



# AT90S ASSEMBLER, LINKER, AND LIBRARIAN

# Programming Guide



## **COPYRIGHT NOTICE**

© Copyright 1995–1996 IAR Systems. All rights reserved.

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

## **DISCLAIMER**

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

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

## **TRADEMARKS**

C-SPY is a trademark of IAR Systems. Windows and MS-DOS are trademarks of Microsoft Corp.

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

Second edition: September 1996

Part no: AA90–1

This documentation was produced by Human-Computer Interface.





## PREFACE

### WELCOME

Welcome to the AT90S Assembler, Linker, and Librarian Programming Guide.

This guide provides reference information about the IAR Systems Assembler, XLINK Linker, and XLIB Librarian for the AT90S family of microprocessors, and applies to both the Embedded Workbench and command line versions of these tools.

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



If you are using the Embedded Workbench refer to the *AT90S Windows Workbench Interface Guide* for information about running the IAR Systems tools from the Workbench interface, a simple tutorial, and complete reference information about the Workbench commands and dialog boxes, and the Workbench editor.



If you are using the command line version refer to the *AT90S Command Line Interface Guide* for general information about running the IAR Systems tools from the command line, and a simple tutorial to illustrate how to use them.

For information about programming with the AT90S C Compiler refer to the *AT90S C Compiler Programming Guide*.

If your product includes the optional AT90S C-SPY debugger refer to the *AT90S C-SPY User Guide* for information about debugging with C-SPY.

### ABOUT THIS GUIDE

This guide consists of the following parts and chapters:

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

#### **AT90S Assembler**

The *Introduction* provides a brief overview of the AT90S Assembler.

The *Tutorial* explains how to use the most important features of the assembler to develop simple AT90S machine-code programs. It also describes a typical development cycle using XLINK and XLIB.





## PREFACE

---

*Assembler options summary* explains how to set the AT90S Assembler options, and gives an alphabetical summary of the options.

*Assembler options reference* then gives reference information about each option.

*Assembler file formats* describes the source format for the AT90S 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 AT90S Assembler operators, with a full description of each one.

*Assembler directives summary* gives an alphabetical summary of the AT90S Assembler directives.

*Assembler directives reference* gives complete reference information about the AT90S Assembler directives, classified into groups according to their function.

*Assembler instructions* lists the AT90S 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.





## PREFACE

### Diagnostics

*Assembler diagnostics* provides a list of error messages specific to the AT90S 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 AT90S processor.
- ◆ The AT90S Assembler language.
- ◆ Windows or MS-DOS, depending on your host system.

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

### CONVENTIONS

This guide uses the following typographical conventions:

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



---

## PREFACE

---





# CONTENTS

<b>INSTALLATION AND DOCUMENTATION ROUTE MAP . . . .</b>	<b>1</b>
Command line versions	1
Windows Workbench versions	2
UNIX versions	3
Documentation route map	4
<b>INTRODUCTION . . . . .</b>	<b>5</b>
Assembler	5
XLINK Linker	6
XLIB Librarian	7
<b>TUTORIAL . . . . .</b>	<b>9</b>
Getting started	9
Creating a program	12
Using macros	20
Using modules	25
<b>ASSEMBLER OPTIONS SUMMARY . . . . .</b>	<b>31</b>
Setting assembler options	32
Options summary	33
<b>ASSEMBLER OPTIONS REFERENCE . . . . .</b>	<b>35</b>
Code generation	35
#define	39
List	40
#undef	44
Include	45
Target	46
Command line	48
<b>ASSEMBLER FILE FORMATS . . . . .</b>	<b>51</b>
Source format	
Expressions and operators	51
Register symbols	56
Listing format	57
Output formats	59

## CONTENTS

<b>ASSEMBLER OPERATOR SUMMARY .....</b>	<b>61</b>
<b>ASSEMBLER OPERATOR REFERENCE .....</b>	<b>65</b>
<b>ASSEMBLER DIRECTIVES SUMMARY .....</b>	<b>79</b>
Directives summary	79
<b>ASSEMBLER DIRECTIVES REFERENCE .....</b>	<b>85</b>
Syntax conventions	86
Module control directives	88
Symbol control directives	90
Segment control directives	91
Value assignment directives	96
Conditional assembly directives	101
Macro processing directives	103
Listing control directives	110
C-style preprocessor directives	115
Data definition or allocation directives	119
Assembler control directives	120
<b>ASSEMBLER INSTRUCTIONS .....</b>	<b>125</b>
CPU instruction mnemonics	125
<b>XLINK LINKER .....</b>	<b>129</b>
Introduction	129
Input files and modules	131
Listing format	134
<b>XLINK OPTIONS SUMMARY .....</b>	<b>139</b>
Setting XLINK options	139
Summary of options	140
<b>XLINK OPTIONS REFERENCE .....</b>	<b>143</b>
Output	143
#define	145
Error	146
List	148
Include	150
Input	151





## CONTENTS

Command line	153
Segment control	156
<b>XLINK OUTPUT FORMATS</b>	<b>163</b>
<b>XLIB LIBRARIAN</b>	<b>167</b>
Introduction	167
<b>XLIB COMMAND SUMMARY</b>	<b>169</b>
<b>XLIB COMMAND REFERENCE</b>	<b>171</b>
<b>ASSEMBLER DIAGNOSTICS</b>	<b>189</b>
Introduction	189
Error messages	191
Warning messages	199
<b>XLINK DIAGNOSTICS</b>	<b>203</b>
Introduction	203
Error messages	204
Warning messages	213
<b>XLIB DIAGNOSTICS</b>	<b>217</b>
XLIB messages	217
<b>INDEX</b>	<b>221</b>



**CONTENTS**

---





# INSTALLATION AND DOCUMENTATION ROUTE MAP

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

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


## COMMAND LINE VERSIONS

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

### WHAT YOU NEED

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

### INSTALLATION

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

When the installation is complete:

- 4 Make the following changes to your `autoexec.bat` file:  
Add the paths to the IAR Systems executable and user interface files to the `PATH` variable; for example:  
`PATH=c:\dos;c:\utils;c:\iar\exe;c:\iar\ui;`

## INSTALLATION AND DOCUMENTATION ROUTE MAP

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

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

```

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

### RUNNING THE TOOLS

Type the appropriate command at the MS-DOS prompt.

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

## WINDOWS WORKBENCH VERSIONS

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

### WHAT YOU NEED

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

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

### INSTALLING FROM WINDOWS 95 OR NT 4.0

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

### RUNNING FROM WINDOWS 95 OR NT 4.0

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



## INSTALLATION AND DOCUMENTATION ROUTE MAP

### INSTALLING FROM WINDOWS 3.1x OR NT 3.51

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

### RUNNING FROM WINDOWS 3.1x OR NT 3.51

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

### RUNNING C-SPY

**Either:**

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

**Or:**

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

## UNIX VERSIONS

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

### WHAT YOU NEED

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

### INSTALLATION

Follow the instructions provided with the media.

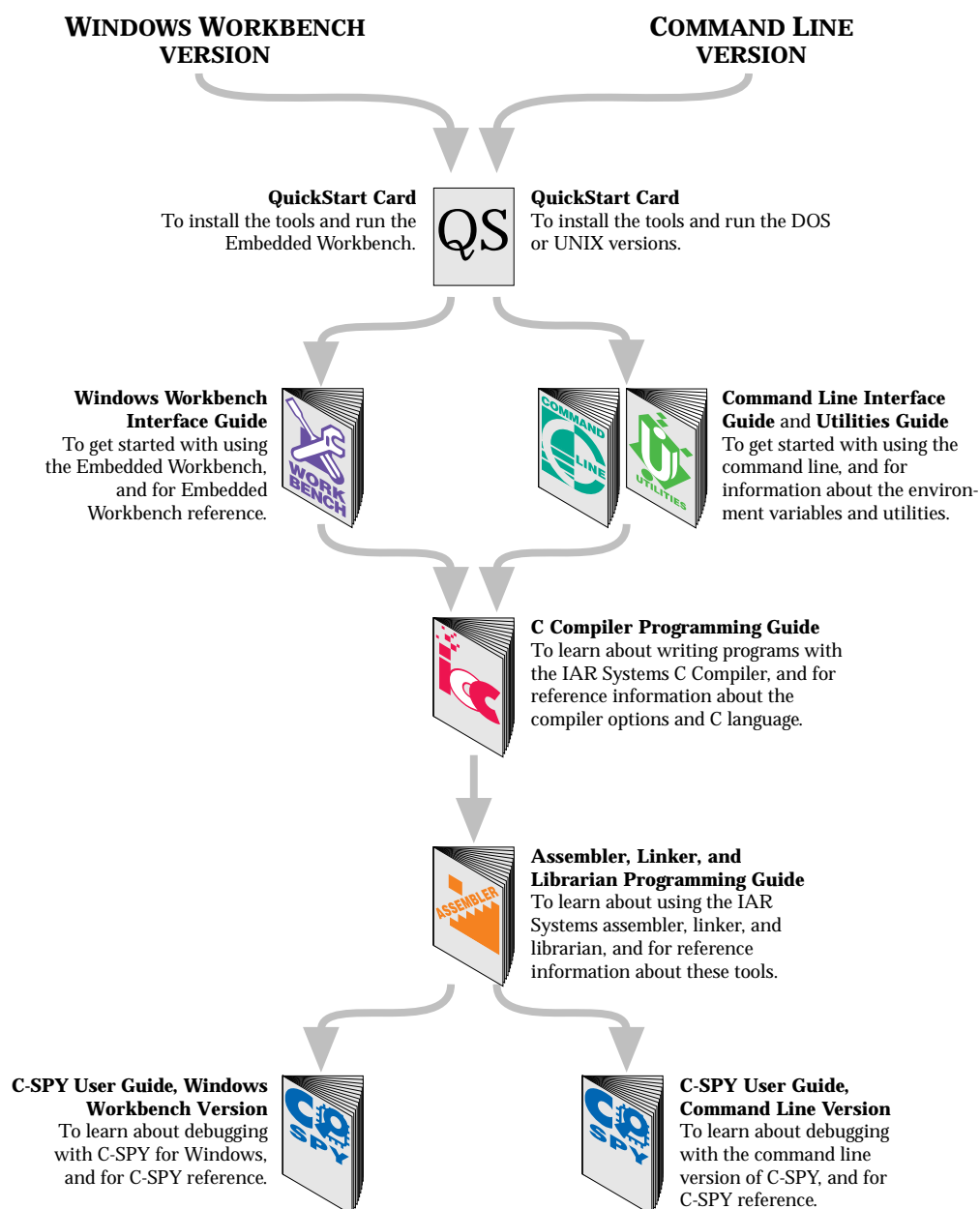
### RUNNING THE TOOLS

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



## INSTALLATION AND DOCUMENTATION ROUTE MAP

### DOCUMENTATION ROUTE MAP





# INTRODUCTION

The IAR Systems AT90S 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 AT90S Assembler is a powerful relocating macro assembler with a versatile set of directives.

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

It provides the following features:

### GENERAL

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

### ASSEMBLER FEATURES

- ◆ Support for AT90S-family microprocessors.
- ◆ 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.



## INTRODUCTION

- ◆ Support for complex expressions with external references.
- ◆ Forward references allowed to any depth.
- ◆ Support for C language pre-processor directives and `sfr` keyword.
- ◆ 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

- ◆ Unlimited number of input files.
- ◆ Searches user-defined library files and loads only those modules needed by the application.
- ◆ Symbols may be up to 255 characters long with all characters being significant. Both upper and lower case may be used.
- ◆ Global symbols can be defined at link time.
- ◆ Flexible segment commands allow full control of the locations of relocatable code and data in memory.
- ◆ Support for over 30 emulator formats.





### **XLIB LIBRARIAN**

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



## INTRODUCTION

---





# TUTORIAL

This tutorial illustrates how you might use the AT90S Assembler to develop a series of simple machine-code programs for the AT90S processor, 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 route map*.
- ◆ Be familiar with the architecture and instruction set of the AT90S processor. 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 *AT90S Windows Workbench Interface Guide* or *AT90S Command Line Interface Guide*, as appropriate, to familiarize yourself with the interface you are using.

## 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 without debugging information to give a file about .a90, which can be downloaded to an emulator with debugging facilities. Use the XLINK -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



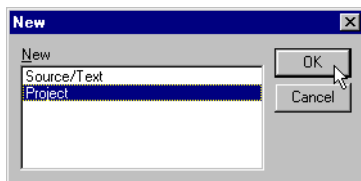
#### Creating a new project using the Embedded Workbench

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



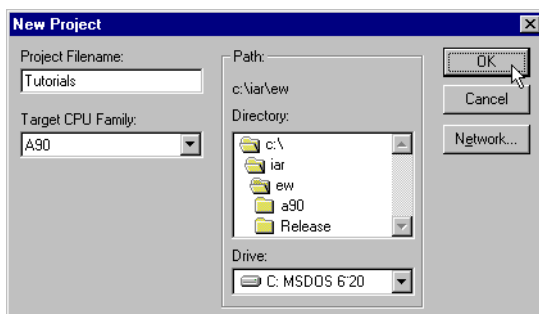
## TUTORIAL

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



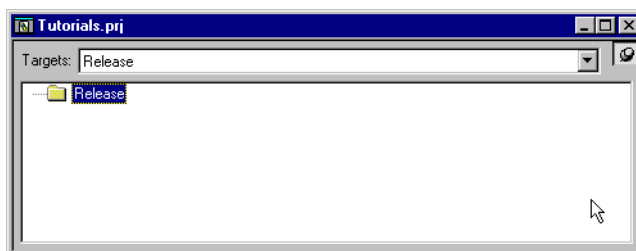
Select **Project** and choose **OK** to display the **New Project** dialog box.

Enter **Tutorials** in the **Project Filename** box, and set the **Target CPU Family** to **A90**:



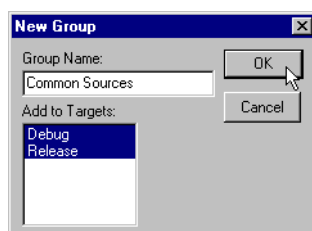
Then choose **OK** to create the new project.

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



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

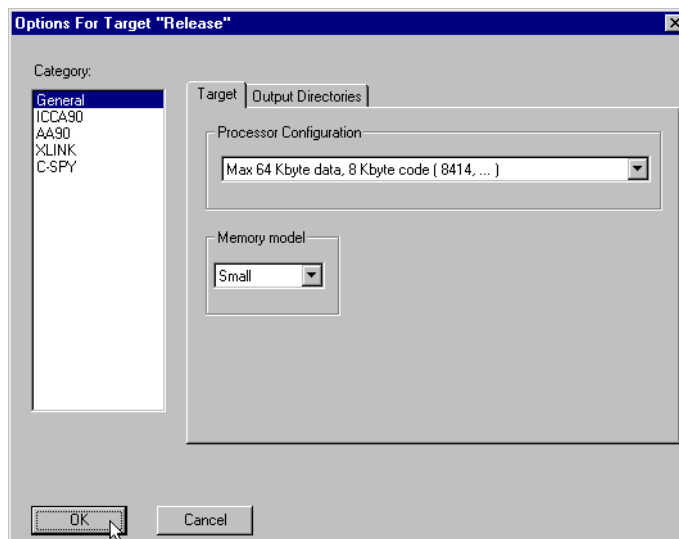
Choose **New Group...** from the **Project** menu and enter the name Common Sources. By default both targets are selected, so the group will be added to both targets:



Choose **OK** to create the group. It will be displayed in the Project window. Now set up the target options to suit the processor and memory model we have chosen.

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

Set the **Processor Configuration** to **Max 64 Kbyte data, 8 Kbyte code** and select the **Small** memory model.



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

## TUTORIAL



### 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 aa90 directory. Select this directory by entering the command:

```
cd c:\iar\aa90
```

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 registers R16 and R17 in binary-coded decimal:

```
NAME      first
ORG       0
RJMP     main

main      ORG      1Ch
          CLR      R17
          CLR      R16
loop      INC      R17
          CPI      R17,10
          BRNE     loop
          CLR      R17
          INC      R16
          CPI      R16,10
          BRNE     loop
done_it   JMP      done_it

          END
```

The ORG directive assembles the program starting at address 0, the AT90S reset address, so the program is executed 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.



## TUTORIAL

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

Enter the program given above and save it in a file `first.s90`. The files associated with the AT90S Assembler have extensions `.s90`, `.a90`, `.d90`, and `.r90` to identify them.

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



### Writing the program using the command line

Enter the program using any standard text editor, such as the MS-DOS `edit` editor, and save it in a file called `first.s90`. The files associated with the AT90S Assembler have extensions `.s90`, `.a90`, `.d90`, and `.r90` to identify them. Alternatively, a copy is provided in the assembler files directory.

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

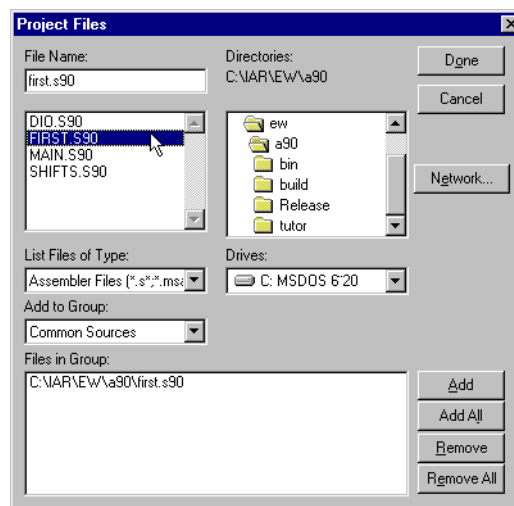
## ASSEMBLING THE PROGRAM



### Assembling the program using the Embedded Workbench


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

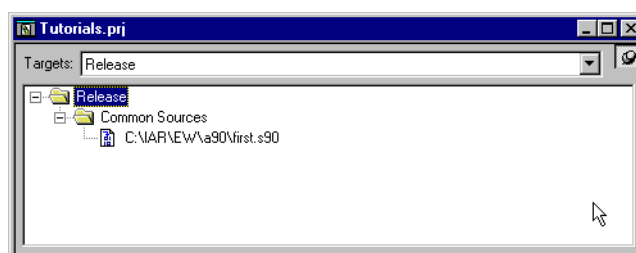
Choose **Files...** from the **Project** menu to display the **Project Files** dialog box. Locate the file `first.s90` in the file selection list in the upper half of the dialog box, and choose **Add** to add it to the **Common Sources** group:



## TUTORIAL

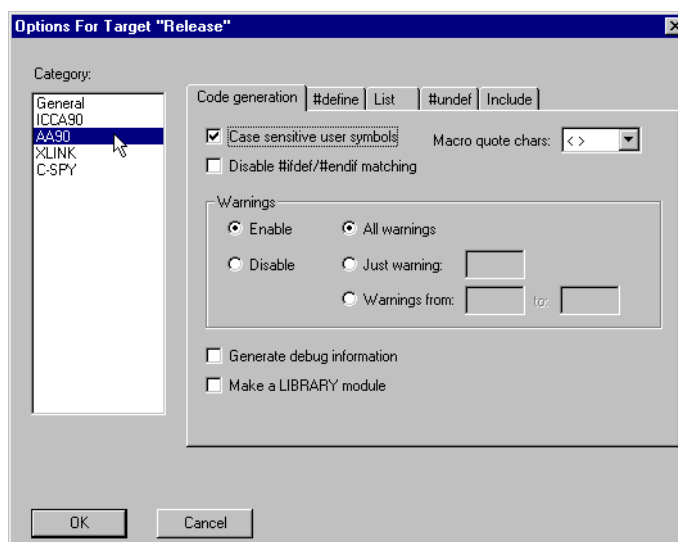
Then click **Done** to close the **Project Files** dialog box.

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



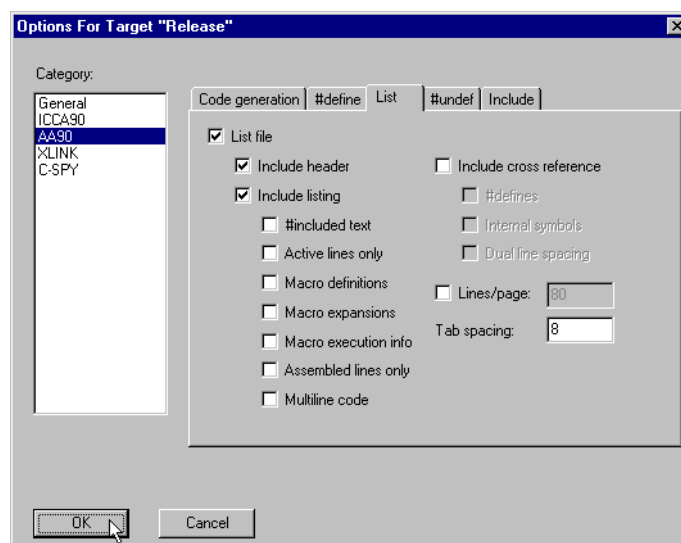
Then set up the assembler options for the project as follows.

Select the **Release** folder in the Project window. Then choose **Options...** from the **Project** menu and select **AA90** in the **Category** list to display the assembler options pages:



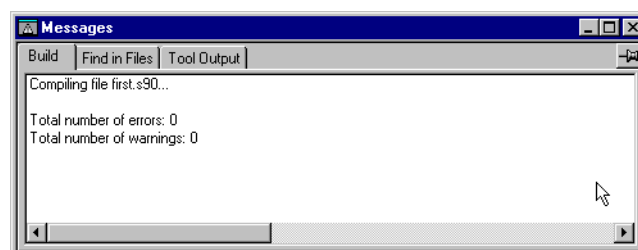


Click **List**, to display the page of list options, and select **List file** to produce an assembler list file. This will enable you to examine the code generated by the assembler:



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:



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





## TUTORIAL

