASSEMBLER DIAGNOSTICS

This chapter lists the errors and warnings for the PICmicro™ Assembler. For details of the XLINK Linker and XLIB Librarian error messages see the chapters XLINK diagnostics, and XLIB diagnostics.

INTRODUCTION

Error messages are printed on the terminal, as well as on the optional list file.

All errors are issued as complete, self-explanatory messages. For example:

```
        ADS    B,C
-----------^
"testfile.s39",4  Error[40]: bad instruction
```

The error message consists of the erroneous source line, with a pointer to the faulty spot, followed by the diagnostic and source line number. If include files are used, error messages will be preceded by the source line number and name of *current* file:

```
        ADS    B,C
-----------^
"subfile.h",4  Error[40]: bad instruction
```

The error messages produced by the assembler fall into six categories:

- Command line error messages.
- Assembly warning messages.
- Assembly error messages.
- Assembly fatal error messages.
- Memory overflow messages.
- Assembler internal error messages.

COMMAND LINE ERROR MESSAGES

Command line errors occur when the assembler is invoked with bad parameters. The most common situation is when a file cannot be opened, or with duplicate, mis-spelled, or missing command line switches. The messages are self-explanatory.

ASSEMBLY ERROR MESSAGES

Assembly error messages are produced when the assembler has found a construct which violates the language rules. These are listed in the section *Error messages*, page 221.

ASSEMBLY WARNING MESSAGES

Assembly warning messages are produced when the assembler has found a construct which probably is due to a programming error or omission. These are listed in the section *Warning messages*, page 231.

ASSEMBLY FATAL ERROR MESSAGES

Assembly fatal error messages are produced when the assembler has found a user error so severe that further processing is not considered meaningful. After the diagnostic message has been issued the assembly is immediately terminated. The fatal error messages are identified as ‘Fatal’ in the error messages list.

MEMORY OVERFLOW MESSAGES

The assembler is a memory-based program that in the case of a system with a small primary memory or in the case of very large source files may run out of memory. This is identified by the special message:

```
* * * ASSEMBLER OUT OF MEMORY * * *
Dynamic memory used: nnnnnn bytes
```

If such a situation occurs the solution is either to add system memory or to split source files into smaller modules. However, with 1 Mbyte RAM the assembler capacity should be sufficient for all reasonably sized source files.

ASSEMBLER INTERNAL ERROR MESSAGES

During assembly a number of internal consistency checks are performed and if any of these checks fail the assembler will terminate after giving a short description of the problem. Such errors should normally not occur and should be reported to the IAR Systems technical support group. Please include all possible information about the problem and, preferably, a disk containing a copy of the program that generated the internal error.

ERROR MESSAGES

GENERAL

The following section lists the general error messages:

0 Invalid syntax

The assembler could not decode the expression.

1 Too deep #include nesting (max. is 10)

Fatal. Assembler limit for nesting of `#include` files exceeded. Recursive `#include` could be the reason.

2 Failed to open #include file < name >

Fatal. Could not open a `#include` file. File does not exist in specified directories. Check `-I` prefixes.

3 Invalid #include file name

Fatal. `#include` file name must be written `<file>` or `"file"`.

4 Unexpected end of file encounted

Fatal. End of file encountered within a conditional assembly, the repeat directive or during macro expansion. Probable cause is a missing end of conditional assembly etc.

5 Too long source line (max. is 512 characters) truncated

Source line length exceeds assembler limit.

6 Bad constant

Character that is not a legal digit was encountered.

7 Hexadecimal constant without digits

Prefix `0x` or `0X` of hexadecimal constant found without following hexadecimal digits.

8 Invalid floating point constant

Too large or invalid syntax of floating-point constant.

9 Too many errors encountered (> 100).

10 Space or tab expected

11 Too deep block nesting (max is 50)

Preprocessor directives are nested too deep.

12 String too long (max is 509)

Assembler string length limit exceeded.

13 Missing delimiter in literal or character constant

No closing delimiter `'` or `"` was found in character or literal constant.

14 Missing #endif

A `#if`, `#ifdef`, or `#ifndef` was found but had no matching `#endif`.

15 Invalid character encountered: < char > ; ignored

16 Identifier expected

A name of a label or symbol was expected.

17 ')' expected

18 No such pre-processor command: < command >

# was followed by an unknown identifier.

19 Unexpected token found in pre-processor line

The preprocessor line was not empty after the argument part was read.

20 Argument to #define too long (max is 512)

21 Too many formal parameters for #define (max is 37)

22 Macro parameter < parameter > redefined

A #`define` symbol's formal parameter was repeated.

23 ',' or ')' expected

24 Unmatched #else, #endif or #elif

Fatal. Missing `#if`, `#ifdef`, or `#ifndef`.

25 #error < error > .

Printout via the `#error` directive.

