

80C196 v6.1

ASSEMBLER USER'S GUIDE



A publication of
TASKING
Documentation Department
Copyright © 1999 TASKING, Inc.

All rights reserved. Reproduction in whole or part is prohibited
without the written consent of the copyright owner.

The following trademarks are acknowledged:

FLEXlm is a registered trademark of Globetrotter Software, Inc.
HP and HP-UX are trademarks of Hewlett-Packard Co.
Intel, MCS and ICE are trademarks of Intel Corporation.
MS-DOS and Windows are registered trademarks of Microsoft Corporation.
SUN is a trademark of Sun Microsystems, Inc.
UNIX is a registered trademark of X/Open Company, Ltd.

All other trademarks are property of their respective owners.

Data subject to alteration without notice.

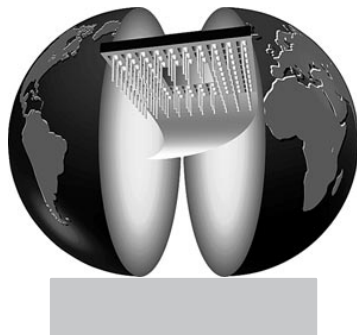
E-mail: support@tasking.com
WWW: <http://www.tasking.com>

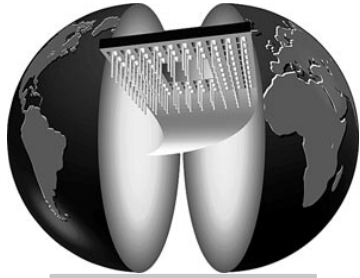
The information in this document has been carefully reviewed and is believed to be accurate and reliable. However, TASKING assumes no liabilities for inaccuracies in this document. Furthermore, the delivery of this information does not convey to the recipient any license to use or copy the software or documentation, except as provided in an executed license agreement covering the software and documentation.

TASKING reserves the right to change specifications embodied in this document without prior notice.

CONTENTS

TABLE OF CONTENTS





CONTENTS

SOFTWARE INSTALLATION **1-1**

1.1	Introduction	1-3
1.2	Installation for Windows	1-3
1.2.1	Setting the Environment	1-4
1.3	Installation for UNIX Hosts	1-5
1.3.1	Setting the Environment	1-8

OVERVIEW **2-1**

2.1	ASM196 and Software Development	2-3
2.1.1	Keeping Track of Files	2-5
2.1.2	Macro Processing Language	2-6
2.2	About This Manual	2-6
2.3	Conventions	2-6
2.4	Customer Support	2-7
2.4.1	If You Have a Problem Using the Software	2-7

ASSEMBLER INVOCATION **3-1**

3.1	Invocation Line	3-3
3.2	Assembler Options	3-4
3.3	Assembler Controls	3-7
3.3.1	Primary and General Controls	3-8
3.3.2	Control Processing	3-10
3.4	Output Object File	3-11
3.5	Listing File	3-11
3.5.1	Header and Introductory Lines	3-15
3.5.2	Source Lines	3-16
3.5.3	Error Lines	3-18
3.5.4	Symbol Table	3-18
3.6	Errorprint File	3-22
3.7	Automatic Assembler Invocation	3-22
3.7.1	Using Make Utility MK196	3-22
3.7.2	Batch Files	3-23
3.7.3	Log File	3-24



ASSEMBLER CONTROLS	4-1
---------------------------	------------

ASSEMBLY LANGUAGE	5-1
--------------------------	------------

5.1	Introduction	5-3
5.2	Assembly Language Components	5-3
5.2.1	Character Set	5-3
5.2.2	Numbers	5-3
5.2.3	Long Constants	5-4
5.2.4	Floating Point Numbers	5-4
5.2.5	Delimiters	5-7
5.2.6	Reserved Words	5-8
5.2.7	Predefined Macros	5-8
5.2.8	Symbols	5-9
5.2.9	Assembler-generated Symbols	5-10
5.2.10	Generic Instructions	5-11
5.2.11	Additional Mnemonics	5-13
5.2.12	Mixed Addressing Modes	5-14
5.2.13	Location Counter	5-15
5.2.14	Strings	5-15
5.3	Expressions and Basic Operands	5-16
5.3.1	Basic Operands	5-17
5.3.2	Attributes of Expression Operands	5-18
5.3.3	Absolute Expressions	5-22
5.3.4	Relocatable Expressions	5-24
5.3.5	External Bit Numbers	5-25
5.4	Statement Format	5-26
5.4.1	Additional Statement Rules	5-27
5.5	Program Format	5-27
5.6	Segments	5-30
5.6.1	Register Segment (Overlayable and Non-overlayable)	5-30
5.6.2	Data Segment (Overlayable and Non-overlayable)	5-30
5.6.3	Stack Segment	5-31
5.6.4	User Defined Stack Segment	5-31
5.6.5	Code Segment	5-32

5.6.6	Constant Segment	5-32
5.7	Absolute and Relocatable Segments	5-32
5.8	Stack Overflow	5-33

ASSEMBLER DIRECTIVES 6-1

MACRO PROCESSING 7-1

7.1	Introduction	7-3
7.2	The Advantages of Using Macros	7-3
7.2.1	An Example of Macro Use	7-4
7.3	Macros and Routines	7-5
7.4	Macro Directives and Macro Calls	7-6
7.4.1	Macro Definition	7-6
7.5	Macro Directives	7-7
7.6	Empty Macro Arguments	7-23
7.7	NARG Symbol	7-23
7.8	Special Macro Operators	7-24
7.9	Nesting Macro Definitions	7-26
7.10	Macro Calls	7-26
7.10.1	Nested Macro Calls	7-27
7.11	Macro Expansion	7-27
7.12	Null Macros	7-29
7.13	Sample Macros	7-30

MESSAGES AND ERROR RECOVERY 8-1

8.1	Console Output	8-3
8.1.1	Sign-on Message	8-3
8.1.2	Error Messages	8-3
8.1.3	Sign-off Message	8-3
8.2	Error Messages and Recovery	8-4
8.2.1	Fatal Error Messages	8-4
8.2.1.1	ASM196 Error Messages	8-4



8.2.1.2	Argument Error Messages	8-6
8.2.1.3	Memory Error Messages	8-7
8.2.1.4	I/O Error Messages	8-7
8.2.2	Warning Messages	8-8
8.2.3	Source File Error Messages	8-9

FLEXIBLE LICENSE MANAGER (FLEXLM) **A-1**

1	Introduction	A-3
2	License Administration	A-3
2.1	Overview	A-3
2.2	Providing For Uninterrupted FLEXlm Operation	A-5
2.3	Daemon Options File	A-6
2.4	License Administration Tools	A-8
3	FLEXlm User Commands	A-11
4	The Daemon Log File	A-17
4.1	Informational Messages	A-18
4.2	Configuration Problem Messages	A-21
4.3	Daemon Software Error Messages	A-23
5	FLEXlm License Errors	A-25

ASSEMBLER DIRECTIVES OVERVIEW **B-1**

ASSEMBLER CONTROLS TABLE **C-1**

ASM196 RESERVED WORDS **D-1**

INDEX

RELEASE NOTE

MANUAL PURPOSE AND STRUCTURE

PURPOSE

This manual is aimed at users of the TASKING ASM196 assembler. It assumes that you are familiar with the 80C196 architecture and the C programming language.

INSTALLING THE ASSEMBLER

To install the ASM196 assembler, see Chapter 1 *Software Installation*. To automate the assembling and linking processes, configure the environment variables listed in the Software Installation chapter, and see Chapter 3 for instructions on how to create a batch or command file.

RUNNING THE ASSEMBLER

To learn how to invoke the assembler, read Chapter 3. To learn how each control affects the assembly process, read Chapter 4. Chapter 8 provides information you can use to interpret an assembler error, including possible causes and suggested actions to recover from the error.

PROGRAMMING IN ASM196

To learn about the basic elements of the assembly language and the ASM196 instruction set, read Chapters 5 and 6. Chapter 7 shows you how to include assembler macros in your program.

MANUAL STRUCTURE

Related Publications

Conventions Used In This Manual

1. Software Installation
Describes the installation of the ASM196 assembler.
2. Overview
Summarizes the functions of the assembler and the utilities.
3. Assembler Invocation
Deals with assembler invocation, output files and explains how the controls affect the assembly process. This chapter also describes how to automate the assembly process.
4. Assembler Controls
Contains an alphabetical list of all assembler controls.
5. Assembly Language
Provides a detailed presentation of the elements of assembly language and statement syntax. It also describes the program format and the different segment types used by the assembler.
6. Assembler Directives
Describes each assembler directive in detail in alphabetical order.
7. Macro Processing
Describes the use of macros and conditional assembly.
8. Messages and Error Recovery
Describes the error/warning messages of the assembler.

APPENDICES

- A. Flexible License Manager (FLEXlm)
Contains a description of the Flexible License Manager.
- B. Assembler Directives Overview
Contains a short description of all assembler directives.
- C. Assembler Controls Table
Contains an overview of all assembler controls.

D. ASM196 Reserved Words

Contains a list of words that are reserved for the assembler. This includes assembler directives, macro directives and instructions.

INDEX

RELEASE NOTE

RELATED PUBLICATIONS

TASKING publications

- 80C196 C Compiler User's Guide [TASKING, MA006022]
- 80C196 Assembler User's Guide [TASKING, MA006020]
- 80C196 Utilities User's Guide [TASKING, MA006009]

Intel publications

- Embedded Microcontrollers and Processors Handbook [270645]
- 8XC196xx User's Manuals

CONVENTIONS USED IN THIS MANUAL

The notation used to describe the format of call lines is given below:

{} Items shown inside curly braces enclose a list from which you must choose an item.

[] Items shown inside square brackets enclose items that are optional.

| The vertical bar separates items in a list. It can be read as OR.

italics Items shown in italic letters mean that you have to substitute the item. If italic items are inside square brackets, they are optional. For example:

filename

means: type the name of your file in place of the word *filename*.

... An ellipsis indicates that you can repeat the preceding item zero or more times.

[,...] You can repeat the preceding item, but you must separate each repetition by a comma.

screen font Represents input examples, keywords, filenames, controls and screen output examples.

bold font Represents a command name, an option or a complete command line which you can enter.

For example

command [*option*]*... filename*

This line could be written in plain English as: execute the command *command* with the optional options *option* and with the file *filename*.

Illustrations

The following illustrations are used in this manual:



This is a note. It gives you extra information.



This is a warning. Read the information carefully.



This illustration indicates actions you can perform with the mouse.



This illustration indicates keyboard input.



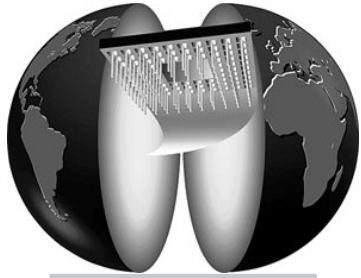
This illustration can be read as “See also”. It contains a reference to another command, option or section.

CHAPTER

1

SOFTWARE INSTALLATION





1

CHAPTER

1.1 INTRODUCTION

This chapter describes how you can install the TASKING 80C196 assembler on Windows 95/NT and several UNIX hosts.

1.2 INSTALLATION FOR WINDOWS

Step 1

Start Windows 95/98 or NT, if you have not already done so.

Step 2

Insert the CD-ROM into the CD-ROM drive.

If the `Auto insert notification` option is enabled for your CD-ROM drive, the TASKING Welcome dialog box appears. Now skip to Step 5.

Step 3

Select the `Start` button and select the `Run . . .` menu item.

Step 4

On the command line type:

d:\setup

(substitute the correct drive letter if necessary) and press the **<Return>** or **<Enter>** key or click on the OK button.

The TASKING Welcome dialog box appears.

Step 5

Select a product to install and click on `Install a Product`.

Step 6

Follow the instructions that appear on your screen.



You can find your serial number on the *Certificate of Authenticity*, delivered with the product.

1.2.1 SETTING THE ENVIRONMENT

Environment variables are definitions that provide direction to a program while it is executing. The assembler uses these definitions to establish directory paths and certain operating parameters. The values can be defined in the `autoexec.bat` file or at the command prompt before invoking the assembler.

PATH

PATH is recognized by DOS/Windows as a list of pathnames of directories containing executable or batch files. If one of the pathnames in this list specifies the directory containing the ASM196 assembler, you need not retype the full pathname each time you invoke the assembler. If you installed the software under `C:\C196`, you can include the executable directory `C:\C196\BIN` in your search path. Your PC literature explains how to define the PATH environment variable.



In EDE, select the `EDE | Directories...` menu item. Add one or more executable directory paths to the `Executable Files Path` field.

TMPDIR

The assembler creates temporary work files, which it normally deletes when assembly is complete. If the assembly is interrupted, for example, by your host system losing power, the work files remain. The TMPDIR environment variable specifies the directory where the assembler is to put these temporary files. If TMPDIR is not defined or is empty, the assembler uses your current working directory for the temporary files. For example, setting TMPDIR as follows causes the assembler to use the `c:\tmp` directory for temporary work files:

```
set TMPDIR=c:\tmp
```

1.3 INSTALLATION FOR UNIX HOSTS

Step 1

Login as a user.

Be sure you have read, write and execute permissions in the installation directory. Otherwise, login as root.

Step 2

If you are a first time user decide where you want to install the product (By default it will be installed in `/usr/local`).

Step 3

For CD-ROM install: insert the CD-ROM into the CD-ROM drive. Mount the CD-ROM on a directory, for example `/cdrom`. Be sure to use a ISO 9660 file system with Rock Ridge extensions enabled. See the manual page for **mount** on your UNIX platform for details.

Or:

For tape install: insert the tape into the tape unit and create a directory where the contents of the tape can be copied to. Consider the created directory as a temporary workspace that can be deleted after installation has succeeded. For example:

```
mkdir /tmp/instdir
```

Step 4

For CD-ROM install: go to the directory on which the CD-ROM is mounted:

```
cd /cdrom
```

For tape install: copy the contents of the tape to the temporary workspace using the following commands:

```
cd /tmp/instdir  
tar xvf /dev/tape
```

where *tape* is the name of your tape device.



If you have received a tape with more than one product, use the non-rewinding device for installing the products.

For **HP** *tape* is usually the name `update.src`.

Step 5

For tape install: remove the installation tape from the device.

Step 6

Run the installation script:

```
sh install
```

and follow the instructions appearing on your screen.

First a question appears about where to install the software. The default answer is `/usr/local`. On certain sites you may want to select another location.

On some hosts the installation script asks if you want to install SW000098, the Flexible License Manager (FLEXlm). If you do not already have FLEXlm on your system, you must install it; otherwise the product will not work on those hosts. See the *Flexible License Manager (FLEXlm)* appendix for more information.

If the script detects that the software has been installed before, the following messages appear on the screen:

```
*** WARNING ***
SW006020 xxxx.xxxx already installed.
Do you want to REINSTALL? [y,n]
```

Answering **n** (no) to this question causes installation to abort and the following message being displayed:

```
=> Installation stopped on user request <=
```

Answering **y** (yes) to this question causes installation to continue. And the final message will be:

```
Installation of SW006020 xxxx.xxxx completed.
```

Step 7

For tape install: remove the temporary installation directory with the following commands:

```
cd /tmp
rm -rf instdir
```

Step 8

For hosts that need the FLEXlm license manager, each user must define an environment variable, **LM_LICENSE_FILE**, to identify the location of the license data file. If the license file is present on the hosts on which the installed product will be used, you must set **LM_LICENSE_FILE** to the pathname of the license file if it differs from the default:

```
/usr/local/flexlm/licenses/license.dat
```

If you have more than one product using the FLEXlm license manager you can specify multiple license files to the **LM_LICENSE_FILE** environment variable by separating each pathname (*lfpath*) with a ':' :

```
setenv LM_LICENSE_FILE lfpath[:lfpath]...
```

If the license file is not available on these hosts, you must set **LM_LICENSE_FILE** to *port@host*; where *host* is the host name of the system which runs the FLEXlm license manager and *port* is the TCP/IP port number on which the license manager listens.

To obtain the port number, look in the license file at *host* for a line starting with "SERVER". The fourth field on this line specifies the TCP/IP port number on which the license server listens. For example:

```
setenv LM_LICENSE_FILE 7594@elliott
```

See the *Flexible License Manager (FLEXlm)* appendix for detailed information.

Step 9

Logout.

License Manager (on some hosts)

If your product has the FLEXlm License Manager the following two files are present:

```
c196/flexlm/
    license.dat      Tasking
```

The file `license.dat` is a template license file for this product. The file `Tasking` is the license daemon for TASKING products. Refer to the *Flexible License Manager (FLEXlm)* appendix for detailed information regarding license management.

1.3.1 SETTING THE ENVIRONMENT

UNIX and the ASM196 assembler recognize several environment variables that you can use to reduce the amount of typing required for an assembler invocation. These environment variables are as follows:

PATH

PATH is recognized by UNIX as a list of pathnames of directories containing executable or scripts. If one of the pathnames in this list specifies the directory containing the ASM196 assembler, you need not retype the full pathname each time you invoke the assembler. Your UNIX literature explains how to define the PATH environment variable.

TMPDIR

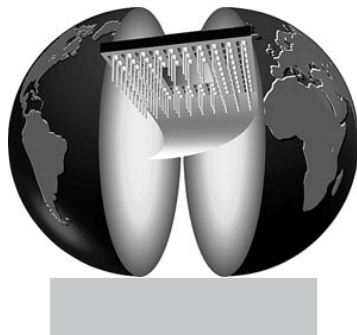
The assembler creates temporary work files, which it normally deletes when assembly is complete. If the assembly is interrupted, for example, by your host system losing power, the work files remain. The TMPDIR environment variable specifies the directory where the assembler is to put these temporary files. If TMPDIR is not defined or is empty, the assembler uses the /tmp directory for the temporary files. For example, setting TMPDIR as follows causes the assembler to use the /tmp directory for temporary work files:

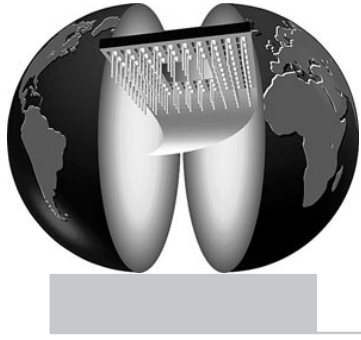
```
setenv TMPDIR /tmp
```

CHAPTER

2

OVERVIEW





2

CHAPTER

This chapter of the *80C196 Assembler User's Guide* summarizes the functions of the assembler and the utilities, and explains the purpose of this manual. This chapter also directs you to sources of detailed and supplemental information.

2.1 ASM196 AND SOFTWARE DEVELOPMENT

The ASM196 assembler translates ASM196 assembly language source code into object code. The assembler performs two passes. During the first pass, the assembler scans your program to determine the values of user-defined symbols. During the second pass, the assembler produces an object file and a listing showing the results of the assembly.

The object file contains machine language and data that you can load into memory for execution or interpretation. This file also contains control information governing the loading process. The object file can be in absolute or relocatable format. You can load an absolute object file without passing it through the RL196 linker. However, you must run a relocatable object file through the RL196 linker so that the linker can locate the relocatable segment to an absolute memory address and resolve its external references.



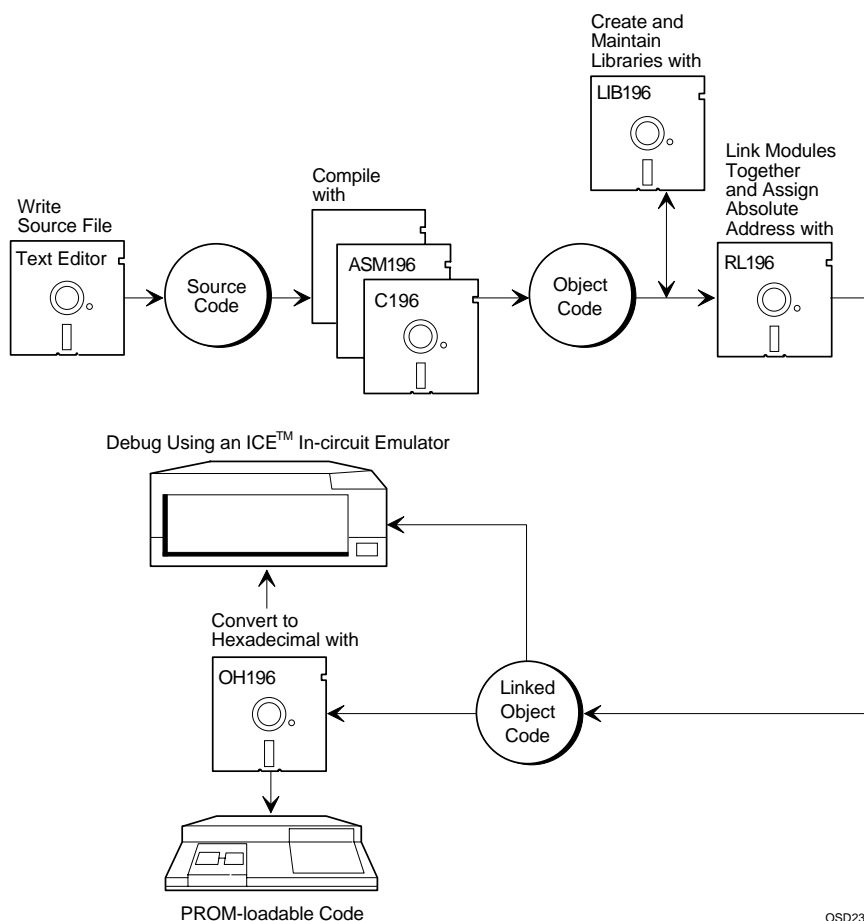
See Chapter 3 for more information about absolute and relocatable object files.

The assembler produces a listing file that provides a record of both the source program and the object code, diagnostic messages for syntax and other coding errors, and a symbol table unless you specify otherwise. The symbol table lists all defined symbols and their attributes. See Chapter 3 for the format of the listing file.

After all modules of the program are assembled, RL196 links all of the object files to form an executable file. RL196 assigns absolute memory locations to all the relocatable segments and resolves all references between modules. RL196 produces either an absolute or a quasi-absolute object module file. The quasi-absolute file must be used as further input to an RL196 process. RL196 also outputs a map file with a .m96 extension showing the results of the link/relocate process. For more information on how to run the linker, see the *80C196 Utilities User's Guide* listed in *Related Publications*.

Your target processor can execute the absolute object code produced by RL196 without further modification. However, certain 80C196 development products require the hexadecimal object code format. For use with these products, you must run the absolute object file through the object-to-hexadecimal conversion program called OH196. See the *80C196 Utilities User's Guide*, listed in *Related Publications*, for instructions on how to use OH196.

Figure 2-1 shows the software development process.



OSD238

Figure 2-1: 80C196 program development process

2.1.1 KEEPING TRACK OF FILES

We suggest to use the following filename extensions. This naming convention is not required, but it allows utilities (like **mk196**) to execute so-called 'suffix rules'. Note that all names and extensions are in lower case, because on UNIX systems it is case sensitive.

Extension	Description
.c .c96	C file (.c is preferred, no extension is forced or assumed by the compiler).
.h .h96	Include files for C (.h is preferred, the compiler does not look for .h96 by itself).
.a96 .asm .src	Assembly source files (mk196 uses .a96).
.inc	Include file for assembly.
.cmd	Command file for asm196 or c196 .
.obj	OMF96 object file produced by c196 or asm196 .
.lst	LIST files from c196 or asm196 .
.lnk	Linker command control file.
.out	File containing linked object with unresolved externals.
.abs	File containing absolute object of application, no remaining unresolved externals (default output file of rl196).
.m96	Linker MAP file.
.mak	For Makefiles other than 'Makefile' or 'makefile'.
.hex	Hexadecimal output file by oh196 .

Table 2-1: Filename extensions

Programmers who at present work on MS-DOS but are thinking of future migration to other platforms (UNIX, Windows NT, etc.) are advised to use lower case characters and forward slashes where possible. This will smoothen the future transition and it will not hurt right now. All the tools are able to find files if forward slashes are used. (Note however that MS-DOS still does not like you to say: `c:/c196/bin/asm196`)

Use caution with the extension `.tmp`, as some operating-system utilities create temporary files with this extension. If you have files with the same name and extension as these, your files are overwritten when the utility creates its temporary file.

2.1.2 MACRO PROCESSING LANGUAGE

ASM196 includes a macro processing utility that allows you to substitute one set of parameters for another when similar code sequences are used several times. Using macros eliminates the tedium of rewriting these sequences, minimizes the probability of entry errors, and increases efficiency by reducing the duplication of effort by programmers.



See Chapter 7 for detailed information on using the macro processing language.

2.2 ABOUT THIS MANUAL

This manual helps you design software based on the 80C196 family of microcontrollers (8096-90, 8096BH, 80C196CA, 80C196CB, 80C196EA, 80C196EC, 80C196JQ, 80C196JR, 80C196JS, 80C196JT, 80C196JV, 80C196KB, 80C196KC, 80C196KD, 80C196KL, 80C196KQ, 80C196KR, 80C196KS, 80C196KT, 80C196LB, 80C196MC, 80C196MD, 80C196MH, 80C196NP, 80C196NT, 80C196NU, etc.) using the ASM196 assembly language. This manual provides examples of assembly language source code and a reference for assembly language directives and instructions. Except when otherwise specified, information in this manual applies to all microcontrollers in the 80C196 family. This manual also helps high-level language programmers to interpret the assembler's output.

To effectively use the ASM196 assembler, you must be familiar with the 80C196 architecture, assembly language programming, high-level language programming, and the software development process.

2.3 CONVENTIONS

This manual follows the notational conventions listed at the beginning of this manual, in addition to the following conventions:

- | | |
|-------------|--|
| [address] | Regular square brackets ([]) in assembly listings denote indirect or based addressing. Type the square brackets as shown in the manual. |
|-------------|--|

2.4 CUSTOMER SUPPORT

The 80C196 software is under warranty. During the warranty period you are entitled to the following:

- Free replacement of any defective media upon notification in writing of the defect and product information.
- Telephone consultation and bug reporting.
- Our best efforts to replace or repair any software that does not meet the specification described in the 80C196 documentation.

TASKING offers various support contracts that provide benefits as free product updates, reduced rate upgrades, and telephone support. Contact your local TASKING sales representative, for information about support contracts and standard warranties. You will find the addresses and telephone numbers in the "Read This First" Envelop included with this package.

2.4.1 IF YOU HAVE A PROBLEM USING THE SOFTWARE

To help expedite your calls, please have the following information available when you contact us for help.

- The serial number of your software distribution. This number is printed on the label of the tape, cassette, or first floppy of your software distribution. In addition, you may be able to obtain the serial number by running ASM196 with option **-V**, you may wish to record the serial number here:

Product: _____

Serial:

- The product name, including host, target processor, and release number.
- The exact command line that you used to invoke our tools when you encountered the problem. Please include all switches.
- The exact error message that was printed. A screen dump will often make this easy to record, and can provide very useful information.
- Any additional information that may be useful in helping to define the problem.



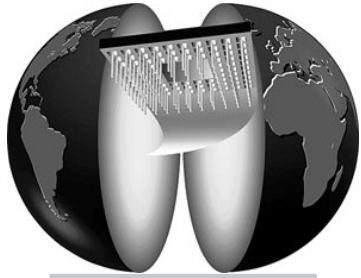
OVERVIEW

CHAPTER

ASSEMBLER INVOCATION

3





3

CHAPTER

This chapter shows you how to invoke the ASM196 assembler and explains how the controls affect the assembly process. This chapter also describes the outputs of the assembler. If you want to automate the assembly process, Section 3.7 discusses the use of batch and command files.

3.1 INVOCATION LINE

The syntax of the invocation line is:

```
[path]asm196 [options] source_file [controls]
```

Where:

- path* is the name of the drive plus directory names indicating the location of the assembler.
- options* is an optional sequence of assembler options. The options are described in detail in Section 3.2.
- source_file* is the name of the ASM196 source file including its path.
- controls* is an optional sequence of assembler controls. The controls are described in detail in Chapter 4.

The ASM196 assembler normally produces two output files. These output files reside on the same device and have the same name as the source file but with the following extensions:

- .lst** This extension indicates that the output file contains a listing of your source code. To specify a different listing filename, use the `print` control. For more information about the listing file, see Section 3.5.
- .obj** This extension indicates that the output file contains the output object code. To specify a different object code filename, use the `object` control.