```
#####  
#          CRC:7920          #  
#      Errors:   0          #  
#      Warnings: 0          #  
#      Bytes: 24          #  
#####
```

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:

6	0000001C	1127	main	CLR	R17
7	0000001E	0027		CLR	R16
8	00000020	1395	loop	INC	R17
9	00000022	1A30		CPI	R17,10

	Address field	Source line
Source line number	Data field	

Assuming that the source assembled successfully, a further file, `first.r90`, 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.

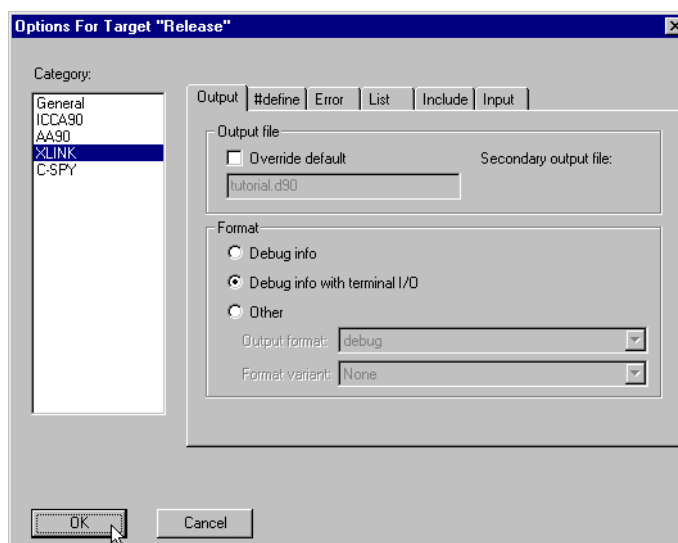
Select the **Release** folder in the Project window. Then choose **Options...** from the **Project** menu, and select **XLINK** in the **Category** list to display the linker option pages.



## TUTORIAL

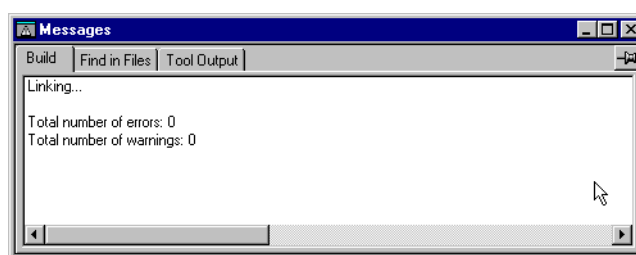
Then click **Output** to display the output options.

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



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 the program using the command line

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

```
xlink first -ca90 -r ↵
```



## TUTORIAL

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 `about.d90`.

### 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 three warning messages will be displayed:

Only assembler level debugging available.

Exit label missing.

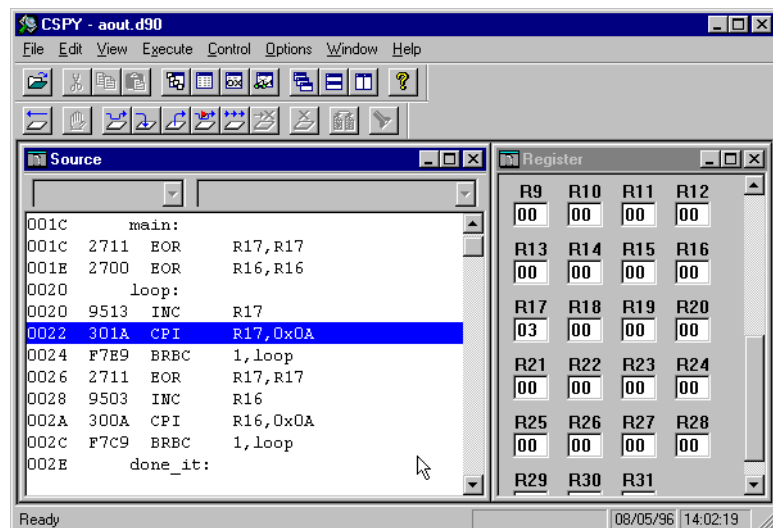
No break on program exit.

You can ignore these warnings, so click **OK** to proceed.

The C-SPY window will then be displayed.

In C-SPY open the Register window, by choosing **Register** from the **Window** menu.

Then choose **Step** from the **Execute** menu, or press **F2**, to step through the program and watch the R17 and R16 registers count in binary-coded decimal.



## TUTORIAL



### Running the program using the command line

To run the example program using the C-SPY debugger type the following command:

```
csa90 aout ↵
```

After the program has loaded enter the command WINDOW REG ON to display the value of the registers.

Then type STEP, or press **F2**, to step through the program and watch the registers R16 and R17 count in binary coded decimal.

## USING MACROS

The second example will demonstrate the use of simple macros. It defines `outdat` which outputs an 8-bit value to port B, and `strobe` which strobes bit 7 of port A. The code to do this is quite short and may need to be executed fast, making a macro an ideal solution.

For a complete explanation of the assembler's macro features see *Macro processing directives*, page 103.

```
strobe  MACRO
        IN      R25,portA
        SBR     R25,128
        OUT     portA,R25
        CBR     R25,128
        OUT     portA,R25
        ENDM

outdat  MACRO    val
        LDI     R25,val
        OUT     portB,R25
        ENDM
```

The `strobe` macro reads port A, sets bit 7 and outputs the result. It then clears bit 7 and outputs the result again.

The `outdat` macro loads the supplied constant into R25, which is then written to port B.



## TUTORIAL

The full listing of the dio assembler program is as follows:

```
NAME      dio

; define the ports
      ASEG      DATA
portA    VAR      0x1B
portB    VAR      0x18

;define the macros
strobe   MACRO
      IN        R25,portA
      SBR       R25,128
      OUT       portA,R25
      CBR       R25,128
      OUT       portA,R25
      ENDM

outdat   MACRO val
      LDI       R25,val
      OUT       portB,R25
      ENDM

;Vector table
      ASEG      CODE
      ORG       0x00
      RJMP      main      ; Reset vector

;main code
      ORG       0x1C      ; Start of main code
main    outdat   23
        strobe
        outdat   40
        strobe
done    JMP      done
        END
```

Type in this listing and save it in a file `dio.s90`. Alternatively, a copy of the source is provided on the installation disk.



## TUTORIAL

### ASSEMBLING THE PROGRAM



#### Assembling the program using the Embedded Workbench

Close the **Tutorial** project, and create a new project, **Tutor2**, by choosing **New** from the **File** menu, and add the file `dio.s90` 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:

```
aa90 dio -r -L -v1
```



#### Viewing the listing

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

```
1  00000000                                NAME    dio
2  00000000
3  00000000                                ; define the ports
4  00000000                                ASEG     DATA
5  0000001B    portA  VAR      0x1B
6  00000018    portB  VAR      0x18
7  00000000
8  00000000                                ;define the macros
16 00000000
21 00000000
22 00000000                                ;Vector table
23 00000000                                ASEG     CODE
24 00000000                                ORG      0x00
25 00000000 0DC0                                RJMP    main    ; Reset vector
26 00000002
27 00000002                                ;main code
28 0000001C                                ORG      0x1C    ; Start of main
                                code
29 0000001C    main  outdat  23
29 0000001C    main  outdat  23
29.1 0000001C 97E1                                LDI      R25,23
29.2 0000001E 98BB                                OUT      portB,R25
29.3 00000020                                ENDM
30 00000020                                strobe
30.1 00000020 98B3                                IN       R25,portA
30.2 00000022 9068                                SBR      R25,128
```





## TUTORIAL

```
30.3 00000024 9BBB      OUT    portA,R25
30.4 00000026 9F77      CBR     R25,128
30.5 00000028 9BBB      OUT     portA,R25
30.6 0000002A           ENDM
31   0000002A           outdat  40
31.1 0000002A 98E2      LDI     R25,40
31.2 0000002C 98BB      OUT     portB,R25
31.3 0000002E           ENDM
32   0000002E           strobe
32.1 0000002E 9BB3      IN      R25,portA
32.2 00000030 9068      SBR     R25,128
32.3 00000032 9BBB      OUT     portA,R25
32.4 00000034 9F77      CBR     R25,128
32.5 00000036 9BBB      OUT     portA,R25
32.6 00000038           ENDM
33   00000038 0C941C00   done    JMP     done
34   0000003C           END
```

The macro-generated lines are numbered with a decimal suffix: eg 30.1, 30.2, etc.

### 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 dio -ca90 -r -l dio.map ↵
```

This generates a file aout.d90.

### RUNNING THE PROGRAM



#### Running the program using the Embedded Workbench

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

The **C-SPY** window will be displayed.

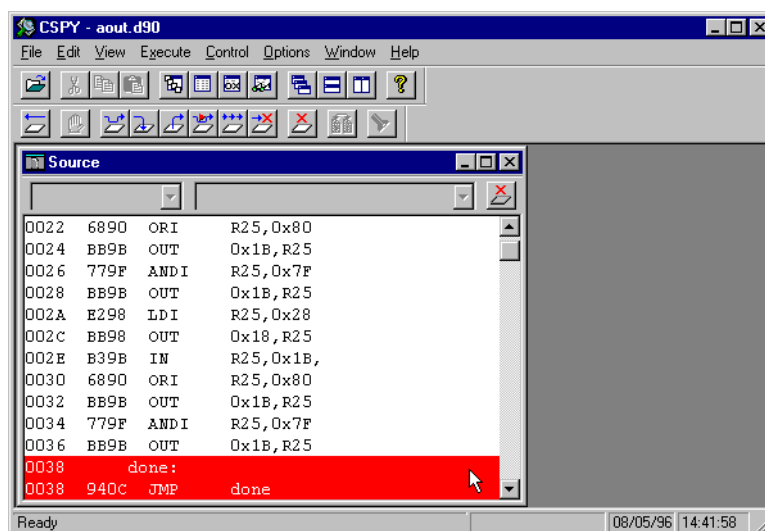


## TUTORIAL

Choose **Step** from the **Execute** menu to display the source program in the Source window.

Then set a breakpoint at the instruction `JMP done` at the end of the main program in the Source window by selecting it and choosing **Toggle Breakpoint** from the **Control** menu.

The `JMP done` instruction will be highlighted to show that there is a breakpoint set there:



Open the Memory window, by choosing **Memory** from the **Window** menu, and display the contents of the locations `portA` and `portB` at `0x1B` and `0x18` respectively.

Then execute from `main` up to the breakpoint by choosing **Go** from the **Execute** menu.

You will see the effect of the program in the Memory window.



### Running the program using the command line




If you have the C-SPY simulator you can run the program with the command:

```
csa90 aout -v1
```



## TUTORIAL

Set C-SPY up with the following commands:

```
MEMORY SFR 0   
WINDOW REG ON   
REG PC=0 
```

These commands open Memory and Register windows, and then set the PC to address 0 which is the reset vector.

Now single step through the code using the **F2** key, and notice how locations 0x1B and 0x18 corresponding to portA and portB change.

### GENERATING A FILE FOR A PROM PROGRAMMER

To generate code which can be read by a PROM programmer, link without the -r option to get a file aout.a90.

## USING MODULES

The final 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 the need to assemble a routine each time you need it 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:

```
NAME      main

PUBLIC    main
EXTERN    r_shift

main      RSEG      MY_CODE
          LDI        R25,H'A
          MOV        R4,R25
          LDI        R25,5
          MOV        R5,R25
          CALL       r_shift
done_it   RJMP       done_it

          END        main
```

This simply uses a routine called `r_shift` to shift the contents of register R4 to the right. The data in register R4 is set to \$A and the `r_shift` routine is called to shift it to the right by four places as specified by the contents of register R5.

The `EXTERN` directive declares `r_shift` as an external symbol, to be resolved at link time.

Enter this program and save it as the file `main.s90` or, alternatively, copy the file provided in the assembler files directory (by default `c:\iar\aa90`).

## CREATING THE LIBRARY ROUTINES

The second program is used to form a separately assembled library. This contains two library routines: the `r_shift` routine called by `main`, and the corresponding `l_shift` routine. These both operate on the contents of register R4 by repeatedly shifting it to the right or left. The number of shifts performed is controlled by decrementing register R5 to zero.



## TUTORIAL

```
MODULE r_shift
public r_shift
RSEG MY_CODE

r_shift TST R5
        BREQ r_shift2
        LSR R4
        DEC R5
        BRNE r_shift
r_shift2 RET
        ENDMOD

MODULE l_shift
PUBLIC l_shift

RSEG MY_CODE
l_shift TST R5
        BREQ l_shift2
        LSL R4
        DEC R5
        BRNE l_shift
l_shift2 RET

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 `r_shift` and `l_shift` entry addresses are made public to other modules with a `PUBLIC` directive.

Save these modules in a source file called `shifts.s90` or, alternatively, copy the file provided in the assembler files directory (by default `c:\iar\aa90`).

### 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.

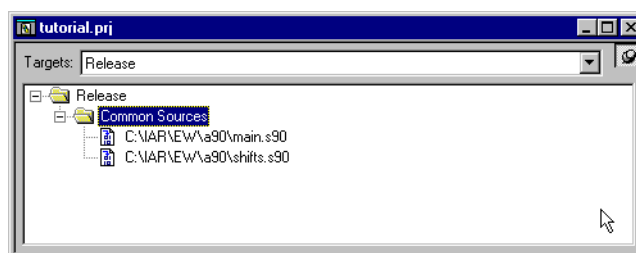


## TUTORIAL



### Assembling and linking using the Embedded Workbench

Create a project containing `main.s90` and `shifts.s90`, 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:

```
aa90 main -r -L ↵
```

Similarly, to assemble the library routines type:

```
aa90 shifts -r -L ↵
```

Assembling the files creates two relocatable files. You need to link these together 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 `l_shift` subroutine. The `r_shift` 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 -ca90 main shifts -ZMY_CODE=0E -xsm -l main.map ↵
```

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

Parameter	Description
-ZMY_CODE=0E	Defines that the code segment is to be relocated to the hex address 0xE.
-xsm	Requests a cross reference listing.
-l main.map	Directs the listing output to <code>main.map</code> .

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 `r_shift` library module, but not the unused `l_shift` library module.

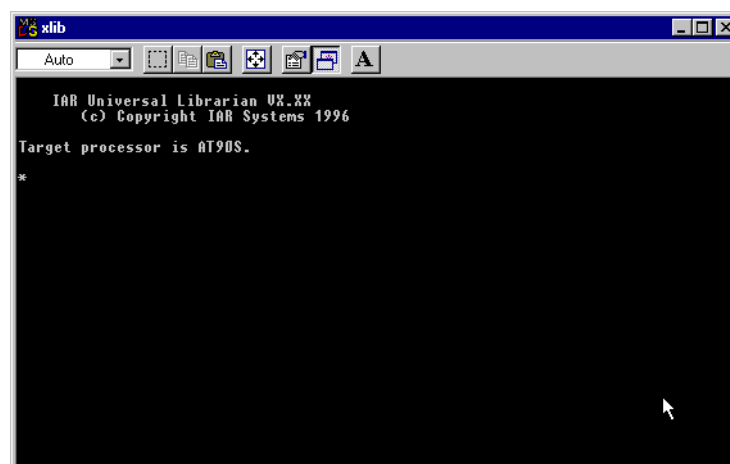
## USING THE XLIB LIBRARIAN

Once you have assembled and debugged a module intended for general use, like the `l_shift` and `r_shift` 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.



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.

## TUTORIAL

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

```
DEFINE-CPU a90 ↵
```



### Giving XLIB commands

Extract the modules you want from `shifts.r90` into a library called `math.r90`. To do this enter the command:

```
FETCH-MODULES ↵
```

This prompts for the following arguments:

<i>Prompt</i>	<i>What you type</i>
Source file	<code>shifts</code> ↵
Destination file	<code>math</code> ↵
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.r90` which contains the code for the `l_shift` and `r_shift` routines.