26 '(' expected

27 Too many active macro parameters (max is 256)

Fatal. Preprocessor limit exceeded.

28 Too many nested parameterized macros (max is 50)

Fatal. Preprocessor limit exceeded.

29 Too deep macro nesting (max is 100)

Fatal. Preprocessor limit exceeded.

30 Actual macro parameter too long (max is 512)

A single macro (in `#define`) argument may not exceed the length of a source line.

31 Macro < macro > called with too many parameters

The number of parameters used was more than the number in the macro declaration.

32 Macro <macro> called with too few parameters

The number of parameters used was less than the number in the macro declaration (`#define`).

33 Too many MACRO arguments

The number of assembler macros exceeds 32.

34 May not be redefined

Assembler macros may not be redefined.

35 No name on macro

Assembler macro definition without a label was encountered.

36 Illegal formal parameter in macro

A parameter that was not an identifier was found.

37 ENDM or EXITM not in macro

`ENDM` directive or `EXITM` directive encountered while not inside macro.

38 '>' expected but found end-of-line

A < was found but no matching >.

39 END before start of module

End-of-module directive has no matching `MODULE` directive.

40 Bad instruction

The mnemonic/directive does not exist.

41 Bad label

Labels must begin with `A-Z`, `a-z`, _, or `?`. The succeeding characters must be `A-Z`, `a-z`, `0-9`, _, or `?`. Labels cannot have the same name as a predefined symbol.

42 Duplicate label

The label has already appeared in the label field or been declared as `EXTERN`.

43 Illegal effective address

The addressing mode (operands) is not allowed for this mnemonic.

44 ',' expected

A comma was expected but not found.

45 Name duplicated

The name of `RSEG`, `STACK`, or `COMMON` segments is already used but for something else.

46 Segment type expected

In `RSEG`, `STACK`, or `COMMON` directive : was found but the segment type that should follow was not valid.

47 Segment name expected

The `RSEG`, `STACK`, and `COMMON` directives need a name.

48 Value out of range <range>

The value exceeds its limits.

49 Alignment already set

`RSEG`, `STACK`, and `COMMON` segment do not allow alignment to be set more than once. Use `ALIGN`, `EVEN`, or `ODD` instead.

50 Undefined symbol: <symbol>

The symbol did not appear in label field nor in an `EXTERN` or `sfr` declaration.

51 Can't be both PUBLIC and EXTERN

Symbols can be declared as either `PUBLIC` or `EXTERN`.

52 EXTERN not allowed

Reference to EXTERN symbols is not allowed in this context.

53 Expression must be absolute

The expression cannot involve relocatable or external symbols.

54 Expression can not be forward

The assembler must be able to solve the expression the first time this expression is encountered.

55 Illegal size

The maximum size for expressions is 32 bits.

56 Too many digits

The value exceeds the size of the destination.

57 Unbalanced conditional assembly directives

Missing conditional assembly IF or ENDIF.

58 ELSE without IF

Missing conditional assembly IF.

59 ENDIF without IF

Missing conditional assembly IF.

60 Unbalanced structured assembly directives

Missing structured assembly IF or ENDIF.

61 ’+’ or ’-’ expected

Plus or minus sign missing.

62 Illegal operation on extern or public symbol

An illegal operation has been used on a public or external symbol; eg SET.

63 Illegal operation on non-constant label

It is not allowed to make a non-constant symbol PUBLIC or EXTERN.

64 Extern or unsolved expression

The expression must be solved at assembly time, ie not include external references.

65 ' = ' expected

Equals sign was missing.

66 Segment too long (max is < max >)

The length of ASEG, RSEG, STACK, or COMMON segments is larger than the addressable length.

67 Public did not appear in label field

A symbol was declared PUBLIC but no label with the same name was found in the source file.

68 End of block-repeat without start

The repeat directive REPT was not found although the ENDR directive was.

69 Segment must be relocatable

The operation is not allowed on ASEG.

70 Limit exceeded: < error text > , value is: < value > (decimal)

The value exceeded the limits set with the LIMIT directive. The error text is set by the user in the LIMIT directive.

71 Symbol <symbol> has already been declared EXTERN

An attempt to redeclare an EXTERN as EXTERN was made.

72 Symbol <symbol> has already been declared PUBLIC

An attempt to redeclare a PUBLIC as PUBLIC was made.

73 End-of-module missing

A PROGRAM or MODULE directive was encountered before ENDMOD was found.

74 Expression must yield non-negative result

The expression was evaluated to a negative number, whereas a positive number was required.

75 Repeat directive unbalanced

This error is caused by a REPT directive without a matching ENDR, or a an ENDR directive without a matching REPT.

76 End of repeat directive is missing

A REPT directive without a closing ENDR was encountered.

77 LOCALs not allowed in this context, (<symbol>)

Local symbols must be declared within macro definitions.

78 End of macro expected

An assembler macro is being defined but there was no end-of-macro.