To illustrate the construction and function of the invocation line, consider the following example:

```
asm196 prog.src
```

ASM196 assembles the source file `prog.src`. Since no controls have been specified, the listing file has the name `prog.lst` and the object file has the name `prog.obj`, the default conditions. Both the listing and the object file reside on the same device and directory as `prog.src`. Default conditions are also in effect for all the other control options.

On UNIX you can continue the invocation line on one or more additional lines by entering the backslash (\) before you enter the carriage return or line feed character. Note that with some UNIX shells arguments containing special characters (such as '(' and '?') must be enclosed with "" or escaped. Example:

```
asm196 prog.src \
title\('Project Review'\)
```

The assembler can detect two types of errors: console errors and source file errors. Console errors are fatal and terminate the assembly process. Source file errors, on the other hand, do not terminate the assembly process. See Chapter 8 for the possible causes of errors and hints on how to fix them.

3.2 ASSEMBLER OPTIONS

You can use assembler options only on the invocation line, as described in Section 3.1. The format for a single option is:

-option_name [[*=* | *:* | *space*] *argument*]

Where:

- (minus sign) must be prefixed to every option name.
- option_name* is the name of the option. This name is case sensitive.
- =*, *:*, or *space* are used to separate the option name from the argument.
- argument* the argument for an option. This is optional.

For example, the following invocation line sets up a listing file with a pagewidth of 80 characters per line and a pagelength of 72 lines per page to be titled `section eight` and to be started on a new page because of the `-e` option.

```
-pw 80 -pl=72 -t:'section eight' -e
```

Some options can toggle conditions on or off. To turn a condition off, you have to append a minus sign to the option name, as in **-p** versus **-p-** or **-x** versus **-x-**.

Most of the options can also be set using controls. However, the **-case** option for example has no equivalent control. Also the **error** control has no equivalent option. When equivalent controls and options exist, they default to the same value.

The following table is a list of all options and their equivalent control (if present). For a detailed description of each option that has an equivalent control, refer to the description of the control in Chapter 4, *Assembler Controls Reference*. Options that have no equivalent control are described below.

Option	Control	Description
-?		Display invocation syntax
-C[-]	[no]cond	Include conditionally-skipped source lines in list file
-D	set	Define a symbol
-G[-]	[no]gen	Include macro expansion lines in list file
-I[-]	[no]searchinclude	Specify alternative search path for include files
-O[-]	[no]optimize	Turn on/off optimization
-U	reset	Undefine a symbol
-R	restore	Restore settings of some general control
-S	save	Save settings of some general control
-V		Display version header only
-c[-]	[no]cmain	Allow public symbol 'main' in source file
-ca[-]	[no]copyattr	Use segment type of current segment
-case[-]	[no]case	Assembler works case sensitive
-da[-]	[no]directaddr	Force direct addressing mode
-e	eject	Start a new listing page
	error	Generate error in list file with user-defined message
-em[-]	[no]extra_mnem	Add extra jump mnemonics
-ep[-]	[no]errorprint	Specify where errors are to be displayed
-f <i>file</i>		Read options and/or controls from <i>file</i>

Option	Control	Description
-fc	farcode	Specify code space configuration for 24-bit models
-fd	fardata	Specify data space configuration for 24-bit models
-fk	farconst	Specify constant space configuration for 24-bit models
-g[-]	[no]debug	Include symbolic debug information in object file
-i	include	Add extra include file
-l[-]	[no]list	Display source lines in list file
-lb[-]	[no]limit_bitno	Do not allow bit number greater than 7
-ld[-]	[no]linedebug	Generate line numbers in object file
-md	model	Specify processor model
-nc	nearcode	Specify code space configuration for 24-bit models
-nd	neardata	Specify data space configuration for 24-bit models
-nk	nearconst	Specify constant space configuration for 24-bit models
-o[-]	[no]object	Specify name of object file
-oc	[no]optionalcolon	Make the colon following a label declaration optional
-omf	omf	Specify OMF96 version to generate
-p[-]	[no]print	Specify name of list file
-pl	pagelength	Set list page length
-pw	pagewidth	Set list page width
-ri[-]	[no]relaxedif	Allow undefined symbols in 'if' statements
-sb[-]	[no]symbols	Include symbol information in list file
-sc	source	Add line number information
-so[-]	[no]signedoper	Set default behavior of operators to signed
-st[-]	[no]subtitle	Set page header subtitle in list file
-t	title	Set page header title in list file
-x[-]	[no]xref	Generate symbol cross-reference table in list file

Table 3-1: ASM196 assembler options

Below are some detailed descriptions of options that have no equivalent control.

- ?** Display an explanation of options on `stdout`.
- V** Display version information on `stdout` and stop.
- f *file*** Use *file* for command line processing. In this way you can extend the command line. This option can be used more than once, even between the controls.

3.3 ASSEMBLER CONTROLS

You can enter assembler controls on the invocation line, as described in Section 3.1, or on a control line in your source code as described below. The format for a control line is

`$control_list [;comment]`

Where:

\$ (dollar sign) must be the first character of any control line, in column 1.

control_list is a list of one or more controls separated by one or more spaces or tabs.

; denotes the start of a comment.

comments is any valid ASCII character. Comment lines are optional.

Control lines must always be terminated by a carriage return or line feed character. Control lines cannot be continued.

For example, the following control lines set up a listing file with a pagewidth of 80 characters per line and a pagelength of 72 lines per page to be titled `section eight` and to be started on a new page because of the `eject` control.

```
$pagewidth (80) pagelength(72)
$title('section eight') eject ; Section 8 listing follows
```

Note that parameters to commands, such as 80 for pagewidth or `section eight` for title, are placed inside parentheses.

Some controls specify conditions that are either positive or negative (on or off). The negative conditions are specified by prefixing `no` to the positive form of the control. Thus, the negative form of `xref` is `noxref` and the negative form of `print` is `noprnt`.

All positive forms of controls have two-letter abbreviations by which they can be specified. Thus the abbreviation for `xref` is `xr` and the abbreviation for `print` is `pr`. The abbreviations for the negative forms of the controls consist of the two-letter abbreviations for the positive form of the command prefixed by `no`. For example, `nopr` is the abbreviation for `noprnt` as `noxr` is the abbreviation for `noxref`.

Most controls have default settings. The assembler automatically uses these defaults unless you specify otherwise. Thus, you only enter controls if you desire assembly conditions different from the default conditions.

You can enter controls and control parameters, both on the invocation line and in source lines. Case is not significant.

3.3.1 PRIMARY AND GENERAL CONTROLS

Controls are categorized as either *primary* or *general*.

Primary controls set conditions that apply throughout the entire assembly of a program. Place primary controls on the primary control lines of the source file or on the invocation line. A primary control line is any control line that appears before the first noncontrol line of the source program. Blank lines and comment lines are considered to be noncontrol lines. A primary control, either positive or negative, can appear only once in a given program. Listing primary controls determine the construction of the listing file, while object primary controls determine the construction of the object file.

General controls cause an immediate action or an immediate change of conditions. In the latter case, the condition specified by the general control remains in effect until another general control causes it to change. You can specify general controls on the invocation line or on control lines anywhere in the source file. You can specify these controls many times within a source file to set conditions during assembly. General controls specified on the invocation line take effect before the assembler reads the first source line, but have no precedence over general controls in the source file.

One exception is the `include` control which can only appear once in a line and only as the last entry of the line.

Table 3-2 lists all the primary and general controls and their abbreviations. The default settings are shown where applicable. See Chapter 4 for a detailed description of each control.

Control Name	Abbreviation	Default	Type
<code>case</code>	<code>cs</code>	<code>cs</code>	Primary
<code>cmain</code>	<code>cm</code>	<code>nocm</code>	Primary
<code>cond</code>	<code>co</code>	<code>co</code>	General
<code>copyattr</code>	<code>ca</code>	<code>noca</code>	Primary
<code>debug</code>	<code>db</code>	<code>nodb</code>	Primary
<code>directaddr</code>	<code>da</code>	<code>noda</code>	Primary
<code>eject</code>	<code>ej</code>	<code>n/a</code>	General
<code>error('string')</code>	<code>er</code>	<code>n/a</code>	General
<code>errorprint</code>	<code>ep</code>	<code>noep</code>	Primary
<code>extra_mnem</code>	<code>em</code>	<code>noem</code>	Primary
<code>gen</code>	<code>ge</code>	<code>noge</code>	General
<code>include(pathname)</code>	<code>ic</code>	<code>n/a</code>	General
<code>limit_bitno</code>	<code>lb</code>	<code>nolb</code>	Primary
<code>linedebug</code>	<code>ld</code>	<code>nold</code>	Primary
<code>list</code>	<code>li</code>	<code>li</code>	General
<code>model(processor)</code>	<code>md</code>	<code>md(kb)</code>	Primary
<code>optimize</code>	<code>ot</code>	<code>noot</code>	Primary
<code>nearcode/farcode</code>	<code>nc/fc</code>	<code>nc</code>	Primary
<code>nearconst/farconst</code>	<code>nk/fk</code>	<code>nk</code>	Primary
<code>neardata/fardata</code>	<code>nd/fd</code>	<code>nd</code>	Primary
<code>object</code>	<code>oj</code>	<code>oj(src.obj)</code>	Primary
<code>omf(number)</code>	<code>omf</code>	<code>omf(2)</code>	Primary
<code>optionalcolon</code>	<code>oc</code>	<code>nooc</code>	Primary
<code>pagelength(number)</code>	<code>pl</code>	<code>pl(60)</code>	Primary
<code>pagewidth(number)</code>	<code>pw</code>	<code>pw(120)</code>	Primary
<code>print</code>	<code>pr</code>	<code>pr(src.lst)</code>	Primary

Control Name	Abbreviation	Default	Type
relaxedif	ri	nori	Primary
save/restore	sa/rs	n/a	General
searchinclude(<i>pathname</i>)	si	nosi	General
set/reset	se/re	n/a	General
signedoper	so	noso	Primary
source	sc	nosc	General
subtitle(<i>'string'</i>)	st	nost	General
symbols	sb	sb	Primary
title(<i>'string'</i>)	tt	tt(<i>modname</i>)	General
xref	xr	noxr	Primary

Table 3-2: ASM196 assembler controls

3.3.2 CONTROL PROCESSING

The assembler processes controls in the following manner. Initially, the assembler uses the default value for each control. The assembler then processes the program text from left to right, starting at the invocation line and then going from the first source line to the final source line.

The assembler sets each primary control encountered to the condition you specified. Any further occurrence of the same primary control produces an error message to that effect and the subsequent occurrence does not affect the assembler processing.

Except for the `list` and `eject` controls, the assembler sets general controls as the assembler scans each line. The assembler uses the last condition of the general control it encounters, unless you insert a `restore` in the middle. For example, in the following control line, `nogen` is in effect because it is the last control on the control line:

```
$gen nogen gen nogen gen nogen
```

Remember that when both positive and negative forms of a control appear on a control line, only the last control is active. Note also that the `list` and `eject` controls become effective only after the current line is printed.

3.4 OUTPUT OBJECT FILE

The assembler produces an object file, with a `.obj` extension, that contains machine language and data that you can load into memory for execution or interpretation. This file also contains control information governing the loading process. The object file can be in absolute or relocatable object code format. An absolute object file contains absolutely located segments and makes no external references. You can load this file without passing it through the RL196 linker. A relocatable object file, however, contains at least one relocatable segment and might make external references. You must run this file through the RL196 linker so that the linker can locate the relocatable segment to an absolute memory address and resolve its external references. See the *80C196 Utilities User's Guide*, listed in *Related Publications*, for instructions on how to run the RL196 linker.

3.5 LISTING FILE

The listing file provides you with a listing of your source program and important information on the assembly process. This file contains any information you supplied and all of the assembler-generated information. Errors detected by ASM196 within the source code also appear in this file. The assembler, by default, produces a listing file that has a pagewidth of 120 characters and a page length of 60 lines. Each page includes a page header and a column heading, followed by the assembly output. Lines that exceed the right margin setting, set by default or the `pagewidth` control, are continued on the line directly below the original line commencing at column 35.

The list file comes in two different formats dependant on the compilation model. If `model(nt)` or `model(np)` is specified, the 24-bit (extended) format is used, otherwise, the standard format (S) is applied. The main difference between the two formats is that in the 24-bit format, addresses are represented as six digit hexadecimal numbers and symbol values as eight digit hex numbers while in the S format both are four digit hex numbers. Because of the additional digits in the 24-bit format format, certain fields are slightly moved to the right. For more details, refer to the discussion on the body of list file, section 3.5.2.

Figure 3-1 shows an example of the format of the list file.

```

80C196 MACRO ASSEMBLER      --> HAND CALCULATOR: MAIN <--                date   PAGE 1

80C196 macro assembler Vx.y Rz SN00000-005   (c) year TASKING, Inc.
(C)1983,1990,1993 Intel Corporation

SOURCE FILE: sample.a96
OBJECT FILE: sample.obj
CONTROLS SPECIFIED IN INVOCATION COMMAND: xref

LOC   OBJECT                STMT   SOURCE STATEMENT
      1 $TITLE('--> HAND CALCULATOR: MAIN <--')
      2
      3 HC_MAIN              MODULE MAIN,STACKSIZE(8)
      4 ;
      5 ;The program simulates a common hand calculator.
      6 ;Available commands (DR the display register, MR the memory
      ;register):
      7 ;
      8 ; [operand1] op operand2 - where 'operand is a decimal
      ;number.
      9 ;                   'op' can be +, -, * and /.
     10 ;   M+                  - add DR to MR.
     11 ;   RM                  - transfer MR to DR.
     12 ;   M-                  - subtract DR from MR (result in MR).
     13 ;   CM                  - clear MR.
     14 ;   ? or H              - a help info is displayed.
     15 ;   EX                  - exit.
     16 ;Commands can be written in upper or lower case letters.
     17
     18                     EXTRN   ER_Invalid_Command
     19                     PUBLIC  MN_Result, MN_Exit_Flag
     20 ;
     21 ;Runtime conventions:
     22 ; 1) Parameters are transferred on the stack.
     23 ; 2) Word and byte funcs return their result in the
     24 ;    global reg MN_Result.
     25 $INCLUDE(8096.INC)
=1 26 $nolist
     86 ;;;;;;;;;;;;;;;;;;;;;;;;;
     87 ;

```

Figure 3-1: Source listing example

```

80C196 MACRO ASSEMBLER      --> HAND CALCULATOR: MAIN <--          date   PAGE   2

LOC   OBJECT                STMT   SOURCE STATEMENT

                                88   ;Common definitions
000D                                89   CR              EQU    0DH
000A                                90   LF              EQU    0AH
0020                                91   BLANK           EQU    20H
0009                                92   TAB              EQU    09H
                                93   STRING           MACRO   Str              ;Defines a length-
                                                ;prefixed string

                                94                       LOCAL   Len
                                95   Start_LC          SET     $
                                96                       DCB     Len, Str
                                97   Len                EQU     $-Start_LC-1
                                98                       ENDM
                                99
0000                                100                      RSEG
0000                                101 MN_Result:      DSW     1              ;Result of byte or
                                                ;word functions
0002                                102 MN_Exit_Flag:    DSB     1              ;Exit flag, only bit
                                                ;0 is used

                                103
2080                                104                      CSEG     at 2080H
                                105                      EXTRN   IO_Put_String, IO_Put_Char
                                106                      EXTRN   RD_Get_Line, RD_Get_Chaqr,
                                                RD_This_Char
                                107                      EXTRN   RD_Skip_Blanks, RD_Check_EPL
                                108                      EXTRN   UT_FindB, UT_Help_Com
                                109                      EXTRN   ER_Put_Message
                                110                      EXTRN   RG_Eval, RG_Mem_Op, RG_Clear_MR,
                                                RG_Recall_MR,
***
*** ERROR #10 IN STMT 110 (LINE 52), SYNTAX ERROR
                                111                      EXTRN   RG_Print_DR
2080                                112 Start:
2080 A1000018                    R 113                      LD      SP, #STAC      ;... of main program
2084 C9DC20                      114                      PUSH     #STR_signon ;Init StackPointer
2087 EF0000                    E 115                      CALL     IO_Put_String ;Print Signon
208A 1102                      R 116                      CLRB     MN_Exit_Flag ;Reset exit
                                ;flag
208C                                117 Main_Loop:
208C 300203E78E00                R 118 !                      BBS     MN_Exit_Flag, 0, Exit ;Check for exit
                                ;character
2092 EF0000                    E 119                      CALL     RG_Print_DR
2095 C93A00                      120                      PUSH     #' ' ;Print prompt
                                ;character
2098 EF0000                    E 121                      CALL     IO_Put_Char
209B EF0000                    E 122                      CALL     RD_Get_Line ;Line is placed in
                                                ;RD_Line

```

Figure 3-1: Source listing example (continued)

```
80C196 MACRO ASSEMBLER      --> HAND CALCULATOR: MAIN <--          date PAGE 3

LOC  OBJECT                  STMT  SOURCE STATEMENT

209E EF0000                  E 123          CALL    RD_Skip_Blanks
                               124
20A1 EF0000                  E 125          CALL    RD_This_Char    ;Find command type
20A4 C9C720                  126          PUSH     #STR_1st_Chars
20A7 C800                    R 127          PUSH     MN_Result
20A9 EF0000                  E 128          CALL    UT_FindB
20AC 89FFFF00                R 129          CMP     MN_Result,#0FFFFH
20B0 D708                    130          BNE     Command_OK
20B2 C90000                  E 131          PUSH     #ER_Invalid_Command
20B5 EF0000                  E 132          CALL    ER_Put_Message
20B8 27D2                    133          BR      Main_Loop
20BA                               134 Command_OK:
20BA 640000 R                 135          ADD     MN_Result, MN_Result ;Do case
                               ;MN_Result

20BD A300FC2000              R 136          LD      MN_Result, Com_Tab[MN_Result]
20C2 C98C20                  137          PUSH     #Main_Loop    ;Simulate indirect
                               ;call

20C5 E300                    R 138 BR      [MN_Result]
                               139 ;
                               140 ;Tables and constants used for the above logic
20C7                               141 STR_1st_Chars: STRING <'0123456789','+-*/','MRC?HE'>
20DC                               145 STR_Signon:  STRING <CR,LF,'8096-BASED CALCULATOR,
                               V1.0',CR,LF>

20FC                               149 Com_Tab:
20FC 00000000000000000000    E 150          DCW     RG_Eval,RG_Eval,RG_Eval,RG_Eval; 0-4
2104 00000000000000000000    E 151          DCW     RG_Eval,RG_Eval,RG_Eval,RG_Eval; 5-9
210C 00000000000000000000    E 152          DCW     RG_Eval,RG_Eval,RG_Eval,RG_Eval; +-
                               */

2114 0000                    E 153          DCW     RG_Mem_Op          ;M+ or M-
2116 0000                    E 154          DCW     RG_Recall_MR        ;RM
2118 0000                    E 155          DCW     RG_Clear_MR        ;CM
211A 00000000                E 156          DCW     UT_Help_com, UT_Help_com    ;? or H
211E 0000                    157          DCW     UT_Exit_com          ;EX

*** ERROR #38 IN STMT 157 (LINE 93), UNDEFINED SYMBOL
                               158 ;
                               159 ; Exit loop
2120                               160 exit:
2120 C92821                  161          PUSH     #STR_Signoff    ;Print signoff
2123 EF0000                  E 162          Call    IO_Put_String
```

Figure 3-1: Source listing example (continued)

```
80C196 MACRO ASSEMBLER      --> HAND CALCULATOR: MAIN <--           date PAGE 4

LOC   OBJECT                STMT   SOURCE STATEMENT

2126  27FE                  163           BR      $              ;Infinite loop
                                164 $GEN
2128                                165 STR_Signoff:  STRING <CR,LF,'      S T O P',CR,LF>
2128                                166+1 Start_LC    SET      $
2128  0F0D0A2020202053      167+1           DCB      ??0003, CR,LF,' S T O P',CR,LF
2130  2054204F20500D0A
000F                                168+1 ??0003 EQU  $-Start_LC-1
2138                                169 Start_LC    SET      $
2138  0F0D0A2020205320      170           DCB      ??0003,CR,LF,'      S T O P',CR,LF
2140  54204F20500D0A
                                171
2147                                172           END
```

Figure 3-1: Source listing example (continued)

3.5.1 HEADER AND INTRODUCTORY LINES

Header information appears at the top of each page of the listing file and contains the title, the date, and the page number as follows:

```
80C196 MACRO ASSEMBLER title date PAGE n
```

Where:

title is either the title specified by the *title* control or the module name. The title appears to the right of the word ASSEMBLER.

date is the system date.

n is the page number.

The first page of the listing file also contains the following introductory lines:

```
80C196 macro assembler vx.y rz SN000000-005 (c) year TASKING, Inc.
(C)1983,1990,1993 Intel Corporation

SOURCE FILE: source
OBJECT FILE: object
CONTROLS SPECIFIED IN INVOCATION COMMAND: control_list
```

Where:

x.y specifies the version number of the assembler.



<i>z</i>	specifies the revision number of the assembler.
<i>year</i>	specifies the copyright year.
<i>source</i>	is the name of the source file including its path.
<i>object</i>	is the name of the object file including its path, or if noobject was specified, the entry <none>.
<i>control_list</i>	is a list of all the controls you specified on the invocation line.

3.5.2 SOURCE LINES

The body of the listing file consists of columns of information (fields). The fields are:

- LOC for the location counter field, starts at column 1.
- OBJECT for the object code field, starts at column 6 (S format), 8 (24-bit format).
- STMT for the line number field, starts at column 28 (S format), 32 (24-bit format).
- SOURCE STATEMENT for the source field, starts at column 35 (S format), 44 (24-bit format).

Source lines contain the following fields, shown in the standard (S) and 24-bit format:

- Location counter field (columns 1 through 4) (S), 1-6 (24-bit). This field contains the hexadecimal value of the location counter. This value is the address where the next byte of code or data is located. The assembler adjusts the value displayed for any required alignment if the next statement is a dsw, dcw, dsp, dcp, ds1, dcl, dsr, dcr, cseg, dseg, kseg, rseg, dseg, odseg, sseg, or org directive. For absolute segments, the LOC field contains an absolute address. For relocatable segments, the LOC field contains the offset from the beginning of the segment currently being assembled. The LOC field is displayed for all machine instructions, the constant-definition directives (dcb, dcw, dcp, dcl, and dcr), the storage-reservation directives (dsb, ds1, dsp, dsq, and dsr), the segment-selection directives (cseg, dseg, kseg, odseg, oseg, reg, and sseg), the org directive, and for empty statements and macro calls that are preceded by labels. Otherwise, the field is blank.

- Set or Equ value (columns 3 through 6) (S), 3-10 (24-bit). If the statement is a `set` or an `equ` directive, this field displays the value of the expression in hexadecimal. The value is not aligned.
- Object code field (columns 6 through 21) (S), 8-25 (24-bit). This field contains the object code in hexadecimal generated by the assembler for the given line.
- Fixup indicator (column 23) (S), 27 (24-bit). When no fixup is required, this field is blank. This field displays an `R` if the code contains a relocatable reference. If the code contains an external reference, this field contains an `E`. If the code contains a complex expression, this field contains a `C`.
- Include indicator (columns 25 through 26) (S), 29-30 (24-bit). All source lines that are from an include file contain an equal sign (=) in column 25. Column 26 contains the include nesting indicator which indicates the level of nesting, if any. For example, if you include an include file, column 26 contains a 1 for the source lines from that include file to indicate nesting level 1. If this include file calls another include file, column 26 contains a 2 for the source lines of the second include file, indicating nesting level 2.
- Line number field (columns 28 through 31) (S), 32-35 (24-bit). This field contains the line number starting from 1. The line number is sequentially incremented by one for every source line and macro expansion line, but the line number is not incremented for conditionally skipped lines, lines not assembled because of an `if` directive. The line number increments for every non-skipped line, whether the line is listed or not.
- Macro expansion indicator (columns 32 through 33) (S), 36-37 (24-bit). The first column contains a plus sign (+) if the line is a macro expansion line. The second column contains macro expansion nesting. Thus, 1 indicates the original macro, 2 indicates a macro nested within the original macro.
- Expanded generic instruction indicator (column 34) (S), 38 (24-bit). This field contains an exclamation point (!) if the source line contains a generic instruction that is expanded into several machine instructions or any of the machine instructions resulting from that expansion.

- Source statement field (column 35 (S), 39 (24-bit) and on). The source line is reproduced exactly as entered, except that tabs are converted to blanks. If a generic instruction is expanded into more than one machine instruction and if the gen control is in effect, the instructions resulting from the expansion are displayed, one instruction per line.

3.5.3 ERROR LINES

When an error occurs, an error line appears in the listing immediately after the source statement which caused the error. The error line format is as follows:

```
*** ERROR #d IN STMT #n (LINE s [OF FILE f ], <error-text>
```

Where:

d is the corresponding error number.

n is the assembly statement number from column 28 through 31 of the listing file.

s is the source line number from your source file. If the source line is from an include file the source line number is the line number within file *f*.

If the error is a syntax error, a line preceding the error message line indicates which token caused the error. The line consists of three asterisks (***) followed by underscores (_) extending to the point in error. The caret symbol (^) points to the token in error.

3.5.4 SYMBOL TABLE

The symbol table is a list of all symbols defined in the program along with status information about the symbols. The assembler automatically includes the symbol table listing in the listing file since `symbols` is the default. If you specify `xref`, the assembler includes cross-reference information in the symbol table.

Figure 3-2 shows the format of the symbol table. A page eject always occurs between the end of the source listing and the start of the symbol table.

```

80C196 MACRO ASSEMBLER ==> HAND CALCULATOR: MAIN <==  date

SYMBOL TABLE AND CROSS-REFERENCE LISTING
-----

  N A M E                VALUE  ATTRIBUTES AND REFERENCES

??0001 ..... 0014H  NULL ABS
                        82  83#
??0002 ..... 001FH  NULL ABS
                        86  87#
??0003 ..... 000FH  NULL ABS
                        107 108#
BLANK ..... 0020H  NULL ABS
                        31#
COM_TAB ..... 20FCH  CODE ABS WORD
                        75  88#
COMMAND_OK ..... 20BAH  CODE ABS ENTRY
                        69  73#
CR ..... 000DH  NULL ABS
                        29# 86  86  107 107
ER_INVALID_COMMAND ... ----- NULL EXTERNAL
                        18# 34  70
ER_PUT_MESSAGE ..... ----- CODE EXTERNAL
                        19# 71
EXIT ..... 2126H  CODE ABS ENTRY
                        57  100#
HC_MAIN ..... ----- MODULE MAIN STACKSIZE(8)
                        3#
IO_PUT_CHAR ..... ----- CODE EXTERNAL
                        46# 60
IO_PUT_STRING ..... ----- CODE EXTERNAL
                        46# 54  102
LF ..... 000AH  NULL ABS
                        30# 86  86  107 107
MAIN_LOOP ..... 208CH  CODE ABS ENTRY
                        35  56# 72  76
MN_EXIT_FLAG ..... 0002H  REG REL PUBLIC BYTE
                        19  43# 55  57
MN_RESULT ..... 0000H  REG REL PUBLIC WORD
                        19  42# 66  68  74  74  75  75  77
RD_CHECK_EOL ..... ----- CODE EXTERNAL
                        47#
RD_GET_CHAR ..... ----- CODE EXTERNAL
                        47#
RD_GET_LINE ..... ----- CODE EXTERNAL
                        47#  61
RD_SKIP_BLANKS ..... ----- CODE EXTERNAL
                        47#  62
RD_THIS_CHAR ..... ----- CODE EXTERNAL
                        47#  64

```

Figure 3-2: Symbol table example

RG_CLEAR_MR	-----	CODE EXTERNAL	50# 94						
RG_EVAL	-----	CODE EXTERNAL	50# 89 89	89 89	89 90	90			
			90 90 90	91 91	91 91	91			
RG_MEM_OP	-----	CODE EXTERNAL	50# 92						
RG_PRINT_DR	-----	CODE EXTERNAL	50# 58						
RG_RECALL_MR	-----	CODE EXTERNAL	50# 93						
SP	0018H	NULL ABS	33# 52						
START	2080H	CODE ABS ENTRY	51#						
START_LC	212EH	CODE ABS	81# 83 85# 87	106# 108					
STR_1ST_CHARS	20C7H	CODE ABS BYTE	65 80#						
STR_SIGNOFF	212EH	CODE ABS BYTE	101 105#						
STR_SIGNON	20DCH	CODE ABS BYTE	53 84#						
STRING	-----	MACRO	34# 80 84 105						
TAB	0009H	NULL ABS	32#						
UT_EXIT_COM	-----	UNDEFINED	96						
UT_FINDB	-----	CODE EXTERNAL	48# 67						
UT_HELP_COM	-----	CODE EXTERNAL	48# 95 95						

Figure 3-2: Symbol table example (continued)

Each page of the symbol table has the same header lines as the pages of the source listing. The page numbers continue from the source listing into the symbol table. In addition, the first page of the symbol table contains the header:

SYMBOL TABLE LISTING

If you specified the xref control, the header reads:

SYMBOL TABLE AND CROSS-REFERENCE LISTING

The symbol table, like the source listing, is divided into fields. The symbols are then listed in alphabetic order, except for the underscore (_) coming first. Fields and symbols are listed as follows:

- **NAME** field (columns 1 through 31). The name of the symbol is listed. Alternating periods and blanks (. . .) fill the field to column 31.
- **VALUE** field (columns 35 through 39)(S), 35-43 (24-bit). The address of the symbol is given in hexadecimal. For relocatable symbols, the address reflects the symbol's offset from the segment. For absolute symbols, the address is the absolute address where the symbol is located. For undefined symbols, external symbols, macros, and the module symbol, the value field contains hyphens (----).
- **ATTRIBUTES** field (column 43 (S), 47 (24-bit) through the end of the line). The attributes associated with each name are listed. The attributes are separated by spaces. The attributes are:

NEAR or FAR appears for near/far attribute of the symbol's segment.