You can confirm this by typing:

```
LIST-MODULES ↵
```

This prompts for the following arguments:

<i>Prompt</i>	<i>What you type</i>
Object file	<code>math</code>
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 ↵
```

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 **AA90** and **General** options in the Embedded Workbench:

Code generation	#undef
#define	Include
List	Target

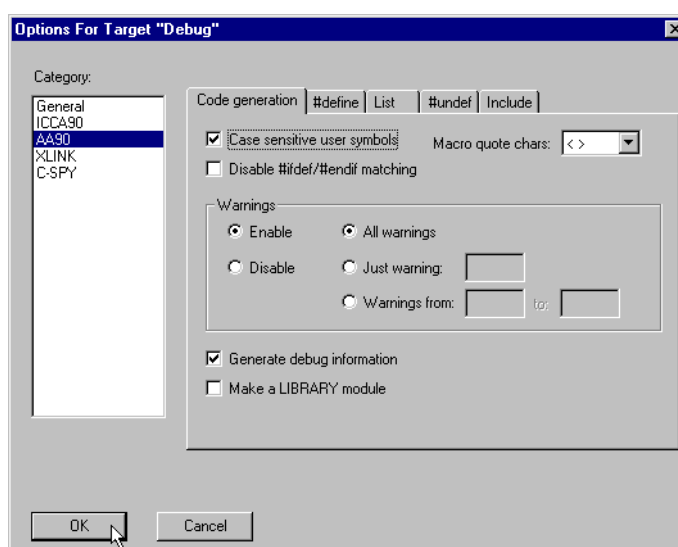
For full reference about each option refer to the following chapter, *Assembler options reference*. The *Command line* section, page 48, provides information about the options which are only available in the command line version.

## 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 **AA90** in the **Category** list to display the assembler options pages:



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



### Setting assembler options from the command line

To set assembler options from the command line, you include them on the command line, after the `aa90` command. For example, when assembling the source `first`, to generate a listing to the default listing filename (`first.lst`):

```
aa90 first -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`:

```
aa90 first -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`:

```
aa90 first -Llist ↵
```



## ASSEMBLER OPTIONS SUMMARY

### 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*.

<i>Option</i>	<i>Description</i>	<i>Section</i>
-B	Macro execution info.	List
-b	Make a LIBRARY module.	Code generation
-c{DMEA0}	Conditional list.	List
-Dsymb[=xx]	Define symbol.	#define
-d	Disable #ifdef/#endif matching.	Code generation
-Enumber	Max number of errors.	Command line
-f filename	Extend the command line.	Command line
-G	Open standard input as source.	Command line
-Iprefix	Include paths.	Include
-i	Included text.	List
-l filename	List to named file.	List
-L[prefix]	List to prefixed source name.	List
-Mab	Macro quote chars.	Code generation
-mn	Memory model.	Target
-N	No header.	List
-Oprefix	Set object filename prefix.	Command line
-o filename	Set object filename.	Command line
-plines	Lines/page.	List
-r	Generate debug information.	Code generation
-S	Set silent operation.	Command line
-s{+ -}	Case sensitive user symbols.	Code generation
-T	Active lines only.	List
-tn	Tab spacing.	List
-Usymb	Undefine symbol.	#undef



---

## ASSEMBLER OPTIONS SUMMARY

---

<i>Option</i>	<i>Description</i>	<i>Section</i>
-vn	Processor configuration.	Target
-w[ <i>string</i> ]	Warnings.	Code generation
-x{DI2}	Cross reference.	List



# ASSEMBLER OPTIONS REFERENCE

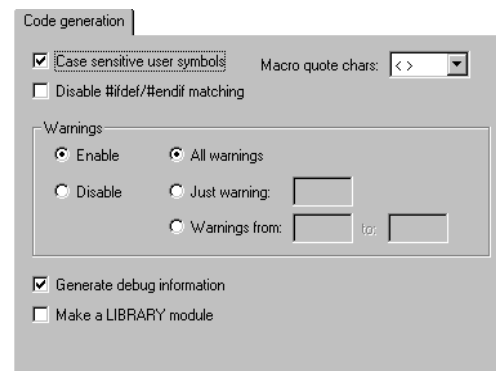
This chapter gives detailed information on each of the AT90S Assembler options, divided into functional categories.

## CODE GENERATION

These options control the assembler's code generation.



### Embedded Workbench



### Command line

- s {+|-} Case sensitive user symbols.
- d Disable #ifdef/#endif matching.
- Mab Macro quote chars.
- w[*string*] Warnings.
- r 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:

<i>Option</i>	<i>Command line</i>
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
```

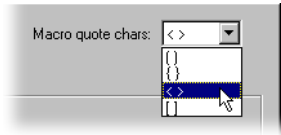
**MACRO QUOTE CHARS (-M)****Syntax:** -Mab

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:



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*]

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:

<i>Range</i>	<i>Effect</i>
+	Enables all warnings.
-	Disables all warnings.
+ <i>n</i>	Enables just warning <i>n</i> .
- <i>n</i>	Disables just warning <i>n</i> .
+ <i>m</i> - <i>n</i>	Enables warnings <i>m</i> to <i>n</i> .
- <i>m</i> - <i>n</i>	Disables warnings <i>m</i> to <i>n</i> .



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

```
aa90 prog -w-0 ↵
```

or to disable warnings 0 to 8:

```
aa90 prog -w-0-8 ↵
```

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

### GENERATE DEBUG INFORMATION (-r)

**Syntax:** -r

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.



### MAKE A LIBRARY MODULE (-b)

**Syntax:** -b

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

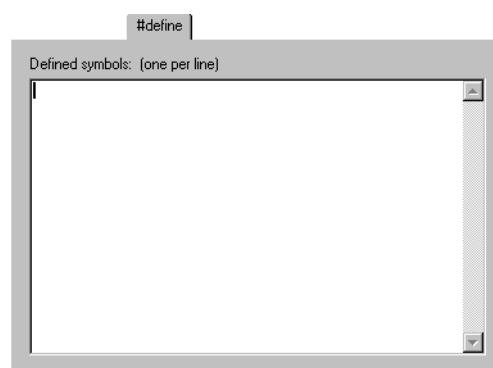
By default, the assembler produces a program module ready to be linked with XLIB. You use the **Make a LIBRARY module** (-b) option if you want it to make a library module for use with XLIB.

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, that is the assembler produces a program module regardless.



**#define**

This option allows you to define symbols.

**Embedded Workbench****Command line**

`-D sym[=xx]` Define symbol.

**DEFINE SYMBOL (-D)**

**Syntax:** `-D sym[=xx]`

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

The **Define symbol** (-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: aa90 prog
test version:       aa90 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:

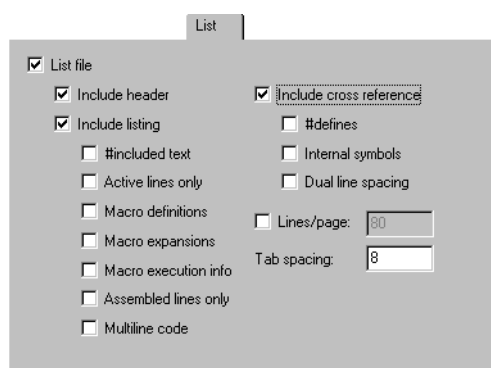
```
aa90 prog -Dframerate=3
```

## 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



### Command line

<code>-l filename</code>	List to named file.
<code>-L[<i>prefix</i>]</code>	List to prefixed source name.
<code>-N</code>	No header.
<code>-i</code>	#included text.
<code>-T</code>	Active lines only.
<code>-c{DMEA0}</code>	Conditional list.
<code>-B</code>	Macro execution info.
<code>-x{DI2}</code>	Cross reference.
<code>-p <i>lines</i></code>	Lines/page.
<code>-t <i>n</i></code>	Tab spacing.



## ASSEMBLER OPTIONS REFERENCE



### 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:

<i>Option</i>	<i>Description</i>
Include header	Includes a header in the listing.
Include listing	Includes the body of the listing.

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

<i>Option</i>	<i>Description</i>
#included text	Includes <i>#include</i> 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:

```
aa90 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.



## ASSEMBLER OPTIONS REFERENCE



### CONDITIONAL LIST (-c)

**Syntax:** -c {DMEA0}

Sets one or more of the following:



<i>Option</i>	<i>Command line</i>
Disable listing	D
Macro definitions	M
No macro expansions	E
Assembled lines only	A
Multiline code	0

### 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.



### 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:



<i>Option</i>	<i>Command line</i>
#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.

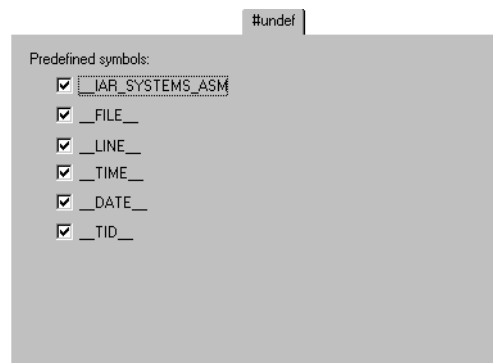
**TAB SPACING (-t)****Syntax:** `-t $n$` 

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****Command line****-U $symb$**       Undefine symbol.**UNDEFINE SYMBOL (-U)****Syntax:** `-U $symb$` 

Undefines the symbol  $symb$ .

By default, the assembler provides certain pre-defined symbols; see *Pre-defined symbols*, page 55. The **Undefine symbol** (-U) option allows you to undefine such a pre-defined symbol to make its name available

for your own use through a subsequent **Define symbol** (-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:

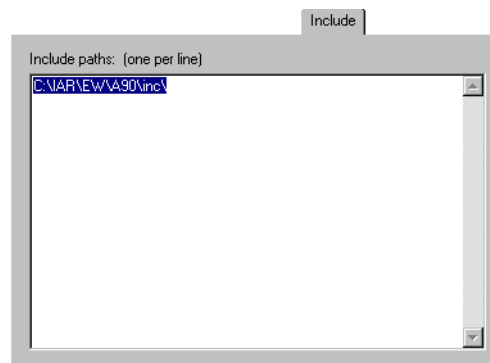
```
aa90 prog -U __TIME__
```

## INCLUDE

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



### Embedded Workbench



### Command line

`-Iprefix` Include paths.

## INCLUDE PATHS (-I)

**Syntax:** `-Iprefix`

Adds the `#include` file search prefix *prefix*.

By default, the assembler searches for `#include` files only in the current working directory. 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" ↵
```

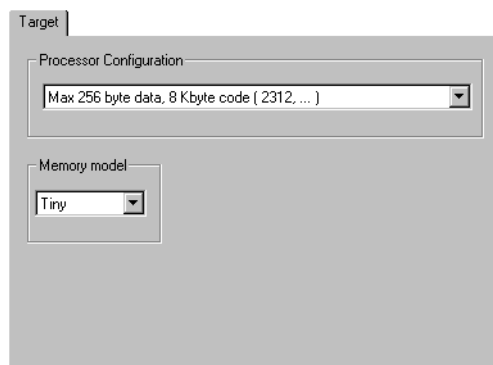
in the source, will make the assembler search first for file `asmlib.hdr`, then for file `c:\global\asmlib.hdr`, and finally for file `c:\thisproj\headers\asmlib.hdr`.

## TARGET

The **Target** options specify the processor and memory model for the assembler and C compiler.



### Embedded Workbench



### Command line

-v n	Processor configuration.
-m n	Memory model.



**PROCESSOR CONFIGURATION (-v)****Syntax:** `-v n`

Selects the processor configuration from one of:

<i>Option</i>	<i>Command line</i>
Max 256 byte data, 8 Kbyte code	-v0
Max 64 Kbyte data, 8 Kbyte code	-v1
Max 256 bytes data, 128 Kbytes code	-v2
Max 64 Kbytes data, 128 Kbytes code	-v3

Versions -v4 to -v6 are for future expansion.

If no **Chip option** (-v) option is specified, the assembler uses -v0 by default.**MEMORY MODEL (-m)****Syntax:** `-m n`

Selects the memory model from the following:

<i>Option</i>	<i>Command line</i>
Tiny	-mt
Small	-ms
Large*	-ml

\* Note that this option is included for future expansion.

## COMMAND LINE

The following additional options are available from the command line.

- Enumber*      Max 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.



### MAX NUMBER OF ERRORS (-E)

**Syntax:**        -*Enumber*

Sets the maximum number of errors the assembler reports.

By default, the maximum number is 100. The **Max number of errors** (-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:

```
aa90 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:** `-Oprefix`

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:

```
aa90 prog -Oobj\ ↵
```

This sends the object to `obj\prog.r90` rather than the default `prog.r90`.

`-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, `.r90` is used.

By default the assembler uses the source filename with the extension changed to `.r90`. The `-o` option lets you use an alternative filename for the object.

For example, the following command puts the object to the file `obj.r90` instead of the default `prog.r90`:

```
aa90 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



---

## ASSEMBLER OPTIONS REFERENCE

---

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.





# ASSEMBLER FILE FORMATS

This chapter describes the source format for the AT90S 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:

<i>label</i>	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.
<i>operation</i>	An assembler instruction or directive. This must not start in the first column.
<i>operands</i>	One or two operands, separated by commas.
<i>comment</i>	A comment, preceded by a ; (semi-colon).

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.

A \* in the first column indicates a comment line.

## 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 (PLC) 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:

```
NAME      prog1
EXTERN    third
RSEG      DATA
first     DB      5
second    DB      3
ENDMOD
MODULE    prog2
RSEG      CODE
start     ...
```

Then in segment CODE the following instructions are legal:

```
LDI      R27,first
LDI      R27,first+1
```

```
LDI    R27,1+first
LDI    R27,(first/second)*third
```

## 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), @ (at), or \_ (underline). 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:

```
RJMP    $    ; 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:

<i>Format</i>	<i>Example</i>	<i>Value</i>
<i>0xhex-digits</i>	0x43	67 in decimal.
<i>H'hex-digits</i>	H'43	67 in decimal.
<i>hex-digitsH</i>	43H	67 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:

<i>Format</i>	<i>Example</i>	<i>Value</i>
<i>'\octal-digits'</i>	<i>'\10'</i>	8 in decimal.
<i>Q'octal-digits</i>	<i>Q'10</i>	8 in decimal.
<i>octal-digitsQ</i>	<i>10Q</i>	8 in decimal.

**Decimal**

Decimal numbers can be written as follows:

<i>Format</i>	<i>Example</i>	<i>Value</i>
<i>digits</i>	<i>123</i>	123 in decimal.
<i>D'digits</i>	<i>D'123</i>	123 in decimal.

**Binary**

Binary numbers can be written as follows:

<i>Format</i>	<i>Example</i>	<i>Value</i>
<i>B'binary-digits</i>	<i>B'10</i>	2 in decimal.
<i>binary-digitsB</i>	<i>10B</i>	2 in decimal.

**ASCII CHARACTER CONSTANTS**

ASCII constants can consist of between zero and four characters enclosed in single 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:

<i>Format</i>	<i>Value</i>
<i>'ABCD'</i>	ABCD (four characters).
<i>"ABCD"</i>	ABCD'\0' (five characters, the last ASCII null).
<i>'A' 'B'</i>	A'B
<i>'A' ''</i>	A'
<i>'' '' (4 quotes)</i>	'



<i>Format</i>	<i>Value</i>
' ' (2 quotes)	Empty string (value= 0).
""	Empty string (an ASCII null character).
\ '	'
\\	\

### PRE-DEFINED SYMBOLS

The AT90S 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 pre-processor directives or include them in the assembled code.

<i>Symbol</i>	<i>Value</i>																
__DATE__	Current date in Mmm dd yy format.																
__FILE__	Current source filename.																
__IAR_SYSTEMS_ASM	IAR assembler identifier.																
__LINE__	Current source line number.																
__TID__	Target identity, consisting of two bytes. The high byte is the target identity, which is 90 for the AT90S. The low byte is the processor option *16. The possible values are therefore as follows:																
<table> <tr> <th><i>Processor option</i></th><th><i>Value</i></th></tr> <tr> <td>-v0</td><td>AT90S2312 0x5A00</td></tr> <tr> <td>-v1</td><td>AT90S8414 0x5A10</td></tr> <tr> <td>-v2</td><td>0x5A20</td></tr> <tr> <td>-v3</td><td>0x5A30</td></tr> <tr> <td>-v4</td><td>0x5A40</td></tr> <tr> <td>-v5</td><td>0x5A50</td></tr> <tr> <td>-v6</td><td>0x5A60</td></tr> </table>		<i>Processor option</i>	<i>Value</i>	-v0	AT90S2312 0x5A00	-v1	AT90S8414 0x5A10	-v2	0x5A20	-v3	0x5A30	-v4	0x5A40	-v5	0x5A50	-v6	0x5A60
<i>Processor option</i>	<i>Value</i>																
-v0	AT90S2312 0x5A00																
-v1	AT90S8414 0x5A10																
-v2	0x5A20																
-v3	0x5A30																
-v4	0x5A40																
-v5	0x5A50																
-v6	0x5A60																
__TIME__	Current time in hh:mm:ss format.																

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 display by the program:

```
timdat DB    __TIME__,"",__DATE__,0 ; time and date
      ...
      LD      WA,timdat      ; load address of string
      CALL    printstring    ; routine to print string
```

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 AT90S family members, you might want to assemble appropriate code for a specific processor. You could do this using the \_\_TID\_\_ symbol as follows:

```
#define TARGET ((__TID__ & 0x0F0)>>4)
#if (TARGET==1)
.
.
.
#else
.
.
.
#endif
```

REGISTER SYMBOLS

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

File	Processor
io2312.h	AT90S2312
io8414.h	AT90S8414



## ASSEMBLER FILE FORMATS

### LISTING FORMAT

The format of the AT90S Assembler listing is as follows:

		#####										
		#										
		# IAR Systems A90 Assembler Vx.xx										
		#										
Header		# Target option = Relative jumps reach entire addr space										
		# Source file = play.s90										
		# List file = play.lst										
		# Object file = play.r90										
		# Command line = play -r -L										
		#										
		# (c) Copyright IAR Systems 1996										
		#####										
Assembler listing		1	00000000			NAME	play					
		2	00000000									
		3	00000000			LSTXRF+						
		4	00000018		portb	VAR	0x18					
		5	00000000			RSEG	DATA					
		6	00000000		buffer	DS	256					
		7	00000200									
		19	00000000			RSEG	CODE					
		20	00000000			play						
		20.1	00000000			LOCAL	loop					
Macro generated lines		20.2	00000000 ....			LDI	R27,HWRD(buffer)					
		20.3	00000002 ....			LDI	R26,LWRD(buffer)					
		20.4	00000004 9FEF			LDI	R25,255					
		20.5	00000006 0D90		loop	LD	R0,X+					
		20.6	00000008 08BA			OUT	portb,R0					
		20.7	0000000A 9A95			DEC	R25					
		20.8	0000000C E1F7			BRNE	loop					
		20.9	0000000E			ENDM						
		21	0000000E			END						
		Segment	Type	Mode								
		-----										
		CODE	UNTYPED	REL								
		DATA	UNTYPED	REL								
		Label	Mode	Type		Segment	Value/Offset					
		-----										
		_args	ABS	CONST PUB LOCAL UNTYP.		ASEG	0					
		buffer	REL	CONST PUB UNTYP.		DATA	0					
		loop	REL	CONST PUB LOCAL UNTYP.		CODE	6					
		portb	ABS	VAR UNTYP.		ASEG	18					
		#####										
		# CRC:86EA										
CRC		# Errors: 0										
		# Warnings: 0										
		# Bytes: 14										
		#####										

The header, with assembly parameters, is only output on listings directed to files other than the terminal.

Assembly list information is put into four fields:

20.5	00000006	0D90	loop	LD	R0,X+
20.6	00000008	08BA		OUT	portb,R0
20.7	0000000A	9A95		DEC	R25
20.8	0000000C	E1F7		BRNE	loop
20.9	0000000E			ENDM	
21	0000000E			END	

Source line number

Address field

Data field

Source line

Source line number

The line number in the source file.

Lines generated by macros will, if listed, have . (full stop) 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:

Segments	Segment	Type	Mode			
	CODE	UNTYPED	REL			
Symbols	Label	Mode	Type	Segment	Value/Offset	
	Label	Mode	Type	Segment	Value/Offset	
	_args	ABS	CONST PUB LOCAL UNTYP.	ASEG	0	
	buffer	REL	CONST PUB UNTYP.	DATA	0	
	loop	REL	CONST PUB LOCAL UNTYP.	CODE	6	
	portb	ABS	VAR UNTYP.	ASEG	18	

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

<i>Information</i>	<i>Description</i>
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.

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



ASSEMBLER FILE FORMATS





# 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:



## ASSEMBLER OPERATOR SUMMARY

### UNARY OPERATORS – 1

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

**OR OPERATORS – 6**

OR (  )	Logical OR.
XOR	Logical exclusive OR.
BITOR ( )	Bitwise OR.
BITXOR (^)	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 SUMMARY

---





# 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	<b>DATE</b>	Date of assembly (1).
Description	<b>DESCRIPTION</b> 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 (1983 → 83).	
Examples	<b>EXAMPLES</b> To assemble the date of assembly: today DB DATE 5, DATE 4, DATE 3	

## 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.





---

**ASSEMBLER 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**

$2 * 2 \rightarrow 4$   
 $-2 * 2 \rightarrow -4$

---

**+**

Unary plus (1).

**DESCRIPTION**

Unary plus operator.

**EXAMPLES**

$+3 \rightarrow 3$   
 $3 * +2 \rightarrow 6$

---

**+**

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**

$92 + 19 \rightarrow 111$   
 $-2 + 2 \rightarrow 0$   
 $-2 + -2 \rightarrow -4$





---

## ASSEMBLER OPERATOR: –

---

---

–

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





## BITOR (|)

### BITOR (|)

Bitwise OR (6).

#### DESCRIPTION

Use BITOR to perform bitwise OR on its operands.

#### EXAMPLES

```
1010B BITOR 0101B → 1111B
1010B BITOR 0000B → 1010B
```

### BITXOR (^)

Bitwise exclusive OR (6).

#### DESCRIPTION

Use BITXOR to perform bitwise XOR on its operands.

#### EXAMPLES

```
1010B BITXOR 0101B → 1111B
1010B BITXOR 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.