79 End of repeat expected

One of the repeat directives is active, but there was no end-of-repeat found.

80 End of conditional assembly expected

Conditional assembly is active but there was no end of if.

81 End of structured assembly expected

One of the directives for structured assembly is active but has no matching `END`.

82 Misplaced end of structured assembly

A directive that terminates one of the structured assembly directives was found but no matching `START` directive is active.

83 Error in SFR attribute definition

The `SFRTYPE` directive was used with unknown attributes.

84 Illegal symbol type in symbol

The symbol cannot be used in this context since it has the wrong type.

85 Wrong number of arguments

Expected a different number of arguments.

86 Number expected

Something other than digits encountered.

87 Label must be public or extern

The label must be declared with `PUBLIC` or `EXTERN`.

88 Label not defined with DEFFN

The label has to be defined via `DEFFN` before used in this context.

89 Sorry DEMO version, bytecount exceeded (max bytes)

90 Different parts of ASEG have overlapping code

91 Internal error

92 Empty macro stack overflow

93 Macro stack overflow

94 Attempt to access out-of-stack value

95 Invalid macro operator

96 No such macro argument

97 Sorry Lite version, bytecount exceeded (max bytes)

98 Option -re cannot handle code in include files, use -r or -rn instead

99 #include within macro not supported

PICMICRO™-SPECIFIC ERROR MESSAGES

In addition to the general errors, the PICmicro™ assembler can generate the following errors:

400 Too many operands

You have supplied more operands than the instruction/directive accept.

401 Destination must be absolute

The destination selector that determines the destination of an instruction result cannot be specifies with an external symbol or an expression that is not fully determined at assembly time.

402 SIZEOF: an offset does not make sense here

Operator `SIZEOF` can only be applied to segment names.

WARNING MESSAGES

GENERAL

The following section lists the general warning messages:

0 Unreferenced label

The label was not used as an operand nor was it declared public.

1 Nested comment

A C comment was nested.

2 Unknown escape sequence

A backslash (\) found in a character constant or string literal was followed by an unknown escape character.

3 Non-printable character

A non-printable character was found in a literal or character constant.

4 Macro or define expected

5 Floating point value out-of-range

Floating point value is too large to be represented by the floating point system of the target.

6 Floating point division by zero

7 Wrong usage of string operator ('#' or '##'); ignored.

The current implementation restricts use of the # and ## operators to the token field of parameterized macros. In addition, the # operator must precede a formal parameter.

8 Macro parameter(s) not used

9 Macro redefined

10 Unknown macro

11 Empty macro argument

12 Recursive macro

13 Redefinition of Special Function Register

The `SFR` has already been defined.

14 Division by zero

Division by 0 in constant expression.

15 Constant truncated

The constant was longer than the size of the destination.

16 Suspicious sfr expression

A Special Function Register `SFR` is used in an expression, and the assembler cannot check access rights.

17 Empty module <module>, module skipped

An empty module was created by using `END` directly after `ENDMOD` or `MODULE`, followed by `ENDMOD` with no statements in between.

18 End of program while in include file

The program ended while a file was being included.

19 Symbol *symbol* duplicated

20 Bit symbol cannot be used as operand

A symbol was declared using the bit directive, but since the bit address is not calculated the symbol should not be used.

21 Label did not appear in label field

22 Set segment alignment the same < value > or larger

When the alignment set by `ALIGN` is larger than the segment alignment it may be lost at link time.

PICMICRO™-SPECIFIC WARNING MESSAGES

In addition to the general warnings the PICmicro™ assembler can generate the following warnings:

400 Number out of range

The expression you have supplied does not fit into the instruction/directive range and is truncated.

401 TRIS/OPTION not recommended for 16CXX devices

`TRIS` and `OPTION` are provided only for backward compatibility. Avoid them to be compatible with future 16CXX devices.

402 - 414

These warnings relate to the limited support for some MPASM-specific directives. See *Compatibility with MPASM directives*, page 139.

XLINK DIAGNOSTICS

This chapter describes the errors and warnings produced by the XLINK Linker.

INTRODUCTION

The error messages produced by the XLINK Linker fall into five categories:

- Linker warning messages.
- Linker error messages.
- Linker fatal error messages.
- Memory overflow message.
- Linker internal error messages.

XLINK WARNING MESSAGES

XLINK warning messages will appear when the linker detects something that may be wrong. The code generated may still be correct.

XLINK ERROR MESSAGES

XLINK error messages are produced when the linker detects something wrong. The linking process will be aborted unless the **Always generate output** (-B) option is specified. The code produced is almost certainly faulty.

XLINK FATAL ERRORS

XLINK fatal error messages abort the linking process. They occur when continued linking is useless, ie the fault is irrecoverable.