REG, OVERLAY, CONSTANT, FAR CONSTANT, CODE, FAR CODE, HIGH CODE, DATA, STACK or NULL is the segment type of the symbol.

NUMBER appears for symbols that do not belong to any segment.

BYTE, WORD, POINTER, LONG, REAL, ENTRY or NULL is the data type of the symbol. The size of a POINTER depends on the processor model used, and is 2 bytes for 16-bit models and 4 bytes for 24-bit models. The POINTER is of type NO_TYPE to avoid symbol attribute mismatch warnings from the linker.

ABS appears for absolute symbols.

REL appears for relocatable symbols.

PUBLIC appears for public symbols.

EXTERNAL appears for external symbols.

MACRO appears for macros.

MODULE appears only for the module name symbol. Each assembly unit has only one module.

MAIN indicates that the module is the main module of the application.

STACKSIZE(*n*) gives the size of the stack required by the module.

UNDEFINED appears for symbols that are not defined in the program.

- REFERENCES (column 43 (S), 47 (24-bit) through the end of the line). The reference field is present only if you specified `xref` with the `symbols` control. This field contains a list of the line numbers of all statements that use the given symbol. If the value of the symbol is defined (or redefined) in the given statement, the line number of the statement is followed by the number sign (#). Spaces separate statement line numbers for a given symbol from one another.

3.6 ERRORPRINT FILE

The errorprint file consists of all listing lines that contain errors. The listing of erroneous lines matches the listing of these lines in the print file. Each error line consists of three asterisks (***) followed by the error number, the statement line number where the error occurred, and the error message. See Chapter 8 for a complete list of error messages produced by ASM196.

3.7 AUTOMATIC ASSEMBLER INVOCATION

TASKING offers three ways of automatically invoking a series of commands: makefiles, batch files and command files. This section demonstrates ways to use these features with TASKING software development tools. Filenames and directory names appearing in this section are examples.

3.7.1 USING MAKE UTILITY MK196

mk196 takes a file of dependencies (a 'makefile') and decides what commands have to be executed to bring the files up-to-date. These commands are either executed directly from **mk196** or written to the standard output without executing them.

For a detailed discription of this utility, see *80C196 Utilities User's Guide*, Chapter *MK196 Make Utility*.

3.7.2 BATCH FILES

Batch files are useful when the same commands are invoked over and over again. Instead of retyping the invocation line, you can create a batch file to execute the command automatically. The same batch file can operate on different sets of input files by passing the input filenames as arguments. See examples in the next section. All batch files must have the `.bat` extension.

Batch files can be chained, meaning one can invoke the next. Batch files can also be nested. Nesting of batch files requires DOS version 3.3 or later and the use of the DOS `call` command. When nesting, the called batch file takes control directly. When the called batch file is complete, control passes back to the calling batch file at the command immediately following the one that invoked the called batch file.

Invoke the batch file by typing in the batch filename without the `.bat` extension.

For example, the `asm_rl.bat` file contains the following lines:

```
asm196 file1.a96 debug
if errorlevel 1 goto :exit
rl196 file1.obj
echo assembled and linked ok
goto :end
:exit
echo asm failed
:end
```

To invoke the batch file, enter the batch filename at the prompt as follows:

```
asm_rl
```

When invoked, ASM196 assembles `file1.a96`. If no error occurs, the assembler creates the object file `file1.obj` and runs it through the RL196 linker; otherwise, DOS displays `asm failed` on your screen.

As mentioned before, passing arguments to a DOS batch file enables the same batch file to perform similar work on different programs or data each time you execute the batch file. To rewrite the previous example, `asm_rl.bat` contains the following:

```
asm196 %1.a96 debug
if errorlevel 1 goto :exit
rl196 %1.obj
echo assembled and linked ok
goto :end
:exit
echo asm failed
:end
```

Invoke the batch file with the following:

```
asm_rl file1
```

The argument `file1` replaces the `%1` parameter and the `.a96` extension is attached after it. The same process happens with the link line. You can have up to nine arguments, `%1 – %9`. The `%0` argument is the command being executed. For more information on batch files, see the *DOS Reference* manual for your system. To increase the number of arguments passed, refer to the `shift` subcommand also found in this manual.

Since most typical applications consist of multiple object files, most likely all your object filenames cannot fit in one line. One way to work around this limitation is to use indirect input. See the *80C196 Utilities User's Guide*, listed in *Related Publications*, for information on how to link multiple object files together.

3.7.3 LOG FILE

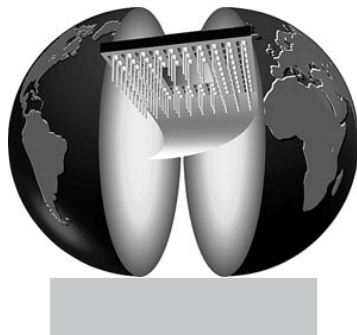
You can redirect the output from a command file to another file to obtain a complete log of all console activity during the command file's execution, including the invocation line for each program executed in the command file. The following example creates a log file named `asm_rl.log`.

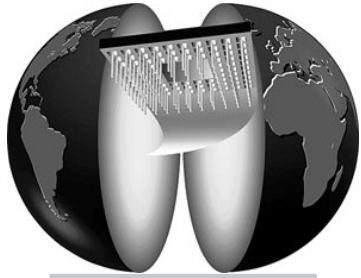
```
command < asm_rl.com > asm_rl.log
```

CHAPTER

4

ASSEMBLER CONTROLS





4

CHAPTER

This chapter explains each control in detail. Each control appears in alphabetical order. Each control except for the `error` control has an equivalent assembler option.



See the *Conventions Used In This Manual* at the beginning of this manual for special meanings of type styles used in this manual.



With controls that can be set from within EDE, you will find a mouse icon that describes the corresponding action.

case

Function

Tells assembler to act case sensitive.

Syntax



Select the EDE | Assembler Options | Project Options... menu item. Enable or disable the Operate in case sensitive mode check box in the Object tab.



case		nocase
-case		-case-

Abbreviation

cs		nocs
----	--	------

Class

Primary control

Default

case

Description

Use this control to tell the assembler to work in a case sensitive manner. However, some general rules regarding case sensitivity must be considered:

1. Options supplied on the command line (**-?** and **-V**) are always handled case sensitive.
2. Controls supplied on the command line are always handled case insensitive.
3. Keywords are always handled case insensitive.

When you use the `nocase` control:

4. All module names, public and external symbols are converted to upper case.
5. All filenames are converted to lower case.

When you use the default case control (or **-case** option):

6. None of the conventions mentioned in (4) or (5) is performed.

Example

The following example turns case sensitivity off:

```
asm196 main.a96 nocase
```

cmain

Function

Allow public symbol 'main' in source.

Syntax



Select the EDE | Assembler Options | Project Options... menu item. Enable or disable the Allow public symbol 'main' in source (needed for cstart.a96) check box in the Object tab.



cmain		nocmain
-c		-c-

Abbreviation

cm		nocm
----	--	------

Class

Primary control

Default

nocmain

Description

The RL196 linker expects a public symbol 'main' in one of the object or library files. When all objects are created from assembler source files, we must define a public symbol 'main' in one of these files. However, 'main' is a reserved word and may not be used in an assembler source file unless the cmain control is used.

Example

```
asm196 main.a96 cmain
```

cond

Function

Specifies whether conditionally-skipped source lines appear in the listing file.

Syntax



Select the EDE | Assembler Options | Project Options... menu item. Enable or disable the Include conditionally-skipped source lines check box in the Listing tab.



cond		nocond
-C		-C-

Abbreviation

co		noco
----	--	------

Class

General control

Default

nocond

Description

Use this control to specify whether to include conditionally-skipped source lines in the listing file. If you specify `cond`, the assembler includes all of the source lines regardless of the result of the conditional assembly. See Chapter 6 for more information on conditional assembly. If you specify `nocond`, only the assembled source lines appear in the listing file, meaning only the conditionally assembled lines evaluated to be TRUE appear in the listing.



`if/else/endif` directive (Chapter 6)
list control

copyattr

Function

Specifies whether to use segment type of current segment.

Syntax



Select the EDE | Assembler Options | Project Options... menu item. Enable or disable the Use segment type of current segment check box in the Object tab.



<code>copyattr</code>		<code>nocopyattr</code>
<code>-ca</code>		<code>-ca-</code>

Abbreviation

<code>ca</code>		<code>noca</code>
-----------------	--	-------------------

Class

Primary control

Default

`nocopyattr`

Description

When symbols are defined using the SET/EQU directives, the segment type of these symbols was the NULL segment type. However, for symbols defined with the EXTRN keyword, the segment type of the current segment was given to the symbol. The assembler will set the segment type for symbols defined with SET/EQU according to the following rules:

1. If the segment type of the expression on the right hand side of SET/EQU is of type NULL and the `copyattr` control is set, the segment type of the current segment is used.
2. For all other segment types of the expression or when the `copyattr` control is not set, the segment type of the expression is assigned to the symbol being defined. This behavior is equal to older versions of the assembler.

Example

When invoked with **asm196 -ca** *source* :

```
13      set      5h              ; type = NULL

      dseg
      extrn     e1:word          ; type = DATA
11      equ      10h             ; type = DATA REL
lab:    dsw      1

      rseg
12      equ      12h             ; type = REGISTER
14      equ      lab            ; type = DATA REL

      cseg
      extrn     e2:byte          ; type = CODE
      end
```


debug

Function

Specifies if symbol-table information is included in the object file.

Syntax



Select the EDE | Assembler Options | Project Options... menu item. Enable or disable the Generate symbolic debug information check box in the Object tab.



debug		nodebug
-g		-g-

Abbreviation

db		nodb
----	--	------

Class

Primary control

Default

nodebug

Description

Use this control to include the symbol table information in the object file, providing the object control is in effect. You must use this control if you plan on debugging your application with an ICE[®]-196 emulator. When you specify nodebug, the assembler suppresses the symbol table information. You cannot specify this control with the noobject control.



object control
symbol table listing (Chapter 3)

directaddr

Function

Force direct addressing mode.

Syntax



Select the EDE | Assembler Options | Project Options... menu item. Enable or disable the Force direct addressing mode check box in the Object tab.



<code>directaddr</code>		<code>nodirectaddr</code>
<code>-da</code>		<code>-da-</code>

Abbreviation

<code>da</code>		<code>noda</code>
-----------------	--	-------------------

Class

Primary control

Default

`nodirectaddr`

Description

This control forces the assembler to use the direct addressing mode, unless another addressing mode is explicitly used in a statement.

The assembler cannot always determine the addressing mode for a variable, and in that case it will choose what is most appropriate: long/short or register direct. Consider the following syntax:

```
LD      reg, var
```

When `var` is within the register range, the assembler will generate a register direct load. If `var` is outside the register segment, the assembler will normally generate a long-indexed load via `r0`. However, with the use of the `directaddr` control, the assembler will always try to generate a direct register load. If `var` is not in the register segment, the assembler will generate an error message.



Mixed Addressing Modes (Chapter 5)

eject

Function

Causes the assembler to start a new listing page.

Syntax

```
eject  
-e
```

Abbreviation

ej

Class

General control

Description

Use this control to continue the listing on a new page. The assembler performs the page ejection after the current line is listed.

Example

Insert the following control line before the desired page division in your source file. Make sure that the dollar sign (\$) is in column 1.

```
$ej
```

error

Function

Generate error in list file with user-defined message.

Syntax

```
error('text')
```

Abbreviation

er

Class

General control

Description

This control cannot be used on the command line and, subsequently, has no associated option. The `error` control can be used to generate an error in the list file with a user-defined error message. If the `error` control is used within a macro definition, all references to the formal and local parameter symbols are replaced with the actual parameter values or the generated labels. This substitution can be suppressed by used angle brackets around the symbol name.

Example

An example usage of the `error` control is:

```
MyAdd MACRO arg1, arg2
    IFB    <arg2>
    $ERROR('Macro called with empty argument: arg1, <arg2>')
    ELSE
        add    arg1, arg2
    ENDIF
END
```

In the above example the formal parameter symbol `arg2` is not replaced with its value as it is surrounded by angle brackets. The formal parameter symbol `arg1` is replaced with the actual parameter value.

errorprint

Function

Specifies where error lines and messages are to be directed.

Syntax

```
errorprint [(filename)] | noerrorprint
-ep [filename]          | -ep-
```

where:

filename is the name of the error print file including its full path.

Abbreviation

```
ep      | noep
```

Class

Primary control

Default

```
noerrorprint
```

Description

Use this control to specify whether error lines and messages are displayed on the screen or stored in a file, specified by *filename*. The filename you specify must differ from the listing filename with the *filename.lst* extension. The assembler automatically displays the messages on the screen if you do not specify a filename.

The `noerrorprint` control is the default. The assembler displays no errors on the screen and does not produce an error print file.

Example

This example tells the assembler to store all messages in the file `file1.err`.


```
asm196 file1.a96 errorprint(file1.err)
```


extra_mnem

Function

Add extra jump mnemonics.

Syntax

 Select the EDE | Assembler Options | Project Options... menu item. Enable or disable the Add extra jump mnemonics check box in the Object tab.



extra_mnem		noextra_mnem
-em		-em-

Abbreviation

em | noem

Class

Primary control

Default

noextra_mnem

Description

With extra_mnem you can use the following additional jump mnemonics and generic jump mnemonics (16-bit or 24-bit) in your assembly source. They have the same meaning as the original mnemonics in the first column.

Original Mnemonics			Description	Equivalent Extra Mnemonics		
JH	BH	EBH	> unsigned	JGTU	BGTU	EBGTU
JNH	BNH	EBNH	<= unsigned	JLEU	BLEU	EBLEU
JC	BC	EBC	>= unsigned	JGEU	BGEU	EBGEU
JNC	BNC	EBNC	< unsigned	JLTU	BLTU	EBLTU
JE	BE	EBE	=	JZ	BZ	EBZ
JNE	BNE	EBNE	<>	JNZ	BNZ	EBNZ
JGT	BGT	EBGT	> signed	JGTS	BGTS	EBGTS



Original Mnemonics			Description		Equivalent Extra Mnemonics		
JLE	BLE	EBLE	<=	signed	JLES	BLES	EBLES
JGE	BGE	EBGE	>=	signed	JGES	BGES	EBGES
JLT	BLT	EBLT	<	signed	JLTS	BLTS	EBLTS

Example

This example tells the assembler to use the extra jump mnemonics.

```
asm196 main.a96 extra_mnem
```

gen

Function

Causes macro expansion lines to be interspersed with original source lines.

Syntax



Select the EDE | Assembler Options | Project Options... menu item. Enable or disable the Expand macros check box in the Listing tab.



gen		nogen
-G		-G-

Abbreviation

ge		noge
----	--	------

Class

General control

Default

nogen

Description

Use this control to include the following items in the listing:

- macro expansion lines
- lines containing object code that cannot fit onto one line such as a long dcb list
- sequences of machine instructions resulting from translation of a generic instruction (e.g., dbnz to a far target)

If you specify nogen, only the original source text appears in the listing.

Example

Your source file contains the following macro and macro call:

```

block macro numb,prefix
    count set 0
    rept numb
    count set count+1
    genlab prefix,%count ; Nested macro call.
                           ; Genlab macro defined
                           ; elsewhere
    endm
endm

cseg
block 3,lab
end

```

If you assemble with the gen control, your macro call line is expanded as follows:

```

block 3,lab
+1      count    set 0
+1      rept     3
+1          count set count+1
+1          genlab lab,%count
+1      endm
+2      count    set count+1
+2      genlab    lab,%count
+3 LAB1: dcb     0
+2      count    set count+1
+2      genlab    lab,%count
+3 LAB2: dcb     0
+2      count    set count+1
+2      genlab    LAB,%count
+3 LAB3: dcb     0

```



list control
macro processing (Chapter 7)

include

Function

Causes the assembler to include the specified file in its processing.

Syntax

```
include(filename)  
-i filename
```

where:

filename is the name of the include file including its full path.

Abbreviation

ic

Class

General control

Description

Use this control to tell the assembler to include the specified file in its processing. You can enter this control on the invocation line or on a control line after the `module` directive. If this control is one of several controls on a control line, the `include` control must always be the last control you specify. The nesting level for include files cannot be more than nine deep.

Example

This invocation line tells the assembler to include the file `kb_sfrs.inc` in its assembly:

```
asm196 file1.a96 include(kb_sfrs.inc)
```

limit_bitno

Function

Do not allow bit numbers greater than 7.

Syntax



Select the EDE | Assembler Options | Project Options... menu item. Enable or disable the Allow bit numbers larger than 7 check box in the Misc tab.



<code>limit_bitno</code>		<code>nolimit_bitno</code>
<code>-lb</code>		<code>-lb-</code>

Abbreviation

<code>lb</code>		<code>nolb</code>
-----------------	--	-------------------

Class

Primary control

Default

`nolimit_bitno`

Description

When you use the JBS or JBC instruction with an external bit number, the assembler will have to fill in the bit number. It is allowed to specify a bit number which is larger than 7. If this is the case, then the bit register will be increased by one and the bit number will be decreased by 8 until the bit number is smaller than 8. If the `limit_bitno` control is used, all external bit number with a value greater than 7 will generate an error.

Example

The following example does not allow bit numbers larger than 7 in its assembly files:

```
asm196 main.a96 limit_bitno
```

linedebug

Function

Generate line numbers in object file.

Syntax



Select the EDE | Assembler Options | Project Options... menu item. Enable or disable the Generate line number information check box in the Object tab.



linedebug		nolinedebug
-ld		-ld-

Abbreviation

ld		nold
----	--	------

Class

Primary control

Default

nolinedebug

Description

To generate line numbers in the object file use the control `linedebug`. It can be used separate from the debug control. If the debug control is used, line numbers are automatically generated. To suppress line numbers use `nold`.



debug control
source control

list

Function

Causes the display of subsequent source lines in the list file.

Syntax

list		nolist
-1		-1-

Abbreviation

li		noli
----	--	------

Class

General control

Default

list

Description

Use this control to list the source lines in the output listing file. If you specify `nolist`, the assembler suppresses the listing of any source line until the next `list` control is encountered. However, the assembler still prints source lines containing errors. When a subsequent `list` control causes the resumption of listing after its suppression due to `nolist`, the assembler follows the listing settings originally in effect, such as the settings of the `pagewidth` and `pagelength` controls.



cond control
gen control
save/restore control

model

Function

Causes code for microcontroller to be generated.

Syntax



Choose a cpu from the EDE | CPU Model... menu item. Optionally select one or more of the radio buttons Near Code/Far Code, Near Const/Far Const, Near Data/Far Data.



```
model(processor)
-mmd processor
```

where:

processor Selects the instruction set the assembler uses in generating code for a specific member of the 80C196 processor family.

Abbreviation

md

Class

Primary control

Default

```
model(kb)
```

Description

This control allows you to specify which processor/instruction set you are using. The *cb*, *ea*, *ec*, *np*, *nt* and *nu* arguments of the *model* control also enable the assembler to recognize the *nearcode*, *farcode*, *nearconst*, *farconst*, *neardata*, and *fardata* controls.

Specify the *processor* as one of the following:

- 61 to select the 8096-61.
- 90 to select the 8096-90.

196	to select the 80C196KB. This argument to <code>model</code> is available for backward compatibility and is equivalent to specifying <code>kb</code> . For future compatibility, use the <code>model(kb)</code> control specification instead of <code>model(196)</code> .
bh	to select the 8096BH.
ca	to select the 80C196CA. Specifying <code>ca</code> is equivalent to specifying <code>kr</code> .
cb	to select the 80C196CB. This argument can have an extra suffix as described in the note below.
ea	to select the 80C196EA. This argument can have an extra suffix as described in the note below.
ec	to select the 80C196EC. This argument can have an extra suffix as described in the note below.
jq	to select the 80C196JQ. Specifying <code>jq</code> is equivalent to specifying <code>kr</code> .
jr	to select the 80C196JR. Specifying <code>jr</code> is equivalent to specifying <code>kr</code> .
js	to select the 80C196JS. Specifying <code>js</code> is equivalent to specifying <code>kr</code> .
jt	to select the 80C196JT. Specifying <code>jt</code> is equivalent to specifying <code>kr</code> .
jv	to select the 80C196JV. Specifying <code>jv</code> is equivalent to specifying <code>kr</code> .
kb	to select the 80C196KB. Specifying <code>kb</code> is equivalent to specifying <code>196</code> .
kc	to select the 80C196KC.
kd	to select the 80C196KD.
kl	to select the 80C196KL. Specifying <code>kl</code> is equivalent to specifying <code>kr</code> .
kq	to select the 80C196KQ. Specifying <code>kq</code> is equivalent to specifying <code>kr</code> .

<code>kr</code>	to select the 80C196KR.
<code>ks</code>	to select the 80C196KS. Specifying <code>ks</code> is equivalent to specifying <code>kr</code> .
<code>kt</code>	to select the 80C196KT. Specifying <code>kt</code> is equivalent to specifying <code>kr</code> .
<code>lb</code>	to select the 80C196LB.
<code>mc</code>	to select the 80C196MC.
<code>md</code>	to select the 80C196MD.
<code>mh</code>	to select the 80C196MH.
<code>np</code>	to select the 80C196NP. This argument can have an extra suffix as described in the note below.
<code>nt</code>	to select the 80C196NT. This argument can have an extra suffix as described in the note below.
<code>nu</code>	to select the 80C196NU. This argument can have an extra suffix as described in the note below.




The `cb`, `ea`, `ec`, `np`, `nt` and `nu` arguments of the `model` control can have an additional suffix. Without a suffix, specifying `xx` is the same as specifying `xx-c`, where `xx` is one of `cb`, `ea`, `np`, `nt` or `nu`. The following six suffixes are possible:

<code>xx-c</code>	to select the compatible mode and to use near code addressing and near data/near const addressing.
<code>xx-cn timer</code>	to select the compatible mode and to use near code addressing and near data/far const addressing.
<code>xx-cf</code>	to select the compatible mode and to use near code addressing and far data/far const addressing.
<code>xx-e</code>	to select the extended mode and to use far code addressing and near data/near const addressing.
<code>xx-en timer</code>	to select the extended mode and to use far code addressing and near data/far const addressing.
<code>xx-ef</code>	to select the extended mode and to use far code addressing and far data/far const addressing.

Example

```
asm196 file1.a96 model(nt)
```



nearcode	nearconst	neardata
farcode	farconst	fardata

nearcode/farcode

Function

Specify code space configuration for 24-bit models.

Syntax



Select the EDE | CPU Model... menu item. Select the Near Code or Far Code radio button.



<code>nearcode</code>		<code>farcode</code>
<code>-nc</code>		<code>-fc</code>

Abbreviation

<code>nc</code>		<code>fc</code>
-----------------	--	-----------------

Class

Primary control

Default

`nearcode`

Description

`nearcode` and `farcode` specify code space configuration for a member of the 24-bit 80C196 family. `nearcode` specifies that the microcontroller is configured in compatible mode. `farcode` specifies that the microcontroller is configured in extended mode. These controls must be preceded by a 24-bit model control. If you specify a 24-bit model without a `nearcode` or `farcode` control, the processor code mode is determined according to the `near` or `far` attribute in the first `cseg` directive having a `near` or `far` attribute. If there is no `cseg` directive with a `near` or `far` attribute, then the default `cseg` attribute is `near`.

Example

```
asm196 file1.a96 model(nt) fc
```



model control

nearconst/farconst

Function

Specify constant space configuration for 24-bit models.

Syntax



Select the EDE | CPU Model... menu item. Select the Near Const or Far Const radio button.



<code>nearconst</code>		<code>farconst</code>
<code>-nk</code>		<code>-fk</code>

Abbreviation

`nk` | `fk`

Class

Primary control

Default

`nearconst`

Description

`nearconst` and `farconst` specify the constant space configuration for the 24-bit 80C196 family of microcontrollers. `nearconst` specifies that all data, unless otherwise indicated, reside in the first 64 kilobytes of the address space. `farconst`, on the other hand, means that all data, unless otherwise specified, are located in the 16-megabytes address space. The assembler uses these controls in the `kseg` directive when the `near` or `far` attribute is omitted. When `nearconst` is in effect, then the `near` segment attribute is assumed, otherwise the `far` attribute is assumed. These controls must be preceded by a 24-bit `model` control.

Example

```
asm196 file1.a96 model(nt) nk
```



`model` control

neardata/fardata

Function

Specify data space configuration for 24-bit models.

Syntax



Select the EDE | CPU Model... menu item. Select the Near Data or Far Data radio button.



neardata		fardata
-nd		-fd

Abbreviation

nd		fd
----	--	----

Class

Primary control

Default

neardata

Description

neardata and fardata specify the data space configuration for the 24-bit 80C196 family of microcontrollers. neardata specifies that all data, unless otherwise indicated, reside in the first 64 kilobytes of the address space. fardata, on the other hand, means that all data, unless otherwise specified, are located in the 16-megabytes address space. The assembler uses these controls in the dseg and odseg directive when the near or far attribute is omitted. When neardata is in effect, then the near segment attribute is assumed, otherwise the far attribute is assumed. These controls must be preceded by a 24-bit model control.

Example

```
asm196 file1.a96 model(nt) nd
```



model control

object

Function

Assigns a name to the object file produced by the assembler.

Syntax



Select the EDE | Assembler Options | Project Options... menu item. Add the control to the Additional options field in the Misc tab.



```
object [(filename)] | noobject
-o filename         | -o-
```

where:

filename is the name assigned to the object file, including its full path.

Abbreviation

oj | nooj

Class

Primary control

Default

```
object(sourcefile.obj)
```

Description

Use this control to assign a name to the object file produced by the assembler, as specified by *filename*. If you do not specify a filename, the assembler uses the default name for the object file, *sourcefile.obj*, where *sourcefile* is the filename being assembled. The *noobject* control suppresses the production of the object file. Use *noobject* if you only want to debug your source program for syntax and not produce an object file.

Example

This invocation line tells the assembler to name the object file *module.obj*.

```
asm196 file1.a96 oj(module1.obj)
```

omf

Function

Specifies OMF96 version.

Syntax



Select the EDE | Assembler Options | Project Options... menu item. Select an OMF96 Version radio button in the Object tab.



`omf (n)`
`-omf : n`

where:

`n` is the number representing the OMF96 version:

- 0 - OMF96 V2.0
- 1 - OMF96 V3.0
- 2 - OMF96 V3.2 (default)

Abbreviation

`omf`

Class

Primary control

Default

`omf (2)`

Description

Use this control is used to specify the OMF96 version to generate. In a previous version of the assembler you could use the control `oldobject` to specify OMF96 V2.0. Also some users were advised to use the internal control `oo1`. The two controls are now combined in one control, `omf`.

Example

This invocation line tells the assembler to use the old OMF96 version V2.0.

```
asm196 file1.a96 omf(0)
```

optimize

Function

Perform some optimizations.