#### EXAMPLES

```
BYTE2 0x12345678 → 0x56
```



## BYTE3

### 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.

#### EXAMPLES

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 (1983 → 83).

#### EXAMPLES

To assemble the date of assembly:

today DB DATE 5, DATE 4, DATE 3







## EQ (=, ==)

### 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 → 0
```

### 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

### 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.

#### EXAMPLES

HIGH 1234ABCDh → ABh

### HWRD

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.

#### EXAMPLES

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.

**EXAMPLES**

LOW 1234ABCDh → 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**

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.

**EXAMPLES**

LWRD 0x12345678 → 0x5678



## MOD (%)

### 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.

$X \text{ MOD } Y$  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**

<i>segment</i>	The name of a relocatable segment, which must be defined before SFB is used.
<i>offset</i>	An optional offset from the start address. The parentheses are optional if <i>offset</i> 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 DW 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

### 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    DW  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 → 11111111111100000B
14 SHL 1 → 28
```





## SHR (>>)

### 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 DW  SIZEOF CODE
sets size to the size of segment CODE.
```





## UGT

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

<i>Option</i>	<i>Description</i>	<i>Section</i>
<code>#define</code>	Assigns a value to a label.	C-style preprocessor.
<code>#elif</code>	Introduces a new condition in an <code>#if...#endif</code> block.	C-style preprocessor.
<code>#else</code>	Assembles instructions if a condition is false.	C-style preprocessor.
<code>#endif</code>	Ends a <code>#if</code> , <code>#ifdef</code> , or <code>#ifndef</code> block.	C-style preprocessor.
<code>#error</code>	Generates an error.	C-style preprocessor.
<code>#if</code>	Assembles instructions if a condition is true.	C-style preprocessor.
<code>#ifdef</code>	Assembles instructions if a symbol is defined.	C-style preprocessor.
<code>#ifndef</code>	Assembles instructions if a symbol is undefined.	C-style preprocessor.
<code>#include</code>	Includes a file.	C-style preprocessor.

## ASSEMBLER DIRECTIVES SUMMARY

<i>Option</i>	<i>Description</i>	<i>Section</i>
<code>#undef</code>	Undefines a label.	C-style preprocessor.
<code>\$</code>	Includes a file.	Assembler control.
<code>/*comment*/</code>	C-style comment delimiter.	Assembler control.
<code>//</code>	C++ style comment delimiter.	Assembler control.
<code>=</code>	Assigns a permanent value local to a module.	Value assignment.
<code>ALIAS</code>	Assigns a permanent value local to a module.	Value assignment.
<code>ALIGN</code>	Aligns the program counter by inserting zero-filled bytes.	Segment control.
<code>ASEG</code>	Begins an absolute segment.	Segment control.
<code>ASSIGN</code>	Assigns a temporary value.	Value assignment.
<code>CASEOFF</code>	Disables case sensitivity.	Assembler control.
<code>CASEON</code>	Enables case sensitivity.	Assembler control.
<code>COL</code>	Sets the number of columns per page.	Listing control.
<code>COMMON</code>	Begins a common segment.	Segment control.
<code>DB</code>	Generates 8-bit byte constants.	Data definition or allocation.
<code>DD</code>	Generates 32-bit double word constants.	Data definition or allocation.
<code>DEFINE</code>	Defines a file-wide value.	Value assignment.
<code>DP</code>	Generates 24-bit double word constants.	Data definition or allocation.
<code>DS</code>	Allocates space for 8-bit bytes.	Data definition or allocation.
<code>DW</code>	Generates 16-bit word constants.	Data definition or allocation.
<code>ELSE</code>	Assembles instructions if a condition is false.	Conditional assembly.



## ASSEMBLER DIRECTIVES SUMMARY

<i>Option</i>	<i>Description</i>	<i>Section</i>
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.
EXPORT	Exports symbols to other modules.	Symbol control.
EXTERN	Imports an external symbol.	Symbol control.
IF	Assembles instructions if a condition is true.	Conditional assembly.
IMPORT	Imports an external symbol.	Symbol control.
LIBRARY	Begins a library module.	Module control.
LIMIT	Checks a value against limits.	Value assignment.
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.



## ASSEMBLER DIRECTIVES SUMMARY

<i>Option</i>	<i>Description</i>	<i>Section</i>
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.
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.
RSEG	Begins a relocatable segment.	Segment control.
sfrb	Creates byte-access SFR labels.	Value assignment.
SFRTYPE	Specifies SFR attributes.	Value assignment.



## ASSEMBLER DIRECTIVES SUMMARY

<i>Option</i>	<i>Description</i>	<i>Section</i>
sfrw	Creates word-access SFR labels.	Value assignment.
STACK	Begins a stack segment.	Segment control.
VAR	Assigns a temporary value.	Value assignment.





---

## ASSEMBLER DIRECTIVES SUMMARY

---



# ASSEMBLER DIRECTIVES REFERENCE

This chapter gives a list of the AT90S 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																																														
Class	<b>SYMBOL CONTROL DIRECTIVES</b>	These directives control how symbols are shared between modules.																																												
Summary		<table><tr><th>Directive</th><th>Description</th></tr><tr><td>PUBLIC (EXPORT)</td><td>Exports symbols to other modules.</td></tr><tr><td>EXTERN (IMPORT)</td><td>Imports an external symbol.</td></tr></table>	Directive	Description	PUBLIC (EXPORT)	Exports symbols to other modules.	EXTERN (IMPORT)	Imports an external symbol.																																						
Directive	Description																																													
PUBLIC (EXPORT)	Exports symbols to other modules.																																													
EXTERN (IMPORT)	Imports an external symbol.																																													
Syntax	<b>SYNTAX</b>  PUBLIC <i>symbol</i> [, <i>symbol</i> ] ...  EXTERN <i>symbol</i> [, <i>symbol</i> ] ...																																													
Parameters	<b>PARAMETERS</b>  <i>symbol</i> Symbol to be imported or exported.																																													
Description	<b>DESCRIPTION</b>  <b>Exporting symbols to other modules</b> 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).  <b>Importing symbols</b> Use EXTERN to import an untyped external symbol.																																													
Examples	<b>EXAMPLES</b>  The following example defines a subroutine to print an error message, and exports the entry address <i>err</i> so that it can be called from other modules.  <table><tr><td>1</td><td>00000000</td><td></td><td>NAME</td><td>error</td></tr><tr><td>2</td><td>00000000</td><td></td><td>EXTERN</td><td>print</td></tr><tr><td>3</td><td>00000000</td><td></td><td>PUBLIC</td><td>err</td></tr><tr><td>4</td><td>00000000</td><td></td><td></td><td></td></tr><tr><td>5</td><td>00000000</td><td>.....</td><td>err</td><td>CALL</td><td>print</td></tr><tr><td>6</td><td>00000004</td><td>2A2A2A2A45</td><td>DB</td><td colspan="2">"*****Error*****"</td></tr><tr><td>7</td><td>00000013</td><td>0895</td><td>RET</td><td colspan="2"></td></tr><tr><td>8</td><td>00000015</td><td></td><td>END</td><td>err</td><td></td></tr></table>	1	00000000		NAME	error	2	00000000		EXTERN	print	3	00000000		PUBLIC	err	4	00000000				5	00000000	.....	err	CALL	print	6	00000004	2A2A2A2A45	DB	"*****Error*****"		7	00000013	0895	RET			8	00000015		END	err		
1	00000000		NAME	error																																										
2	00000000		EXTERN	print																																										
3	00000000		PUBLIC	err																																										
4	00000000																																													
5	00000000	.....	err	CALL	print																																									
6	00000004	2A2A2A2A45	DB	"*****Error*****"																																										
7	00000013	0895	RET																																											
8	00000015		END	err																																										

85

## 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 VAR 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:

<i>Parameter</i>	<i>What it consists of</i>
<i>symbol</i>	An assembler symbol.
<i>label</i>	A symbolic label.
<i>expr</i>	An expression; see <i>Expressions and operators</i> , page 51.

## 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.

<i>Directive</i>	<i>Description</i>
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.

## SYNTAX

NAME *symbol* [(*expr*)]

MODULE *symbol* [(*expr*)]

ENDMOD [*label*]

END [*label*]

## PARAMETERS

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

## 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.



## MODULE CONTROL DIRECTIVES

### 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`.

## EXAMPLES

The following example defines three modules:

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

SYMBOL CONTROL DIRECTIVES

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

SYMBOL CONTROL DIRECTIVES

These directives control how symbols are shared between modules.

Directive	Description
PUBLIC (EXPORT)	Exports symbols to other modules.
EXTERN (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).

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, BYTE2, and BYTE3 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.



## SEGMENT CONTROL DIRECTIVES

### 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.

Since the message is enclosed in double quotes, the string will be followed by a zero byte.

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

1	00000000		NAME	error	
2	00000000		EXTERN	print	
3	00000000		PUBLIC	err	
4	00000000				
5	00000000	.....	err	CALL	print
6	00000004	2A2A2A2A2A45		DB	"*****Error*****"
7	00000013	0895		RET	
8	00000015			END	err

## SEGMENT CONTROL DIRECTIVES

The segment directives control how code and data are generated.

<i>Directive</i>	<i>Description</i>
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 program counter by inserting zero-filled bytes.
EVEN	Aligns the program counter to an even address.

## SYNTAX

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

## PARAMETERS

<i>start</i>	A start address which has the same effect as using an ORG directive at the beginning of the absolute segment.
<i>segment</i>	The name of the segment.
<i>type</i>	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, and CONST.
<i>expr</i>	Address to set location counter to.
<i>align</i>	Power of two to which the address should be aligned, in the range 0 to 30.

## 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 where you desire to have a number of different routines 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 156.

Specifying the *align* parameter in any of the above directives is equivalent to including an ALIGN directive with the same value.

**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 program counter to a specified address boundary. The expression gives the power of two to which the program counter should be aligned. The EVEN directive aligns the program counter to an even address (which is equivalent to ALIGN 1).

## EXAMPLES

### Beginning an absolute segment

The following example assembles interrupt routine entry addresses in the appropriate AT90S interrupt vectors using an absolute segment:

```

        EXTERN  reset,IRQ0,IRQ1,TIM1CAPT

        ASEG
        ORG     0h
vec0    DW      TIM1CAPT 0
vec1    DW      TIM1CAPT 1
vec2    DW      TIM1CAPT 2
vec3    DW      TIM1CAPT .. etc

        ORG     15h
reset

        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 ORG directive is used to create a gap of six bytes in the table.

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

```

        EXTERN  divrtn,mulrtn

        RSEG    table
        DW      divrtn,mulrtn

        ORG     $+6
        DW      subrtn
    
```



```

                                RSEG    code
subrtn  MOV    R16,R17
        SUBI   R16,20
        END
    
```

### Beginning a stack segment

The following example defines two 100-byte stacks in a relocatable segment called `rpystack`:

```

                                STACK   rpystack
parms   DS      100
opers   DS      100

                                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      DB      1
                                ORG      $+2
down    DB      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

### Aligning a segment

This example starts a relocatable segment, moves to an even address and adds some data. It then aligns to a 64-byte boundary before creating a 64-byte table.

```
                RSEG    data    ; Start a relocatable data
                                segment
                EVEN     ; Ensure it is on an even
                                boundary
target DW      1          ; target and best will be on an
                                even boundary
best  DW      1
      ALIGN   6          ; Now align to a 64 byte
                                boundary
results DS     64        ; And create a 64 byte table
      END
```

## VALUE ASSIGNMENT DIRECTIVES

These directives are used to assign values to symbols.

<i>Directive</i>	<i>Description</i>
VAR (ASSIGN)	Assigns a temporary value.
EQU (ALIAS,=)	Assigns a permanent value local to a module.
DEFINE	Defines a file-wide value.
LIMIT	Checks a value against limits.
sfrb	Creates byte-access SFR labels.
sfrw	Creates word-access SFR labels.
SFRTYPE	Specifies SFR attributes.

### SYNTAX

*label* VAR *expr*

*label* EQU *expr*

*label* = *expr*

*label* DEFINE *expr*



## VALUE ASSIGNMENT DIRECTIVES

LIMIT *label,min,max,message*

[const] sfrb *register* = *value*

[const] sfrw *register* = *value*

[const] SFRTYPE *register attribute* [,*attribute*] = *value*

### PARAMETERS

*label*            Symbol to be defined.

*expr*            Value assigned to symbol.

*register*        The special function register.

*attribute*       One or more of the following:

READ            You can read from this SFR.

WRITE           You can write to this SFR.

BYTE            The SFR must be accessed as a byte.

WORD            The SFR must be accessed as a word.

*value*            The SFR port address.

*min, max*        The minimum and maximum values allowed for *label*.

*message*        A text message that will be printed when the symbol is out of range.

### DESCRIPTION

#### Defining a temporary value

Use VAR to define a symbol which may be redefined, such as for use with macro variables. Symbols defined with VAR 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.

### Defining special function registers

Use `sfrb` to create special function register labels with attributes `READ`, `WRITE`, and `BYTE` turned on. Use `sfrw` to create special function register labels with attributes `READ`, `WRITE`, or `WORD` turned on. Use `SFRTYPE` to create special function register labels with specified attributes.

Prefix the directive with `const` to disable the `WRITE` attribute assigned to the SFR. You will then get an error/warning when trying to write to the SFR.

### Checking symbol values

Use `LIMIT` to check that symbols lie within a specified range. If the symbol is assigned a value outside the range an error message will be printed.

*min, max*      The minimum and maximum values allowed for *label*.

*message*      A text message that will be printed when the symbol is out of range.

The check will occur as soon as the value is resolved, which will be during linking if the expression contains external references. The *min* and *max* expressions cannot involve references to forward or external labels, ie they must be resolved when encountered.

## 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
cons      VAR      1
buildit   MACRO    times
          DW        cons
```



## VALUE ASSIGNMENT DIRECTIVES

```
cons    VAR      cons * 3
        IF      times > 1
        buildit times - 1
        ENDIF
        ENDM
main    buildit 4
        END
```

It generates the following code:

1	00000000		NAME	table
2	00000001	cons	VAR	1
10	00000000	main	buildit	4
10	00000000	main	buildit	4
10.1	00000000 0100		DW	cons
10.2	00000003	cons	VAR	cons * 3
10.3	00000002		IF	4 > 1
10.4	00000002		buildit	4 - 1
10.5	00000002 0300		DW	cons
10.6	00000009	cons	VAR	cons * 3
10.7	00000004		IF	4 - 1 > 1
10.8	00000004		buildit	4 - 1 - 1
10.9	00000004 0900		DW	cons
10.10	00000018	cons	VAR	cons * 3
10.11	00000006		IF	4 - 1 - 1 > 1
10.12	00000006		buildit	4 - 1 - 1 - 1
10.13	00000006 1800		DW	cons
10.14	00000051	cons	VAR	cons * 3
10.15	00000008		IF	4 - 1 - 1 - 1 > 1
10.16	00000008		buildit	4 - 1 - 1 - 1 - 1
10.17	00000008		ENDIF	
10.18	00000008		ENDM	
10.19	00000008		ENDIF	
10.20	00000008		ENDM	
10.21	00000008		ENDIF	
10.22	00000008		ENDM	
10.23	00000008		ENDIF	
10.24	00000008		ENDM	
11	00000008		END	

### 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` symbol is used to declare `locn` for use anywhere in the file:

```

locn    NAME    add1
value   DEFINE  020h
value   EQU     77
        CLR     R27
        LDI     R26,locn
        LD      R16,X
        LDI     R17,value
        ADD     R16,R17
        RET
        ENDMOD

```

```

value   NAME    add2
value   EQU     88
        CLR     R27
        LDI     R26,locn
        LD      R16,X
        LDI     R17,value
        ADD     R16,R17
        RET
        END

```

The symbol `locn` defined in module `add1` is also available to module `add2`.

### Using special function registers

In this example a number of `sfr` variables are declared with a variety of access capabilities.

```

sfrb portd                = 0x12  /* byte read/write
                                access */
sfrw ocr1                 = 0x2A  /* word read/write
                                access */
const sfrb pind           = 0x10  /*byte read only access
                                */
SFRTYPE portb write, byte = 0x18  /* byte write only
                                access */

```



## CONDITIONAL ASSEMBLY DIRECTIVES

### Using the LIMIT directive

The following example sets the value of a variable called speed and then checks it (at assembly time) to see if it is in the range 10 to 30. This might be useful if speed was often changed at compile time, but values outside a defined range would cause undesirable behaviour.

```
speed    VAR        23
LIMIT   speed,10,30,"fred out of range"
```

## CONDITIONAL ASSEMBLY DIRECTIVES

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

<i>Directive</i>	<i>Description</i>
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 condition
ENDIF
```

## PARAMETERS

<i>condition</i>	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.
	<i>string1=string2</i>	The condition is true if <i>string1</i> and <i>string2</i> have the same length and contents.
	<i>string1&lt;&gt;string2</i>	The condition is true if <i>string1</i> and <i>string2</i> have different length or contents.

## DESCRIPTION

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 an ELSE or ENDIF directive is found.

Use ELSEIF to introduce a new condition after an IF directive.

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 IFxx directive must be terminated by an 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.

## EXAMPLES

The following macro subtracts a constant from the register 'r'.