MEMORY OVERFLOW MESSAGE

XLINK is a memory-based linker. If run on a system with a small main memory or if very large source files are being used, XLINK may run out of memory. This is recognized by the following message:

```
* * * LINKER OUT OF MEMORY * * *
```

```
Dynamic memory used: nnnnnn bytes
```

If this occurs, the solution is either to add system memory, or to enable file bound processing with the -m option. The `-t` option can also be used to save memory.

XLINK INTERNAL ERRORS

During linking, a number of internal consistency checks are performed. If any of these checks fail, the linker will terminate after giving a short description of the problem. These errors will not normally occur, but if they do please report them to the IAR Systems technical support group. Please include all possible information about the problem and also a disk with the files that generated the error.

ERROR MESSAGES

If you get a message that indicates a corrupt object file, reassemble or recompile the faulty file since an interrupted assembly or compilation may produce an invalid object file.

The following table lists the XLINK error messages:

0 Format chosen cannot support banking

Format unable to support banking.

1 Corrupt file. Unexpected end of file in module *`module`* (*`file`*) encountered

Linker aborts immediately. Recompile or reassemble, or check the compatibility between the linker and C compiler.

2 Too many errors encountered (>100)

Linker aborts immediately.

3 Corrupt file. Checksum failed in module *`module`* (*`file`*). Linker checksum is *`linkcheck`*, module checksum is *`modcheck`*

Linker aborts immediately. Recompile or reassemble.

4 Corrupt file. Zero length identifier encountered in module *`module`* (*`file`*)

Linker aborts immediately. Recompile or reassemble.

5 Address type for CPU incorrect. Error encountered in module *`module`* **(*`file`*)**

Linker aborts immediately. Check that you are using the right files and libraries.

6 Program module *`module`* **redeclared in file** *`file`***. Ignoring second module**

XLINK will not produce code unless the **Always generate output** (`-B`) option (forced dump) is used.

7 Corrupt file. Unexpected UBROF – format end of file encountered in module *`module`* **(*`file`*)**

Linker aborts immediately. Recompile or reassemble.

8 Corrupt file. Unknown or misplaced tag encountered in module *`module`* **(*`file`*). Tag** *`tag`*

Linker aborts immediately. Recompile or reassemble.

9 Corrupt file. Module *`module`* **start unexpected in file** *`file`*

Linker aborts immediately. Recompile or reassemble.

10 Corrupt file. Segment no. *`segno`* **declared twice in module** *`module`* **(*`file`*)**

Linker aborts immediately. Recompile or reassemble.

11 Corrupt file. External no. *`ext no`* **declared twice in module** *`module`* **(*`file`*)**

Linker aborts immediately. Recompile or reassemble.

12 Unable to open file *`file`*

Linker aborts immediately. If you are using the command line, check the environment variable `XLINK_DFLTDIR`.

13 Corrupt file. Error tag encountered in module *module* (*file*)

A UBROF error tag was encountered. Linker aborts immediately. Recompile or reassemble.

14 Corrupt file. Local *local* defined twice in module *module* (*file*)

Linker aborts immediately. Recompile or reassemble.

15

This error message has been deleted.

16 Segment *segment* is too long for segment definition

The segment defined does not fit into the memory area reserved for it. Linker aborts immediately.

17 Segment *segment* is defined twice in segment definition -Zsegdef

Linker aborts immediately.

18 Range error in module *module* (*file*), segment *segment* at address *address*. Value *value*, in tag *tag*, is out of bounds

The address is out of the CPU address range. Locate the cause of the problem using the information given in the error message.

The check can be suppressed by the `-R` option.

19 Corrupt file. Undefined segment referenced in module *module* (*file*)

Linker aborts immediately. Recompile or reassemble.

20 Undefined external referenced in module *module* (*file*)

Linker aborts immediately. Recompile or reassemble.

21 Segment *segment* in module *module* does not fit bank

The segment is too long. Linker aborts immediately.

22 Paragraph no. is not applicable for the wanted CPU. Tag encountered in module *module* **(***file***)**

Linker aborts immediately. Delete the paragraph no. declaration in the .xcl file.

23 Corrupt file. T_REL_FI_8 or T_EXT_FI_8 is corrupt in module *module* **(***file***)**

The tag T_REL_FI_8 or T_EXT_FI_8 is faulty. Linker aborts immediately. Recompile or reassemble.

24 Segment *segment* **overlaps segment** *segment*

The segments overlap each other; ie both include the same address.

25 Corrupt file. Unable to find module *module* **(***file***)**

A module is missing. Linker aborts immediately.

26 Segment *segment* **is too long**

This error should never occur unless the program is extremely large. Linker aborts immediately.

27 Entry entry in module *module* **(***file***) redefined in module** *module* **(***file***)**

There are two or more entries with the same name. Linker aborts immediately.

28 File *file* **is too long**

The program is too large. Split the file. Linker aborts immediately.