Syntax



Select the EDE | Assembler Options | Project Options... menu item. Enable or disable the Perform register optimization check box in the Object tab.



optimize		nooptimize
-O		-O-

Abbreviation

ot		noot
----	--	------

Class

Primary control

Default

nooptimize

Description

When the optimize control is used, some register optimization is performed.

Example

```
asm196 file1.a96 optimize
```

optionalcolon

Function

Make the colon following a label declaration optional.

Syntax



Select the EDE | Assembler Options | Project Options... menu item. Enable or disable the Treat the colon of a label as optional check box in the Misc tab.



<code>optionalcolon</code>		<code>nooptionalcolon</code>
<code>-oc</code>		<code>-oc-</code>

Abbreviation

<code>oc</code>		<code>nooc</code>
-----------------	--	-------------------

Class

Primary control

Default

`nooptionalcolon`

Description

This control will make the colon following a label declaration optional.

Example

```
asm196 file1.a96 oc
```


pagelength

Function

Specifies the listing's maximum number of lines per page.

Syntax



Select the EDE | Assembler Options | Project Options... menu item. Enter the page length in the Page length (lines per page) field in the Listing tab.



```
pagelength(n)
```

```
-pl n
```

where:

n is the number of lines desired per page.

Abbreviation

pl

Class

Primary control

Default

```
pagelength(60)
```

Description

Use this control to set the listing's maximum number of lines per page. The header lines are counted toward the total. The minimum pagelength is 10 lines per page. Specifying a number less than 10 causes an invocation line error. If no pagelength is specified, the default is 60 lines per page.

Example

This invocation line tells the assembler to list 70 lines per page.

```
asm196 file1.a96 pl(70)
```

pagewidth

Function

Specifies the listing's maximum number of characters per line.

Syntax



Select the EDE | Assembler Options | Project Options... menu item. Enter the number of *characters* in the Page width (characters per line) field in the Listing tab.



```
pagewidth(n)  
-pw n
```

where:

n is the maximum number of characters desired per line.

Abbreviation

pw

Class

Primary control

Default

```
pagewidth(120)
```

Description

Use this control to set the listing's maximum number of characters per line. The pagewidth can range from 72 to 255 characters per line. Specifying a number less than 72 or greater than 255 causes an invocation-line error. The assembler wraps the lines that exceed the right margin setting to column 35 of the line directly below. Source lines of more than 255 characters appear in the listing as truncated to 255 characters and are accompanied by an error message to that effect.

Example

This invocation line tells the assembler to print 80 characters per line.

```
asm196 file1.a96 pw(80)
```

print

Function

Sets the name of the listing file to the specified filename.

Syntax



Select the EDE | Assembler Options | Project Options... menu item. Enable or disable the Generate listing file (.lst) check box in the Listing tab.



```
print [(filename)] | noprint
-p filename         | -p-
```

where:

filename is the desired name of the listing file, including its full path.

Abbreviation

```
pr    |  nopr
```

Class

Primary control

Default

```
print(sourcefile.lst)
```

Description

Use this control to set the name of the listing file. If you do not specify a filename, the assembler uses the default filename for the listing file, *sourcefile.lst*, where *sourcefile* is the filename being assembled. The noprint control suppresses the production of a listing file.

Example

This invocation line tells the assembler to produce a listing file called *main.lst*.

```
asm196 file1.a96 pr(main.lst)
```



errorprint control
symbols control
xref control

relaxedif

Function

Specifies whether an undefined symbol in an IF is allowed.

Syntax



Select the EDE | Assembler Options | Project Options... menu item. Enable or disable the Allow undefined symbol in an IF check box in the Misc tab.



<code>relaxedif</code>		<code>norelaxedif</code>
<code>-ri</code>		<code>-ri-</code>

Abbreviation

<code>ri</code>		<code>nori</code>
-----------------	--	-------------------

Class

Primary control

Default

`norelaxedif`

Description

The `relaxedif` control specifies whether an undefined symbol in an IF statement is allowed. For example: the statement `IF SYMBOL` will normally result in an error if `SYMBOL` is not defined. However, when specifying the `relaxedif` option the IF statement will act as an `IFDEF` and the statement is correct. `IF SYMBOL` will be `FALSE`.

Example

```
asm196 file1.a96 ri
```



set control

save/restore

Function

Saves or restores the current setting of some general controls.

Syntax

save		restore
-S		-R

Abbreviation

sa		rs
----	--	----

Class

General control

Description

Use the `save` control to save the current settings of the `list`, `cond`, and `gen` controls if a temporary setting is desired for a section of the source.

Use `restore` then to reinstate the saved setting where desired. An error occurs if the assembler encounters a `restore` with no control saved or if the number of `restore` operations exceeds the number of `save` operations. You can nest these controls nine deep.

Example

The following lines shows how `save` and `restore` work:

```
$list
$save nolist      ; This line is listed
$restore          ; This line is not listed
$nolist save      ; This line is listed
$restore list     ; This line is not listed
TRUE equ 1        ; This line is listed
```

searchinclude

Function

Specifies or suppresses search paths for include files.

Syntax



Select the `EDE | Directories...` menu item. Add one or more directory paths to the `Include Files Path` field.



```
searchinclude(pathprefix [,...]) | nosearchinclude
-I pathprefix | -I-
```

where:

pathprefix is a string of characters that the assembler prepends to an include file's filename. This string must include any special characters that the operating system expects in a path prefix.

Abbreviation

si | nosi

Class

General control

Default

nosearchinclude

Description

Use this control to specify a list of possible path prefixes for include files.

Each *pathprefix* argument is a string that, when concatenated to a filename, specifies the relative or absolute path of a file (including a device name and directory name, if necessary). The assembler tries each prefix in the order in which they are specified, until a legal filename is found. If a legal filename is not found, the assembler issues an error.

An include file is a source text file specified with the `include` control in the assembler invocation or on a control line in the source text. The contents of each include file are inserted into the source text during preprocessing.

When searching for a file specified with the `include(filename)` control, the assembler tests the path prefixes in the following order:

1. The current directory (no prefix).
2. The directories specified by the `searchinclude` list.
3. The directories in the `C196INC` environment variable, if defined.
4. The `include` directory, one directory higher than the directory containing the **asm196** binary. For example, **asm196** is installed in `/usr/local/c196/bin`, then the directory searched for include files is `/usr/local/c196/include`.

The number of directories searched for include files are not limited by the assembler.

The `searchinclude` and `nosearchinclude` controls affect only the subsequent source text and remain in effect until the assembler encounters a contradictory control. Specifying the `searchinclude` control more than once adds to the search path prefix list. Specifying the `nosearchinclude` control after the `searchinclude` control suppresses the search path prefix list until the next occurrence of `searchinclude`. You can specify these controls on the invocation line or on a control line.

Example

This example demonstrates the paths searched by the assembler when a `C196INC` environment variable is defined and the `searchinclude` control is specified.

The `C196INC` environment variable is defined as follows:

PC:

```
set C196INC=\proj001;\proj001\headers
```

UNIX:

```
setenv C196INC /proj001:/proj001/headers
```

The `searchinclude` control is specified in the assembler invocation as follows:

```
searchinclude (/proj001/test_inc,/generic/stubs)
```


The source text contains the following `include` control:

```
include(t_locate.inc)
```

The assembler is invoked in the root directory and executed from `/usr/local/c196/bin`. The assembler searches for filenames in the following order (UNIX notation is used):

1. The current directory: `/t_locate.inc`
2. From the `searchinclude` control: `/proj001/test_inc/t_locate.inc`
3. From the `searchinclude` control: `/generic/stubs/t_locate.inc`
4. From `C196INC`: `/proj001/t_locate.inc`
5. From `C196INC`: `/proj001/headers/t_locate.inc`
6. From the relative path: `/usr/local/c196/include/t_locate.inc`



`include`

set/reset

Function

Set or reset a symbol.

Syntax

<code>set (name)</code>		<code>reset (name)</code>
<code>-D name</code>		<code>-U name</code>

where:

name is the name of a symbol that will have the value TRUE in conditional expressions.

Abbreviation

<code>se</code>		<code>re</code>
-----------------	--	-----------------

Class

General control

Description

When symbols are used in the expression of an IF directive, the symbols used must be defined by the programmer using the `set` and `reset` controls. When the `set` control is used to define a symbol, the value of the symbol is TRUE, and symbols defined with `reset` have value FALSE. All undefined symbols have the value FALSE. Therefore, it is not necessary to define all symbols, but only those symbols that have value TRUE.

Example

The following lines shows how `set` and `reset` work:

```
$set(I_AM_TRUE) reset(I_AM_FALSE)

if (I_AM_TRUE or I_AM_FALSE)
...
else
...
endif
```

signedoper

Function

Changes default behavior of some operators to signed.

Syntax



Select the EDE | Assembler Options | Project Options... menu item. Enable or disable the Treat relational operators (LE, GT, etc) as signed check box in the Object tab.



<code>signedoper</code>		<code>nosignedoper</code>
<code>-so</code>		<code>-so-</code>

Abbreviation

<code>so</code>		<code>noso</code>
-----------------	--	-------------------

Class

Primary control

Default

`nosignedoper`

Description

In a previous version of the assembler all relational operators like 'LE', 'GT', et cetera, all used unsigned comparison of the 32-bit operand values. However, the new unsigned relational operators should be used for this, and not the old relational operators. To get the proper behavior you need to include the control `signedoper` on the command line. If the control is not used, the assembler is compatible with the old style of unsigned comparison.

The control `signedoper` also changes the default unsigned behavior of operators like addition, multiplication, etc. to signed behavior.

Example

```
asm196 test.a96 so
```

source

Function

Specifies if line number information is based on actual line number or statement number.

Syntax



Select the EDE | Assembler Options | Project Options... menu item. Enable or disable the Base line information on source (not statement number) check box in the Object tab.



source		nosource
-sc		-sc-

Abbreviation

sc		nosc
----	--	------

Class

General control

Default

nosource

Description

This control will generate line number information based on the actual line numbers in the source file or the statement number (default). Use the `linedebug` control to enable generation of line number information.

Example

```
asm196 test.a96 linedebug source
```



debug control
linedebug control

symbols

Function

Causes symbol-table information to be included in the listing file.

Syntax



Select the EDE | Assembler Options | Project Options... menu item. Enable or disable the Include symbol information check box in the Listing tab.



<code>symbols</code>		<code>nosymbols</code>
<code>-sb</code>		<code>-sb-</code>

Abbreviation

<code>sb</code>		<code>nosb</code>
-----------------	--	-------------------

Class

Primary control

Default

`symbols`

Description

Use this control to include the symbol table information in the listing file. Do not specify `noprint` with this control because `noprint` suppresses the production of the listing file. The `nosymbols` control suppresses the inclusion of symbol table information in the listing file.



`print` control
`xref` control

subtitle

Function

Sets the page header subtitle of the listing file.

Syntax



Select the EDE | Assembler Options | Project Options... menu item. Enter the *subtitle* in the Subtitle of listing file field in the Listing tab.



<code>subtitle('string')</code>		<code>nosubtitle</code>
<code>-st 'string'</code>		<code>-st-</code>

where:

string is a set of any valid ASCII characters.

Abbreviation

<code>st</code>		<code>nost</code>
-----------------	--	-------------------

Class

General control

Default

`nosubtitle`

Description

Aside from the `title` control it is also possible to use a subtitle for a number of pages. The control `subtitle` must be used for this feature. The given subtitle is used on the next generated page in the list file. To get a proper subtitle for each page, use the following combination of controls in the assembler source file:

```
$SUBTITLE('This is a subtitle')
$EJECT
```

When the control is used, you must enclose the string in either apostrophes (' ') or quotation marks (" "). The same type of mark must be used on both sides of the string. Thus `subtitle('string')` is not a valid entry. If literal quotation marks appear in the title, the other type of mark must be used as the string delimiters.

To suppress the use of subtitles, use `nosubtitle`. This will turn off subtitles of the next generated page in the list file.

Example

This example shows how to include an apostrophe (') in the desired subtitle.

```
asm196 file1.a96 st("TASKING's module")
```

title

Function

Sets the page header title of the listing file.

Syntax



Select the EDE | Assembler Options | Project Options... menu item. Enter the *title* in the Title of listing file field in the Listing tab.



```
title('string')  
-t 'string'
```

where:

string is a set of any valid ASCII characters.

Abbreviation

tt

Class

General control

Default

```
title(module_name)
```

Description

Use this control to assign the title of the listing. If you do not specify a title, the assembler uses the module name as the default title. When the control is used, you must enclose the string in either apostrophes (' ') or quotation marks (" "). The same type of mark must be used on both sides of the string. Thus `title('string')` is not a valid entry. If literal quotation marks appear in the title, the other type of mark must be used as the string delimiters.

Example

This example shows how to include an apostrophe (') in the desired title.

```
asm196 file1.a96 tt("TASKING's module")
```


xref

Function

Causes a symbol cross-reference listing to be included in the listing file.

Syntax



Select the `EDE | Assembler Options | Project Options...` menu item. Enable or disable the `Include a cross-reference check` box in the `Listing` tab.



`xref | noxref`

Abbreviation

`xr | noxr`

Class

Primary control

Default

`noxref`

Description

Use this control to include a symbol cross-reference listing in the listing file. The cross-reference listing shows the line numbers on which a particular symbol appear. See Chapter 3 for the format of the cross-reference listing. You cannot specify `xref` in the same assembly with either `nosymbols` or `noprint`. Specifying `noxref` suppresses the inclusion of a cross-reference table in the list file.

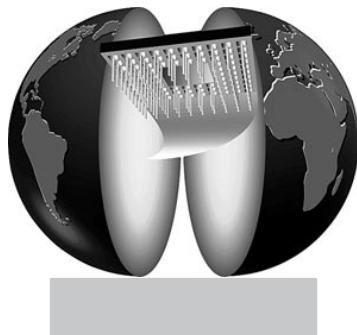


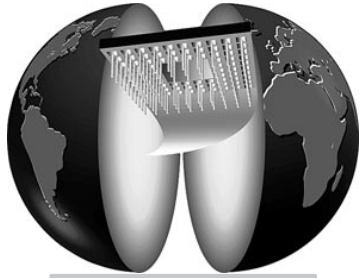
`print control`
`symbols control`
 cross-reference listing (Chapter 3)

CHAPTER

ASSEMBLY LANGUAGE

5





5

CHAPTER

5.1 INTRODUCTION

This chapter provides a detailed presentation of the elements of assembly language and statement syntax. It also describes the program format and the different segment types used by the assembler.

The basic unit of the assembly language is the statement. The three types of statements are instructions, directives, and controls. Chapter 6 describes the assembler directives and Chapter 4 discusses the controls. This chapter explains the types of terms that appear in a statement and the order in which they must appear.

5.2 ASSEMBLY LANGUAGE COMPONENTS

An assembly language statement consists of fields, and the correct order of these fields is defined by the statement format. Only the correct operands appear in the given field, and each operand must be composed of valid elements. The character set defines the set of valid elements that can appear in any term. If any field contains invalid characters or if the fields are in an incorrect order, the assembler generates an error.

5.2.1 CHARACTER SET

ASM196 uses the ASCII character set. All ASCII characters are valid, but some of them have special meanings for the assembler, unless they appear in a comment or in a string. Numbers, delimiters, reserved words, and symbols all have special meanings and are discussed in the following sections.

5.2.2 NUMBERS

ASM196 supports modulo- 2^{32} unsigned arithmetic, meaning the numbers are unsigned 32-bit integers. Numbers must be presented to the assembler as a series of valid digits followed by a base specifier. Table 5-1 shows the base specifiers and their valid digits. Case is not significant.



When no base specifier is explicitly stated, the decimal base is used.



Number	Base
binary	the digits 0 to 1, followed by B (or b).
octal	the digits 0 to 7, followed by O or Q (or o or q).
decimal	the digits 0 to 9, followed by D (or d) or no base specifier.
hexadecimal	the digits 0 to 9, the supplementary hexadecimal A to F (or a to f), followed by H (or h). The first digit must be in the range of 0 to 9. Use a leading 0 if the number begins with A to F.

Table 5-1: ASM196 numbers and bases

5.2.3 LONG CONSTANTS

Long constants are 32-bit unsigned numbers. Use the `dc1` directive to declare a long constant. See Chapter 6 for a description of the `dc1` directive.

5.2.4 FLOATING POINT NUMBERS

Use the `dcr` directive, discussed in Chapter 6, to declare a floating point number. The format of a floating point number is:

`[sign]int_dec_digit[frac_dec_digit][E [sign][exp_dec_digit]]`

where:

- sign* is the optional sign (+ or -). If not specified, + is used.
- int_dec_digit* are the decimal digits representing the integer part of the floating point number.
- frac_dec_digit* are the decimal digits representing the fractional part of the floating point number.
- exp_dec_digit* are the decimal digits representing the exponent part of the floating point number. The exponent part is optional. If not specified or if no digit is entered, the exponent part is assumed to equal 1 (i.e., 10 to the 0th power). If the sign is not explicitly entered, it is assumed to be positive.

The maximum absolute value of a floating point number is $3.37\text{E}38$. The minimum absolute value is $1.17\text{E}-38$.

A floating point value occupies four contiguous memory bytes, which can be viewed as 32 contiguous bits. The bits are divided into fields as follows:

sign	exponent	fraction
(1 bit)	(8 bits)	(23 bits)

Where:

sign bit is 0 if the floating point value is positive or zero, or 1 if the floating point value is negative.

exponent field contains a value offset by 127. In other words, the actual exponent can be obtained from the exponent field value by subtracting 127. This field is all 0s if the floating point value is zero.

fraction field contains the binary digits of the fractional part of the floating point value, when it is represented in binary scientific notation (see the following). This field is all 0s if the floating point value is zero.

The byte with the lowest address contains the least-significant eight bits of the fraction field. The byte with the highest address contains the sign bit and the most-significant seven bits of the exponent field.



See the *80C196 Utilities User's Guide*, listed in *Related Publications*, for more information about floating point numbers.

The following examples illustrate these concepts.

Consider the following binary number, equivalent to the decimal value 10.25:

1010.01B

The period (.) in this number is a binary point. The same number can be represented as:

1.01001B * 23

This is binary scientific notation, with the binary point immediately to the right of the most significant digit. The digits 01001 are the fractional part, and 3 is the exponent. This value is represented as follows:

- The sign bit is 0, since the value is positive.
- The exponent field contains the binary equivalent of $127 + 3 = 130$.
- The leftmost digits of the fraction field is 01001, and the remainder of this field is all 0's.

The complete 32-bit representation is:

```
0 10000010 010010000000000000000000
```

and the contents of the four contiguous memory bytes is as follows:

highest address:	01000001
	00100100
	00000000
lowest address:	00000000

Note that the most significant digit is not actually represented, since by definition it is a 1 unless the floating point value is zero. If the floating point value is zero, the entire 32-bit representation is all 0's.

For a second example, consider the fraction $1/16$, or 0.0625. In binary, this is:

```
0.0001B
```

In binary scientific we have:

```
1.0000B * 2-4
```

The actual exponent, -4 , is represented as 123 (i.e., $127 - 4$), and the fraction field contains all 0's.

The largest possible value for a valid exponent field is 254, which corresponds to an actual exponent of 127. The largest possible absolute value for a positive or negative floating point value is therefore:

```
1.111111111111111111111111B * 2127
```

or approximately 3.37×10^{38}

The lowest permissible exponent field value for a non-zero floating point value is 1, which corresponds to an actual exponent of -126 . The smallest possible absolute value for a positive or negative floating point value is therefore:

$$1.0B * 2^{-126}$$

or approximately 1.17×10^{-38}

5.2.5 DELIMITERS

Delimiters separate and terminate operands in an ASM196 statement just as spaces serve to delimit words in an English sentence. The position of a delimiter is important because it can radically affect the meaning of a statement to the assembler. Table 5-2 shows the complete set of ASM196 delimiters and their definitions.

Character(s)	Definition
space	One or more spaces can be entered. Spaces serve as field separators or symbol delimiters.
,	Commas separate operands.
'	Apostrophes serve to delimit character strings.
()	Parentheses delimit expressions.
[]	Square brackets enclose base or index registers in instruction operands.
< >	Angle brackets enclose macro arguments.
%	The percent sign indicates that the following macro argument is to be evaluated as an expression before replacing the formal parameter.
!	The exclamation mark is used as a literalizer in macro processing.
&	The ampersand is used to separate formal parameters from adjacent text.
LF	The Line Feed (ASCII 0AH) is the line terminator.
CR	The Carriage Return (ASCII 0DH) can optionally precede the Line Feed.
HT	The Horizontal Tab is treated as a space.
;	The semicolon delimits the beginning of a comment.



Character(s)	Definition
::	Two successive semicolons mark the start of a comment in a macro definition. This comment is suppressed when the macro is expanded.
#	The number sign prefixes an immediate value in immediate addressing mode.
:	The colon separates the name from the rest of a statement.

Table 5-2: ASM196 delimiters

5.2.6 RESERVED WORDS

Reserved words are names that have a specific meaning in the assembly language so you cannot use them as symbols. Instruction mnemonics, assembler and macro directives, and the predefined symbol `stack` are all reserved words. If a reserved keyword is used, the following error is generated:

```
ERROR #74: Reserved keyword used in invalid context
```

Appendix D contains a list with all the reserved words used by the assembler.

5.2.7 PREDEFINED MACROS

Based on the `MODEL` control, the assembler sets/resets some predefined macros:

Name	Value
<code>_SFR_INC_</code>	name of the SFR include file
<code>_MODEL_xx_</code>	boolean to indicate if model <code>xx</code> is used
<code>_FAR_CODE_</code>	boolean to indicate far code
<code>_FAR_CONST_</code>	boolean to indicate far const
<code>_FAR_DATA_</code>	boolean to indicate far data
<code>_16_BITS_</code>	boolean to indicate a 16-bit model

Name	Value
<code>_24_BITS_</code>	boolean to indicate a 24-bit model
<code>_OMF96_VERSION_</code>	OMF version: 0 V2.0 1 V3.0 2 V3.2

Table 5-3: ASM196 predefined macros

The above macro symbols will not appear in the symbol table or in the resulting object file. The linker will search certain model specific directories for libraries and object files.

5.2.8 SYMBOLS

Symbols are names that you can define and use to represent memory addresses, constants, macros, etc. A symbol contains up to 31 characters. The first character must be a letter or a special character. The special characters are the question mark (?) and the underscore (_). The remaining characters can be letters, special characters, or the digits 0 to 9. Case is not significant within symbols. Thus, the assembler treats `io_control` the same as `IO_CONTROL` or `Io_CoNtRoL`. The special characters are useful in creating more readable symbol names, especially since the hyphen is not valid in a symbol name. Thus, `DATA_Buffer_START` or `DATA?Buffer?START` are more easily recognizable than is `DATABufferSTART`.

You can define a symbol by using a directive or by using the symbol as a label. You can use the following directives to define a symbol: `module`, `extrn`, `equ`, `set`, `macro`, `dcl`, `dcb`, `dcw`, `dcr`, `dscr`, `dsw`, `dsl`, `dsq`, and `dsb`. To define a symbol as a label, append a colon (:) after the symbol name.

Define each symbol only once, unless the symbol is defined by the `set` directive, in which case that symbol can be redefined only by another `set` directive. Symbols local to macros override other symbols defined outside the scope of the macro. See Chapter 7 for more information on the macro directive.

5.2.9 ASSEMBLER-GENERATED SYMBOLS

The assembler generates numeric symbols for each local macro symbol definition. The generated numeric symbols have the form `??dddd` where *d* is a decimal digit. For example, your program includes the following macro which declares `loop` and `eom` as local symbols:

```
block macro g1,g2,g3,g4
    local loop, eom
    ld reg1,g1
    ld reg2, #g2
loop: g3 reg3,[reg2]+
    g4 eom
    dbnz reg1,loop
eom:  endm
```

When you assemble your program with the `gen` control, the following output can be seen in the listing file:

```
test module main

rseg at 30h
    reg1: dsw 1
    reg2: dsw 1
    reg3: dsw 1
    size: dsw 1
    tab:  dsw 1

block macro g1,g2,g3,g4
    local loop, eom

    ld reg1,g1
    ld reg2, #g2
loop: g3 reg3,[reg2]+
    g4 eom
    dbnz reg1,loop
eom:  endm
```

```

cseg
    ld size, #3
    ld tab, #4
    clr reg3
    block size, tab, add, !;
        +1      ld reg1,size      ;macro expansion
starts
        +1      ld reg2, #tab
        +1      ??0001: add reg3,[reg2]+
        +1      ; ??0002
        +1      dbnz reg1,??0001
        +1      ??0002:          ; macro expansion
ends
    div reg3, #size
    block size, tab, cmp, bge
        +1      ld reg1,size      ; macro expansion
starts
        +1      ld reg2, #tab
        +1      ??0003: cmp reg3,[reg2]+
        +1      bge ??0004
        +1      dbnz reg1,??0003
        +1      ??0004:          ; macro expansion
ends

    end

```

Note that in the listing file, the `loop` and `eom` local symbols are replaced with `??0001` through `??0004`. These numbers represent the number of symbols created when the assembler expanded the macro. The most recent symbol name generated always indicates the total number of symbols created for all macro expansions.

Treat assembler-generated symbols the same way as reserved words to avoid duplication of symbol names.

5.2.10 GENERIC INSTRUCTIONS

In the **asm196** assembler generic instructions can be used to let the assembler decide what instructions can be used best. Generic instructions are available for jump and call instructions.

The generic call instruction (mnemonic `call`) will select the proper call instruction (`scall`, `lcall`) based on the offset to the label being called to the current address. When the offset ranges from -1024 to $+1023$ decimal, the `scall` instruction is generated. For offsets ranges between -32768 and $+32767$ decimal, the `lcall` instruction is generated. For offsets outside the latter range, we must use the `ecall` instruction and one of the 24-bit memory models. This instruction cannot be generated using the generic call instruction.

The generation of the correct instructions for the generic jump instructions is more complex. The generic jump instructions are:

16-bit memory model:

```
bbc  bbs  bc   be   bge
bgt  bh   ble  blt  bnc
bne  bnh  bnst bnv  bnvt
bst  bv   bvt  dbnz dbnzw
```

and

24-bit memory model:

```
ebbc ebbs ebc  ebe  ebge
ebgt ebh  eble eblt ebnc
ebne ebnh ebnstebnv ebnavt
ebst ebv  ebvt edbnzedbnzw
```

The normal jump instructions (except `sjmp`, `ljmp` and `ejmp`) can jump within the range of -128 to $+127$ decimal. When the offset for the generic jump is within this range, the corresponding normal jump instruction is generated. However, when the offset falls outside of this range, the generic jump instruction will be negated and an appropriate direct jump instruction is generated. Consider the following example:

```
.
.
jmp_label:      bge      my_label
.
.
my_label:
.
.
```

When the offset between `jmp_label` and `my_label` is not within the range of -128 to $+127$ decimal, the **asm196** assembler will generate object code as if the source code was given as:

```
.
.
jmp_label:      blt      tmp_label
                sjmp     my_label
tmp_label:
.
.
my_label:
.
.
```

The object code for the `sjmp` instruction will only be generated if the offset is within the range -1024 to 1023 decimal. The offsets for each of the jumps are (in decimal):

```
sjmp -1024      +1023
ljmp -32768     +32767
ejmp -8388608   +8388607
```

The `ejmp` instruction will only be generated when a 24-bit memory model is used.

5.2.11 ADDITIONAL MNEMONICS

In the **asm196** assembler you can use additional mnemonics as a substitute for some jump instructions and their corresponding generic instructions (16-bit and 24-bit). The additional mnemonics are listed in Table 5-4. Note that you can only use the additional mnemonics if the `extra_mnem` control is in effect.