```
sub    MACRO    r,c
        IF      c=1
        DEC     r
        ELSE
        SUBI    r,c
```





## MACRO PROCESSING DIRECTIVES

ENDIF  
ENDM

If the argument to the macro is 2 it generates an SUBI instruction to save instruction cycles; otherwise it generates a DEC instruction.

It could be tested with the following program:

```
main    LDI    R16,17
        sub    R16,2
        LDI    R17,22
        sub    R17,1
        RET
        END
```

## MACRO PROCESSING DIRECTIVES

These directives allow user macros to be defined.

<i>Directive</i>	<i>Description</i>
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*



## MACRO PROCESSING DIRECTIVES

```
REPTC formal,actual
REPTI formal,actual [,actual] ...
ENDR
```

### PARAMETERS

<i>name</i>	The name of the macro.
<i>argument</i>	A symbolic argument name.
<i>symbol</i>	Symbol to be local to the macro.
<i>expr</i>	An expression.
<i>formal</i>	Argument into which each character of <i>actual</i> (REPTC) or each <i>actual</i> (REPTI) is substituted.
<i>actual</i>	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 just 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 ERROR as follows:

```
errmac MACRO    text
          CALL    abort
          DB      text,0
        ENDM
```





## MACRO PROCESSING DIRECTIVES

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

```
errmac    'Disk not ready'
```

This will be expanded by the assembler to:

```
CALL      abort
DB        'Disk not ready',0
```

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
          CALL      abort
          DB        \1,0
          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
          LDI      op
          ENDM
```



## MACRO PROCESSING DIRECTIVES

It could be called using:

```
macld    <R16, 1>
END
```

You can redefine the macro quote characters with the `-M` command line option; see *Macro quote chars (-M)*, page 36.

### 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.





## MACRO PROCESSING DIRECTIVES

### 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.

For example, the following subroutine outputs bytes from a buffer to a port:

```
NAME      play

portb     VAR      0x18
          RSEG     DATA
buffer    DS       256

          RSEG     CODE
play      LDI      R27,HIGH(buffer)
          LDI      R26,LOW(buffer)
          LDI      R25,255
loop      LD       R0,X+
          OUT      portb,R0
          DEC      R25
          BRNE     loop
          RET
          END
```

The main program calls this routine as follows:

```
doplay    CALL     play
```

For efficiency we can recode this as the following macro:

```
NAME      play

portb     VAR      0x18
          RSEG     DATA
buffer    DS       256

play      MACRO
          LOCAL    loop
          LDI      R27,HIGH(buffer)
```



## MACRO PROCESSING DIRECTIVES

```

                                LDI    R26,LOW(buffer)
                                LDI    R25,255
loop    LD      R0,X+
                                OUT    portb,R0
                                DEC    R25
                                BRNE   loop
                                ENDM

                                RSEG   CODE
                                play
                                END
```

Note the use of the `LOCAL` directive to make the label `loop` local to the macro; otherwise an error will be generated if the macro is used twice, as the `loop` label will already exist.

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

```
doplay  play
```

### 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
banner  REPTC   chr,"Welcome"
                                LDI      R16,'chr'
                                CALL     plotc
                                ENDR

                                END
```

This produces the following code:

1	00000000	NAME	reptc
2	00000000		
3	00000000	EXTERN	plotc
4	00000000	banner REPTC	chr,"Welcome"
5	00000000	LDI	R16,'chr'
6	00000000	CALL	plotc
7	00000000	ENDR	
7.1	00000000 07E5	LDI	R16,'W'
7.2	00000002 .....	CALL	plotc



## MACRO PROCESSING DIRECTIVES

```
7.3 00000006 05E6      LDI    R16,'e'
7.4 00000008 .....    CALL   plotc
7.5 0000000C 0CE6      LDI    R16,'l'
7.6 0000000E .....    CALL   plotc
7.7 00000012 03E6      LDI    R16,'c'
7.8 00000014 .....    CALL   plotc
7.9 00000018 0FE6      LDI    R16,'o'
7.10 0000001A .....    CALL   plotc
7.11 0000001E 0DE6      LDI    R16,'m'
7.12 00000020 .....    CALL   plotc
7.13 00000024 05E6      LDI    R16,'e'
7.14 00000026 .....    CALL   plotc
8    0000002A
9    0000002A          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
          LDI      R30,LOW(adds)
          LDI      R31,HIGH(adds)
          LDI      R16,0
          STD      Z+0,R16
          ENDR

          END
```

This produces the following code:

```
1    00000000          NAME      repti
2    00000000
3    00000000          EXTERN    base,count,init
4    00000000
5    00000000          banner    REPTI    adds,base,count,init
6    00000000          LDI      R30,LOW(adds)
7    00000000          LDI      R31,HIGH(adds)
8    00000000          LDI      R16,0
9    00000000          STD      Z+0,R16
10   00000000          ENDR
10.1 00000000 ....    LDI      R30,LOW(base)
```

## LISTING CONTROL DIRECTIVES

10.2	00000002	....	LDI	R31,HIGH(base)
10.3	00000004	00E0	LDI	R16,0
10.4	00000006	0083	STD	Z+0,R16
10.5	00000008	....	LDI	R30,LOW(count)
10.6	0000000A	....	LDI	R31,HIGH(count)
10.7	0000000C	00E0	LDI	R16,0
10.8	0000000E	0083	STD	Z+0,R16
10.9	00000010	....	LDI	R30,LOW(init)
10.10	00000012	....	LDI	R31,HIGH(init)
10.11	00000014	00E0	LDI	R16,0
10.12	00000016	0083	STD	Z+0,R16
11	00000018			
12	00000018		END	

## LISTING CONTROL DIRECTIVES

These directives provide control over the assembler listing.

<i>Directive</i>	<i>Description</i>
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, PTITL, and PSTITL.





## LISTING CONTROL DIRECTIVES

### 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 132.

*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 for error messages. This 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.



## LISTING CONTROL DIRECTIVES

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 132.

Use PAGESIZ 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 CONTROL DIRECTIVES

### 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   VAR    0
begin   IF      debug
        CALL    print
        ENDIF

LSTCND+
begin2   IF      debug
        CALL    print
        ENDIF
END
```

This will generate the following listing:

```
1  00000000      NAME    lstcndtst
2  00000000      EXTERN  print
3  00000000
4  00000000      RSEG    prom
5  00000000      debug   VAR    0
6  00000000      begin   IF      debug
7  00000000      CALL    print
8  00000000      ENDIF
9  00000000
10 00000000      LSTCND+
11 00000000      begin2   IF      debug
13 00000000      ENDIF
14 00000000      END
```

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

```
1  00000000      NAME    lstcodtst
2  00000000 01000A006400  DW      1,10,100,100,10000
3  0000000A
4  0000000A      LSTCOD+
5  0000000A 01000A006400  DW      1,10,100,1000,10000
    E8031027
6  00000014      END
```

LISTING CONTROL DIRECTIVES

```
Controlling the listing of macros
The following example shows the effect of LSTMAC and LSTEXP:

dec2      MACRO      arg
          DEC        arg
          DEC        arg
          ENDM

          LSTMAC-

inc2      MACRO      arg
          INC        arg
          INC        arg
          ENDM

begin     dec2      R16

          LSTEXP-
          inc2      R17
          RET

          END        begin
```

This will produce the following output:

```
5      00000000
6      00000000          LSTMAC-
7      00000000
12     00000000
13     00000000          begin  dec2      R16
13     00000000          begin  dec2      R16
13.1   00000000 0A95          DEC        R16
13.2   00000002 0A95          DEC        R16
13.3   00000004          ENDM
14     00000004
15     00000004          LSTEXP-
16     00000004          inc2      R17
17     00000008 0895          RET
18     0000000A
19     0000000A          END        begin
```



## C-STYLE PREPROCESSOR DIRECTIVES

### Formatting listed output

The following example formats the output into pages of 66 lines each with 80 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 80
LSTPAG+
...
ENDMOD
MODULE
...
PAGE
...
```

## C-STYLE PREPROCESSOR DIRECTIVES

The following C-language preprocessor directives are available:

<i>Directive</i>	<i>Description</i>
<code>#define</code>	Assigns a value to a label.
<code>#undef</code>	Undefines a label.
<code>#if</code>	Assembles instructions if a condition is true.
<code>#ifdef</code>	Assembles instructions if a symbol is defined.
<code>#ifndef</code>	Assembles instructions if a symbol is undefined.
<code>#elif</code>	Introduces a new condition in an <code>#if...#endif</code> block.
<code>#else</code>	Assembles instructions if a condition is false.
<code>#endif</code>	Ends a <code>#if</code> , <code>#ifdef</code> , or <code>#ifndef</code> block.
<code>#include</code>	Includes a file.
<code>#error</code>	Generates an error.

### SYNTAX

```
#define label text
#undef label
#if condition
```

## C-STYLE PREPROCESSOR DIRECTIVES

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

### PARAMETERS

<i>label</i>	Symbol to be defined, undefined, or tested.	
<i>text</i>	Value to be assigned.	
<i>condition</i>	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.
	<i>string1</i> = <i>string2</i>	The condition is true if <i>string1</i> and <i>string2</i> have the same length and contents.
	<i>string1</i> <> <i>string2</i>	The condition is true if <i>string1</i> and <i>string2</i> have different length or contents.
<i>filename</i>	Name of file to be included.	
<i>message</i>	Text to be displayed.	

### DESCRIPTION

#### Defining and undefining labels

Use `#define` to define a temporary label.

```
#define label value
```



## C-STYLE PREPROCESSOR DIRECTIVES

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.

Use `#elif` to introduce a new condition after a `#if` directive.

`#if ... #elif ... #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 defines the labels `tweek` and `adjust`. If `adjust` is defined then register 16 is decremented by an amount that depends on `adjust`, in this case 30.

```
#define tweek 1
#define adjust 3

#ifdef tweek
#if adjust=1
    SUBI    R16,4
#elif adjust=2
    SUBI    R16,20
#elif adjust=3
    SUBI    R16,30
#endif
#endif /* ifdef tweek*/
```

**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.s90`:

```
xch    MACRO    a,b
        PUSH    a
        MOV     a,b
        POP     b
    ENDM
```

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

```
NAME    include

;Standard macro definitions
#include "macros.s90"

; Program
main    xch      R16,R17
        RET
        END      main
```





## DATA DEFINITION OR ALLOCATION DIRECTIVES

### DATA DEFINITION OR ALLOCATION DIRECTIVES

These directives define temporary values or reserve memory.

<i>Directive</i>	<i>Description</i>
DS	Allocates space for 8-bit bytes.
DB	Generates 8-bit byte constants.
DW	Generates 16-bit word constants.
DP	Generates 24-bit double word constants.
DD	Generates 32-bit double word constants.

### SYNTAX

DS *expr*

DB *expr*[, *expr*]

DW *expr*[, *expr*]

DP *expr*[, *expr*]

DD *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 DS to allocate space. The memory contents are not initialized in any way.

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

### EXAMPLES

The following example generates a lookup table of addresses to routines:

```
NAME    table

table   DW    addsubr,subsubr,clrsubr
```



## ASSEMBLER CONTROL DIRECTIVES

```
addsubr ADD      R16,R17
        RET
```

```
subsubr SUB      R16,R17
        RET
```

```
clrsubr CLR      R16
        RET
```

```
END
```

### Defining strings

To define a string:

```
mymess DB 'Please enter your name'
```

To define a string which includes a trailing zero:

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

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

```
errmess DB '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.

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

**SYNTAX**

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

**PARAMETERS**

<i>filename</i>	Name of file to be included. The \$ character must be the first character on the line.
<i>comment</i>	Comment ignored by the assembler.
<i>expr</i>	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 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.s90`:

```
xch      MACRO    a,b
          PUSH     a
          MOV      a,b
          POP      b
          ENDM
```

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

```
NAME      include

;Standard macro definitions
$macros.s90

; Program
main      xch      R16,R17
          RET
          END      main
```

### Defining comments

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

```
/*
Program to read serial input.
Version 2: 19.6.94
Author: mjp
*/
```

### Changing the base

To set the default base to 16:

```
RADIX D'16
LD     A,12
```

The immediate argument will then be interpreted as H'12.

**Controlling case sensitivity**

By default CASEOFF is active, so in the following example label and LABEL are identical:

```
label    NOP        ;stored as "LABEL"  
        JMP        LABEL
```

However, the following will generate a duplicate label error:

```
label    NOP  
LABEL    NOP        ;Error: "LABEL" already defined  
END
```



---

## ASSEMBLER CONTROL DIRECTIVES

---





# ASSEMBLER INSTRUCTIONS

This chapter lists the mnemonics of the AT90S processor.

## CPU INSTRUCTION MNEMONICS

The following symbols are used in the list of instruction mnemonics:

<i>Symbol</i>	<i>What it means</i>
b	A bit in a register in the range 0 to 7.
s	A bit in the status register in the range 0 to 255.
k	Any address.
k64	Any constant in the range -64 to + 63.
k255	Any constant in the range 0 to 255.
k4096	Any constant in the range -4094 to + 4096.
k64K	Any constant in the range 0 to 65535.
k4M	Any constant in the range 0 to 4194303.
p	A port number in the range 0 to 63.
p32	A port number in the range 0 to 31.
q	An offset in the range 0 to 63.
r16	Any of R16 to R31.
r32	Any of R0 to R31.
rd1	R24, R26, R28, or R30.
X	R27:R26.
Y	R29:R28.
Z	R31:R30.



## ARITHMETIC AND LOGIC INSTRUCTIONS

	<i>First argument</i>	<i>Second argument</i>
ADD, ADC, SUB, SBC AND, OR, EOR, MUL <sup>1</sup>	r32	r32
<sup>1</sup> MUL is not available with all processors, but it is accepted by the assembler regardless of the processor option.		
SUBI, SBCI, ANDI ORI, SBR, CBR	r16	k255
COM, NEG, INC DEC, TST, CLR	r32	
SER	r16	
ADIW, SBIW	rd1	k64

## BRANCH INSTRUCTIONS

	<i>First argument</i>	<i>Second argument</i>
JMP	k4M	
CALL	k	
RJMP, RCALL	k4096	
BRCC, BRCS, BREQ BRGE, BRSH, BRID BRIE, BRLO, BRLT BRMI, BRNE, BRHC BRHS, BRPL, BRTC BRTS, BRVC, BRVS	k64	
CPI	r16	k255
BRBC, BRBS	s	k64
CPSE, CP, CPC	r32	r32
SBRC, SBRS	r32	b



## ASSEMBLER INSTRUCTIONS

	<i>First argument</i>	<i>Second argument</i>
SBIC, SBIS	p32	b
IJMP, ICALL, RET, RETI	-	

## DATA TRANSFER INSTRUCTIONS

	<i>First argument</i>	<i>Second argument</i>
MOV	r32	r32
LDI	r32	k255
LD	r32	X
	r32	X+
	r32	-X
	r32	Y
	r32	Y+
	r32	-Y
	r32	Z
	r32	Z+
	r32	-Z
LDD	r32	Y+q
	r32	Z+q
LDS <sup>2</sup>	r32	k64K
ST	X	r32
	X+	r32
	-X	r32
	Y	r32
	Y+	r32
	-Y	r32
	Z	r32
	Z+	r32
	-Z	r32
STD	Y+q	r32
	z+q	r32

<sup>2</sup> Not available with processor options -v0 or -v2.

## ASSEMBLER INSTRUCTIONS

	<i>First argument</i>	<i>Second argument</i>
STS <sup>3</sup>	r32	k64K
<sup>3</sup> Not available with processor options -v0 or -v2.		
LPM	None	
IN, OUT	r32	p
PUSH, POP	r32	
CBI, SBI	p32	b

## BIT AND BIT-TEST INSTRUCTIONS

	<i>First argument</i>	<i>Second argument</i>
LSL, LSR, ROL ROR, ASR, SWAP	r32	
BST, BLD	r32	b
CLC, CLIM, CLN CLH, CLS, CLT CLV, CLZ, SEC SEI, SEN, SEH SES, SET, SEV, SEZ	None	
BSET, BCLR	s	

## MISCELLANEOUS

	<i>First argument</i>	<i>Second argument</i>
NOP, SLEEP, WDR	None	

# XLINK LINKER

This chapter describes the IAR Systems XLINK Linker, and gives examples of how it can be used.

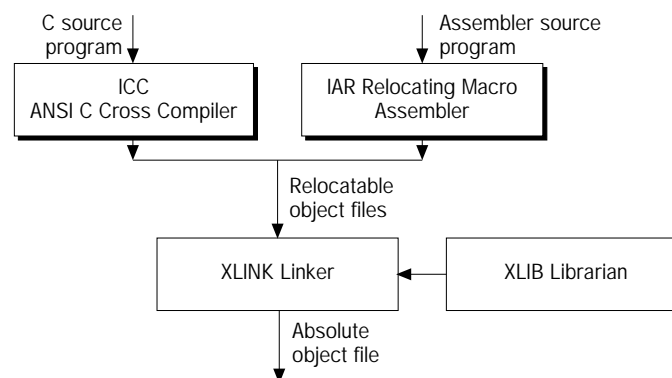
Note that some of the options described in the following chapters may not be available for all assemblers.

## 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.

### **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 locates each segment of code or data at a user-specified address.
- ◆ 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 libraries.

### **LIBRARIES**

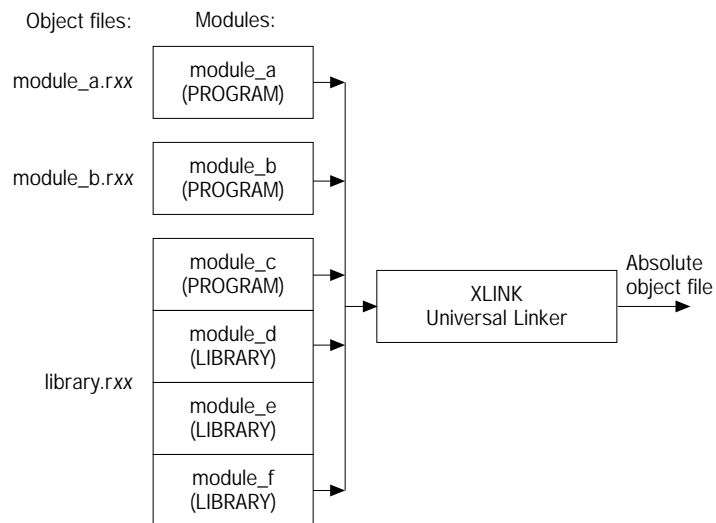
When XLINK reads a library file (which can contain multiple C or assembler modules) it will only load those modules which are actually needed by the program you are linking. This avoids having to load all the modules in a library file when you only need one routine. The XLIB Librarian is used to manage 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 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.sxx` and `module_b.sxx`, 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; ie the filenames are given as arguments.

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.rxx` 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.rxx` 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.rxx` 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, and no more. 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 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 `.rxx` 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 force a C module to 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 you have only limited control over segment naming and usage.