29 No object file specified in command-line

There is nothing to link. Linker aborts immediately.

30 Option *-option* **also requires the -***option* option

Linker aborts immediately.

31 Option *`-option`* **cannot be combined with the** *`-option`* `option`

Linker aborts immediately.

32 Option *`-option`* **cannot be combined with the** *`-option`* **option and the** *`-option`* **option**

Linker aborts immediately.

33 Faulty value *`val`***, (range is 10-150)**

Faulty page setting. Linker aborts immediately.

34 Filename too long

The filename is more than 255 characters long. Linker aborts immediately.

35 Unknown flag *`flag`* **in cross reference option** *`option`*

Linker aborts immediately.

36 Option *`op`* **does not exist**

Linker aborts immediately.

37 - not succeeded by character

The `-` marks the beginning of an option, and must be followed by a character. Linker aborts immediately.

38 Option *`option`* **must not be defined more than once**

Linker aborts immediately.

39 Illegal character specified in option *`op`*

Linker aborts immediately.

40 Argument expected after option *`op`*

This option must be succeeded by an argument. Linker aborts immediately.

41 Unexpected '-' in option *`op`*

Linker aborts immediately.

42 Faulty symbol definition -D*`symbol definition`*

Incorrect syntax. Linker aborts immediately.

43 Symbol in symbol definition too long

The symbol name is more than 255 characters. Linker aborts immediately.

44 Faulty value *`val`***, (range 80-300)**

Faulty column setting. Linker aborts immediately.

45 Unknown CPU *`CPU`* **encountered in** *`context`*

Linker aborts immediately. Check the argument to `-c` is valid. If you are using the command line you can get a list of CPUs by typing `xlink -c?` ↵.

46 Undefined external *`external`* **referred in** *`module`* **(***`file`***)**

Entry to external is missing.

47 Unknown format *`format`* **encountered in** *`context`*

Linker aborts immediately.

48

This error message has been deleted.

49

This error message has been deleted.

50 Paragraph no. not allowed for this CPU, encountered in option *`option`*

Linker aborts immediately. Do not use paragraph no. in declarations.

51 *`Input base`* **value expected in option** *`option`*

Linker aborts immediately.

52 **Overflow on value in option** *`option`*

Linker aborts immediately.

53 **Parameter exceeded 255 characters in extended command line file** *`file`*

Linker aborts immediately.

54 **Extended command line file** *`file`* **is empty**

Linker aborts immediately.

55 **Extended command line variable XLINK_ENVPAR is empty**

Linker aborts immediately.

56 **Non-increasing range in segment definition** *`segment def`*

Linker aborts immediately.

57 **No CPU defined**

No CPU defined, either in the command line or in `XLINK_CPU`. Linker aborts immediately.

58 **No format defined**

No format defined, either in the command line or in `XLINK_FORMAT`. Linker aborts immediately.

59 **Revision no. for file is imcompatible with XLINK revision no.**

Linker aborts immediately.

If this error occurs after recompilation or reassembly, the wrong version of XLINK is being used. Check with your supplier.

60 Segment ***segment*** **defined in bank definition and segment definition.**

Linker aborts immediately.

61

This error message has been deleted.

62 Input file ***file*** **cannot be loaded more than once**

Linker aborts immediately.

63 Trying to pop an empty stack in module module (file)

Linker aborts immediately. Recompile or reassemble.

64 Module ***module*** **(*file*) has not the same debug type as the other modules**

Linker aborts immediately.

65 Faulty replacement definition -e ***replacement*** **definition**

Incorrect syntax. Linker aborts immediately.

66 Function with F-index ***index*** **has not been defined before indirect reference in module** ***module*** **(*file*)**

Indirect call to an undefined in module. Probably caused by an omitted function declaration.

67 Function ***name*** **has same F-index as** ***function-name*****, defined in module** ***module*** **(*file*)**

Probably a corrupt file. Recompile file.

68 External function *name* in module *module* (*file*) has no global definition

If no other errors have been encountered, this error is generated by an assembly language call from C where the required declaration using the `$DEFFN` assembly language support directive is missing. The declaration is necessary to inform the linker of the memory requirements of the function.

69 Indirect or recursive function *name* in module *module* (file) has parameters or auto variables in nondefault memory

The recursively or indirectly called function name is using extended language memory specifiers (bit, data, idata, etc) to point to non-default memory, memory which is not allowed.

Function parameters to indirectly called functions must be in the default memory area for the memory model in use, and for recursive functions, both local variables and parameters must be in default memory.

70

This error message has been deleted.

71 Segment *name* is incorrectly defined (in a bank definition, has wrong segment type or mixed with other segment types)

This is usually due to misuse of a predefined segment; see the explanation of *name* in the *PICmicro™ C Compiler Programming Guide*. It may be caused by changing the predefined linker control file.