Original Mnemonics			Description	Equivalent Extra Mnemonics		
JH	BH	EBH	> unsigned	JGTU	BGTU	EBGTU
JNH	BNH	EBNH	<= unsigned	JLEU	BLEU	EBLEU
JC	BC	EBC	>= unsigned	JGEU	BGEU	EBGEU
JNC	BNC	EBNC	< unsigned	JLTU	BLTU	EBLTU
JE	BE	EBE	=	JZ	BZ	EBZ
JNE	BNE	EBNE	<>	JNZ	BNZ	EBNZ



Original Mnemonics			Description		Equivalent Extra Mnemonics		
JGT	BGT	EBGT	>	signed	JGTS	BGTS	EBGTS
JLE	BLE	EBLE	<=	signed	JLES	BLES	EBLES
JGE	BGE	EBGE	>=	signed	JGES	BGES	EBGES
JLT	BLT	EBLT	<	signed	JLTS	BLTS	EBLTS

Table 5-4: Additional mnemonics

5.2.12 MIXED ADDRESSING MODES

The **asm196** assembler sometimes uses two different addressing modes for one and the same variable: long/short-indexed and register direct addressing. When the address is within the window range (and windowing is used) at run-time two different physical memory locations are addressed. To avoid this problem the following syntactical additions are made to the assembler:

Whenever a base register (either implicit or explicit) is used in an indexed addressing mode you can use the following modifiers to force some form of addressing mode:

- @r register direct
- @e extended-indexed
- @l long-indexed
- @s short-indexed

To illustrate the above, consider the following example:

```
RSEG
r0 EQU 0H:WORD
r1: DSW 1
reg: DSW 1

DSEG
var: DSW 1

CSEG
LD reg, r0           ; register direct
LD reg, var          ; long-indexed via r0
LD reg, var[r1]      ; long-indexed via r1
```

```

; If we know that 'var' is within the range -128 to 127, we
; can force short-indexed addressing.

LD    reg, var@s      ; short-indexed via r0
LD    reg, var@s[r1]  ; short-indexed via r1

; If we know that 'var' is within the register file we can
; use 'var' as a register.

LD    reg, var@r      ; register direct

; However it is not possible to use the '@r' when a base
; register is explicitly given.

LD    reg, var@r[r1]  ; ERROR
END

```

The above example shows only some of the possible uses of the @-letter suffix. In general, the following syntax can be used in the operands:

```

reg, [base-reg]           ; indirect, no @-letter
reg, #byte                ; immediate, no @-letter
reg, offset @-letter [base-reg] ; indexed, @-letter

```

When three-operand instructions are used, only one indexed addressing is allowed. This indexed addressing may contain one of the @-letter suffixes to force some other addressing mode.

5.2.13 LOCATION COUNTER

Use the dollar sign (\$) to reference the current location within the active segment of the program.

5.2.14 STRINGS

Strings are a sequence of printable ASCII characters delimited by a beginning and an ending apostrophe ('). Characters that appear within the string delimiters are treated literally by the assembler and do not possess their usual special meaning. Thus, 'STACK' is the character string S-T-A-C-K and not the reserved word `stack`, 'FFH' is the character string F-F-H and not the hexadecimal number FF, and '\$' is the literal dollar sign and not the location counter.

If the apostrophe is itself to be the initial or the terminal character of the string, two apostrophes must appear before the initial apostrophe string. For example, type `''tis the season'` to delimit the string `'tis the season`. Entering only one apostrophe does not preserve the initial apostrophe as part of the string.

Within strings, case is significant. Thus, `'Bigfoot'` is not the same string as `'BIGFOOT'`.

You can use strings as operands to the `dcb` directive, discussed in Chapter 6, and as operands in an expression. When used in a `dcb` directive, the string length can be anywhere from zero (i.e., a null string) to 255. When a string is used as an operand in an expression, the string can contain only one or two characters.

5.3 EXPRESSIONS AND BASIC OPERANDS

In the new definition of the object format, OMF96 v3.2, it is possible to include complete expressions in the object file. This enables us to have a better control over the various expressions that can be used. One simple example of a new type of expression that is allowed is:

```
DSEG
  EXTRNA, B
Var EQU A + B
```

To maintain compatibility with previous versions of the assembler, use the `omf` control or `-omf` option to select the version of the object format that is used. To enable the new expression syntax, use `omf (2)` in the invocation syntax of the assembler.

Basic operands are numbers, user-defined symbols, the predefined symbol (\$) for the active location counter, the predefined variable (`stack`) for the bottom of the stack, and strings of one to four characters. An expression consists of one of the following items:

- a single basic operand
- a single expression acted upon by a prefix unary operator
- two expressions operated on by an infix operator

A prefix unary operator is meaningful only if it precedes a single basic operand. Thus, `-25` is meaningful, as is `NOT a`. The text `/32` and `> limit` are not meaningful. In order for the latter two operations to be meaningful, two basic operands are required. Thus, `166/32` is meaningful, as is `total > limit`.

Use expressions to define constants. Expressions are either defined at assembly time (absolute expressions) or at relocation time (termed relocatable expressions). When any expression is defined, each operand in the expression is evaluated as a 16-bit unsigned integer, an integer in the range of 0 to 0FFFFH.

5.3.1 BASIC OPERANDS

The five kinds of basic operands are:

Symbols	User-defined symbols can represent a memory address, a constant, or an external name that is defined in some other module as either a constant or an address.
Numbers	Scalar quantities are represented as numbers. You can express them in binary, octal, decimal or hexadecimal. When no base is explicitly stated, decimal is used by default.
\$	The dollar sign (\$) is a predefined symbol that represents the present value of the active location counter.
stack	The predefined variable <code>stack</code> represents the bottom of the stack.
Strings	Only one- and two-character strings can serve as operands in expressions. The assembler interprets a one-character string as a one-byte constant and sets it equal to the ASCII value of the character. The assembler interprets a two-character string as a word. The low-order byte of the word is set equal to the ASCII value of the first character of the string, and the high-order byte is set equal to the ASCII value of the second character of the string. The assembler interprets a three or four-character string as a double word.

5.3.2 ATTRIBUTES OF EXPRESSION OPERANDS

Every expression has three attributes: type, relocatability, and value. Each attribute's description is as follows.

TYPE ATTRIBUTE

The type attribute states the type of segment (`register`, `code`, `data`, `stack`, `overlay` or `null`) to which the operand belongs. For example, an operand defined in the code segment is of type `code`. For each of the five kinds of basic operands and for unary and binary expressions, type is defined according to the following rules:

Symbols

For user-defined symbols, the type depends upon whether the symbol is defined as a label or via a directive such as `equ`, `set`, or `extrn`.

If you define a symbol as a label or with the `extrn` directive, the type is the type of the active segment. For example, if the active segment is `code` and you declare an external symbol, the type of the symbol is `code`.

If you define a symbol via a `set` or `equ` directive, its type is defined according to the following rules:

1. If the segment type of the expression on the right-hand side of `set` or `equ` is of type `null` and the new `copyattr` control is set, the segment type of the current segment is used.
2. For all other segment types of the expression or when the `copyattr` control is not set, the segment type of the expression is assigned to the symbol being defined.

For example (when invoked with **asm196 -ca source_file**):

```

13   SET      5H          ; type = NULL

      DSEG
      EXTRN   e1:WORD     ; type = DATA
11   EQU      10H         ; type = DATA REL
lab:  DSW      1

      RSEG
12   EQU      12H         ; type = REGISTER
14   EQU      lab         ; type = DATA REL

      CSEG
      EXTRN   e2:BYTE     ; type = CODE
      END

```

Numbers and strings

For numbers and for strings the segment type is null. Strings are treated like numbers when the length of the string is less or equal to four characters.

The location counter symbol (\$)

For \$, the type is the type of the active segment.

Bottom of the stack (stack)

For stack, the type is always stack.

Unary and Binary Operators

Expressions resulting from unary operations (omf(0) and omf(1)):

+ is the only unary operation that can be applied to a memory address. The type of the resulting expression is the type of the memory address. When any unary operation is applied to numbers, the type of the resulting expression is always null.

Expressions resulting from binary operations (omf(0) and omf(1)):

The only valid expressions containing a number and a memory address are the following:

```

memory_address - number
memory_address + number
number + memory_address

```

The resulting type is always that of the memory address.

When any binary operation is applied to two numbers, the resulting type is always `null`.

The only binary operations you can apply to two memory addresses are subtraction and comparison. Note that both memory addresses must belong to the same segment (i.e., they both have the same type and relocatability). The resulting expression is always of type `null`.

When an expression is the result of a binary or unary operation, the type attribute is determined using the following rules (`omf(2)`):

1. When the unary plus '+' operator is used, the type attribute of the expression is equal to the type attribute of the operand.
2. When any unary operator, other than unary plus, is used, the type attribute of the expression is `null`.
3. When the addition operator is used, the type attribute is defined using the following rules:
 - a. When both operands have type `null`, the expression will also be of type `null`.
 - b. When neither operands are of type `null`, the expression will be of type `null`.
 - c. When one operand is of type `null` and the other operand is not, the expression will have the type of the operand that is not of type `null`.
4. When the subtraction operator is used, the type attribute is defined using the following rules:
 - a. When the operand being subtracted is not of type `null`, the expression will be of type `null`.
 - b. When the operand being subtracted is of type `null`, the expression will have the type of the operand that is subtracted from.
5. For all other binary operators other than addition and subtraction, the expression will be of type `null`.

The above rules are applied to each sub-expression to determine the type attribute of complex expressions.

RELOCATABILITY ATTRIBUTE

The relocatability attribute determines whether you declared the operand as absolute, relocatable, or external.

Expressions can be absolute, relocatable, external, or complex. The type of the expressions are included in the assembler list file using the following indicating letters:

R Relocatable expression

E External expression

C Complex expression

The value of an expression or operand depends on the relocatability attribute, and is based on the following rules:

1. **ABSOLUTE:** The value of the expression or operand is fixed and there is no need for further evaluation. After the linker/locater phase, the value remains the same regardless of the location in memory. Examples of absolute expressions are numeric constants.
2. **RELOCATABLE:** The value defines the offset relative to the beginning of the current segment. After the linker/locater phase the value is incremented with the base address of the corresponding segment.
3. **EXTERNAL:** The value is unknown in the current module, except for the name of the symbol. The linker/locater will search among the known public symbols for the value of the external.
4. **COMPLEX:** The value is unknown in the current module, and consists of an expression with one or more external symbols. The complete expression tree is stored in the object file. The linker/locater will evaluate the expression to determine the value (which should be ABSOLUTE).

VALUE ATTRIBUTE

The value attribute is always a 32-bit two's-complement number. If the type is `null` (pure number), the value of the expression is a typeless number. If the segment is anything other than `null`, the value is a memory address. The precise address cannot be defined until link-time in the case of relocatable expressions. All other addresses can be evaluated by the assembler as absolute addresses.

5.3.3 ABSOLUTE EXPRESSIONS

Absolute expressions are expressions that are evaluated during assembly time and they can appear anywhere in the program. Table 5-5 lists the valid operators for absolute expressions according to their precedence, the order in which they are evaluated.

Operator(s)	Function
nul	test argument in the macro
**	calculate the power of two operands
MSW, LSW HIGH, LOW	mask some part of an unsigned 32-bit value. MSW selects the most significant word, LSW the least significant word, HIGH the top 8 bits of the LSW, and LOW the lower 8 bits of the LSW
*, /, mod, shr, shl	multiply, divide, mod (take remainder only), shift right, shift left
+, -	addition, subtraction (unary and binary)
eq (=), ne (<>), gt (>), ge (>=), lt (<), le (<=) ugt, uge, ult, ule	relational operators and their symbolic representations within parentheses. equal, not equal, greater than, greater than or equal, less than, less than or equal
NOT	logical inversion (bit by bit)
AND	logical conjunction (bit by bit)
OR, XOR	logical disjunction and exclusive OR (bit by bit)

Table 5-5: Valid operators for ASM196 absolute expressions listed by precedence (highest to lowest)

You can use the nul unary operator in a macro definition to test for the existence of a macro argument. If the actual argument is not defined, the operator returns TRUE, and FALSE otherwise. This kind of testing can also be accomplished by using the new IF conditional directives. See Chapter 7 for more information on macro definitions.

Relational operators and nul produce a result that is interpreted as being either logical true (represented by 0FFFFH) or logical false (represented by 0H).

The four relational operators `lt`, `le`, `gt` and `ge` can either compare unsigned or signed values depending on the `signedoper` control. The four operators `ult`, `ule`, `ugt` and `uge` always do an unsigned comparison of its operands.

An operator composed of letters, such as `NOT`, `XOR`, `MSW`, or `LSW`, must be separated from an adjacent symbol by at least one space. Thus, for the operator `XOR` and the symbol `size`, `size XOR 1` or `size XOR 1` are both valid, while `sizeXOR 1` and `size XOR1` are not and result in an error.

The `MSW` and `LSW` operators can be used to get the lower or upper 16 bits of a 32-bit value. Here is an example that uses both:

```
$model(NT)
    RSEG
ind_addr:  DSL 1           ; indirect address of routine
           CSEG AT 0FF2080h
           ; ...
           LD 1CH,ind_addr ; get lower 16 bits
           LD 1EH,ind_addr+2 ; get upper 16 bits
           PUSH #MSW backhere ; push msw of return addr
           PUSH #LSW backhere ; push lsw of return addr
           EBR [1CH]         ; jump to the routine
backhere: ; and get back here later
           ; ...
           RET
           END
```

As a rule, for an expression to be both absolute and be completely computable at assembly time, all operands in the expression must be absolute. However, the assembler can evaluate an expression as being absolute and of `null` type at assembly time if the expression meets the following requirements:

- The expression contains subtraction or any of the relational operators listed in Table 5-5.
- The operands subtracted are relocatable but of the same segment type.

During the evaluation, segments of the same types cancel out.

5.3.4 RELOCATABLE EXPRESSIONS

Relocatable expressions are those expressions that are evaluated only during link-time. Relocatable expressions are valid only when they are composed of either one relocatable or one external symbol, plus or minus an optional two's-complement constant that is computed during assembly-time. Thus, the permissible forms are a relocatable or an external symbol plus or minus a constant, or a constant plus a relocatable or an external symbol. A constant minus a relocatable or an external symbol is not permitted. Thus, `relo-5`, `exto+3`, and `12+relo` are valid, but `13-exto` is not.

Relocatable expressions that are based upon an external symbol can appear only when specifying operands to statements that generate object code (i.e., only in machine instructions and code definition directives).

The following example illustrates the possible relocatable expressions:

```

DSEG AT 24H
abslab:   DSW  1

DSEG
EXTRN extlab:WORD
rellab:   DSW  1

CSEG
DCW  abslab + abslab
DCW  abslab - abslab
DCW  abslab + rellab
DCW  abslab - rellab; ERROR
DCW  abslab + extlab
DCW  abslab - extlab; ERROR
DCW  rellab + abslab
DCW  rellab - abslab
DCW  rellab + rellab; ERROR
DCW  rellab - rellab
DCW  rellab + extlab; ERROR
DCW  rellab - extlab; ERROR
DCW  extlab + abslab
DCW  extlab - abslab
DCW  extlab + rellab; ERROR
DCW  extlab - rellab; ERROR
DCW  extlab + extlab; ERROR
DCW  extlab - extlab; ERROR
END

```

The lines with the comment "ERROR" will generate the following error:

```
ERROR #12, ILLEGAL BINARY OPERATION.
```

This error is generated because of an illegal combination of absolute and relative expressions.

5.3.5 EXTERNAL BIT NUMBERS

With `omf (2)` it is possible to use external bit numbers in the conditional bit jump instructions like `BBS`, `EBBC`, and `JBC`. For example, the following statement is legal when using the `omf (2)` control:

```
RSEG
    EXTRN Bitno
BReg:DSB 1

CSEG
Lab:
    ...
    BBS BReg, Bitno, Lab
    ...
END
```

5.4 STATEMENT FORMAT

The assembly source program statement consists of four fields: the label field, the operation field, the operand field, and the comment field. Each field is optional. Thus, the correct syntax for a statement is as follows:

```
[label[:] | name][operation][operand,...];comment<CR,LF>
```

Where:

- | | |
|------------------|--|
| <i>label:</i> | contains a label name. A label differs from a name in that a label must be followed by a colon unless the control optional colon is used. In that case the colon is optional. A label can appear before machine instructions, macro calls, storage reservation directives, constant-definition directives, and empty statements. The value of a label is the value of the program counter at the beginning of the instruction or constant, after achieving the required alignment. The value of a label of an empty statement is the value of the program counter at the beginning of the next instruction or constant. Use a label to quickly reference a position within the code. |
| <i>name</i> | contains a tag name that is used to associate a set of attributes. The assembler does not reserve a storage space for this field. |
| <i>operation</i> | contains a machine-instruction mnemonic code, a directive mnemonic, or a user-defined macro name. The contents of the operation field determine which instruction the machine is to execute. |
| <i>operand</i> | contents and syntax depend on the specific operation in the statement. Some operations take more than one operand. The contents of the operand field specify which operands are used in the operations. The operand(s) can be simple names, expressions involving names and numeric operators, or an address expression that defines the method (indirect or indexed) by which another operand is to be accessed. Only certain operands are allowed with certain operations. See Section 5.3 for more information on basic operands. |

*;**comment* allows the placement of a natural-language description on each statement. The assembler takes no action on the comment, but includes it in the listing file. Therefore, special characters appearing in the comment field do not cause the assembler to perform any special function. The comment field must begin with a semicolon (*;*) and can contain any symbols in the ASM196 character set. All text appearing between the semicolon and the next line feed is taken as a comment.

5.4.1 ADDITIONAL STATEMENT RULES

Statements must also meet the following requirements:

- No continuation lines are allowed in ASM196. All parameters in a given statement must be placed on the same line.
- Each statement must be terminated by a line feed. A carriage return can optionally precede the line feed.
- Blank characters (spaces) can appear anywhere in a statement except within numbers, names, and special symbols. Thus, *data buffer* is two names, not one, and *;* *;* is not the same as *;;*.
- An empty line, one containing only blanks, comments, or null, can appear anywhere within a program.
- A line that contains only a label can appear only within a segment. It cannot appear before the first *xseg* directive.

5.5 PROGRAM FORMAT

When assembly language statements are combined to form programs that direct the processor to do something, the assembler follows a set of rules that govern the format of its input. The format for such assembly units is presented below. See the notation conventions and Chapter 2 for conventions used in this manual.

```
[ { primary control line(s) } ]  
[ module attribute ]  
[ statement(s) ]  
end
```

Generally, an assembly unit begins with one or more primary control lines, although you can omit primary control lines entirely. See Chapter 4 for a detailed discussion of the form and function of ASM196 controls, which determine the operation of the assembler as it encounters statements.

The module directive allows the assignment of attributes to the module being assembled. See Chapter 6 for a description of these attributes.

The ASCII SUB, or Ctrl-Z, character can appear as the last character in a file. The ASCII value of this character is 1A hexadecimal. DOS uses this character for an end-of-file marker and all characters following the Ctrl-Z are skipped by the ASM196 assembler.

Many statements are meaningful and valid only if they appear in a specific context. Thus, a machine instruction can appear only within a code segment. The rules governing a statement are stated in the description of the statements.

Figure 5-1 presents an assembly-unit program structure.

```
$TITLE('--> HAND CALCULATOR: MAIN <--')

HC_MAIN                MODULE MAIN,STACKSIZE(8)
;
;The program simulates a common hand calculator.
;Available commands (DR the display register, MR the memory register):
;
;Commands can be written in upper or lower case letters.
    EXTRN    ER_Invalid_Command
    PUBLIC   MN_Result, MN_Exit_Flag
;

$INCLUDE(8096.INC)
$NOLIST
$LIST
;

    RSEG
MN_Result:    DSW      1          ;Result of byte or word functions
MN_Exit_Flag: DSB      1          ;Exit flag, only bit 0 is used

    CSEG      at 2080H
    EXTRN    IO_Put_String, IO_Put_Char
    EXTRN    RD_Get_Line, RD_This_Char
    EXTRN    RD_Skip_Blanks
    EXTRN    UT_FindB, UT_Help_Com, UT_Exit_Com
    EXTRN    ER_Put_Message
    EXTRN    RG_Eval, RG_Mem_Op, RG_Clear_MR, RG_Recall_MR,
    EXTRN    RG_Print_DR
```

Figure 5-1: Assembly-unit program structure

```

Start:                                ;... of main program
        LD      SP,#STACK              ;Init StackPointer
        PUSH    #STR_signon           ;Print Signon
        CALL    IO_Put_String
        CLRB    MN_Exit_Flag          ;Reset exit flag

Main_Loop:
        BBS     MN_Exit_Flag,0,Exit    ;Check for exit
        CALL    RG_Print_DR
        PUSH    #':'                  ;Print prompt character
        CALL    IO_Put_Char
        CALL    RD_Get_Line            ;Line is placed in RD_Line
        CALL    RD_Skip_Blanks
        CALL    RD_This_Char           ;Find command type
        PUSH    #STR_1st_Chars
        PUSH    MN_Result
        CALL    UT_FindB
        CMP     MN_Result,#0FFFFH
        BNE     Command_OK
        PUSH    #ER_Invalid_Command
        CALL    ER_Put_Message
        BR      Main_Loop

Command_OK:
        ADD     MN_Result, MN_Result  ;Do case MN_Result
        LD      MN_Result, Com_Tab[MN_Result]
        PUSH    #Main_Loop            ;Simulate indirect call
        BR      [MN_Result]

;
;Tables and constants used for the above logic
STR_1st_Chars: DCB '0123456789','+*/*','MRC?HE',0
STR_Signon:   DCB CR,LF,'8096-BASED CALCULATOR, V1.0',CR,LF,0
Com_Tab:
        DCW     RG_Eval,RG_Eval,RG_Eval,RG_Eval, RG_Eval ; 0-4
        DCW     RG_Eval,RG_Eval,RG_Eval,RG_Eval, RG_Eval ; 5-9
        DCW     RG_Eval,RG_Eval,RG_Eval,RG_Eval ; +-*/*
        DCW     RG_Mem_Op                                ;M+ or M-
        DCW     RG_Recall_MR                             ;RM
        DCW     RG_Clear_MR                              ;CM
        DCW     UT_Help_com, UT_Help_com                  ;? or H
        DCW     UT_Exit_com                              ;EX

;
; Exit loop
exit:
        PUSH    #STR_Signoff          ;Print signoff
        Call    IO_Put_String
        BR      $                     ;Infinite loop

$GEN
STR_Signoff:   DCB CR,LF,'      S T O P',CR,LF, 0

END

```

Figure 5-1: Assembly program structure (continued)

5.6 SEGMENTS

A segment is a piece of memory defined within a module or a section of memory used by the application programs. Each segment has a specific function. The five segment types in ASM196 are:

- the non-overlayable register segment
- the overlayable register segment
- the non-overlayable data segment
- the overlayable data segment
- the stack segment
- the user defined stack segment
- the code segment
- the constant segment

5.6.1 REGISTER SEGMENT (OVERLAYABLE AND NON-OVERLAYABLE)

The register segment is a portion of memory allocated in the register section. You can use variables belonging to the register segment as registers throughout the program. You can specify only storage reservation variables within this segment. If a register segment is declared using the `oseg` directive (see Chapter 6), it is tagged as overlayable. The RL196 linker can overlay it with other overlayable register segments if they are not active simultaneously. Non-overlayable segments cannot be overlaid.

5.6.2 DATA SEGMENT (OVERLAYABLE AND NON-OVERLAYABLE)

The data segment is a portion of memory allocated in a RAM section. You can use variables belonging to this segment as memory-referenced operands using direct, indirect, indexed, or based-addressing modes or extended indirect and extended index modes for 24-bit models. You can specify only storage reservation variables within this segment. If a data segment is declared using the `odseg` directive (see Chapter 6), it is tagged as overlayable. The RL196 linker can overlay it with other overlayable data segments if they are not active simultaneously. Non-overlayable segments cannot be overlaid.

5.6.3 STACK SEGMENT

The stack segment is a portion of memory allocated in a RAM section. You can reference variables belonging to this segment using the based addressing mode using the stack pointer or any other register used as a pointer. You cannot define code or storage variables within this segment, which is solely for efficient stack use. To reference a static variable within the stack segment, specify the (negative) offset from the bottom of the stack (highest memory address), the initial value of the stack pointer. Note that the stack grows downwards. Because of the special character of stack segments, all stack segments are placed contiguously in one section of RAM.

5.6.4 USER DEFINED STACK SEGMENT

The user defined stack segment, declared using the `sseg` directive (see Chapter 6), provides you with a way to declare a stack segment in an assembly module. Since the linker will locate the stack in this user defined stack, the user defined stack segment must be treated like the stack segment (see 5.6.3). Note that the user defined stack segment can only be specified in assembly. It does not define a second stack segment, it overrides the default stack segment "computed" by the linker.

The following example declare a user defined stack with a size of 256 bytes:

```
SSEG
DSB  0100H
.
.
END
```

Note that you can still override the user defined stack size in the module with the linker `STACKSIZE` control.

5.6.5 CODE SEGMENT

The code segment is a portion of memory allocated in a ROM section. Use this segment to store program constants and code. You can reference variables belonging to the code segment as memory-referenced operands using direct, indirect, based, or indexed-addressing modes, or extended-indirect or indexed modes for 24-bit models or as the targets of branch and call instructions.

5.6.6 CONSTANT SEGMENT

The constant segment is a portion of memory allocated in a ROM section. This segment is used for constants only. Variables which belong to such segments can be used as memory referenced operands.

5.7 ABSOLUTE AND RELOCATABLE SEGMENTS

You can define a segment as either absolute or relocatable. When you define an absolute segment, the assembler creates a new segment. This segment displaces a previous absolute segment, if one existed.

When a relocatable segment is defined, the assembler continues from the previous relocatable segment of that type, if any exists. Thus, the object module contains at most one relocatable segment of each type and an unlimited number of absolute segments. Absolute segments do not require relocation, but they can contain public symbols and references to other segments or modules.

The assembler attaches an alignment attribute to each non-empty relocatable segment. This attribute determines whether the segment starts on a byte, a word, or a long-word boundary, thus ensuring proper alignment of all symbols within the given segment. When object files are linked, the segments are placed in the required section of memory and aligned according to their alignment attribute. Because of the special character of stack segments, all stack segments are placed contiguously in one section of RAM.

When writing code for a 24-bit processor, code, data, and constant segments can be defined as near or far. When near code segment configuration is selected, the processor is configured in compatible code mode and all code addresses are specified as 16-bit offsets from 0FF0000H base. Code which ends up in this page is referred to as 'high code'. When far code segment is selected, referencing code may require usage of 24-bit offsets. Defining data/constants segments as near provides the possibility of using 16-bit addressing for accessing variables in data/constant segments, while defining those segments as far means for some of the variables can only be accessed by using extended load and store instructions. Two relocatable code, data or constant segments are of the same type if they are defined with the same directive (`cseg`, `odseg`, `dseg`, or `kseg`) and have the same segment attribute (`near` or `far`).

5.8 STACK OVERFLOW

Some 80C196 models have support to detect stack overflow. This StackOverflow Module (SOM) has 2 SFRs that store the upper and lower SP boundaries. The linker generates two symbols, `_TOP_OF_STACK_` and `_BOTTOM_OF_STACK_`, that represent the upper and lower stack boundaries. It is up to you to load the SFRs with the linker generated symbols in your program. For example:

```

EXTRN  _TOP_OF_STACK_
EXTRN  _BOTTOM_OF_STACK_
.
.
LD      TMPREG0, #_TOP_OF_STACK_
ST      stack_top, TMPREG0
LD      TMPREG0, #_BOTTOM_OF_STACK_
ST      stack_bottom, TMPREG0
.
.
```

The two symbols `_TOP_OF_STACK_` and `_BOTTOM_OF_STACK_` will be set to the boundaries on the stack. If the stack is located at 0300H with a size of 0100H the stack pointer SP will be initialized with 0400H and `_TOP_OF_STACK_` and `_BOTTOM_OF_STACK_` will have the values 0402H and 02FEH respectively. This is conform the specification of the SOM. The upper limit comparator compares for a `SP >= stack_top` condition while the lower limit comparator compares for a `SP <= bottom_stack` condition. If at a later date the behavior of SOM changes, you can easily load other values, for example:

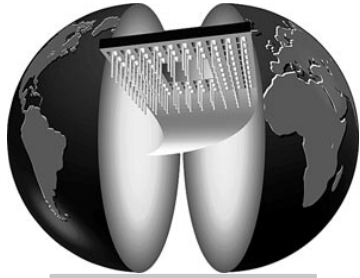
```
LD    TMPREG0, #_TOP_OF_STACK_ - 2
ST    stack_top, TMPREG0
LD    TMPREG0, #_BOTTOM_OF_STACK_ + 2
ST    stack_bottom, TMPREG0
```

CHAPTER

6

ASSEMBLER DIRECTIVES





6

CHAPTER

This chapter describes the directives recognized by the ASM196 assembler. Directives specify auxiliary information such as storage reservation, segment selection, location counter control, code and symbol definition, and conditional assembly.