## LISTING FORMAT

The default XLINK listing format is shown below:

```
#####
#
# IAR Universal Linker Vx.xx
#
# Target CPU = xxxxx
# List file = c:\iar\ew\program\release\list\out.map
# Output file 1 = c:\iar\ew\program\release\exe\out.hex
# Output format = debug
# Command line = -o C:\IAR\EW\PROGRAM\Release\exe\out.hex
#               -rt -f C:\IAR\EW\PROGRAM\ICCxxx\Lnk_kbs.xcl
#               -l C:\IAR\EW\PROGRAM\Release\list\out.map
#               -x -Ic:\Program Files\iar\ew\program\iccxxx\
#               C:\IAR\EW\PROGRAM\Release\obj\tutor1.rxx
#
# (c) Copyright IAR Systems 1996 #
#####

*****
*
* CROSS REFERENCE
*
*****

Program entry at : 00002080 Relocatable, from module : CSTARTUP

*****
*
* MODULE MAP
*
*****

DEFINED ABSOLUTE ENTRIES
PROGRAM MODULE, NAME : ?ABS_ENTRY_MOD
ABSOLUTE ENTRIES      ADDRESS      REF BY MODULE
-----
SET_CCB3               0000FFFF      CSTARTUP
SET_CCB2               0000FFFF      CSTARTUP
SET_CCB1               000027FE      CSTARTUP
SET_CCB0               000020FF      CSTARTUP

*****

FILE NAME : c:\program files\iar\ew\program\release\obj\tutor1.rxx
PROGRAM MODULE, NAME : tutor1

SEGMENTS IN THE MODULE

CODE
Relative segment, address : 0000210C - 00002141
ENTRIES      ADDRESS      REF BY MODULE
do_foreground_process  0000210C      Not referred to
calls direct
main           00002120      CSTARTUP
calls direct
LOCALS
?0001         00002133
?0000         0000213F

-----
CONST
Relative segment, address : 00002146 - 00002146
ENTRIES      ADDRESS      REF BY MODULE
con_char     00002146      Not referred to

-----
WRKSEG
Common segment, address : 00000024 - 00000043

*****
*
* SEGMENTS IN DUMP 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
IDATA0      Not in use      rel      flt      pos      1
```



It consists of the following sections:

## HEADER

Shows the command line, and options selected for the XLINK command:

Target CPU type	#####
Output file or device name for the listing	#####
Absolute output filename	#####
Output file format	#####
Full list of options	#####

```

#####
#
# IAR Universal Linker Vx.xx
#
# Target CPU = xxxxx
# List file = ncr.map
# Output file 1 = aout.dxx
# Output format = debug
# Command line = -cxxxxx -rt -x -l ncr.map ncr
#
# (c) Copyright IAR Systems 1996
#
#####

```

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 88.

### Segment list (-xs)

The segment list gives the segments in the order in which they were linked:

List of segments	SEGMENT	START ADDRESS	END ADDRESS	TYPE	ORG	P/N	ALIGN
	GLOBREG	0000001C	- 00000023	rel	stc	pos	2
	WRKSEG	00000024	- 00000043	com	flt	pos	2
	IDATA0	Not in use		rel	flt	pos	1

Segment name	Segment load address range	Segment type	Allocation direction	Origin	Segment alignment
--------------	----------------------------	--------------	----------------------	--------	-------------------

## XLINK LINKER

This lists the start and end address for each segment, and the following parameters:

<i>Parameter</i>	<i>Description</i>
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 segments.
ALIGN	The segment is aligned to the next $2^{\text{ALIGN}}$ address boundary.

### Module listing (-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:

	Input file containing the module	
Module type (PROGRAM/LIBRARY) and name	FILE NAME : c:\program files\iar\ew\program\release\obj\tutor1.rxx PROGRAM MODULE, NAME : tutor1	
List of segments	SEGMENTS IN THE MODULE	
Segment name	CODE	
List of public symbols	Relative segment, address : 0000210C - 00002141 ENTRIES do_foreground_process 0000210C REF BY MODULE calls direct Not referred to main 00002120 CSTARTUP calls direct	
List of local symbols	LOCALS ?0001 00002133 ?0000 0000213F	
Segment name	CONST	
	Relative segment, address : 00002146 - 00002146 ENTRIES con_char 00002146 REF BY MODULE Not referred to	
	WRKSEG Common segment, address : 00000024 - 00000043	

For each segment the module map also lists locals and entries.

### Symbol listing (-xe)

Shows the entry name and address for each module and filename.

Module name	DEFINED ABSOLUTE ENTRIES PROGRAM MODULE, NAME : ?ABS_ENTRY_MOD	
List of symbols	ABSOLUTE ENTRIES SET_CCB3 0000FFFF CSTARTUP SET_CCB2 0000FFFF CSTARTUP SET_CCB1 000027FE CSTARTUP SET_CCB0 000020FF CSTARTUP	REF BY MODULE
	Symbol	Value



---

## XLINK LINKER

---



# XLINK OPTIONS SUMMARY

XLINK options allow you to control the operation of XLINK from the command line.

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

Output	List
#define	Include
Error	

The *Command line* and *Segment control* sections provide information about additional options which are only available in the command line version, or in an XCL command file.

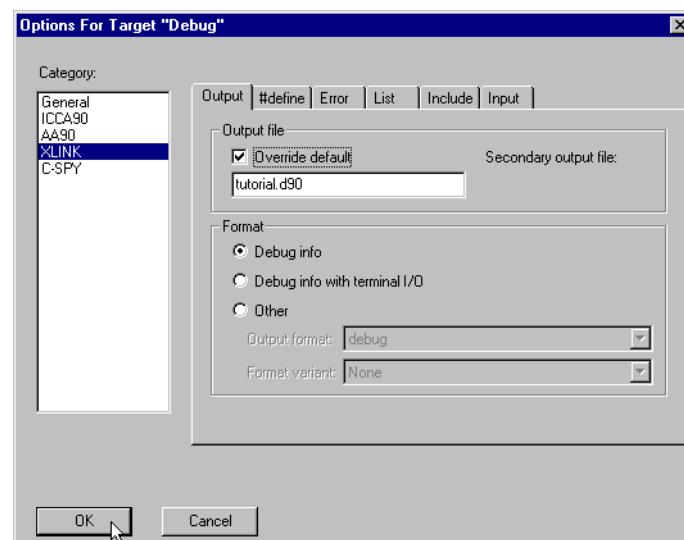
For full reference about each option refer to the following chapter, *XLINK options reference*.

## 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:



## XLINK OPTIONS SUMMARY

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



### 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 an XCL command file, and include this on the command line with `-f file` command.
- ◆ Specify the options in the `XLINK_ENVPAR` environment variable; see the *AT90S Command Line Interface Guide*.

## 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 delimiter.	Command line
-A <i>file</i> ,...	Load as PROGRAM.	Input
-B	Always generate output.	Error
-bbank_def	Define banked segments.	Segment control
-C <i>file</i> , ...	Conditionally load input files.	Command line
-ccpu	Processor type.	Command line
-Dsymbol=value	Define symbol.	#define
-d	Disable code generation.	Command line
-E <i>file</i> ,...	Inherent, no object code.	Input
-enew=old[,old] ...	Rename external symbols.	Command line
-Fformat	Output format.	Output
-f <i>file</i>	XCL filename.	Include
-G	No global type checking.	Error
-Ipathname	Include paths.	Include

## XLINK OPTIONS SUMMARY

<i>Option</i>	<i>Description</i>	<i>Section</i>
-l <i>file</i>	Generate linker listing.	List
-m	Use less host memory.	Command line
-n	Ignore local symbols.	Command line
-o <i>file</i>	Output file.	Output
-p <i>lines</i>	Lines/page.	List
-R	Disable range check.	Error
-r	Debug info.	Output
-rt	Debug info with terminal I/O.	Output
-S	Silent operation.	Command line
-t	Temporary file.	Command line
-w	Disable warnings.	Error
-x[ <i>sem</i> ]	Cross reference.	List
-Y[ <i>char</i> ]	Format variant.	Output
-Z <i>seg_def</i>	Define segments.	Segment control
-z	Segment overlap warnings.	Error



---

## **XLINK OPTIONS SUMMARY**

---





# XLINK OPTIONS REFERENCE

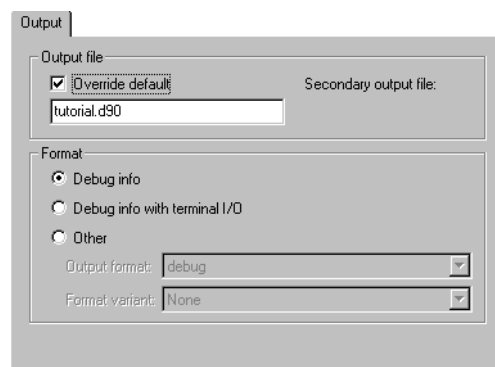
This section gives details of the XLINK options classified according to their function.

## OUTPUT

The output options are used to specify the output format and the level of debugging information.



### Embedded Workbench



### Command line

- |                               |                               |
|-------------------------------|-------------------------------|
| <code>-o <i>file</i></code>   | Output file.                  |
| <code>-r</code>               | Debug info.                   |
| <code>-rt</code>              | Debug info with terminal I/O. |
| <code>-F <i>format</i></code> | Output format.                |
| <code>-Y[<i>char</i>]</code>  | Format variant.               |



## OUTPUT

### 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 `out.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 (`.a90`) will only affect the primary output file (first format).

### DEBUG INFO (-r)

**Syntax:** `-r`

Use **Debug info** (-r) to output a file in DEBUG (AUBROF) format, with a `.d90` extension, to be used with the C-SPY debugger, or emulators which support the IAR Systems DEBUG format.

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 *AT90S Command Line Interface Guide*.

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.

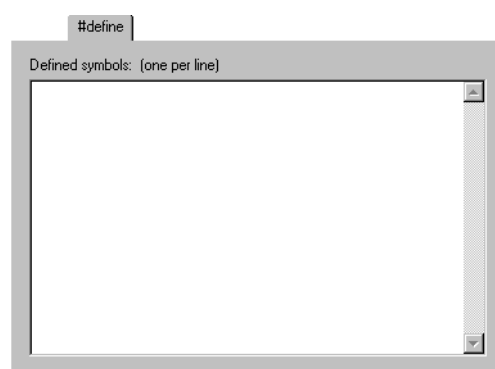


**FORMAT VARIANT (-Y)****Syntax:** `-Y[char]`

Use **Format variant** (-Y) to select enhancements available for some output formats. For more information see the chapter *XLINK output formats*.

**#define**

The **#define** option allows you to define symbols.

**Embedded Workbench****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.

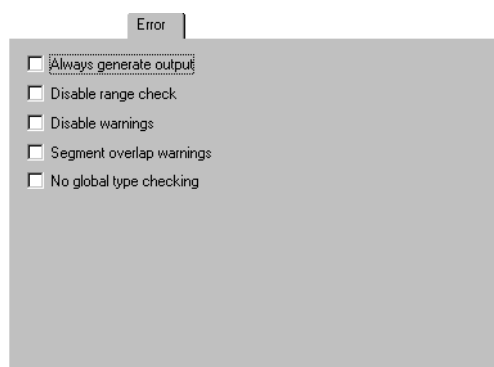
## ERROR

### ERROR

The **Error** options determine the error and warning messages generated by the XLINK Linker.



#### Embedded Workbench



#### Command line

- |    |                           |
|----|---------------------------|
| -B | Always generate output.   |
| -R | Disable range check.      |
| -W | 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 -B.

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 XLINK segment definition list (-Z command).

## DISABLE WARNINGS (-w)

**Syntax:** -w

Use **Disable warnings** (-w) to suppress all warning messages. They will, however, still be counted and shown in the linker's final statistics.

## 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.

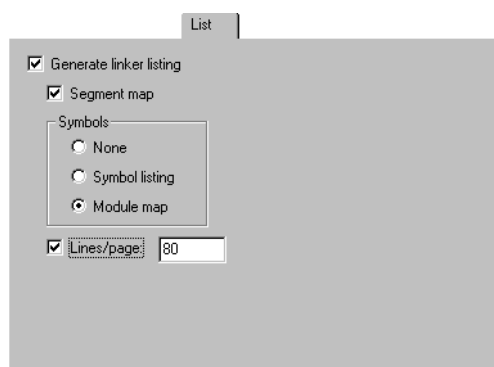
By default, XLINK performs link-time type checking between modules by comparing the external references to an entry with the PUBLIC entry (if the information exists in the object modules involved). A warning is printed if there are mismatches; otherwise the linker will continue and not abort.

## LIST

The **List** options determine the generation of an XLINK cross-reference listing.



### Embedded Workbench



### Command line

`-l file`      Generate linker listing.  
`-x[sem]`      Cross reference.  
`-plines`      Lines/page.

## GENERATE LINKER LISTING (-l)

**Syntax:**      `-l file`

Use **Generate linker listing** (-l) to generate a linker listing.



The name of the file or device to which a listing is directed. If an extension is not 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.



## CROSS REFERENCE (-x)

**Syntax:** -x[*sem*]



Use **Cross reference** (-x) to include a segment map in the XLINK listing file.



The following options are available:

<i>Option</i>	<i>Command line</i>	<i>Description</i>
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.
- ◆ A segment load map in dump order.

## 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 *AT90S Command Line Interface Guide*.

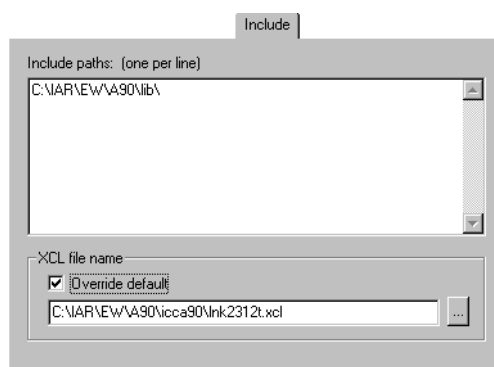
## INCLUDE

## INCLUDE

The **Include** option allows you to set the include path for linker command files, and specify the linker command file.



### Embedded Workbench



### Command line

-*Ipathname* Include paths.  
-f *file* XCL filename.

## INCLUDE PATHS (-I)

**Syntax:** -*Ipathname*

Specifies the pathname to be searched for linker command files.

By default, XLINK searches for linker command 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 *AT90S Command Line Interface Guide*.

## 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.







## INPUT

### 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.r03` 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`.



## COMMAND LINE

### 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.

## COMMAND LINE

The following additional options can be set from the command line or in XCL files:

-! <i>comment</i> -!	Comment delimiter.
-C <i>file</i> , ...	Conditionally load input files.
-ca90	Processor type.
-d	Disable code generation.
- <i>new=old</i> [, <i>old</i> ] ...	Rename external symbols.
-m	Use less host memory.
-n	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* -!

Use -! 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.

For example, to load the user-written library module putchar.r90 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.

**CONDITIONALLY LOAD INPUT FILES (-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.

**PROCESSOR TYPE (-c)**

**Syntax:**       -c a90

Use -c to set the CPU type to AT90S.

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 *AT90S Command Line Interface Guide*.

**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:** -e new=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 *AT90S Command Line Interface Guide*.

**IGNORE LOCAL SYMBOLS (-n)****Syntax:** -n

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.

Note that local symbols are only included in files if they were compiled or assembled with the appropriate option to specify this.



## SEGMENT CONTROL

---



### 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

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 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 *AT90S Command Line Interface Guide*.

---

## SEGMENT CONTROL

These options control the allocation of segments.

-b *bank\_def* Define banked segments.

-Z *seg\_def* Define segments.



### DEFINE BANKED SEGMENTS (-b)

**Syntax:** -b [*addrtype*] [(*type*)]  
*segments=first,length,increment*

where the parameters are as follows:





## SEGMENT CONTROL

<i>addrtype</i>	The type of load addresses used when dumping the code:  omitted      Logical addresses with bank number. #              Linear physical addresses. @              64180-type physical addresses.
<i>type</i>	Specifies the memory type for all segments in <i>segments</i> or <i>bankedsegments</i> , if applicable for the target processor. If omitted it defaults to UNTYPED.
<i>segments</i>	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.
<i>first</i>	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.
<i>length</i>	The length of each bank, in bytes. This is a 16-bit value.
<i>increment</i>	The incremental factor between banks, ie the number that will be added to <i>first</i> 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.

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.

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
```



DEFINE SEGMENTS (-Z)

**Syntax:** -Z [(*type*)] *segments* [=|#] [*start-end*,] ...  
[*address*]

where the parameters are as follows:

<i>type</i>	Specifies the memory type for all segments in <i>segments</i> or <i>bankedsegments</i> , if applicable for the target processor. If omitted it defaults to UNTYPED.				
<i>segments</i>	<p>A list of one or more segments to be linked, separated by commas.</p> <p>The segments are allocated in memory in the same order as they are listed. Appending +<i>nnnn</i> to a segment name increases the amount of memory that XLINK will allocate for that segment by <i>nnnn</i> bytes.</p>				
= or #	<p>Specifies how segments are allocated.</p> <table><tr><td>=</td><td>Allocates the segments so they begin at the start of the specified range (upwards allocation).</td></tr><tr><td>#</td><td>Allocates the segments so they finish at the end of the specified range (downwards allocation).</td></tr></table> <p>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.</p>	=	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).
=	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).				
<i>start, end</i>	Addresses defining a range within which the listed <i>segments</i> should be placed.				
<i>address</i>	Start address for placing any remaining segments to be allocated.				





## SEGMENT CONTROL

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 or -b (banked definition command), a warning will be displayed by the linker. However, the segment will still be allocated as if it were listed in the last segment definition; ie at the next available address.

There can be more than one -Z definition.

Additional related topics and optional forms for -Z are described below.

### Allocation segment types

The following table lists the different types of segments that can be processed by XLINK:

<i>Segment type</i>	<i>Description</i>
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.
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
BIT	Bit memory.*
CODE	Code memory.
DATA	Data memory.
FAR	Data in FAR memory. XLINK will not check access to it, and a part of a segment straddling a 64 Kbyte boundary will be moved upwards to start at the boundary.
FARC, FARCONST	Constant in FAR memory (behaves as above).
FARCODE	Code in FAR memory.
HUGE	Data in HUGE memory. No straddling problems.
HUGECONST, HUGECONST	Constant in HUGE memory.
HUGECODE	Code in HUGE memory.
NEAR	Data in NEAR memory. Accessed using 16-bit addressing, this segment can be located anywhere in the 32-bit address space.
NEARCONST, NEARCONST	Constant in NEAR memory.
NPAGE	Absolute-addressed data memory.
UNTYPED	Default type.
ZPAGE	Zero-page data memory.

\* The address of a BIT segment is specified in bits, not in bytes. BIT memory is allocated first.



## SEGMENT CONTROL

### 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 `-Z` command and absolute segments) do not overlap in memory.

### Examples

To locate SEGA at address 0, followed immediately by SEGB:

```
-Z(CODE)SEGA,SEGB=0
```

To allocate SEGA downwards from 1000H, followed by SEGB below it:

```
-Z(CODE)SEGA,SEGB#1000
```

To allocate specific areas of memory to SEGA and SEGB:

```
-Z(CODE)SEGA,SEGB=100-200,400-700,1000
```

In this example SEGA will be placed between address 100 and 200, if it fits in that amount of space. If it does not, XLINK will try the range 400–700. 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 100–200 range then XLINK will try to put SEGB there as well (following SEGA). Otherwise, SEGB will go into the 400 to 700 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).



## SEGMENT CONTROL

---





# 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:

<i>Format</i>	<i>Type</i>	<i>Extension</i>	<i>Address type</i>
AOMF8051†	binary	from CPU	N
AOMFH8†	binary	from CPU	NL
AOMF8096†	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†	binary	.dbg	NL
EXTENDED-TEKHEX†	ASCII	from CPU	NLPS
HP-CODE	binary	.x	NLPS
HP-SYMB	binary	.l	NLPS
INTEL-STANDARD	ASCII	from CPU	N
INTEL-EXTENDED	ASCII	from CPU	NLPS
MILLENIUM (Tektronix)	ASCII	from CPU	N
MOTOROLA	ASCII	from CPU	NLPS
MPDS-CODE	binary	.tsk	N
MPDS-SYMB	binary	.sym	NLPS
MSD	ASCII	.sym	N

## XLINK OUTPUT FORMATS

<i>Format</i>	<i>Type</i>	<i>Extension</i>	<i>Address type</i>
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
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
ZAX	ASCII	from CPU	NLPS

† format depends on the typing of the segments; ie the *type* field specified in the XLINK -Z option is important.

### 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:

<i>Format</i>	<i>Code format</i>	<i>Exten.</i>	<i>Symbolic format</i>	<i>Exten.</i>
DEBUG-MOTOROLA	DEBUG	.axx	MOTOROLA	.obj
DEBUG-INTEL-STD	DEBUG	.axx	INTEL-STD	.hex
DEBUG-INTEL-EXT	DEBUG	.axx	INTEL-EXT	.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
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
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

## XLINK OUTPUT FORMATS

### OUTPUT FORMAT VARIANTS

The following enhancements can be selected for the specified output formats, using the **Format variant** (-Y) option:

<i>Output format</i>	<i>Option</i>	<i>Description</i>
PENTICA-A,B,C,D and MPDS-SYMB	Y0	Symbols as <code>modules:symbolname</code> .
	Y1	Labels and lines as <code>module:symbolname</code> .
	Y2	Lines as <code>module:symbolname</code> .
AOMF8051	Y0	Extra type of information for Hitex.
INTEL-STANDARD	Y0	End only with <code>:00000001FF</code> .
	Y1	End with <code>PGMENTRY</code> , else <code>:0000001FF</code> .
MPDS-CODE	Y0	Fill with <code>0xFF</code> instead.
DEBUG, -r	Y#	Old UBROF version.
INTEL-EXTENDED	Y0	Segmented variant.
	Y1	32-bit linear variant.

Refer to the file `XLINK.DOC` for additional options that are available.





# 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.





## **XLIB LIBRARIAN**

---

- ◆ 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.

## **XLIB COMMAND SUMMARY**

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.

## PARAMETERS

The following parameters are common to many of the XLIB commands.

<i>Parameter</i>	<i>What it means</i>										
<i>objectfile</i>	File containing object modules.										
<i>start, end</i>	The first and last modules to be processed, in one of the following forms: <table><tr><td><i>n</i></td><td>The <i>n</i>th module.</td></tr><tr><td><i>\$</i></td><td>The last module.</td></tr><tr><td><i>name</i></td><td>Module <i>name</i>.</td></tr><tr><td><i>name+n</i></td><td>The module <i>n</i> modules after <i>name</i>.</td></tr><tr><td><i>\$-n</i></td><td>The module <i>n</i> modules before the last.</td></tr></table>	<i>n</i>	The <i>n</i> th module.	<i>\$</i>	The last module.	<i>name</i>	Module <i>name</i> .	<i>name+n</i>	The module <i>n</i> modules after <i>name</i> .	<i>\$-n</i>	The module <i>n</i> modules before the last.
<i>n</i>	The <i>n</i> th module.										
<i>\$</i>	The last module.										
<i>name</i>	Module <i>name</i> .										
<i>name+n</i>	The module <i>n</i> modules after <i>name</i> .										
<i>\$-n</i>	The module <i>n</i> modules before the last.										
<i>listfile</i>	File to which a listing will be sent.										
<i>source</i>	A file from which modules will be read.										
<i>destination</i>	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:

<i>Name</i>	<i>Description</i>
3	The third module.
\$	The last module.
<i>name</i> +4	The module 4 modules after <i>name</i> .
<i>name</i> -12	The module 12 modules before <i>name</i> .
\$-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:

<i>Prefix</i>	<i>Description</i>
<i>nn</i> .Pgm	A program module with relative number <i>nn</i> .
<i>nn</i> .Lib	A library module with relative number <i>nn</i> .
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

### 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%, to give library files which take up less time during the loader/linker process.

#### EXAMPLES

The following command compacts the file `maxmin.rxx`:

```
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 IAR2000:

```
DEF-CPU IAR2000 ↵
```



## DELETE-MODULES

### 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.rxx`:

```
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

### 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

---

---

### EXIT

Returns to operating system.

#### SYNTAX

EXIT

#### DESCRIPTION

Use EXIT to exit from XLIB after an interactive session.

#### EXAMPLES

To exit from XLIB:

EXIT ↵

---

## 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 module mean from *math.rxx* to *general.rxx*:

FETCH-MOD math general mean ↵

---

## HELP

Displays help information.

#### SYNTAX

HELP [*command*] [*listfile*]





## INSERT-MODULES

### 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 module *min* in the file *math.rxx*:

```
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 172.

### EXAMPLES

The following command lists all the symbols in `math.rxx`:

```
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 172.

**EXAMPLES**

The following command lists the CRCs for all modules in *math.rxx*:

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 172.

**EXAMPLES**

The following command lists the date stamps for all the modules in *math.rxx*:

LIST-DATE-STAMPS math ↵



## LIST-ENTRIES

This displays:

```
09/Jan/96      1.  Lib  max
09/Jan/96      2.  Lib  mean
09/Jan/96      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 172.

### EXAMPLES

The following command lists the entries for all the modules in `math.rxx`:

```
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*]





## LIST-MODULES

### 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 172.

### EXAMPLES

The following command lists the externals for all the modules in `math.rxx`:

```
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 172.

### EXAMPLES

The following command lists all the modules in `math.rxx`:

```
LIST-MOD math ↵
```

It produces the following output:

1. Lib max
2. Lib min
3. Lib mean



## LIST-OBJECT-CODE

### 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 *objectfile* on the *listfile* in an ASCII format.

Each symbol is identified with a prefix; see *List format*, page 172.

#### EXAMPLES

The following command lists the object code of *math.rxx* 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 172.

#### EXAMPLES

The following command lists the segments in module *mean* in the file *math.rxx*:

```
LIST-SEG math,,mean mean ↵
```

Note the use of two commas to skip the *listfile* parameter.





## MAKE-LIBRARY

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.rxx` 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.rxx` into a program module:

```
MAKE-PROG main start ↵
```



---

## ON-ERROR-EXIT

---

---

### 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 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*



## RENAME-ENTRY

### 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.rxx* from *mean* to *average*:

```
RENAME-ENTRY math mean average 2 4 ↵
```

## RENAME-EXTERNAL

Renames EXTERN symbols.

### SYNTAX

RENAME-EXTERNAL *objectfile old new [start] [end]*

### DESCRIPTION

Use RENAME-EXTERNAL to rename all occurrences of an external from *old* to *new* in the specified modules.



## RENAME-GLOBAL

---

### EXAMPLES

The following command renames all external symbols in `math.rxx` 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 or entry from *old* to *new* in the specified modules.

### EXAMPLES

The following command renames all occurrences of `mean` to `average` in `math.rxx`:

```
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 name *old*, only the first encountered is changed.

### EXAMPLES

The following example renames the module `average` to `mean` in the file `math.rxx`:

```
RENAME-MOD math average mean ↵
```





## RENAME-SEGMENT

### 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 name *old* to *new* in the specified modules.

#### EXAMPLES

The following example renames all CODE segments to ROM in the file `math.rxx`:

```
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.rxx` with modules from `newmath.rxx`:

```
REPLACE-MOD math newmath ↵
```

This displays:

```
Replacing module 'max'  
Replacing module 'mean'  
Replacing module 'min'
```



---

## REPLACE-MODULES

---





# ASSEMBLER DIAGNOSTICS

This chapter lists the errors and warnings for the AT90S 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.s90",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 191.

## 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 199.

## 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 table lists the general error messages:

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
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 encountered	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.

## ASSEMBLER DIAGNOSTICS

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
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 pre-processor line was not empty after the argument part was read.
20	Argument to #define too long (max is <max>)	



## ASSEMBLER DIAGNOSTICS

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
21	Too many formal parameters for #define (max is 127)	
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. Pre-processor limit exceeded.
28	Too many nested parameterized macros (max is <max>)	Fatal. Pre-processor limit exceeded.
29	Too deep macro nesting (max is 100)	Fatal. Pre-processor 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.

## ASSEMBLER DIAGNOSTICS

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
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.

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
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.

## ASSEMBLER DIAGNOSTICS

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
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.

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
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.

### AT90S-SPECIFIC ERROR MESSAGES

The following table lists the AT90S-specific error messages:

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
400	Absolute operand is not possible here	
401	Accessing SFR incorrectly, check read/write flags	

## ASSEMBLER DIAGNOSTICS

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
402	Accessing SFR using incorrect size	
403	Number out of range. Valid range is -128 (-0x80) to 255 (0xFF).	
404	Bit-number out of range. Valid range is 0 to 7 (0x07).	
405	Address can't be negative.	
406	Register not valid. Use register R16 - R31 here	
407	Register not valid. Use register Y or Z	
408	Port address out of range. Valid range is 0 to 63 (0x3F).	
409	Register displacement out of range. Valid range is 0 to 63 (0x3F).	
410	Address out of range. Valid range is 0 to 8388606 (0x7FFFFE).	
411	Address must be even.	
412	PC offset out of range. Valid range is -128 (-0x80) to 126 (0x7E).	
413	PC offset must be even.	
414	Address out of range. Valid range is 0 to 8190 (0x1FFE).	



<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
415	PC offset out of range. Valid range is -4096 (-0x1000) to 4094 (0x0FFE).	
416	Port address out of range. Valid range is 0 to 31 (0x1F).	
417	Number out of range. Valid range is -32 (-0x20) to 63 (0x3F).	
418	Register not valid. Use any of register R24, R26, R28 and R30 here.	

## WARNING MESSAGES

### GENERAL

The following table lists the general warning messages:

<i>No</i>	<i>Warning message</i>	<i>Suggestion</i>
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.

## ASSEMBLER DIAGNOSTICS

No	Warning message	Suggestion
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 name>', 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 '<symb>' duplicated	



## ASSEMBLER DIAGNOSTICS

<i>No</i>	<i>Warning message</i>	<i>Suggestion</i>
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.

### AT90S-SPECIFIC WARNING MESSAGES

The following table lists the AT90S-specific warning messages:

<i>No</i>	<i>Warning message</i>	<i>Suggestion</i>
400	SFR neither defined as READ nor WRITE	
401	More than one SFR size attribute defined, using default (byte)	
402	No SFR size attribute defined, using default (byte)	



ASSEMBLER DIAGNOSTICS





# 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 not be aborted but the code produced may be 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 program 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:

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
0	Format chosen cannot support banking	Format unable to support banking.
1	Corrupt file. Unexpected end of file in module <i>module (file)</i> 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.



## XLINK DIAGNOSTICS

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
3	Corrupt file. Checksum failed in module <i>module (file)</i> . Linker checksum is linkcheck, module checksum is modcheck	Linker aborts immediately. Recompile or reassemble.
4	Corrupt file. Zero length identifier encountered in module <i>module (file)</i>	Linker aborts immediately. Recompile or reassemble.
5	Address type for CPU incorrect. Error encountered in module <i>module (file)</i>	Linker aborts immediately. Check that you are using the right files and libraries.
6	Program module <i>module</i> declared twice, redeclaration in file <i>file</i> . Ignoring second module	XLINK will not produce code unless the -B option (forced dump) is used.
7	Corrupt file. Unexpected UBROF - format end of file encountered in module <i>module (file)</i>	Linker aborts immediately. Recompile or reassemble.
8	Corrupt file. Unknown or misplaced tag encountered in module <i>module (file)</i> . Tag <i>tag</i>	Linker aborts immediately. Recompile or reassemble.
9	Corrupt file. Module <i>module</i> start unexpected in file <i>file</i>	Linker aborts immediately. Recompile or reassemble.
10	Corrupt file. Segment no. <i>segno</i> declared twice in module <i>module (file)</i>	Linker aborts immediately. Recompile or reassemble.
11	Corrupt file. External no. <i>ext no</i> declared twice in module <i>module (file)</i>	Linker aborts immediately. Recompile or reassemble.

## XLINK DIAGNOSTICS

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
12	Unable to open file <i>file</i>	Linker aborts immediately. If you are using the command line check the environment variable XLINK_DFLTDIR.
13	Corrupt file. Error tag encountered in module <i>module (file)</i>	A UBROF error tag was encountered. Linker aborts immediately. Recompile or reassemble.
14	Corrupt file. Local <i>local</i> defined twice in module <i>module (file)</i>	Linker aborts immediately. Recompile or reassemble.
15	Faulty bank definition -bbank def	Incorrect syntax. Linker aborts immediately.
16	Segment <i>segment</i> is too long for segment definition	The segment defined does not fit into the memory area reserved for it. Linker aborts immediately.
17	Segment <i>segment</i> is defined twice in segment definition -Zsegdef	Linker aborts immediately.
18	Range error in module <i>module (file)</i> , segment <i>segment</i> at address <i>address</i> . Value <i>value</i> , in tag <i>tag</i> , 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 <i>module (file)</i>	Linker aborts immediately. Recompile or reassemble.
20	Undefined external referenced in module <i>module (file)</i>	Linker aborts immediately. Recompile or reassemble.
21	Segment <i>segment</i> in module <i>module</i> does not fit bank	The segment is too long. Linker aborts immediately.





## XLINK DIAGNOSTICS


No	Error message	Suggestion
22	Paragraph no. is not applicable for the wanted CPU. Tag encountered in module <i>module (file)</i>	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 <i>module (file)</i>	The tag T_REL_FI_8 or T_EXT_FI_8 is faulty. Linker aborts immediately. Recompile or reassemble.
24	Segment <i>segment</i> overlaps segment <i>segment</i>	The segments overlap each other; ie both have code on the same address.
25	Corrupt file. Unable to find module <i>module (file)</i>	A module is missing. Linker aborts immediately.
26	Segment <i>segment</i> is too long	This error should never occur unless the program is extremely large. Linker aborts immediately.
27	Entry <i>entry</i> in module <i>module (file)</i> redefined in module <i>module (file)</i>	There are two or more entries with the same name. Linker aborts immediately.
28	File <i>file</i> 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 <i>-option</i> also requires the <i>-option</i> option	Linker aborts immediately.
31	Option <i>-option</i> cannot be combined with the <i>-option</i> option	Linker aborts immediately.
32	Option <i>-option</i> cannot be combined with the <i>-option</i> option and the <i>-option</i> option	Linker aborts immediately.

## XLINK DIAGNOSTICS

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
33	Faulty value <i>val</i> (in command line or in XLINK_PAGE), (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 <i>flag</i> in cross reference option <i>option</i>	Linker aborts immediately.
36	Option <i>op</i> 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 <i>option</i> multiply defined	Linker aborts immediately.
39	Illegal character specified in option <i>op</i>	Linker aborts immediately.
40	Argument expected after option <i>op</i>	This option must be succeeded by an argument. Linker aborts immediately.
41	Unexpected '-' in option <i>op</i>	Linker aborts immediately.
42	Faulty symbol definition - <i>Dsymbol</i> definition	Incorrect syntax. Linker aborts immediately.
43	Symbol in <i>symbol</i> definition too long	The symbol name is more than 255 characters. Linker aborts immediately.
44	Faulty value <i>val</i> (in command line or in XLINK_COLUMNS), (range 80-300)	Faulty column setting. Linker aborts immediately.



## XLINK DIAGNOSTICS

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
45	Unknown CPU <i>CPU</i> encountered in command line (or in XLINK_CPU)	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  .
46	Undefined external <i>external</i> referred in module ( <i>file</i> )	Entry to external is missing.
47	Unknown format <i>format</i> encountered in command line or XLINK_FORMAT	Linker aborts immediately.
48	Faulty segment definition -Zsegdef	Incorrect syntax. Linker aborts immediately.
49	Segment name in segment definition too long	255 characters long. Linker aborts immediately.
50	Paragraph no. not allowed for this CPU, encountered in option <i>option</i>	Linker aborts immediately. Do not use paragraph no. in declarations.
51	Hexadecimal or decimal value expected in option <i>option</i>	Linker aborts immediately.
52	Overflow on value in option <i>option</i>	Linker aborts immediately.
53	Parameter exceeded 255 characters in extended command line file <i>file</i>	Linker aborts immediately.
54	Extended command line file <i>file</i> is empty	Linker aborts immediately.
55	Extended command line variable XLINK_ENVPAR is empty	Linker aborts immediately.
56	Overlapping ranges in segment definition <i>segment def</i>	Linker aborts immediately.

## XLINK DIAGNOSTICS

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
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 incompatible 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 <i>segment</i> defined in bank definition and segment definition.	Linker aborts immediately.
61	Symbol in bank definition is too long	Linker aborts immediately.
62	File <i>file</i> multiply defined in command line	Linker aborts immediately.
63	Trying to pop an empty stack in module <i>module</i> ( <i>file</i> )	Linker aborts immediately. Recompile or reassemble.
64	Module <i>module</i> ( <i>file</i> ) has not the same debug type as the other modules	Linker aborts immediately.
65	Faulty replacement definition <i>-rreplacement</i> definition	Incorrect syntax. Linker aborts immediately.
66	Function with F-index <i>index</i> has not been defined before indirect reference in module <i>module</i> ( <i>file</i> )	Indirect call to an undefined in module. Probably caused by an omitted function declaration.



## XLINK DIAGNOSTICS

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
67	Function <i>name</i> has same F-index as function- <i>name</i> , defined in module <i>module</i> ( <i>file</i> )	Probably a corrupt file. Recompile file.
68	External function <i>name</i> in module <i>module</i> ( <i>file</i> ) 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 <i>name</i> in module <i>module</i> ( <i>file</i> ) has parameters or auto variables in nondefault memory	<p>The recursively or indirectly called function <i>name</i> is using extended language memory specifiers (bit, data, idata, etc) to point to non-default memory, which is not allowed.</p> <p>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.</p>
70	Module <i>module</i> ( <i>file</i> ) has not the same memory as previously linked modules	Only modules compiled under the same memory model may be linked together.
71	Segment <i>name</i> is incorrectly defined (in a bank definition, has wrong segment type or mixed segment types)	This is usually due to misuse of a predefined segment; see the explanation of <i>name</i> in the <i>AT90S C Compiler Programming Guide</i> . It may be caused by changing the predefined linker control file.

## XLINK DIAGNOSTICS

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
72	Segment <i>name</i> must be defined in a -Z definition	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 need 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 <i>file</i>	Either the linker or your host system is corrupt, or the two are incompatible.
75	SFR address in module <i>module (file)</i> , segment <i>segment</i> at address <i>address</i> , value <i>value</i> is out of bounds	An SFR has been defined to a bad address. Change the definition.
76	Absolute segments overlap in module <i>module</i>	The linker has found two or more absolute segments in <i>module</i> overlapping each other.
77	Absolute segments in module <i>module (file)</i> overlaps absolute segment in module <i>module (file)</i>	The linker has found two or more absolute segments in <i>module (file)</i> and <i>module (file)</i> overlapping each other.
78	Absolute segment in module <i>module (file)</i> overlaps segment <i>segment</i>	The linker has found an absolute segment in <i>module (file)</i> overlapping a relocatable segment.
79	Faulty allocation definition -a <i>definition</i>	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.



## XLINK DIAGNOSTICS

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
81	Unknown flag in extended format option -Y	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.