72 Segment *name* must be defined in a segment option definition (-Z, -b, or -P)

This is either an omission of a segment in the linker (usually a segment needed by the C system control) file or a spelling error (segment names are case sensitive).

73 Label ?ARG_MOVE not found (recursive function needs it)

In the library there should be a module containing this label. If it has been removed it must be restored.

74 There was an error when writing to file *`file`*

Either the linker or your host system is corrupt, or the two are incompatible.

75 SFR address in module *`module`* **(***`file`***), segment** *`segment`* **at address** *`address`***, value** *`value`* **is out of bounds**

An SFR has been defined to a bad address. Change the definition.

76 Absolute segments overlap in module *`module`*

The linker has found two or more absolute segments in *`module`* overlapping each other.

77 Absolute segments in module *`module`* **(***`file`***) overlaps absolute segment in module** *`module`* **(***`file`***)**

The linker has found two or more absolute segments in *`module`* (*`file`*) and *`module`* (*`file`*) overlapping each other.

78 Absolute segment in module *`module`* **(***`file`***) overlaps segment** *`segment`*

The linker has found an absolute segment in *`module`* (*`file`*) overlapping a relocatable segment.

79 Faulty allocation definition -a*`definition`*

The linker has discovered an error in an overlay control definition.

80 Symbol in allocation definition (-a) too long

A symbol in the `-a` command is too long.

81 Unknown flag in extended format option *`option`*

Check flags.

82 Conflict in segment *name*. Mixing overlayable and not overlayable segment parts.

These errors only occur with the 8051 and converted PL/M code.

83 The overlayable segment *name* may not be banked.

These errors only occur with the 8051 and converted PL/M code.

84 The overlayable segment *name* must be of relative type.

These errors only occur with the 8051 and converted PL/M code.

85 The far/farc segment *name* in module *mod* (*file*) is larger than *size*

The segment *name* is too large to be a `far` segment.

86

This error message has been deleted.

87 Function with F-index *i* has not been defined before tiny_func referenced in module *mod* (*file*)

Check that all tiny functions are defined before they are used in a module.

88 Wrong library used (compiler version or memory model mismatch). Problem found in *mod* (*file*). Correct library tag is *tag*

Code from this compiler needs a matching library. A library belonging to a later or earlier version of the compiler may have been used.

92 Cannot use this format with this cpu

Some formats need CPU-specific information and are only supported for some CPUs.

93 Non-existant warning number *no*, (valid numbers are 0-*max*)

An attempt to suppress a warning that does not exist gives this error.

94 Unknown flag *x* in local symbols option -n*x*

The character *x* is not a valid flag in the local symbols option.

95 Module *mod* (*file*) uses source file references, which are not available in UBROF 5 output

This feature cannot be filtered out by the linker when producing UBROF 5 output.

96 Unmatched -! comment in extended command file

An odd number of `-!` (comment) options were seen in a `.xcl` file.

97 Unmatched -! comment in extended command line variable XLINK_ENVPAR

As above, but for the environment variable `XLINK_ENVPAR`.

98 Unmatched /* comment in extended command file

No matching `*/` was found in the `.xcl` file.

99 Syntax error in segment definition: *option*

There was a syntax error in the option.

100 Segment name too long: "*seg*" in *option*

The segment name exceeded the maximum length (255 characters).

101 Segment already defined: "*seg*" in *option*

The segment has already been mentioned in a segment definition option.

102 No such segment type: *option*

The segment type given is not a valid one.

103 Ranges must be closed in *`option`*

The `-P` option requires all memory ranges to have an end.

104 Failed to fit all segments into specified ranges. Problem discovered in segment *`seg`***.**

The packing algorithm used in the linker didn't manage to fit all the segments.

105 Recursion not allowed for this system. Check module map for recursive functions

The run-time model used does not support recursion. Each function determined by the linker to be recursive is marked as such in the module map part of the linker list file.

106 Syntax error or bad argument in *`option`*

There was an error when parsing the command line argument given.

107 Banked segments do not fit into the number of banks specified

The linker did not manage to fit all of the contents of the banked segments into the banks given.

108 Cannot find function *`function`* **mentioned in -a#**

All the functions specified in an indirect call option must exist in the linked program.

109 Function *`function`* **mentioned as callee in -a# is not indirectly called**

Only functions that actually can be called indirectly can be specified to do so in an indirect call option.

110 Function *function* mentioned as caller in -a# does not make indirect calls

Only functions that actually make indirect calls can be specified to do so in an indirect call option.

111 The file *file* is not a UBROF file

The contents of the file are not in a format that XLINK can read.

112 The module *module* is for an unknown cpu (tid = *tid*). Either the file is corrupt or you need a later version of XLINK

The version of XLINK used has no knowledge of the cpu that the file was compiled/assembled for.