These directives are grouped into seven categories. Table 6-1 describes each category and its function.

Category	Function	Directives
module level	These directives specify information that affect the module as a whole.	module, end public, extrn
segment selection	These directives specify the segment to be created.	cseg, dseg, kseg, rseg, oseg, odseg, sseg
location counter	These directives explicitly sets the location counter.	org, at
symbol definition	These directives assign constant values to symbols.	equ, set
constant definition	These directives define constants to be inserted in the program.	dcb, dcw, dcl, dcr
storage reservation	These directives reserve storage space for program variables.	dsb, dsw, dsp, dsq, dsl, dsr
conditional assembly	These directives allows control over the flow of of the assembler processing of statements.	if, endif else

Table 6-1: Functions performed by directives

You can intersperse directives between instructions throughout your program. Only one directive can appear on a given statement line.

The remainder of this chapter explain each directive in detail. The directives appear in alphabetical order.



See the *Conventions Used In This Manual* at the beginning of this manual for special meanings of type styles used in this manual.

cseg, dseg, kseg, oseg, odseg, rseg and sseg

Function

Specify the type of segment to be created.

Syntax

```
{ cseg | dseg | odseg | oseg | rseg | kseg | sseg }  
[ rel | at base_address ]
```

where:

- cseg designates a code segment.
- dseg designates a data segment.
- odseg designates an overlayable data segment.
- oseg designates an overlayable register segment.
- rseg designates a non-overlayable register segment.
- kseg designates a constant segment.
- sseg designates a user defined stack segment.
- rel specifies a relocatable segment. The `rel` option is the default if you do not explicitly specify `rel` or `at`.
- at specifies an absolute segment, beginning at address *base_address*.

Type

Segment selection

Description

Use these directives to specify the type of segment to be created. These directives terminate the current segment (deactivated segment), and initiate a new segment as specified in the directive. The specified segment remains active until you specify another segment or until the assembler reaches the end of the program.

If the deactivated segment was absolute, the directive you specify closes that segment. The assembler then creates a new absolute segment if you specify the `at` option. If the deactivated segment was relocatable, then the assembler records its location counter so that it can be resumed when that relocatable segment is reselected.

If you specify a relocatable segment using the `rel` option, the assembler continues the previous relocatable segment of that type, if any exists. If none exists, the assembler creates a relocatable segment of that type. Specifying `at` creates an absolute segment that starts at the address given by *base_address*.

The assembler restricts statements appearing within segments according to the type of the segment as shown below.

- Register segments (overlayable or not) and data segments cannot contain constant-definition directives or machine instructions.
- Code segments cannot contain storage-reservation directives.
- Data segments cannot contain constant-definition directives and machine instructions.
- Constant segments cannot contain storage reservation directives and machine instructions.

These directives also specify the segment type of all external symbols specified within their scope while the segment is active. Thus, the segment type of an external symbol is always that of the currently active segment.

Since the active segment has no default initial value, you must place a segment-selection directive before the first piece of memory is define (i.e., before entering any machine instruction, constant-definition directive, or storage-reservation directive).

If a 24-bit model() control is specified, each segment defined with the `cseg`, `dseg`, `odseg` and `kseg` directives has a near or far attribute. That attribute, if not specified in the directive, is determined according to the following rules:

- For data and constant segments, if a `far`data/`far`const primary control is specified segments are far; otherwise, they are near, except if they are absolute and the starting address is above 0FFFFH.
- For the first code segment defined in a program, the segment attribute is far if the `far`code primary control is specified; otherwise it is near.

- For the second, third, etc. code segment define with the `cseg` directive, the same value of the near/far segment attribute is assumed as in the first `cseg` directive.

If a 24-bit `model()` control is specified, you are not allowed to have both near and far code segments in one module simultaneously. If no 24-bit `model()` control is specified, neither near or far segment attribute are allowed in segment definition directives.

dcb

Function

Defines a byte constant in a const or code segment.

Syntax

```
[label:] dcb {expression | string} [,...]
```

where:

label is a valid label name followed by a colon (:).

expression is a valid expression.

string can be a string of *m* characters, where *m* can be any non-negative integer.

Type

Constant definition

Description

Use this directive to specify a list of byte values, each represented by an expression, to be inserted one after the other starting at the position of the active location counter. The expressions can be relocatable or absolute, but must be in the range of -128 to +255 inclusive. Since the assembler does not use the higher bit, sign information is lost for values outside the range of -128 to +127, inclusive. For relocatable expressions, the linker checks the range of the expression during link-time.

The `dcb` directive is the only statement that can accept a null string or a string of more than two characters. A null string (i.e., length = zero) generates no code. A string of *m* characters is coded as a sequence of *m* bytes, each byte having the ASCII value of the respective character in the string.

If you place the location counter symbol (\$) in any of the expressions in the `dcb` directive, the assembler interprets its value as the address before the first byte in the list. You can also use the value of the location counter to define the optional label, and to define any previous labels if they are separated from the present statement by statements that do not affect the value of the location counter.

Constant-definition directives can appear only in code segments (cseg) and data segments (dseg).

Example

```
kseg

A    equ    2           ;variable A is 2
B    equ    24          ;variable B is 24

    dcb     'ABCDEF'    ;six bytes with ASCII
                        ;of the characters
    dcb     10*A         ;one byte, value 20
    dcb     A,B+10      ;two bytes, values 2,34
```

dcl

Function

Defines a long constant in a const or code segment.

Syntax

```
[label:] dcl expression [, ...]
```

where:

label is a valid label name followed by a colon (:).

expression is 32-bit value expression.

Type

Constant definition

Description

Use this directive to specify a list of 32-bit values, each represented by expressions, to be inserted one after the other, starting at the position of the location counter. The assembler adjusts the location counter by incrementing it by one, if necessary, to correct alignment so that first element begins on a word boundary.

The expression can be relocatable. For each expression, the 32-bit value is stored with the low-order word first in the lowest address, followed by the high-order word.

If you place the location counter symbol (\$) in any of the expressions, the assembler interprets its value as the address before the first element in the list. You can also use this value to define the label in the directive and to define previous labels if they are separated from this statement by statements which do not affect the location counter.

Constant-definition directives can appear only in code segments (*cseg*) and constant segments (*kseg*).

dcp

Function

Defines a pointer constant in a const or code segment.

Syntax

```
[label:] dcp expression [, ...]
```

where:

label is a valid label.

expression is a valid expression.

Type

Constant definition

Description

The `dcp` directive specifies a list of pointer values, each represented by an expression, to be inserted one after the other, starting at the position the location counter is pointing to. The pointer size is 2 bytes in 16-bit mode and 3 bytes in 24-bit mode. The pointers are always aligned on a word boundary.

The expression can be relocatable. For each expression, the value is stored with the low-order byte first.

If the segment type of the expression is null, the value is treated as an offset and must be in the range $(-0FFFFFFH, +0FFFFFFH)$ for 24-bit mode or $(-0FFFFH, +0FFFFH)$ for 16-bit mode; since the highest byte is ignored, the sign value is lost for values outside the range $(-800000H, +7FFFFFFH)$ for 24-bit mode or $(-8000H, +7FFFH)$ for 16-bit mode. If the segment type is not Null, then the value is treated as an address and must be in the range $(0, 0FFFFFFH)$ for 24-bit mode or $(0, 0FFFFH)$ for 16-bit mode.

If the location counter symbol appears in any of the expressions, it stands for its value before the first element in the list. This value is also used for defining the label in the directive, and for defining previous labels if they are separated from the current statement by statements which do not affect the location counter.

dcr

Function

Defines a floating point constant in a const or code segment.

Syntax

```
[label:] dcr float_num [, ...]
```

where:

label is a valid label name followed by a colon (:).

float_num is a floating point number.

Type

Constant definition

Description

Use this directive to specify a list of one or more floating point numbers (not expressions) to be inserted one after the other starting at the position of the location counter. The assembler adjusts the location counter by incrementing it by one, if necessary, for correct alignment, so that the first element begins on a word boundary. For more information on floating point numbers, see the *80C196 Utilities User's Guide* listed in *Related Publications*.

Constant-definition directives can appear only in code segments (cseg) and constant segments (kseg).

dcw

Function

Defines a word constant in a const or code segment.

Syntax

```
[label:] dcw expression [, ...]
```

where:

label is a valid label name followed by a colon (:).

expression is a valid expression.

Type

Constant definition

Description

Use this directive to specify a list of word values, each represented by an expression, to be inserted one after the other, starting at the position of the location counter. The assembler adjusts the location counter by incrementing it by one, if necessary, for correct alignment so that the first word begins on a word boundary.

The expressions can be relocatable or absolute. For each expression, the assembler stores the expression's 16-bit value with the low-order byte in the first (lower-order) address, followed by the high-order byte in the higher order address.

If you place the location counter symbol (\$) in any of the expressions, the assembler interprets its value as the address before the first word in the list. You can also use this value to define any previous labels if they are separated from the present statement by statements that do not affect the value of the location counter.

Constant-definition directives can appear only in code segments (*cseg*) and constant segments (*kseg*).

dsb, dsl, dsp, dsr, dsw, and dsq

Function

Reserve storage space for program variables.

Syntax

```
[label:] {dsb | dsw | dsp | dsl | dsq | dsr} expr
```

where:

<i>label</i>	is a valid label name followed by a colon (:).
<i>expr</i>	is a valid absolute expression of null segment type containing no forward references.
dsb	specifies byte variables, no alignment required.
dsw	specifies word variables, adjustment to a word boundary required.
dsp	specifies a pointer variable. In 16-bit mode this is a 2-byte variable, adjusted to a word boundary. In 24-bit mode this is a 3-byte variable. If the dsp is in register space it is long-word aligned. If it is in normal data space it is word aligned.
dsl	specifies long-word variables, adjustment either to a long-word boundary if in a register segment or to a word boundary if in a data segment.
dsq	specifies quad-word variables, adjustment either to a quad-word boundary if in a register segment or to a word boundary if in a data segment.
dsr	specifies floating point number variables, adjustment to word boundary.

Type

Storage reservation

Description

Use these directives to reserve storage for program variables. Specify `dsb` to define storage for byte variables. No adjustment is made to the boundary. Specify `dsw` to define storage for word variables. The assembler adjusts the location counter so that the symbols start on a word boundary. `dsp` defines storage for 24-bit pointer variables and adjustment for long word or word is performed depending on the segment type. Specify `dsl` to define storage for long-word variables. The assembler adjusts the location counter to ensure that the symbols start on word or long-word boundaries.

Specify `dsq` to reserve the storage for a quad word aligned piece of memory. The corresponding `dcq` keyword is not implemented as it would require 64-bit addressing, which is not possible. The `dsq` keyword can be used to allocate the storage needed for a register in conjunction with the `ebmovi` instruction:

```

RSEG
Tmp0 : DSQ      1
Tmp8 : DSW      1

CSEG
EBMOVI  Tmp0 , Tmp8

END
```



In the list file of the assembler, the register segment will be `QUAD WORD` alignment, and the type of the `Tmp0` register will be `LONG`.

Specify `dsr` to define storage for floating point numbers. The assembler adjusts the location counter so that the symbols starts on word boundaries. The storage-reservation directives cannot appear in the code segment.

The label, if specified, is set to the value of the location counter after the assembler aligns this value as required by `dsw`, `dsp`, `dsl`, or `dsr`. You can also use this value to define previous labels if they are separated from this statement by statements that do not affect the value of the location counter.

The assembler reserves storage by incrementing the active location counter by $expression * n$ where $n=1$ for `dsb`, $n=2$ for `dsw`, or $n=4$ for `dsr` and `dsl`. The location counter is incremented by $expression * 2$ for `dsp` in 16-bit mode, and by $expression * 4 - 1$ in 24-bit mode. This last expression means that a `dsp` always leaves one byte free at the odd address following the `dsp`. For all storage reservation directives the *expression* must be absolute and of null segment type and must not contain forward references.

end

Function

Signifies the end of the module.

Syntax

end

Type

Module level

Description

Use this directive to signify the end of the module. You must place this directive as the final statement in a program. The assembler flags any statement placed after the end directive as an error.

equ, set

Function

Assign constant values to symbols.

Syntax

```
symbol_name { equ | set } expression [:data_type]
```

where:

symbol_name is the name of the symbol to be defined.

expression defines the segment type, relocatability, value, and data type of the symbol.

data_type is byte, word, dword, long, pointer, entry, real, or null.

Type

Symbol definition

Description

The directives `equ` and `set` are used to define symbols.

Use `set` and `equ` to define the attributes (segment type, relocatability, value and data type) of the *symbol* to be that of *expression*. If *expression* has the null segment type, the segment type and relocatability for *symbol* is that of the current segment type, otherwise the segment type and relocatability of *expression* is used. You cannot redefine symbols defined by `equ` directive. However, you can redefine symbols defined by `set` but only with another `set` directive.

The *expression* value cannot contain any external symbols, but can contain one level of forward reference. You must then define this forward-referenced symbol somewhere in your program.

The optional *data_type* must match the segment type of the expression. Thus, `byte`, `word`, `dword`, `long`, `pointer`, `real`, and `null` are allowed everywhere, while `entry` is allowed only for a code segment.

Example

```
level set 1           ;level is 1
A      equ level:byte  ;A is a byte with value 1

level set level+1     ;increment level
B      equ level:dword ;B is a double word with value 2
```

extrn

Function

Declares symbols as externals.

Syntax

```
extrn {symbol_name [:data_type] } [,...]
```

where:

symbol_name is the name of a symbol to be declared as external.

data_type must be `byte`, `word`, `dword`, `long`, `pointer`, `entry`, `real` or `null`.

Type

Module level

Description

Use this directive to declare one or more symbols as externals (i.e., defined and located in other modules but used by the present module). You must declare each such symbol `public` within the other modules. These symbols cannot be defined in the present program.

The segment type of the symbols is set to that of the active segment. For example, to define an external symbol of type `code`, you must place the `extrn` directive that specifies the symbol within a code segment (`cseg`).

The *data_type* value must match the segment type. Thus, `byte`, `word`, `dword`, `long`, `pointer`, `real`, and `null` are allowed everywhere, while `entry` is allowed only in a code segment.

To define an external symbol with the `null` segment type, define the symbol before entering the first segment-selection directive in the program.

if, else, and endif

Function

Determines the lines of code the assembler processes.

Syntax

```
if_expression
    statement(s)
[else]
    statement(s)
endif
```

where:

if_expression is one of the following conditional if directives:

<i>if_expression</i>	True condition
IF <i>expr</i>	least significant bit of <i>expr</i> = 1
IFEQ <i>expr</i>	<i>expr</i> = 0
IFNE <i>expr</i>	<i>expr</i> <> 0
IFLT <i>expr</i>	<i>expr</i> < 0
IFLE <i>expr</i>	<i>expr</i> <= 0
IFGT <i>expr</i>	<i>expr</i> > 0
IFGE <i>expr</i>	<i>expr</i> >= 0
IFDEF <i>symbol</i>	<i>symbol</i> defined
IFNDEF <i>symbol</i>	<i>symbol</i> not defined
IFB < <i>str</i> >	<i>str</i> is empty
IFNB < <i>str</i> >	<i>str</i> is non-empty
IFIDN < <i>str1</i> >, < <i>str2</i> >	<i>str1</i> is equal to <i>str2</i>
IFNIDN < <i>str1</i> >, < <i>str2</i> >	<i>str1</i> is not equal to <i>str2</i>



All comparisons between *expr* and 0 are signed. The state of the signedoper control has no influence here.



When *symbol* is a forward reference, the symbol is considered as not defined.



These four conditional directives can be used in normal assembly code, but are mostly used in macro definitions to test for availability of arguments. The angle brackets are required in the assembler source file.

expr is a valid absolute expression (that contains no forward references).

symbol is a valid identifier.

str, str1, str2 is a valid string.

statement(s) is one or more valid statements.

Type

Conditional assembly

Description

Use these directives to control which lines of code the assembler processes. These directives provide for conditional assembly processing of statements based upon assembly-time evaluation of user-defined expressions.

The *expr* value must be a valid absolute expression (that contains no forward reference). If the expression in the *if_expression* line is evaluated and proves true (least significant bit = 1), the assembler assembles the statements between *if_expression* and the optional *else* or the mandatory *endif* (whichever comes first). If the *if_expression* line proves false (lsb = 0), the assembler skips the statements between the *if_expression* line and the optional *else* line, and assembles any statements between the *else* line or the *endif*, if *else* does not appear.

All undefined symbols and forward references are evaluated to false by the assembler if the *relaxedif* control is specified. *norelaxedif* is the default control, so if you have undefined symbols in an *if_expression* the assembler will issue an error.

The statements between an *if_expression* directive and its ending *endif* constitute a conditional assembly block. Conditional-assembly directives can appear anywhere in a program, including within macros. However, if a conditional assembly block is initiated in a macro definition, its *endif* must also lie in that macro.

When statements are not assembled due to conditional-assembly directives they are effectively not in the program, though they can be listed if you specify the `cond` control. Therefore, any symbols defined in assembler-skipped statements are not available to the rest of the program. Avoid referring to any symbols that are defined in lines that can be skipped due to conditional assembly.

Conditional assembly blocks can be nested within one another (i.e., `if/else/endif` blocks within another `if/else/endif` block). Be sure to maintain congruence of `if` lines and their respective `endif` lines. The maximum depth of nesting is nine.

Example

The following macro definition shows an example usage of the `IFB` conditional directive:

```
MyAddMACROsrc1, src2, res
    IFB <src2>
        add res, src1
    ELSE
        add res, src1, src2
    ENDIF
ENDM
```

module

Function

Assigns the module name and type.

Syntax

```
module_name module [attr,...]
```

where:

module_name is the desired name of the module.

attr is either `main` or `stacksize(n)`.

`main` specifies that the module is to be of type `main`. If none is entered, the default is `non-main`.

`stacksize(n)` defines the amount of stack in bytes required by the module. *n* must be an even number. The default is zero bytes.

When the `cmain` control is in effect, `cmain` must be used instead of `main`.

Type

Module level

Description

Use this directive to assign a name to the module. The assembler includes this name in the object file and uses this name to identify the object module. If you omit the `module` directive, the assembler uses the source filename without the extension as the default module name.

The `module` directive can appear at most once in a program. You must place this directive at the beginning of the program before non-control lines.

org

Function

Sets the value of the location counter to the evaluated expression.

Syntax

```
org expression
```

where:

expression is an expression that can be evaluated to an address.

Type

Location counter control

Description

Use this directive to set the value of the location counter equal to the evaluated expression. The expression cannot contain a forward reference. The segment type of the expression must match the type of either the active segment or `null`. For example, if the active segment's type is `data`, the expression's type must be `data` or `null`.

If the active segment is absolute, the expression also must be absolute and represent some absolute memory address not below the *base_address* of the segment. If the active segment is relocatable, the assembler interprets the expression as an offset from the beginning of the given segment and so the expression can be absolute or relocatable.

Refrain from making backward references where the `org` directive points back into a code segment thus causing redefinition of previously defined code or data. Unpredictable results can occur.

Since the `org` directive does not create a new segment, it can create gaps in the segment. Such gaps are not filled by any other segments and remain after the relocation and linkage process has been executed.

public

Function

Declares symbols as public.

Syntax

```
public symbol_name [,...]
```

where:

symbol_name is the name of a symbol to be declared public.

Type

Module level

Description

Use this directive to declare one or more symbols public. You must define public symbols somewhere in the program either by using a symbol-definition directive, or by naming the symbol as a label. Declaring a symbol public allows other modules to access the symbol. Macro names, module names, reserved words, and external symbols cannot be declared public.



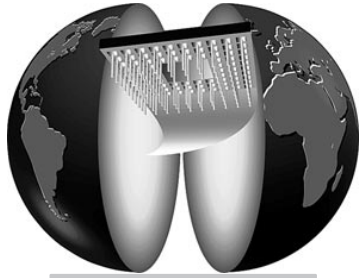
DIRECTIVES

CHAPTER

7

MACRO PROCESSING





7

CHAPTER

7.1 INTRODUCTION

A macro is a facility for simplifying coding of often-used sequences of instructions and assembler directives replacing one set of parameters with another. Often during a program development, instruction sequences are repeated several times with only certain parameters changed.

For example, suppose you write a routine that moves thirty-five bytes of data from one memory location to another. A little later, you find yourself writing another routine that moves forty-five bytes from a different source field to a different destination field. If the two routines use the same coding techniques, they probably are identical except for three parameters: the character count, the source field starting address, and the destination field starting address. Certainly, it would be handy if you had some way to regenerate that original routine substituting the new parameters rather than rewriting the code. The macro facility provides this capability and offers several other advantages over simply rewriting the code every time it is needed.

7.2 THE ADVANTAGES OF USING MACROS

Macros provide four distinct advantages:

1. By eliminating the tedium of rewriting, using macros minimizes the probability of entry error.
2. Symbols used in macros can be restricted so that they have meaning only within the macro itself. Therefore, you need not worry that you will accidentally duplicate a symbol used in the macro. Also, a macro can be used any number of times in the same program without duplicating any of its own symbols.
3. An error detected in a macro need be corrected only once regardless of how many times the macro appears in the program, thus reducing debugging time.
4. Increased efficiency is gained by reducing the duplication of effort by programmers. Frequently used macros can be made available to all programmers.

Macros also aid the development of structured programming and thereby help produce more readable documentation. Using macros to segment code blocks provides clear program notation and simplifies tracing the flow of the program.

7.2.1 AN EXAMPLE OF MACRO USE

A macro can be described as a routine defined by a formal sequence of prototype instructions that, when called within a program, results in the replacement of each such call with a code expansion consisting of the actual instructions represented. The following example illustrates macro definition, call, and expansion.

Suppose you want to place a standard greeting in a typical business form letter then list the specific flight number, departure time, destination, and arrival time for the given specific passenger. A macro, `cnfirm`, can be defined that

1. welcomes the passenger, then
2. lists the flight number (`fno`), departure time (`dtime`), arrival time (`atime`) at the destination (`dest`).

The macro `cnfirm` has four formal parameters to be replaced. Thus the macro could be:

```
cnfirm 13, "1:00", "Jamaica", "11:00"
```

Another macro, `greet`, is the standard message. Its only parameter is the passenger's name. Thus the entire text of the message can be placed in a source file in the form:

```
greet "Mr. Smith"
cnfirm 13, "1:00", "Jamaica", "11:00"
```

We trust you will enjoy the dream trip of your life.

Sincerely,
D. Jones, Manager Air Freight

After the source file is passed through a macro processor and the macro calls are expanded, the following letter is produced:

Dear Mr. Smith:

Peninsula-Panhandle Airlines welcomes you as a passenger. We are pleased to confirm your reservation on Flight #13 which leaves at 1:00 PM and arrives in Jamaica at 6:00 PM (local time).

We trust you will enjoy the dream trip of your life.

Sincerely,
D. Jones, Manager Air Freight

While this example illustrates the substitution of parameters in a macro, it overlooks the relationship of the macro processor to the assembler. The purpose of the macro processor is to generate source code that is then assembled.

7.3 MACROS AND ROUTINES

Macros differ from routines invoked by the `call` instruction as follows:

- Routines necessarily branch to another part of the program, while macros generate in-line code. Thus, a program contains only one version of a given routine, but it contains as many versions of a given macro as there are calls for that macro.
- A macro does not always generate the same source code; a routine does. By changing the parameters in a macro call, the source code produced by the macro changes. Thus, the macro is more of a general-purpose tool used to generate customized source code as required by the particular programming situation.
- Macro expansion and code customization occur at assembly time and at the source-code level. Routines, on the other hand, reside within the program and require additional execution time for parameter passing and transfer of control.

Whether to use a macro or a routine depends on the situation. The main tradeoff is between the additional space required by the macro or the additional time required to pass parameters and transfer control to a routine. In some cases, a single routine is more efficient than multiple in-line macros. In situations involving a large number of parameters, the use of macros can be more efficient. Also note that macros can call routines and routines can contain macros. Though both routines and macros can be used to implement common code, only macros can generate common data structures.

7.4 MACRO DIRECTIVES AND MACRO CALLS

The assembler recognizes the following macro directives:

- `macro` directive
- `endm` directive
- `local` directive
- `rept` directive
- `irp` directive
- `irpc` directive
- `exitm` directive
- `.strlen.` directive
- macro call

All of the preceding directives except the `.strlen.` directive and macro call are related to macro definition. The `.strlen.` directive determines the length of a string. The macro call initiates the macro expansion and parameter substitution process.

7.4.1 MACRO DEFINITION

Macros must be defined in your program before you use them. A `macro` directive initiates a macro definition. The macro definition lists the macro's name and the formal parameters to be replaced during the macro expansion. The `endm` directive terminates a macro definition. The prototype instructions, bounded by the `macro` and the `endm` directives, are called the macro body.

When labels used in a macro body have global scope, duplication errors occur if the macro is called more than once. To limit the scope of a label, use the `local` directive. This directive assigns a unique value to the symbol each time the macro is called and expanded. Formal parameters also have limited scope.



See Section 5.2.9 for information on assembler-generated symbols and the `local` directive discussed in this chapter.

Occasionally you may wish to duplicate a block of code several times either within a macro or in line with other source code. This duplication can be accomplished using the repeat block (`rept`), indefinite repeat (`irp`), and the indefinite repeat character (`irpc`) directives. Use the `endm` directive to terminate these directives. You can also use the `exitm` directive as an alternate exit from a macro. When encountered, `exitm` terminates the current macro just as if `endm` had been encountered. However, `endm` must still appear at the end to terminate the macro.

When using macros, note that you cannot redefine macros and you cannot prefix macro directives by a label. However, you can prefix a label to a macro call statement. Macro symbols cannot be forward referenced. The definition of the macro must appear before any reference is made to it. Also that the literalization character (!) loses its function unconditionally after its application. The literalization character does not preserve its function when the corresponding formal parameter is used in building another (nested) macro argument.

7.5 MACRO DIRECTIVES

The following section explains all valid macro directives. The directives are listed in alphabetical order.

endm

Function

Terminate a macro definition.

Syntax

endm

Description

Use this directive to terminate a macro definition. Each macro definition must end with an **endm** directive. This directive follows the last prototype instruction. You must also use **endm** to terminate code repetition blocks defined by the **rept**, **irp**, and **irpc** directives. If any characters are placed as a label or operand to the **endm** directive, the assembler reports an error. Because nested macro calls are not expanded during macro definition, the **endm** directive closing an outer macro cannot be contained in the expansion of an inner, nested macro call.



See Section 7.10 for further information on nesting macro calls.

Example

The following example illustrates an empty macro body.

```
do_nothing macro  
endm
```

exitm

Function

Terminates a macro expansion.

Syntax

exitm

Description

Use this directive as an alternate method of terminating a macro expansion or terminating the repetition of a `rept`, `irp`, or `irpc` code sequence. As soon as the assembler recognizes `exitm`, it skips all macro prototype instructions located between the `exitm` and the `endm` directive for this macro. You can use `exitm` in addition to but not in place of `endm`.

When used in nested macros, `exitm` causes an exit to the previous level of macro expansion. An `exitm` within a `rept`, `irp`, or `irpc` terminates not only the current expansion, but all subsequent iterations as well.

Any characters appearing as an operand to the `exitm` directive causes an error.

Example

In the following example, the macro expansion process is terminated as soon as the `x` equals 0. So long as `x` is not equal to 0 the assembler skips the line beneath that statement so the `exitm` is not executed. When `x` equals 0, the assembler executes the `exitm` directive.

```
route3 macro x,y
    ld    reg2,#y
    addb  reg2,#10
    ld    reg1,#x
    if x eq 0
        exitm
    cmpb  x,#10
    be    target_label
endm
```

irp

Function

Executes the macro body indefinite times.

Syntax

```
irp formal_parameter, <actual_arguments>
```

where:

formal_parameter is a single formal parameter.

actual_arguments is a list of one or more actual arguments, separated by commas. The list of actual arguments must be enclosed within angle brackets (<>).

Description

Use this directive to perform an indefinite repeat. The assembler executes the macro body one time for each actual argument. Each time the macro body is executed, the successive actual argument are substituted for the formal parameter. If you do not specify an actual argument, the macro body is executed once, with a null (blank) used for the parameter. If two commas are next to each other with no actual parameter between them, a null (blank) is substituted for the actual argument.