## WARNING MESSAGES

The following table lists the linker warning messages:

<i>No</i>	<i>Warning message</i>	<i>Suggestion</i>
0	Too many warnings	Too many warnings encountered.
1	Error tag encountered in module <i>module (file)</i>	A UBROF error tag was encountered when loading file <i>file</i> . This indicates a corrupt file and will generate an error in the linking phase.
2	Symbol <i>symbol</i> is redefined in command-line	A symbol has been redefined.
3	Type conflict. Segment <i>segment</i> , in module <i>module</i> , is incompatible with earlier segment(s) of the same name	Segments of the same name should have the same type.

## XLINK DIAGNOSTICS

<i>No</i>	<i>Warning message</i>	<i>Suggestion</i>
4	Close/open conflict. Segment <i>segment</i> , in module <i>module</i> , is incompatible with earlier segment of the same name	Segments of the same name should be either open or closed.
5	Segment <i>segment</i> cannot be combined with previous segment	The segments will not be combined.
6	Type conflict for external/entry <i>entry</i> , in module <i>module</i> , against external/entry in module <i>module</i>	Entries and their corresponding externals should have the same type.
7	Module <i>module</i> declared twice, once as program and once as library. Redeclared in file <i>file</i> , ignoring library module	The program module is linked.
8	Segment <i>segment</i> undefined in segment or bank definition	Undefined segment exists. All segments should be defined in either the segment or the bank definition.
9	Ignoring redeclared program entry	Only the program entry found first is chosen.
10	No modules to link	The linker has no modules to link.
11	Module <i>module</i> declared twice as library. Redeclared in file <i>file</i> , ignoring second module	The module found first is linked.
12	Using SFB in banked segment <i>segment</i> in module <i>module</i> ( <i>file</i> )	The SFB assembler directive may not work in a banked segment.





## XLINK DIAGNOSTICS

No	Warning message	Suggestion
13	Using SFE in banked segment <i>segment</i> in module <i>module (file)</i>	The SFE assembler directive may not work in a banked segment.
14	Entry <i>entry</i> duplicated. Module <i>module (file)</i> loaded, module <i>module (file)</i> 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 <i>module (file)</i> and <i>module (file)</i>	The modules have been compiled with different options for predefined types, such as different sizes of basic C types (eg integer, double).
16	Function <i>name</i> in module <i>module (file)</i> is called from two function trees (with roots <i>name1</i> and <i>name2</i> )	The probable cause is that an interrupt function calls another function that also could be executed by a foreground program, and this could lead to execution errors.
17	Segment <i>name</i> 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 <i>segment</i> overlaps segment <i>segment</i>	The linker has found two relocatable segments overlapping each other. Check the -Z option parameters.

## XLINK DIAGNOSTICS

<i>No</i>	<i>Warning message</i>	<i>Suggestion</i>
19	Absolute segments overlaps in module <i>module (file)</i>	The linker has found two or more absolute segments in module <i>module</i> overlapping each other.
20	Absolute segment in module <i>module (file)</i> overlaps absolute segment in module <i>module (file)</i>	The linker has found two or more absolute segments in module <i>module (file)</i> and module <i>module (file)</i> overlapping each other. Change the ORG directives.
21	Absolute segment in module <i>module (file)</i> overlaps segment <i>segment</i>	The linker has found an absolute segment in module <i>module (file)</i> overlapping a relocatable segment. Change either the ORG directive or the -Z relocation command.
22	Interrupt function <i>name</i> in module <i>module (file)</i> is called from other functions	Interrupt functions may not be called.



# XLIB DIAGNOSTICS

This chapter lists the messages produced by the XLIB Librarian.

## XLIB MESSAGES

The following table lists the XLIB messages. Commands flagged as erroneous never alter object files.

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
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 <i>file</i>	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.

## XLIB DIAGNOSTICS

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
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 <i>file</i> .
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 <i>destination</i> in the INSERT-MODULES command must not be within the <i>start-end</i> 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.



## XLIB DIAGNOSTICS

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
24	Corrupt file, error occurred in <i>tag</i>	A fault is detected in the object file <i>tag</i> . Reassembly or recompilation may help. Otherwise contact your supplier.
25	<i>File</i> is write protected	The file <i>file</i> is write protected and cannot be written to.
26	Non-matching replacement module <i>name</i> found in source file	In the source file, a module name with no corresponding entry in the destination file was found.



---

## XLIB DIAGNOSTICS

---





## INDEX

### A

absolute segments, beginning	92	assembler directives ( <i>continued</i> )		assembler directives ( <i>continued</i> )	
address field, in listing	58	#undef	115	LSTOUT	110
ALIAS (assembler directive)	96	\$	120	LSTPAG	110
ALIGN (assembler directive)	91	/*	120	LSTREP	110
AND (assembler operator)	68	//	120	LSTXRF	110
ASCII character constants	54	=	96	MACRO	103
ASEG (assembler directive)	91	ALIAS	96	MODULE	88, 168
assembler		ALIGN	91	NAME	88, 168
expressions	51	ASEG	91	ORG	91
features	5	ASSIGN	96	PAGE	110
labels	53	CASEOFF	120	PAGSIZ	110
listing format	17, 57	CASEON	120	PROGRAM	88
operator format	65	COL	110	PUBLIC	90
operators	51	COMMON	91	RADIX	120
output formats	59	DB	119	REPT	103
source format	51	DD	119	REPTC	103
symbols	53	DEFINE	96	REPTI	103
assembler diagnostics	189	DP	119	RSEG	91
command line errors	190	DS	119	sfrb	96
error messages	190, 191	DW	119	SFRTYPE	96
fatal errors	190	ELSE	101	sfrw	96
internal errors	191	ELSEIF	101	STACK	91
memory overflow	190	END	88	VAR	96
warning messages	190, 199	ENDIF	101	assembler mnemonics	125
assembler directive syntax		ENDM	103	assembler operator	
comments	87	ENDMOD	88	precedence	61
conventions	86	ENDR	103	summary	61
labels	87	EQU	96	assembler operators	
parameters	87	EVEN	91	!	74
assembler directives		EXITM	103	!=	74
#define	115	EXPORT	90	%	74
#elif	115	EXTERN	90	&	68
#else	115	IF	101	&&	68
#endif	115	IMPORT	90	*	66
#error	115	LIBRARY	88	+	66
#if	115	LIMIT	96	-	67
#ifdef	115	LOCAL	103	/	67
ifndef	115	LSTCND	110	<	73
#include	115	LSTCOD	110	< <	76
		LSTEXP	110	< =	72
		LSTMAC	110	< >	74

## INDEX

### assembler operators (*continued*)

=	71
= =	71
>	71
> =	71
> >	77
^	69
	69
	75
~	68
AND	68
BITAND	68
BITNOT	68
BITOR	69
BITXOR	69
BYTE2	69
BYTE3	70
DATE	70
EQ	71
GE	71
GT	71
HIGH	72
HWRD	72
LE	72
LOW	73
LT	73
LWRD	73
MOD	74
NE	74
NOT	74
OR	75
SFB	75
SFE	76
SHL	76
SHR	77
SIZEOF	77
UGT	78
ULT	78
XOR	78
assembler option summary	31, 79

### assembler options

-B	43
-b	38
-c	43
-D	39
-d	36
-E	48
-f	48
-G	48
-I	45
-i	42
-L	42
-l	41
-M	36
-N	42
-O	49
-o	49
-p	44
-r	38
-S	49
-s	36
-T	42
-t	44
-U	44
-v	47
-w	37
-X	43
ASSIGN (assembler directive)	96
assumptions	v
<b>B</b>	
binary numbers	54
BIT (segment type)	160
BITAND (assembler operator)	68
BITNOT (assembler operator)	68
BITOR (assembler operator)	69
BITXOR (assembler operator)	69
BYTE2 (assembler operator)	69
BYTE3 (assembler operator)	70

## C

C preprocessor directives	115
C-SPY	
running	3
using	9, 25
case sensitivity, controlling	121
CASEOFF (assembler directive)	120
CASEON (assembler directive)	120
character constants	54
CODE (segment type)	160
code generation options	35
COL (assembler directive)	110
command line errors	190
command line options	48
command line options (XLINK)	153
comments, in assembler directives	87
COMMON (assembler directive)	91
COMMON (segment type)	159
common segments, beginning	93
COMPACT-FILE (XLIB	
command)	173
conventions	v

## D

DATA (segment type)	160
data definition or allocation	
directives	119
data field, in listing	58
DATE (assembler operator)	70
DB (assembler directive)	119
DD (assembler directive)	119
decimal numbers	54
DEF-CPU (XLIB command)	30
DEFINE (assembler directive)	96
#define options	39
DEFINE-CPU (XLIB command)	173
defining macros	104





## INDEX

- |                                |         |                              |         |                                              |              |
|--------------------------------|---------|------------------------------|---------|----------------------------------------------|--------------|
| DELETE-MODULES (XLIB command)  | 174     | EXTERN (assembler directive) | 90      | Inherent (XLINK option)                      | 151          |
| diagnostics                    |         |                              |         | input options (XLINK)                        | 151          |
| assembler                      | 189     | <b>F</b>                     |         | INSERT-MODULES (XLIB command)                | 177          |
| XLIB                           | 217     | false value                  | 52      | installation, requirements                   | 1            |
| XLINK                          | 203     | FAR (segment type)           | 160     | instruction mnemonics                        | 125          |
| DIRECTORY (XLIB command)       | 174     | FARC (segment type)          | 160     | integer constants                            | 53           |
| DISPLAY-OPTIONS (XLIB command) | 175     | FARCONST (segment type)      | 160     |                                              |              |
| documentation route map        | 4       | features                     |         | <b>L</b>                                     |              |
| DP (assembler directive)       | 119     | assembler                    | 5       | labels                                       |              |
| DS (assembler directive)       | 119     | XLIB Librarian               | 7       | defining and undefining                      | 116          |
| DW (assembler directive)       | 119     | XLINK Linker                 | 6       | in assembler                                 | 53           |
|                                |         | FETCH-MODULES (XLIB command) | 30, 176 | in assembler directives                      | 87           |
| <b>E</b>                       |         |                              |         | LE (assembler operator)                      | 72           |
| ECHO-INPUT (XLIB command)      | 175     | <b>G</b>                     |         | librarian                                    |              |
| ELSE (assembler directive)     | 101     | GE (assembler operator)      | 71      | command summary                              | 169          |
| ELSEIF (assembler directive)   | 101     | global value, defining       | 98      | error messages                               | 217          |
| Embedded Workbench             |         | GT (assembler operator)      | 71      | introduction                                 | 167          |
| installing                     | 2, 3    |                              |         | using                                        | 29           |
| running                        | 2       | <b>H</b>                     |         | librarian commands. <i>See</i> XLIB commands |              |
| END (assembler directive)      | 88      | HELP (XLIB command)          | 176     | libraries                                    | 132, 167     |
| ENDIF (assembler directive)    | 101     | hexadecimal numbers          | 53      | using                                        | 25           |
| ENDM (assembler directive)     | 103     | HIGH (assembler operator)    | 72      | using with assembler programs                | 168          |
| ENDMOD (assembler directive)   | 88      | HUGE (segment type)          | 160     | using with C programs                        | 167          |
| ENDR (assembler directive)     | 103     | HUGE (segment type)          | 160     | LIBRARY (assembler directive)                | 88           |
| entry list, linker             | 137     | HUGE (segment type)          | 160     | library modules, beginning                   | 89           |
| EQ (assembler operator)        | 71      | HUGE (segment type)          | 160     | library routines, creating                   | 26           |
| EQU (assembler directive)      | 96      | HUGE (segment type)          | 160     | LIMIT (assembler directive)                  | 96           |
| error messages                 |         | HUGE (segment type)          | 160     | linker                                       |              |
| assembler                      | 191     | HUGE (segment type)          | 160     | error messages                               | 203          |
| XLIB                           | 217     | HWRD (assembler operator)    | 72      | input files and modules                      | 131          |
| XLINK                          | 203     |                              |         | introduction                                 | 129          |
| error options (XLINK)          | 146     | <b>I</b>                     |         | libraries                                    | 130          |
| errors, displaying             | 117     | IF (assembler directive)     | 101     | listing format                               | 134          |
| EVEN (assembler directive)     | 91      | IMPORT (assembler directive) | 90      | object format                                | 129, 132     |
| EXIT (XLIB command)            | 30, 176 | in-line coding using macros  | 107     | output formats                               | 59, 130, 163 |
| EXITM (assembler directive)    | 103     | include options              | 45      | warning messages                             | 213          |
| EXPORT (assembler directive)   | 90      | include options (XLINK)      | 150     | linker options. <i>See</i> XLINK options     |              |
| expressions, in assembler      | 51      |                              |         |                                              |              |

## INDEX

linking	23	LSTREP (assembler directive)	110	numbers ( <i>continued</i> )	
list options	40	LSTXRF (assembler directive)	110	hexadecimal	53
list options (XLINK)	148	LT (assembler operator)	73	octal	54
LIST-ALL-SYMBOLS (XLIB command)	178	LWRD (assembler operator)	73		
LIST-CRC (XLIB command)	179			<b>O</b>	
LIST-DATE-STAMPS (XLIB command)	179	<b>M</b>		octal numbers	54
LIST-ENTRIES (XLIB command)	180	MACRO (assembler directive)	103	ON-ERROR-EXIT (XLIB command)	184
LIST-EXTERNALS (XLIB command)	180	macro processing directives	103	operators, in assembler	51
LIST-MODULES (XLIB command)	30, 181	macro-generated lines	23	options, assembler	31, 79
LIST-OBJECT-CODE (XLIB command)	182	macros		OR (assembler operator)	75
LIST-SEGMENTS (XLIB command)	182	defining	104	ORG (assembler directive)	91
listings		processing	106	output formats, assembler	59
address and data fields	58	using special characters	105	output formats, XLINK	163
assembler	57	MAKE-LIBRARY (XLIB command)	183	variants	166
conditional code and strings	111	MAKE-PROGRAM (XLIB command)	183	output options (XLINK)	143
cross reference table	112	memory overflow error	190		
formatting	112	mnemonics, assembler	125	<b>P</b>	
generated lines	112	MOD (assembler operator)	74	PAGE (assembler directive)	110
macros	112	MODULE (assembler directive)	88, 168	PAGSIZ (assembler directive)	110
source line	58	module map, linker	137	PATH variable	1
source line number	58	modules		pre-defined symbols	
turning on and off	111	terminating	89	__DATE__	55
LOCAL (assembler directive)	103	using	25	__FILE__	55
local symbols, using	100	<b>N</b>		__IAR_SYSTEMS_ASM	55
local value, defining	97	NAME (assembler directive)	88, 168	__LINE__	55
location counter	53	NE (assembler operator)	74	__TID__	55
setting	93	NEAR (segment type)	160	PROGRAM (assembler directive)	88
LOW (assembler operator)	73	NEARC (segment type)	160	program modules, beginning	88
LSTCND (assembler directive)	110	NEARCONST (segment type)	160	PROMable code, generating	25
LSTCOD (assembler directive)	110	NOT (assembler operator)	74	PUBLIC (assembler directive)	90
LSTEXP (assembler directive)	110	NPAGE (segment type)	160		
LSTMAC (assembler directive)	110	numbers		<b>Q</b>	
LSTOUT (assembler directive)	110	binary	54	QUIT (XLIB command)	184
LSTPAG (assembler directive)	110	decimal	54		



## INDEX

### R

RADIX (assembler directive)	120
RELATIVE (segment type)	159
relocatable expressions, using symbols in	52
relocatable segments, beginning	93
REMARK (XLIB command)	184
RENAME-ENTRY (XLIB command)	185
RENAME-EXTERNAL (XLIB command)	185
RENAME-GLOBAL (XLIB command)	186
RENAME-MODULE (XLIB command)	186
RENAME-SEGMENT (XLIB command)	187
repeating statements	106
REPLACE-MODULES (XLIB command)	187
REPT (assembler directive)	103
REPTC (assembler directive)	103
REPTI (assembler directive)	103
requirements	1
route map	4
RSEG (assembler directive)	91
running	
a program	24
C-SPY	3
Embedded Workbench	2

### S

segment control directives	91
segment control options (XLINK)	156
segment location	133
segment map, linker	135
segment types	
BIT	160
CODE	160

#### segment types (*continued*)

COMMON	159
DATA	160
FAR	160
FARC	160
FARCONST	160
HUGE	160
HUGECL	160
HUGECLCODE	160
HUGECONST	160
NEAR	160
NEARCL	160
NEARCONST	160
NPAGE	160
RELATIVE	159
STACK	159
UNTYPED	160
ZPAGE	160
SFB (assembler operator)	75
SFE (assembler operator)	76
sfrb (assembler directive)	96
SFRTYPE (assembler directive)	96
sfrw (assembler directive)	96
SHL (assembler operator)	76
SHR (assembler operator)	77
SIZEOF (assembler operator)	77
source files, including	117, 122
source format, assembler	51
source line, in listing	58
source line number, in listing	58
STACK (assembler directive)	91
STACK (segment type)	159
stack segments, beginning	93
symbol and cross reference table	58
symbols	
exporting to other modules	90
importing	91
in assembler	53
in relocatable expressions	52
pre-defined	55
redefining	98

### T

target options	46
temporary value, defining	97
true value	52
tutorial	9
tutorial programs	
dio	21
first	12
main	26
shifts	27

### U

UGT (assembler operator)	78
ULT (assembler operator)	78
UNTYPED (segment type)	160

### V

value assignment directives	96
VAR (assembler directive)	96

### W

warning messages, XLINK	213
Workbench	
installing	3
running	2

### X

XLIB commands	171
COMPACT-FILE	173
DEFINE-CPU	30, 173
DELETE-MODULES	174
DIRECTORY	174
DISPLAY-OPTIONS	175
ECHO-INPUT	175





## INDEX

-I (assembler option)	45	-S (XLINK option)	156	< = (assembler operator)	72
-i (assembler option)	42	-T (assembler option)	42	< > (assembler operator)	74
-L (assembler option)	42	-t (assembler option)	44	= (assembler directive)	96
-l (assembler option)	41	-t (XLINK option)	156	= (assembler operator)	71
-l (XLINK option)	148, 150	-U (assembler option)	44	= = (assembler operator)	71
-M (assembler option)	36	-v (assembler option)	47	> (assembler operator)	71
-m (XLINK option)	155	-w (assembler option)	37	> = (assembler operator)	71
-N (assembler option)	42	-w (XLINK option)	147	> > (assembler operator)	77
-n (XLINK option)	155	-X (assembler option)	43	^ (assembler operator)	69
-O (assembler option)	49	-x (XLINK option)	135, 137, 149	__DATE__ (pre-defined symbol)	55
-o (assembler option)	49	-Y (XLINK option)	145	__FILE__ (pre-defined symbol)	55
-o (XLINK option)	144	-Z (XLINK option)	158	__IAR_SYSTEMS_ASM	
-p (assembler option)	44	-z (XLINK option)	147	(pre-defined symbol)	55
-p (XLINK option)	149	/ (assembler operator)	67	__LINE__ (pre-defined symbol)	55
-r (assembler option)	38	/* (assembler directive)	120	__TID__ (pre-defined symbol)	55
-R (XLINK option)	147	// (assembler directive)	120	(assembler operator)	69
-r (XLINK option)	144	< (assembler operator)	73	(assembler operator)	75
-S (assembler option)	49	< < (assembler operator)	76	~ (assembler operator)	68
-s (assembler option)	36				



## INDEX

---