113 Corrupt input file: "*symptom*" in module *module* (*file*)

The input file indicated appears to be corrupt. This can occur either because the file has for some reason been corrupted after it was created, or because of a problem in the compiler/assembler used to create it. If the latter appears to be the case, please contact IAR.

WARNING MESSAGES

The following section lists the linker warning messages:

0 Too many warnings

Too many warnings encountered.

1 Error tag encountered in module *module* (*file*)

A UBROF error tag was encountered when loading file *file*. This indicates a corrupt file and will generate an error in the linking phase.

2 Symbol *symbol* is redefined in command-line

A symbol has been redefined.

3 Type conflict. Segment *`segment`*, in module *`module`*, is incompatible with earlier segment(s) of the same name

Segments of the same name should have the same type.

4 Close/open conflict. Segment *`segment`*, in module *`module`*, is incompatible with earlier segment of the same name

Segments of the same name should be either open or closed.

5 Segment *`segment`* cannot be combined with previous segment

The segments will not be combined.

6 Type conflict for external/entry *`entry`*, in module *`module`*, against external/entry in module *`module`*

Entries and their corresponding externals should have the same type.

7 Module *`module`* declared twice, once as program and once as library. Redeclared in file *`file`*, ignoring library module

The program module is linked.

8

This warning message has been deleted.

9 Ignoring redeclared program entry in module *`module`* (*`file`*), using entry from module *`module1`*

Only the program entry found first is chosen.

10 No modules to link

The linker has no modules to link.

11 Module *`module`* declared twice as library. Redeclared in file *`file`*, ignoring second module

The module found first is linked.

12 Using SFB in banked segment *segment* **in module** *module* **(***file***)**

The SFB assembler directive may not work in a banked segment.

13 Using SFE in banked segment *segment* **in module** *module* **(***file***)**

The SFE assembler directive may not work in a banked segment.

14 Entry *entry* **duplicated. Module** *module* **(***file***) loaded, module** *module* **(***file***) discarded**

Duplicated entries exist in conditionally loaded modules; ie library modules or conditionally loaded program modules (with the `-C` option).

15 Predefined type sizing mismatch between modules *module* **(***file***) and** *module* **(***file***)**

The modules have been compiled with different options for predefined types, such as different sizes of basic C types (eg `integer`, `double`).

16 Function *name* **in module** *module* **(***file***) is called from two function trees (with roots** *name1* **and** *name2***)**

The probable cause is *module* interrupt function calls another function that also could be executed by a foreground program, and this could lead to execution errors.

17 Segment name is too large or placed at wrong address

This error occurs if a given segment overruns the available address space in the named memory area. To find out the extent of the overrun do a dummy link, moving the start address of the named segment to the lowest address, and look at the linker map file. Then relink with the correct address specification.

18 Segment *segment* **overlaps segment** *segment*

The linker has found two relocatable segments overlapping each other. Check the segment placement option parameters.

19 Absolute segments overlaps in module `module` (`file`)

The linker has found two or more absolute segments in module *`module`* overlapping each other.

20 Absolute segment in module `module` (`file`) overlaps absolute segment in module `module` (`file`)

The linker has found two or more absolute segments in module *`module`* (*`file`*) and module *`module`* (*`file`*) overlapping each other. Change the `ORG` directives.

21 Absolute segment in module `module` (`file`) overlaps segment `segment`

The linker has found an absolute segment in module *`module`* (*`file`*) overlapping a relocatable segment. Change either the `ORG` directive or the `-Z` relocation command.

22 Interrupt function `name` in module `module` (`file`) is called from other functions

Interrupt functions may not be called.

23 *`limitation specific warning`*

Due to some limitation in the chosen output format, or in the information available, XLINK cannot produce the correct output. Only one warning for each specific limitation is given.

24 *`num`* **counts of** *`warning`* **total**

For each warning of type 23 emitted, a summary is provided at the end.

25 Using -Y# discards and distorts debug information. Use with care. If possible find an updated debugger that can read modern UBROF

Using the UBROF format modifer `-Y#` is not recommended.

26 No reset vector found

Failed in determining the LOCATION setting for XCOFF output format for the 78400 processor, because no reset vector was found.

27 No code at the start address

Failed in determining the LOCATION setting for XCOFF output format for the 78400 processor, because no code was found at the address specified in the reset vector.

28 Parts of segment `name` are initialized, parts not

This is not useful if the result linking is to be promable.

29 Parts of segment `name` are initialized, even though it is of type `type` (and thus not promable)

Initing DATA memory is not if the result of linking is to be promable.

30 Module `name` is compiled with tools for `cpu1` expected `cpu2`

You are building an executable for CPU `cpu2`, but module name is compiled for CPU `cpu1`.

31 Modules have been compiled with possibly incompatible settings: `more info`

According to the contents of the modules, they are not compatible.

32 Format option set more than once. Using format `format`

The format option can only be given once. The linker uses the format `format`.