Example

Consider the following macro:

```
ld    reg1, #storit
irp   x,<fld1,#3E20H,fld3>
    ld    reg2, x
    st    reg2, [reg1]+
endm
```

When assembled, this macro generates the following code:

```
ld    reg1, #storit
ld    reg2, #fld1
st    reg2, [reg1]+
ld    reg2, #3E20H
st    reg2, [reg1]+
ld    reg2, fld3
st    reg2, [reg1]+
```

Thus, the command sets `ld` and `st` are executed successively for each of the actual parameters, `fld1`, `#3E20H`, and `fld3`.

irpc

Function

Executes the macro body one time per character.

Syntax

```
irpc formal_parameter, actual_argument
```

where:

formal_parameter is a single formal parameter.

actual_argument is sequence of characters.

Description

Use this directive to cause the macro body to be executed one time for each character of the character list. On each successive execution, the assembler substitute a successive character of the list for the formal parameter. If the character list is null (empty), the assembler executes the macro body one time with a null (blank) substituted for the formal parameter. If a delimiter is to substitute a formal parameter, include it in the list and enclose the list in angle brackets (<>).

Example

Consider the following macro code:

```
clr    reg1
irpc   x, 1982
    mul    reg1, #10
    add    reg1, #x
endm
```

When assembled, this macro generates the following code:

```
clr    reg1
mul    reg1, #10
add    reg1, #1
mul    reg1, #10
add    reg1, #9
mul    reg1, #10
add    reg1, #8
mul    reg1, #10
add    reg1, #2
```

Thus, reg1 repeatedly accumulates the result of the conversion of 1982 to internal form.

local

Function

Defines a symbol local to the macro body.

Syntax

```
local local_symbol_name [, ...]
```

where:

local_symbol_name is the name of a label to be defined only within the current macro expansion.

Description

Use this directive to define one or more symbols valid only within the macro body. This directive must appear only within a macro definition and must precede all other statements within the macro body.

The local symbol names are meaningful only within the current macro expansion. Each time the macro is called or expanded, the assembler assigns each local symbol a unique global symbol in the form *??nnnn*, where *nnnn* is 0001 for the first local symbol in the program, 0002 for the second, and so on. The most recent symbol name generated always indicates the total number of symbols created for all macro expansions. The assembler never duplicates these symbols. To avoid conflict, do not define symbols that begin with *??*.

Formal parameters included in a macro call cannot be operands of a `local` directive. A formal parameter is always restricted to its own macro definition. Local symbols can be defined only within a macro definition. A maximum of 16 local symbols can appear in a macro definition. They must all appear between the `macro` directive and the first line of prototype code.

Example

In the following example, `loop` is declared as a local symbol so you eliminate the possibility of errors arising from `route2` being called twice, making `loop` multiply defined, or from the symbol `loop` being defined elsewhere in the code.

```
route2 macro g1,g2,g3
    local loop
    ld    reg1, g1
    ld    reg2, #g2
    ld    reg3, #g3
loop:   ld    reg4, [reg2]+
        st    reg4, [reg3]+
        dbnz  reg1, loop
    endm
```

macro

Function

Defines the macro.

Syntax

name **macro** [*formal_parameters*]

where:

name specifies the name of the macro body being defined. Any valid user-defined symbol name can be used as a macro name.

formal_parameters are user-defined symbol names. You can have up to 16 formal parameters, separated by commas.

Description

Use this directive to define a macro body. You can have up to 16 formal parameters in the macro definition. When you specify multiple parameters, separate each parameter with a comma. The scope of a formal parameter is limited to its specific macro definition.

You cannot use reserved words as formal parameter names. The assembler does not recognize formal parameters embedded in a comment. You can, however, place these parameters in a character string. In this case, you must use the ampersand (&) as explained in Section 7.6.

The macro body can contain any statement including nested macro definitions. If you want to add comment lines, precede each line with a double semicolon (;). You can include any machine instruction or applicable assembler directive in the macro body. The distinguishing feature of macro prototype text is that you can make parts of the text variable by placing substitutable formal parameters in instruction fields. These formal parameters are the same as the symbols in the operand field of the macro directive. The assembler then replaces these parameters with the actual argument values, arguments passed to the macro, when you call the macro.

The macro body can also contain expressions that test the existence of actual arguments. To do this, place the `nul` operator before the formal parameter. The `nul` operator is a unary operator which returns true if the argument is omitted, and zero otherwise.

Examples

1. The following code defines a macro called `route1` that contains the formal parameters `g1`, `g2`, and `g3`. Note that `g1`, `g2`, and `g3` cannot be operands of a `local` directive. Macro names cannot be redefined.

```
route1 macro g1,g2,g3      ;; macro directive
        ld    reg1, g1      ;; start of macro body
        ld    reg2, #g2
        ld    reg3, g3
loop:   ld    reg4, [reg2]+
        st    reg4, [reg3]+
        dbnz  reg1, loop    ;; end of macro body
        endm                ;; endm directive
```

2. For the following examples, assume `a` and `b` are formal parameters defined in the macro definition, and `x` and `y` are actual arguments passed when the macro is called.

```
check_mac macro nul a      ;; is evaluated as TRUE only
                           ;; if x is omitted.
check_mac macro nul a&b    ;; is evaluated as TRUE only
                           ;; if both x and y are
                           ;; omitted.
check_mac macro nul SP     ;; is evaluated to FALSE.
                           ;; Only macro arguments can
                           ;; be nulls.
```

rept

Function

Repeats a sequence of source code lines.

Syntax

rept *expression*

where:

expression is a valid absolute expression of null segment type containing no forward references.

Description

Use this directive to cause a sequence of source-code lines to be repeated *expression* times. All lines appearing between the **rept** directive and subsequent **endm** directive constitute the block to be repeated.

The assembler inserts the repeat blocks in-line as it encounters the **rept** directive. No explicit call is required to cause the code insertion as the definition is an implied call for expansion.

Example

This macro moves the six words pointed at by `src_reg` to the memory area pointed at by `dst_reg`.

```
rept 6
ld    reg1, [src_reg]+
st    reg1, [dst_reg]+
endm
```

.strlen.

Function

Determine the length of a string.

Syntax

```
.strlen. string
```

where:

string is a valid string enclosed by single quotes.

Description

Use this directive to determine the length of a string. This string must be enclosed by single quotes. It can be used both inside and outside of a macro definition.

Example

The following example illustrates the use of `.strlen.` inside a macro definition:

```
DefStr  MACRO    label, string
label:  DCB      .STRLEN. string
        DCB      string
        ENDM
```

After the following macro call

```
DefStr  MyLab, 'Hello'
```

this will expand to:

```
MyLab:  DCB      .STRLEN. 'Hello' ;result is 5
        DCB      'Hello'
```


Macro Call

Function

Call a macro.

Syntax

```
[label:]  macro_name [actual_arguments ]
```

where:

label is a user-defined label.

macro_name is a user-defined name for the macro.

actual_arguments is a list of up to 16 actual arguments.

Description

The assembler must encounter the macro definition before the first call for that macro. Otherwise, an error occurs. The assembler inserts the macro body identified by *macro_name* each time it encounters a call to a previously defined macro in the program.

The positioning of actual arguments in a macro call is critical since the substitution of parameters is based solely on position. The first listed actual parameter replaces each occurrence of the first listed formal parameter; the second actual argument replaces the second formal parameter, and so on. When coding a macro call, always list actual arguments in the appropriate sequence for that macro.

Blanks are usually treated as delimiters. Therefore, when an actual argument contains blanks (passing the instruction `ld reg1, [reg2]` for example), the parameter must be enclosed in angle brackets (<>), as in `<ld reg1, [reg2]>`. You must also enclose in angle brackets any other delimiter that is to be passed as part of an actual parameter. Carriage returns cannot be passed as actual arguments.

If a macro call specifies more actual arguments than are listed in the macro definition, an error occurs. If fewer parameters appear in the call than in the definition, a null (blank) replaces each missing parameter.

Example

The following example shows two calls for the macro block, defined as follows:

```

block    macro    g1,g2,g3,g4
          local   loop, eom
          ld      reg1, g1
          ld      reg2, #g2
loop:    g3      reg3, [reg2]+
          g4      eom
          dbnz    reg1, loop
eom:
          endm

```

The block macro operates on the g1 words starting at g2 by the opcode g3. Then, it conditionally performs the branch specified by g4. The first time you call the block macro, it gets the average of the list of words making up the block. The second time you call block, it finds the first word greater than or equal to that average. The following code shows the expanded macro lines preceded by plus sign (+).

```

test module main

rseg at 30h
    reg1: dsw 1
    reg2: dsw 1
    reg3: dsw 1
    size: dsw 1
    tab:  dsw 1

block macro g1,g2,g3,g4
    local loop, eom
    ld reg1,g1
    ld reg2, #g2
loop: g3 reg3,[reg2]+
      g4 eom
      dbnz reg1,loop
eom:  endm

```

```

cseg
    ld size, #3
    ld tab, #4
    clr r3
    block size, tab, add, !;
        +1    ld reg1,size      ;macro expansion starts
        +1    ld reg2, #tab
        +1    ??0001: add reg3,[reg2]+
        +1    ; ??0002
        +1    dbnz reg1,??0001
        +1    ??0002:          ; macro expansion ends
    div reg3, #size
    block size, tab, cmp, bge
        +1    ld reg1,size      ; macro expansion starts
        +1    ld reg2, #tab
        +1    ??0003: cmp reg3,[reg2]+
        +1    bge ??0004
        +1    dbnz reg1,??0003
        +1    ??0004:          ; macro expansion ends
end

```

7.6 EMPTY MACRO ARGUMENTS

To be able to pass empty macro arguments that are not at the end of the actual parameter list, you can just omit the argument. The following example illustrates this:

```
MyAddMACROsrc1, src2, res
    IFB  <src2>
        add  res, src1
    ELSE
        add  res, src1, src2
    ENDIF
ENDM
```

A call to the macro MyAdd:

```
MyAdd#a, , R0
```

will expand to:

```
add  R0, #a
```

7.7 NARG SYMBOL

The symbol `narg` is defined only within the expanded macro body and is equal to the actual number of parameters when the macro was called. If empty macro arguments are used between non-empty macro arguments, they are counted as a given argument. If less than the number of formal parameters is given, you can use `narg` to check how many arguments were given. An example:

```
MyAddMACROres, src1, src2
    IF    NARG = 2
        add  res, src1
    ELSE
        add  res, src1, src2
    ENDIF
ENDM
```

7.8 SPECIAL MACRO OPERATORS

The special macro operators help prevent ambiguity in specifying parameters to the assembler. For example, if you want to specify three actual arguments, of which the second is a comma, entering the following line appears to the assembler as a list of four parameters with the second and third parameters missing:

```
macro_name param_1,,param_3
```

To avoid this ambiguity, prefix the comma by an exclamation mark (!), as follows:

```
macro_name param_1,!,,param_3.
```

Placing the comma or any other character after the exclamation mark causes the assembler to treat the character as a literal, possessing no special significance.

When a macro is expanded, the assembler removes any ampersand (&) preceding or following a formal parameter in macro definition and substitutes the actual argument at that point. When the ampersand is not adjacent to a formal parameter, it is not removed and is passed as part of the macro expansion text. Note that the ampersand must be right next to the text being concatenated; intervening blanks are not allowed.

If nested macro definitions contain ampersands, the only ampersands removed are those adjacent to formal parameters belonging to the macro definition currently being expanded. All ampersands must be removed by the time the expansion macro body is performed. Exceptions force illegal character errors. Ampersands are not recognized as operators in comments. See the examples in Section 7.13.

Table 7-1 lists and defines the special macro operators, in addition to the exclamation mark and ampersand.

Operator	Definition
&	The ampersand specifies concatenation (linking) of text and formal parameters.
<>	Angle brackets undefine delimiters that are treated as literal characters instead. Since blanks are delimiters, literal blanks must be enclosed in angle brackets. To pass text to nested macro calls, use one set of angle brackets for each level of nesting.
::	Double semicolons placed before a comment in a macro definition prevent the inclusion of the comment in expansions of the macro and thereby reduce storage requirements. The comment still appear in the listing of the macro definition.
!	The exclamation mark is placed before a character (usually a delimiter) that is to be passed as literal text in an actual argument. To pass a literal exclamation point, enter two exclamation points (! !). The exclamation point is not preserved while building an actual argument (e.g., when substitution is used to build another actual argument). Carriage returns cannot be passed as actual argument.
nul	<p>Use the nul to indicate the omission of a parameter. The omitted (or null) parameter can be represented by two consecutive delimiters (e.g., <code>param_1,,param_3</code>). A null parameter can also be represented by two consecutive single quotes (e.g., <code>" ,param_2,param_3</code>). Notice that a null is not a blank. The blank is the ASCII character represented by 20 hexadecimal. The nul operator has no character representation. A typical use of the nul operator is in if directives; for example,</p> <pre> if nul x&y exitm endif </pre> <p>The example causes macro expansion to terminate if both the actual arguments of <code>x</code> and <code>y</code> are null.</p>

Table 7-1: Macro operators

7.9 NESTING MACRO DEFINITIONS

A macro definition can be contained completely (nested) within the body of another macro definition. The body of a macro consists of all text (including any nested macros) bounded by matching macro and `endm` directives. The assembler allows any number of macro definitions to be nested.

When a higher-level macro is called for expansion, the next lower-level macro cannot be called unless the assembler have called and expanded all higher-level macro definitions.

Since macro names cannot be redefined, the nested macro definition must have the macro name specified as a local symbol unless the enclosing macro is called only once.

Since `irp`, `irpc`, and `rept` blocks constitute macro definitions, they can also be nested within another definition created by `irp`, `irpc`, `rept`, or macro directives. In addition, an element in an `irp` or `irpc` actual argument list (enclosed in angle brackets) can itself be a list of bracketed parameters; that is, lists of parameters can contain elements that are also lists. For example,

```
lists    macro    param1,param2
          ld reg1, param1
          ld reg2, param2
          endm
```

```
lists a,<b,c          ; macro invocation
```

In this example, when the `lists` macro is invoked, `a` is passed to `param1`, and the `b,c` string is literally passed to `param2`.

7.10 MACRO CALLS

Once a macro is defined, you can call the macro any number of times in your program. The call consists of the macro name and any actual arguments that are to replace formal parameters during macro expansion. During assembly, the assembler replaces each macro call by the macro definition code; actual arguments replace formal parameters.



See the Macro Call reference page in this chapter for syntax and examples.

7.10.1 NESTED MACRO CALLS

You can nest macro calls, including any combination of nested `irp`, `irpc`, and `rept` constructs, within macro definitions up to nine levels. The macro being called need not be defined when the enclosing macro is defined or called. However, the macro must be defined before it is called.

A macro definition can also contain nested calls to itself (recursive macro calls) up to nine levels, as long as the recursive macro expansions are terminated eventually. You can control this operation by using the conditional-assembly directives described in Chapter 6.

For example, the macro below, called `recall`, calls itself five times after you call it from elsewhere in your program.

```
param1 set 5

recall macro
    if param1 ne 0
        param1 set param1 -1
        recall           ; recursive call
    endif
endm
```

7.11 MACRO EXPANSION

When you call a macro, the actual arguments to be substituted into the prototype code can be passed in one of two modes. Normally, the substitution of actual arguments for formal parameters is simply a text substitution. The parameters are not evaluated until the macro is expanded.

If a percent sign (%) precedes the actual argument in the macro call, the assembler evaluates the argument immediately before any expansion occurs. The assembler then passes this argument as a decimal number representing the value of the parameter. You cannot use the percent sign with the `irpc` macro directive.

The normal method for passing actual parameters works for most applications. Using the percent sign to pre-evaluate parameters is necessary only when the value of the parameter is different within the local context of the macro definition as compared to its global value outside the macro definition.

The following macro generates a number of multi-byte arithmetic instructions. The parameters passed in the macro call determine the number of bytes in an operand, and whether an addition or subtraction operation is to be performed. The addresses of the source and destination operand are assumed to be in reg1 and reg2, respectively. Some typical calls for this macro is as follows:

```
mbyte    'a',5
mbyte    's',%count-1
```

The second call shows an expression used as an argument. The assembler evaluates this expression immediately rather than passing it simply as text.

The definition of the `mbyte` macro follows. This macro uses the conditional `if` directive to test the validity of the first parameter. Also, the `rept` macro directive is nested within the `mbyte` macro.

```
mbyte    macro    x,y
            if (x ne 'a') and (x ne 's')
                exitm
            endif
            clrc
            rept y
                ldb reg3,[reg1]
                if x eq 'a'
                    addcb reg3,[reg2]+
                else
                    subcb reg3,[reg2]+
                endif
                stb reg3,[reg1]+
            endm
        endm
```

The indentation shown in the definition of the `mbyte` macro illustrates the relationship of the `if`, `else`, `endif` directives and the `rept` and `endm` directives. Such indentation is not required in the program.

The `mbyte` macro generates nothing if the first parameter is neither a nor s. Therefore, the following calls produce no code.

```
mbyte    5
mbyte    'b',5
```

The result in the object program is as though the `mbyte` macro does not appear in the source program.

The following is another form of the `mbyte` macro. Note that in this definition uses concatenation to form the `addcb` or `subcb` instruction code. If a call to the `mbyte` macro specifies a string other than `add` or `sub`, the assembler reports an error. The assembler flags all operation codes as undefined symbols.

```
mbyte    macro    x,y
          clr     clrc
          rept    y
              ldb  reg3,[reg1]
              x&cb reg3,[reg2]+
              stb  reg3,[reg1]+
          endm
endm
```

7.12 NULL MACROS

A macro can legally begin and end only with the `macro` and `endm` directives, respectively. Thus, the following is a legal macro definition:

```
nada     macro    p1,p2,p3,p4
          endm
```

A call to this macro produces no source code and therefore has no effect on the program.

The null (or empty) macro body has a practical application. For example, all the macro prototype instruction might be enclosed with `if/endif` conditional directives. When none of the specified conditions is satisfied, all that remains of the macro is the macro directive and the `endm` directive.

7.13 SAMPLE MACROS

The following examples further demonstrate the use of macro directives and operators.

Example 1: Nesting of `irpc` within a macro

The following macro definition contains a nested `irpc` directive. Note that the third operand of the outer macro becomes the character string for the `irpc`.

```

move    macro    x,y,z
          irpc    param,z
              ld    reg1,x&param
              st    reg1,y&param
          endm
endm

```

Assume that the program contains the call `move src,dst,123`. The assembler passes the third parameter (123) of this call to the `irpc`, the same effect as coding `irpc param,123`. When expanded, the `move` macro generates the following source code:

```

          ld      reg1,src1
          st      reg1,dst1
          ld      reg1,src2
          st      reg1,dst2
          ld      reg1,src3
          st      reg1,dst3

```

Note that concatenation forms the labels in this example.

Example 2: Nested macros used to generate `dc` directives

This example generates a number of `dc` directives, each with its own label. Two macros are used for this purpose: `genlab` and `block`. The `genlab` macro is defined as follows:

```

          genlab  macro    f1,f2
$save gen
          f1&f2:  dc      0          ; generate labels & dcbs
$restore
          endm

```

The block macro, which accepts the number of dcbs to be generated (numb) and a label prefix (prefix), is defined as follows:

```

        block  macro  numb,prefix
$save  nogen
                count  set  0
                rept  numb
                    count  set  count+1
                    genlab  prefix,%count ;nested macro call
                endm
$restore
        endm

```

The macro call block 3,lab generates the following source code:

```

        block  3,lab
lab1:   dcb      0
lab2:   dcb      0
lab3:   dcb      0

```

The assembler controls specified in these two macros (the lines beginning with \$) clean up the assembly listing for easier reading. The source code shown for the call block 3,lab is what appears in the listing file when the controls are used. Without the controls, the assembly listing appears as follows:

```

block 3,lab
+1      count    set  0
+1      rept     3
+1          count    set  count+1
+1          genlab   lab,%count
+1      endm
+2      count    set  count+1
+2      genlab   lab,%count
+3 lab1:   dcb      0
+2      count    set  count+1
+2      genlab   lab,%count
+3 lab2:   dcb      0
+2      count    set  count+1
+2      genlab   lab,%count
+3 lab3:   dcb      0

```

Example 3: A Macro that Converts Itself into Routine

In some cases, the in-line coding substituted for each macro call imposes an unacceptable memory requirement. The next two examples show two different methods for converting a macro call into a routine call. The first time you call the `sbmac` macro, it generates a full line substitution that defines the `subr` routine. Each subsequent call to `sbmac` generates only a `call` instruction to the `subr` routine.

Within the following examples, the label `subr` must be global so that it can be called from outside the first expansion. Such calls are possible only when that part of the macro definition containing the global label is called only once in the entire program.

Example 3, Method 1: Conditional Assembly.

This method of altering the expansion of the sample macro, `sbmac`, uses conditional assembly. Use the variable `first` as a switch based upon whether the value of `first` when tested is `true` or `false`. The variable `first` is set `true` just before the first call for `sbmac`. The macro `sbmac` is defined as follows:

```

rseg
reg1:  dsw 1
reg2:  dsw 1

true   equ 0ffh
false  equ 0
first  set true

sbmac  macro
      call subr
      if first
          first set false
          br    dun
subr:  ld reg1, #01
      ld reg2, #02
      ret
dun:
      endif
      endm

```

The first call to `sbmac` expands the full definition (assembled with `gen` control), including the call to and definition of `subr`:

```
sbmac
+1      call  subr
+1      if    first
+1          first set  false
+1          br   dun
+1  subr:
+1      ld reg1, #01
+1      ld reg2, #01
+1      ret
+1  dun:
+1      endif
```

Because `first` is true when encountered during the first expansion of `sbmac`, ASM196 assembles all the statements between `if` and `endif` into the program. In subsequent calls, since `first` is set to `false`, the assembler skips the conditionally-assembled code, meaning no opcode is produced even though they appear in the list file. The assembler only generates opcode for the following statements :

```
submac
+1      call  subr
```

Example 3, Method 2: Conditional assembly with `exitm`.

The other method of altering the expansion of `sbmac` also uses conditional assembly, but uses the `exitm` directive to suppress unwanted macro expansion after the first call. The `exitm` directive is effective when `first` is `false`, the case after the first call to `sbmac`.

```
true    equ    0ffh
false   equ    0
first   set    true

rseg
reg1:   dsw    1
reg2:   dsw    1

sbmac   macro
        call  subr
        if first
            first set  false
            br   dun
        else
```

```

        exitm
    endif
subr:   ld reg1, #01
        ld reg2, #02
        ret
dun:
        endm

```

In subsequent calls, only the following lines appear in the list file:

```

sbmac
+1      call subr
+1      if first
+1          first set false
+1      br dun
+1      else
+1          exitm

```

The assembler only generates opcode for the call to subr.

Example 4: Computed goto macro

This sample macro presents an implementation of a computed goto for the 80C196. The computed goto, a common feature of many high-level languages, allows the program to jump to one of a number of different locations depending on the value of a variable. For example, if the variable has the value 0, the program jumps to the first item in the list; if the variable has the value 3, the program jumps to the fourth item, and so on. In this example, the first parameter is the address of the branch table (specified via a sequence of `dcws`), and the second is the table index.

```

tjump macro i,t                ; Jump to i-th addr in
table t
    ld     reg1,i              ; Compute offset of entry
    add    reg1, reg1          ; "
    ld     reg1, t[reg1]       ; Get address of entry
    br     [reg1]              ; Indirect branch
endm

```

Example 5: Using *irp* to define the jump table

The `tjump` macro becomes even more useful when a second macro (`goto`) defines the jump table, loads the address of the table, and then calls `tjump`. The `goto` macro is defined as follows:

```
goto    macro    index,list
        local    jtab
        tjump    index,jtab    ; ld accum with index
jtab:
        irp      formal,<list>
            dcw   formal        ; set up table
        endm
endm
```

A typical call to the `goto` macro appears in this example:

```
goto    case,<count,timer,date,ptr>
```

This call to the `goto` macro builds a table of `dcw` directives for the labels `count`, `timer`, `date`, and `ptr`. The macro `goto` then calls the `tjump` macro. If the value of the variable `case` is 2 when the `goto` macro is called, the `goto` and the `tjump` macros together cause a jump to the address of the `date` routine.

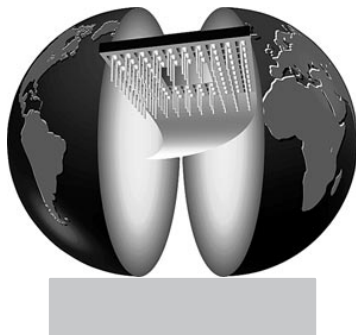
You can specify any number of addresses in the list for the `goto` routine as long as they all fit on a single source line.

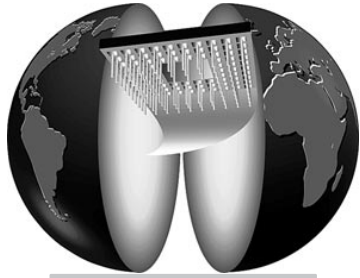


CHAPTER

8

MESSAGES AND ERROR RECOVERY





8

CHAPTER

This chapter lists all messages produced by the assembler. Where possible, it explains the cause of the error and possible remedies.

8.1 CONSOLE OUTPUT

During assembly, ASM196 produces the following output to the screen: the sign-on message, fatal error messages, and the sign-off message.

8.1.1 SIGN-ON MESSAGE

The sign-on message appears in the following format:

```
80C196 macro assembler vx.y rz SN00000-005 (c) year TASKING, Inc.
```

where:

vx.y identifies the version of the assembler.

rz identifies the revision of the assembler.

year identifies the copyright year.

8.1.2 ERROR MESSAGES

Error messages are discussed in the next section.

8.1.3 SIGN-OFF MESSAGE

The sign-off message is displayed on the screen and in the listing file when the assembly is terminated. The format of the sign-off messages is:

```
ASSEMBLY COMPLETED, nnnn ERROR(S) FOUND
```

where:

nnnn is the number of errors found. NO appears if no errors are found.

8.2 ERROR MESSAGES AND RECOVERY

The ASM196 assembler issues three types of error messages: fatal error messages, warnings and source file error messages. Fatal errors terminate the assembly process. The messages are displayed on the screen. Example of fatal error messages are invocation line errors, internal errors, and I/O errors. Warnings and source file errors do not terminate the assembly process. The object file produced, however, might not be executable. Correct the errors by looking at the error messages in the listing or errorprint file and then reassemble the program.

8.2.1 FATAL ERROR MESSAGES

Upon detecting a fatal error within the system hardware or on the invocation line, ASM196 prints a message on the screen, terminates the assembly processing, and returns control to the host system.

Fatal error messages can be caused by the following:

- invocation-line errors
- I/O errors
- insufficient-host-memory errors
- internal-synchronization errors

8.2.1.1 ASM196 ERROR MESSAGES

The assembler displays fatal ASM196 errors in the following form:

```
FATAL ASM196 ERROR num: message
```

```
ASM196 TERMINATED
```

where:

num is an error number.

message is a message describing the cause of the error.

The following list of error messages provides their decimal codes and their meanings.

1 Synchronization error

This error should not occur. If it does, report it to your local TASKING representative.

2 Insufficient memory

Remove some processes from memory or break down your program into smaller pieces or add more memory.

3 File not found

The assembler cannot open the file. Check the filename or directory.

4 File write**5 File not created****7 NEAR/FARCONST obsolete, same as NEAR/FARDATA**

Use the NEARDATA and FARDATA instead of NEARCONST and FARCONST respectively.

8 MODEL(EX) obsolete, use MODEL(NT)

Specify a 24-bit model, NT or NP.

9 Invalid model name: name

Replace *name* by a valid model name. See the description of the `model` control for details.

11 Unrecognized control or misplaced primary

This message indicates that an invalid control was used or a primary control is specified after one or more source lines. A primary control must be specified before any source line, and cannot be specified more than once. Recheck each control and its position in your source file.

13 Missing parameter for control

A parameter was omitted for a control that requires a parameter. Include a parameter with the control.

69 Pathname too long

Reduce the pathname.

8.2.1.2 ARGUMENT ERROR MESSAGES

The assembler displays fatal argument errors in the following form:

```
FATAL ARGUMENT ERROR num: message
```

where:

num is an error number.

message is a message describing the cause of the error.

The following list of error messages provides their decimal codes and their meanings.

0 *Unexpected end of argument: arg*

Check and correct the syntax.