33 Using -r overrides format option. Using UBROF

The `-r` option specifies UBROF format and C-SPY library modules. It overrides any `-F` (format) option.

34 The 20 bit segmented variant of the INTEL EXTENDED format cannot represent the addresses specified. Consider using -Y1 (32 bit linear addressing).

The program uses addresses higher than 0xFFFFF, and the segmented variant of the chosen format cannot handle this. The linear addressing variant can handle full 32 bit addresses.

35 There is more than one definition for the struct/union type with tag *`tag`*

Two or more different structure/union types with the same tag exist in the program. If this is not intentional, it is likely that the declarations differ slightly. It is very likely that there will also be one or more warnings about type conflicts (warning 6). If this is intentional, consider turning this warning off.

36 There are indirectly called functions doing indirect calls. This can make the static overlay system unreliable

XLINK does not know what functions can call what functions in this case, which means that it cannot make sure static overlays are safe.

37 More than one interrupt function makes indirect calls. This can make the static overlay system unreliable. Using -ai will avoid this

If a function is called from an interrupt while it is already running its params and locals will be overwritten.

38 There are indirect calls both from interrupts and from the main program. This can make the static overlay system unreliable. Using -ai will avoid this

If a function is called from an interrupt while it is already running its params and locals will be overwritten.

39 The function *`function`* in module *`module`* (*`file`*) does not appear to be called. No static overlay area will be allocated for its params and locals

As far as XLINK can tell, there are no callers for the function, so no space is needed for its params and locals. To make XLINK allocate space anyway use `-a(function)`.

40 The module *`module`* contains obsolete type information that will not be checked by the linker

This kind of type information was replaced in 1988.

41 The function *`function`* in module *`module`* (*`file`*) makes indirect calls but is not mentioned in the left part of any -a# declaration

If any `-a#` indirect call options are given they must, taken together, specify the complete picture.

42

This warning message does not exist.

43 The function *`function`* in module *`module`* (*`file`*) is indirectly called but is not mentioned in the right part of any -a# declaration

If any `-a#` indirect call options are given they must, taken together, specify the complete picture.

44 C library routine localtime failed. Timestamps will be wrong

XLINK is unable to determine the correct time. This primarily affects the dates in the list file. This problem has been observed on one host platform if the date is after the year 2038.

XLIB DIAGNOSTICS

This chapter lists the messages produced by the XLIB Librarian.

XLIB MESSAGES

The following section lists the XLIB messages. Commands flagged as erroneous never alter object files.

1 **Bad object file, EOF encountered**

Bad or empty object file, which could be the result of an aborted assembly or compilation.

2 **Unexpected EOF in batch file**

The last command in a command file must be `EXIT`.

3 **Unable to open file** *`file`*

Could not open the command file or, if `ON-ERROR-EXIT` has been specified, this message is issued on any failure to open a file.

4 **Variable length record out of bounds**

Bad object module, could be the result of an aborted assembly.

5 **Missing or non-default parameter**

A parameter was missing in the direct mode.

6 **No such CPU**

A list with the possible choices is displayed when this error is found.

7 **CPU undefined**

`DEFINE-CPU` must be issued before object file operations can begin. A list with the possible choices is displayed when this error is found.

8 **Ambiguous CPU type**

A list with the possible choices is displayed when this error is found.

9 No such command

Use the HELP command.

10 Ambiguous command

Use the HELP command.

11 Invalid parameter(s)

Too many parameters or a misspelled parameter.

12 Module out of sequence

Bad object module, could be the result of an aborted assembly.

13 Incompatible object, consult distributor!

Bad object module, could be the result of an aborted assembly, or that the assembler/compiler revision used is incompatible with the version of XLIB used.

14 Unknown tag: hh

Bad object module, could be the result of an aborted assembly.

15 Too many errors

More than 32 errors will make XLIB abort.

16 Assembly/compilation error?

The T_ERROR tag was found. Edit and re-assemble/re-compile your program.

17 Bad CRC, hhhh expected

Bad object module; could be the result of an aborted assembly.

18 Can't find module: xxxxx

Check the available modules with LIST-MOD file.

19 Module expression out of range

Module expression is less than one or greater than `$`.

20 Bad syntax in module expression: xxxxx

The syntax is invalid.

21 Illegal insert sequence

The specified destination in the `INSERT-MODULES` command must not be within the *`start-end`* sequence.

22 < End module > found before < Start module > !

Source module range must be from low to high order.

23 Before or after!

Bad `BEFORE/AFTER` specifier in the `INSERT-MODULES` command.

24 Corrupt file, error occurred in *`tag`*

A fault is detected in the object file *`tag`*. Reassembly or recompilation may help. Otherwise contact your supplier.

25 *`File`* **is write protected**

The file *`file`* is write protected and cannot be written to.

26 Non-matching replacement module name found in source file

In the source file, a module *`name`* with no corresponding entry in the destination file was found.