1 *Control or option cannot be negated: name*

The control *name* cannot have a no prefix, or the option *name* cannot have a minus sign appended. Remove the negation.

2 *Syntax error in control: control*

Check and correct the syntax.

3 *Argument expected for control or option: name*

Specify an argument to *name*.

4 *Syntax error in option: option*

Check and correct the syntax.

5 *Unknown option specified: name*

Replace *name* with the correct option.

6 *Maximum depth in buffer stack reached*

The control or option has too many argument levels. Reduce the number of argument levels.

7 *Buffer stack is empty*

This error should not occur. If it does, report it to your local TASKING representative.

8 *Argument too long*

Reduce the length of the argument.

9 *Unexpected argument for control: name*

Control *name* cannot have an argument. Remove the argument.

10 *Unexpected internal error: message*

This error should not occur. If it does, report it to your local TASKING representative.

8.2.1.3 MEMORY ERROR MESSAGES

The assembler displays fatal memory errors in the following form:

```
FATAL MEMORY ERROR num: message
```

where:

num is an error number.

message is a message describing the cause of the error.

The following list of error messages provides their decimal codes and their meanings.

0 *Cannot allocate memory block of size size***1 *Cannot reallocate memory block to size size***

There is not enough memory left to allocate. Remove some processes from memory or break down your program into smaller pieces or add more memory.

8.2.1.4 I/O ERROR MESSAGES

The assembler displays fatal I/O errors in the following form:

```
FATAL I/O ERROR num: message
```

where:

num is an error number.

message is a message describing the cause of the error.

The following list of error messages provides their decimal codes and their meanings.

0 *Unexpected end of file detected*

Check your file.

1 *Cannot write to standard input*

Specify a file or standard output to write to. Standard input is used for input only.

2 *Cannot read from standard output*

Specify a file or standard input to read from. Standard output is used for output only.

3 *Filename too long*

Give your file a shorter name.

4 *Filename not conform DOS standard*

Check your DOS Reference Manual for the correct filename syntax.

5 *Cannot read from null device*

Specify another device or filename to read from.

6 *Cannot rename :WORK:*

This is a temporary file. So, you cannot rename it.

8.2.2 WARNING MESSAGES

Warning messages do not terminate the assembly process. The assembler still produces an object file and a listing file. In the listing file, an error line appears following the source statement that caused the warning. Correct any warning and reassemble the program before attempting to execute it. While not fatal, warnings can cause the object code to be unexecutable.

The assembler displays warning messages in the form:

```
WARNING num: message
```

where:

num is an error number.

message is a message describing the cause of the error.

The following list of warning messages provides their decimal codes and their meanings.

6 *Control character ignored in next line: char*

The control character *char* is ignored.

88 *Variable 'name' in next line used with multiple addressing modes*

Use the variable *name* in one addressing mode only, otherwise the object code may be unexecutable.

8.2.3 SOURCE FILE ERROR MESSAGES

Source file error messages do not terminate the assembly process. The assembler still produces an object file and a listing file. In the listing file, an error line appears following the source statement that caused the error. The error is indicated by three asterisks (***), the word ERROR, a number sign (#) followed by the error number and error message. Correct any source file error and reassemble the program before attempting to execute it. While not fatal, source file errors can cause the object code to be unexecutable. The following list of source file error messages provides their decimal codes and their meanings.

10 *Syntax error*

The line contains a syntax error. Check and correct the syntax.

11 *Unrecognized control or misplaced primary*

12 *Primary control specified more than once*

This message indicates that an invalid control was used or a primary control is specified after one or more source lines. A primary control must be specified before any source line, and cannot be specified more than once. Recheck each control and its position in your source file.

13 Missing parameter for control

A parameter was omitted for a control that requires a parameter. Include a parameter with the control.

14 No parameter allowed with control

A parameter was specified for a parameterless control. Remove the parameter.

15 Invalid number

A number contains illegal characters for its base, or its value exceeds 429,496,725 ($2^{32}-1$). Make sure that the number is within its allowable range.

16 Bad parameter to control

The parameter of the control is illegal. Recheck the type of parameter accepted by the control.

18 Contradicting controls

One or more of the following combinations of controls have been specified:

xref	+	nosymbols or noprint
noprint	+	symbols, pagewidth, or pagelength
noobject	+	debug or linedebug
oldobject	+	omf
nosource	+	omf(2) (and higher)
nearcode	+	farcode
nearconst	+	farconst
neardata	+	fardata

Avoid using these combinations.

19 Save-Restore stack overflow

Your program contains too many saves without corresponding restores. The maximum nesting is nine, meaning nine consecutive saves can be done before a restore. Make sure that the number of saves do not exceed nine before a restore.

20 Save-Restore stack underflow

Your program contains a restore control without a corresponding save. Make sure that a save precedes a restore.

21 Invalid symbol name

Check the spelling of the symbol and replace it by an existing symbol name.

22 Conflict between model and symbol name

The symbol name is not valid for the specified model. Choose the correct model.

25 Include not rightmost control**26 Include stack overflow**

The include control is not the last control on the control line or the nesting level is too deep. The maximum nesting is nine. Place the include control after any other control on the control line. Check that nesting level does not exceed nine.

27 Input line too long

The maximum length of each source line is 255 characters including the terminating line feed. Do not exceed 255 characters.

28 Invalid character

A misplaced character or a control character was embedded in the input. The character is replaced in the listing line by a #. Delete the misplaced character or retype the line to delete the control character.

29 Missing apostrophe

A string was not terminated. Insert the closing apostrophe.

30 Invalid character at end of line

The end of the file was encountered before a line feed. Make sure that the file is terminated by a line feed.

31 Parse stack overflow

The statement is too complicated. Break the statement into smaller parts.

32 Text found beyond end statement - ignored**33 Premature end of file**

Text was found after the assembler encountered the end statement, or no end statement was found at the end of the source file.

34 Statement not allowed in this context

Incorrect directives are used in certain statements so the assembler took no action on the statement. Check the following:

<code>end</code> statement	must be the last line in the source program.
<code>local</code> and <code>exitm</code> macro directives	are valid only within macro definitions.
<code>endm</code> macro directive	is valid only when terminating a macro definitions.
<code>else</code> directive	is valid only between <code>if</code> and <code>endif</code> directives.
<code>endif</code> directive	is valid only when matching a preceding <code>if</code> directive.
<code>module</code> directive	can be preceded only by empty statements and control lines.
<code>org</code> directive	cannot appear before the first segment selection directive.
Machine instruction and code definition directives	must appear in code segments.
Storage reservation directives	can appear only within register, overlay, or data segments.

35 Expression stack overflow

The expression nesting is too deep. Break the expression down.

36 Invalid relocatable expression

The relocatable expression contains an illegal operation or operand. Check expression syntax.

37 Attempt to divide by zero

The expression contains a zero divider. Do not divide by zero.

38 Undefined symbol

The line contains an undefined symbol. Check the symbol name spelling. Define if the symbol is undefined.

39 *Illegal binary operation*

Binary operation is executed on non-null segment type operands. Binary operation is illegal if both operands have a non-null segment type, except for subtraction and comparing operands that belong to the same segment.

40 *Invalid code address*

Code address indicated is below 255. Code address must have a code or null segment type and be above 255.

41 *Invalid register address*

Register address was above 256. Register address must have a register or null segment type and be below 256.

42 *Data type not compatible with segment type*

You cannot use the entry attribute to specify the data type of a non-code or non-null expression. Assign entry attribute only to a code segment type or null expression over 256.

43 *Public attribute not allowed for this symbol*

You cannot specify the public attribute to a macro or module definition, or to an external symbol. Delete the public attribute.

44 *External reference not allowed in this context*

External references are allowed only in code generation statements (machine instructions and the `dc`b and `dc`w directives).

45 *Expression with forward reference not allowed*

Forward reference is not allowed for this statement (e.g., `dsb` directive).

46 *Absolute expression expected*

The directive requires an absolute expression as an operand.

47 *Symbol already defined*

The symbol name was redefined. Symbol cannot be redefined unless it is first defined by a `set` directive and it is redefined by another `set` directive.

48 Invalid bit number

The bit number used was less than 0 or greater than 7. A bit number must be an expression in the range of 0 to 7.

49 Alignment error

The address or offset is not properly aligned. Be sure symbols are aligned properly. Word symbols are on two-byte alignment. Long symbols are on four-byte alignment.

50 Reference not to current segment

The expression used with `org` directive is out of the current segment. The `org` directive requires that the expression refer to the current segment or be absolute and neutral.

51 Location counter too low

The address specified is below 256. The operand of a `cseg at` directive must be at least 256. The operand of an `org` directive must not point below the base address of the segment (applicable only within an absolute segment).

52 Location counter overflow

The location counter is incremented beyond 16777216 (2^{24}) for `far`, beyond 65535 (2^{16}) for `near` code and data and constant segments, or beyond 256 for register segment. Locate the segment lower in memory or reduce the size of the segment.

53 Decision list overflow

Too many generic jumps. Further generic jumps are expanded in their long form. Reduce the number of generic jumps (`br` instructions). Use `l jmp` or `s jmp` where possible.

54 Expression not within range

This error can be caused by the following:

- A register expression is not in the range of 0 to 255.
- An immediate count in a shift is not in the range of 0 to 15.
- A register count in a shift is not in the range of 16 to 255.
- An immediate operand in byte instructions or a `decb` constant is not in the range of -256 to +255.

- The offset in the `jbs`, `jbc`, or `djnz` instruction is not in the range of -128 to +127.
- The offset in the `sjmp` or `scall` instruction is not in the range of -1024 to +1023.

55 Conditional assembly stack overflow

Your program contains too many `if` statements without corresponding `endifs`. The maximum nesting is nine. Make sure that there is at least one `endif` statement after nine consecutive `ifs`.

56 Missing endif

An `if` block was not terminated until the end of the macro expansion, or until the assembler reached the `end` directive or the end of the file. Terminate `if` blocks with an `endif` directive.

57 Too many formal parameters

The number of formal macro parameters (in the macro definition) exceeded 16.

58 Too many local parameters

The number of local parameters, parameters designated by the `local` directive, exceeded 16.

59 Macro stack overflow

The macro call nesting is too deep. The maximum nesting is nine.

60 Too many actual parameters

A macro is called with more actual arguments than formal parameters in its definition. Pass the same number of arguments as the number of parameters inside the definition.

61 Missing endm

A macro definition was not terminated until the assembler reached the `end` directive, or the end of the file. Terminate the macro definition with `endm`.

62 Stacksize not even

The `stacksize` attribute given is not an even size.

63 Floating point constant underflow

The absolute value of floating-point constant is not above 1.17E-38.

64 Floating point constant overflow

Absolute value of floating point constant above 3.37E38.

Absolute value of floating point constants must be below 3.37E38.

65 Invalid segment attribute

All code segments in a module must be either near or far.

Assemble with the `nearcode` or `farcode` control or specify near or far with the `cseg` directive.

66 Instruction invalid in model(ex) compatible mode

Extended jump and call instructions as well as instruction generating them are not allowed in the `compatible` mode.

Make sure that you are using the extended 24-bit model.

67 Location counter too high

The operand of the directive is too high.

Keep the following rules in mind:

1. The operand of a `cseg at` directive must be at most 0FFFFH if a 24-bit model is not specified and 0FFFFFFH if a 24-bit model is specified.
2. The operand of an `rseg at` or `oseg at` directive must be at most 0FFH.
3. The operand of `dseg at`, `odseg at` and `kseg at` directives must be at most 0FFFFH if a 24-bit model is not specified or segment type is near, and 0FFFFFFH if a 24-bit model is specified and the segment type is far.
4. The same holds true for operands of the corresponding `org` directives in absolute segments.
5. The operand of an `org` directive in the relocatable segment must be at most a) 0FFFFH for nonregister and 0FFH for register segments if a 24-bit model is not specified, b) 0FFFFFFH for nonregister far, 0FFFFH for nonregister near, and 0FFH for register segments when a 24-bit model is specified.

68 LSW and MSW not allowed in this context

Use LSW and MSW to get the lower or upper 16 bits of a 32-bit value.

69 Pathname too long

Reduce the pathname.

70 Evaluation version can only generate absolute files**71 Exceeded code size limit of evaluation version**

You have a restricted version of the assembler. Contact TASKING for a registered version.

72 Search path count overflow, ignoring path: path

You specified too many search paths. Reduce the number of paths.

73 Multiple type indices for type POINTER

The type POINTER has two different indices. Check your source code for redeclaration of POINTER.

75 Cannot change addressing mode for variable

An addressing mode is requested with the @-character which cannot be generated for this instruction.

76 error_message

A user specified error. See the error control.

77 Missing closing angle bracket

An identifier string was not terminated. Insert the closing '>'.

78 Invalid symbol name or string constant

An invalid string is used in an IFB, IFNB, IFIDN or IFNIDN statement.

79 Control not allowed on the command line: control

The ERROR control has been used on the command line. See the error control.

80 Invalid floating point expression

An operand is non-integral, but the operator requires integral operands. That is, for example, NOT, AND, OR, XOR all require integral operands.

81 Expression evaluation stack overflow

The expression which is being evaluated is too complex. Try to simplify the expression.

82 Expression evaluation stack underflow

This error should not occur. If it does, report it to your local TASKING representative.

83 Cannot swap expressions on evaluation stack

This error should not occur. If it does, report it to your local TASKING representative.

84 Character string value expected

A string is expected.

85 Expression contains unresolved references

An expression is used which could not be solved. Check for the presence of external variables in this expression and replace them with local variables.

86 Operator "operator" can only be used in absolute expression

The LOW or HIGH keyword is not used with an absolute expression.

87 Missing colon after label declaration: label

A colon is missing after a label declaration. Add a colon after the label.

100 Internal fatal error, please report: message

This error should not occur. If it does, report it to your local TASKING representative.

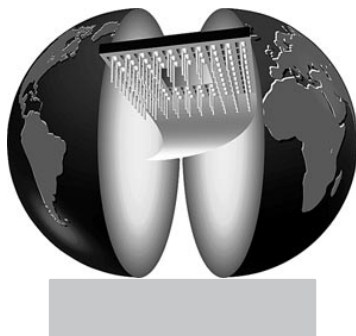
101 This DEMO ASM196 has reached its limit.

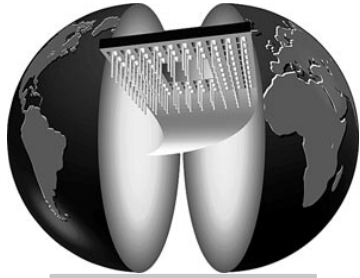
You have a restricted demo version of the assembler. Contact TASKING for a registered version.

APPENDIX

FLEXIBLE LICENSE MANAGER (FLEXLM)

A





A

APPENDIX

1 INTRODUCTION

This appendix discusses Highland Software's Flexible License Manager and how it is integrated into the TASKING toolchain. It also contains descriptions of the Flexible License Manager license administration tools that are included with the package, the daemon log file and its contents, and the use of daemon options files to customize your use of the TASKING toolchain.

2 LICENSE ADMINISTRATION

2.1 OVERVIEW

The Flexible License Manager (FLEXlm) is a set of utilities that, when incorporated into software such as the TASKING toolchain, provides for managing access to the software.

The following terms are used to describe FLEXlm concepts and software components:

feature	A feature could be any of the following: <ul style="list-style-type: none">• A TASKING software product.• A software product from another vendor.
license	The right to use a feature. FLEXlm restricts licenses for features by counting the number of licenses for features in use when new requests are made by the application software.
client	A TASKING application program.
daemon	A process that "serves" clients. Sometimes referred to as a <i>server</i> .
vendor daemon	The daemon that dispenses licenses for the requested features. This daemon is built by an application's vendor, and contains the vendor's personal encryption code. Tasking is the vendor daemon for the TASKING software.

license daemon

The daemon process that sends client processes to the correct vendor daemon on the correct machine. The same license daemon is used by all applications from all vendors, as this daemon neither performs encryption nor dispenses licenses. The license daemon processes no user requests on its own, but forwards these requests to other daemons (the vendor daemons).

server node A computer system that is running both the license and vendor daemon software. The server node will contain all the dynamic information regarding the usage of all the features.

license file An end-user specific file that contains descriptions of the server nodes that can run the license daemons, the various vendor daemons, and the restrictions for all the licensed features.

The TASKING software is granted permission to run by FLEXlm daemons; the daemons are started when the TASKING toolchain is installed and run continuously thereafter. Information needed by the FLEXlm daemons to perform access management is contained in a license data file that is created during the toolchain installation process. As part of their normal operation, the daemons log their actions in a daemon log file, which can be used to monitor usage of the TASKING toolchain.

The following sections discuss:

- Installation of the FLEXlm daemons to provide for access to the TASKING toolchain.
- Customizing your use of the toolchain through the use of a daemon options file.
- Utilities that are provided to assist you in performing license administration functions.
- The daemon log file and its contents.

For additional information regarding the use of FLEXlm, refer to the chapter *Software Installation*.

2.2 PROVIDING FOR UNINTERRUPTED FLEXLM OPERATION

TASKING products licensed through FLEXlm contain a number of utilities for managing licenses. These utilities are bundled in the form of an extra product under the name SW000098. TASKING products themselves contain two additional files for FLEXlm in a *flexlm* subdirectory:

Tasking	The Tasking daemon (vendor daemon).
license.dat	A template license file.

If you have already installed FLEXlm (e.g. as part of another product) then it is not needed to install the bundled SW000098. After installing SW000098 the directory `/usr/local/flexlm` will contain two subdirectories, `bin` and `licenses`. The exact location may differ if FLEXlm has already been installed as part of a non-TASKING product but in general there will be a directory for executables such as `bin`. That directory must contain a copy of the **Tasking** daemon shipped with every TASKING product. It also contains the files:

lmgrd	The FLEXlm daemon (license daemon).
lm*	A group of FLEXlm license administration utilities.

Next to it, a `licenses` directory must contain a file with all licenses. If you did install SW000098 then the `licenses` directory will be empty. In that case the `license.dat` file from the product should be copied to the `licenses` directory after filling in the data from your license data sheet. Be very careful not to overwrite an existing `license.dat` file because it contains valuable data.

Example `license.dat`:

```
SERVER HOSTNAME HOSTID PORT
DAEMON Tasking /usr/local/flexlm/bin/Tasking
FEATURE SW008002-32 Tasking 3.000 EXPDATE NUSERS PASSWORD SERIAL
```

After modifications from a license data sheet (example):

```
SERVER elliot 5100520c 7594
DAEMON Tasking /usr/local/flexlm/bin/Tasking
FEATURE SW008002-32 Tasking 3.000 1-jan-00 4 0B1810310210A6894 "123456"
```


If the `license.dat` file already exists then you should make sure that it contains the DAEMON and FEATURE lines from your license data sheet. An appropriate SERVER line should already be present in that case. You should only add a new SERVER line if no SERVER line is present. The third field of the DAEMON line is the pathname to the **Tasking** daemon and you may change it if necessary.

If the pathname of the resulting license file differs from:

```
/usr/local/flexlm/licenses/license.dat
```

then you must set the environment variable **LM_LICENSE_FILE** to the correct pathname. If you have more than one product using the FLEXlm license manager you can specify multiple license files by separating each pathname (*lfpath*) with a ':' :

```
setenv LM_LICENSE_FILE lfpath[:lfpath]...
```

When the main license daemon **lmgrd** already runs it is sufficient to type the command:

```
lmreread
```

for notifying the daemon that the `license.dat` file has been changed. Otherwise, you must type the command:

```
lmgrd >/usr/tmp/lmgrd.log &
```

Both commands reside in the flexlm bin directory mentioned before.

2.3 DAEMON OPTIONS FILE

It is possible to customize the use of TASKING software using a daemon options file. This options file allows you to reserve licenses for specified users or groups of users, to restrict access to the TASKING toolchain, and to set software timeouts. The following table lists the keywords that are recognized at the start of a line of a daemon options file.

Keywords	Function
RESERVE	Ensures that TASKING software will always be available to one or more users or on one or more host computer systems.
INCLUDE	Allows you to specify a list of users who are allowed exclusive access to the TASKING software.
EXCLUDE	Allows you to disallow certain people use of the TASKING software.
GROUP	Allows the specification of a group of users for use in the other commands.
TIMEOUT	Allows licenses that are idle to be returned to the free pool, for use by someone else.
NOLOG	Causes messages of the specified type to be filtered out of the daemon's log output.

Table A-1: Daemon options file keywords

In order to use the daemon options capability, you must create a daemon options file and list its pathname as the fourth field on the **DAEMON** line for the **Tasking** daemon in the license file. For example, if the daemon options were in file `/usr/local/license.opt`, then you would modify the license file **DAEMON** line as follows:

```
DAEMON Tasking /usr/local/Tasking /usr/local/license.opt
```

A daemon options file consists of lines in the following format:

```
RESERVE      number feature{USER | HOST | DISPLAY | GROUP} name
INCLUDE      feature{USER | HOST | DISPLAY | GROUP} name
EXCLUDE      feature{USER | HOST | DISPLAY | GROUP} name
GROUP        name <list_of_users>
TIMEOUT      feature timeout_in_seconds
NOLOG        {IN | OUT | DENIED | QUEUED}
REPORTLOG    file
```

Lines beginning with the sharp character (#) are ignored, and can be used as comments. For example, the following options file would reserve one copy of feature SWxxxxxx-xx for user “pat”, three copies for user “lee”, and one copy for anyone on a computer with the hostname of “terry”; and would cause QUEUED messages to be omitted from the log file. In addition, user “joe” and group “pinheads” would not be allowed to use the feature SWxxxxxx-xx:

```
GROUP      pinheads moe larry curley
RESERVE 1  SWxxxxxx-xx USER pat
RESERVE 3  SWxxxxxx-xx USER lee
RESERVE 1  SWxxxxxx-xx HOST terry
EXCLUDE    SWxxxxxx-xx USER joe
EXCLUDE    SWxxxxxx-xx GROUP pinheads
NOLOG      QUEUED
```

2.4 LICENSE ADMINISTRATION TOOLS

The following utilities are provided to facilitate license management by your system administrator. In certain cases, execution access to a utility is restricted to users with root privileges. Complete descriptions of these utilities are provided at the end of this section.

lmstat

License administration is simplified by the **lmstat** utility. **lmstat** allows you to instantly monitor the status of all network licensing activities. **lmstat** allows a system administrator to monitor license management operations including:

- Which daemons are running
- Users of individual features
- Users of features served by a specific DAEMON

The usage of **lmstat** is as follows:

```
lmstat      [-a] [-S [DAEMON]] [-f [feature]]
            [-s [server]] [-t value] [-c license_file]
            [-A] [-l [regular expression]]

-a          - Display everything
-A          - List all active licenses
-c license_file - Use "license_file"
-S [DAEMON] - List all users of DAEMON's features
-f [feature_name] - List users of feature(s)
-l [regular expression] - List users of matching license(s)
-s [server_name] - Display status of server node(s)
-t value    - Set lmstat timeout to "value"
```

lmdown

The **lmdown** utility allows for the graceful shutdown of all license daemons (both **lmgrd** and all vendor daemons, such as **Tasking**) on all nodes. To use **lmdown**, simply type "lmdown" with the correct license file in either /usr/local/license.dat, or the license file pathname in the environment variable LM_LICENSE_FILE. In addition, **lmdown** takes the "-c license_file_path" argument to specify the license file location. Since shutting down the servers will cause loss of licenses, execution of **lmdown** is restricted to users with root privileges.

lmremove

The **lmremove** utility allows the system administrator to remove a single user's license for a specified feature. This could be required in the case where the licensed user was running the software on a node that subsequently crashed. This situation will sometimes cause the license to remain unusable. **lmremove** will allow the license to return to the pool of available licenses.

lmremove is used as follows:

```
lmremove [-c file] feature user host display
```

lmremove will remove all instances of "user" on node "host" on display "display" from usage of "feature". If the optional -c file is specified, the indicated file will be used as the license file. Since removing a user's license can be disruptive, execution of **lmremove** is restricted to users with root privileges.

lmreread

The **lmreread** utility will cause the license daemon to reread the license file and start any new vendor daemons that have been added. In addition, all pre-existing daemons will be signaled to re-read the license file for changes in feature licensing information. Usage is:

lmreread [-c *license_file*]



If the **-c** option is used, the license file specified will be read by **lmreread**, NOT by **lmgrd**; **lmgrd** re-reads the file it read originally. Also, **lmreread** cannot be used to change server node names or port numbers. Vendor daemons will not re-read their option files as a result of **lmreread**.

3 FLEXLM USER COMMANDS

lmdown(1)

Name

lmdown – graceful shutdown of all license daemons

Synopsis

lmdown [**-c** *license_file*] [**-q**]

Description

lmdown allows the system administrator to send a message to every license daemon asking it to shut down. The license daemons write out their last messages to the log file, close the file, and exit. All licenses which have been given out by those daemons will be revoked, so that the next time a client program goes to verify his license, it will not be valid.

Options

-c *license_file*

Use the specified *license_file*. If no **-c** option is specified, **lmdown** looks for the environment variable **LM_LICENSE_FILE** in order to find the license file to use. If that environment variable is not set, **lmdown** looks for the file `/usr/local/flexlm/licenses/license.dat`.

-q

Quiet mode. If this switch is not specified, **lmdown** asks for confirmation before asking the license daemons to shut down. If this switch is specified, **lmdown** will not ask for confirmation.



lmgrd(1), lmstat(1), lmrmread(1)

Imgrd(1)

Name

Imgrd – flexible license manager daemon

Synopsis

Imgrd [**-c** *license_file*] [**-l** *logfile*] [**-t** *timeout*] [**-s** *interval*]

Description

Imgrd is the main daemon program for the FLEXlm distributed license management system. When invoked, it looks for a license file containing all required information about vendors and features.

Options

-c *license_file*

Use the specified *license_file*. If no **-c** option is specified, **Imgrd** looks for the environment variable `LM_LICENSE_FILE` in order to find the license file to use. If that environment variable is not set, **Imgrd** looks for the file `/usr/local/flexlm/licenses/license.dat`.

-l *logfile*

Specifies the output log file to use. Instead of using the **-l** option you can use output redirection (`>` or `>>`) to specify the name of the output log file.

-t *timeout*

Specifies the *timeout* interval, in seconds, during which the license daemon must complete its connection to other daemons if operating in multi-server mode. The default value is 10 seconds. A larger value may be desirable if the daemons are being run on busy systems or a very heavily loaded network.

-s *interval*

Specifies the log file timestamp *interval*, in minutes. The default is 360 minutes. This means that every six hours **Imgrd** logs the time in the log file.



`Imdown(1)`, `Imstat(1)`

lmhostid(1)

Name

lmhostid – report the hostid of a system

Synopsis

lmhostid

Description

lmhostid calls the FLEXlm version of `gethostid` and displays the results.

The output of **lmhostid** looks like this:

```
lmhostid - Copyright (C) 1989, 1990 Highland Software, Inc.  
The FLEXlm host ID of this machine is "1200abcd"
```

Options

lmhostid has no command line options.

Imremove(1)

Name

Imremove – remove specific licenses and return them to license pool

Synopsis

Imremove [**-c** *license_file*] *feature user host* [*display*]

Description

Imremove allows the system administrator to remove a single user's license for a specified feature. This could be required in the case where the licensed user was running the software on a node that subsequently crashed. This situation will sometimes cause the license to remain unusable. **Imremove** will allow the license to return to the pool of available licenses.

Options

-c *license_file*

Use the specified *license_file*. If no **-c** option is specified, **Imremove** looks for the environment variable LM_LICENSE_FILE in order to find the license file to use. If that environment variable is not set, **Imremove** looks for the file /usr/local/flexlm/licenses/license.dat.



lmstat(1)

Imreread(1)

Name

Imreread – tells the license daemon to reread the license file

Synopsis

Imreread [**-c** *license_file*]

Description

Imreread allows the system administrator to tell the license daemon to reread the license file. This can be useful if the data in the license file has changed; the new data can be loaded into the license daemon without shutting down and restarting it.

Imreread uses the license file from the command line (or the default file, if none specified) only to find the license daemon to send it the command to reread the license file. The license daemon will always reread the file that it loaded from the original path. If you need to change the path to the license file read by the license daemon, then you must shut down the daemon and restart it with that new license file path.

You can not use **Imreread** if the **SERVER** node names or port numbers have been changed in the license file. In this case, you must shut down the daemon and restart it in order for those changes to take effect.

Imreread does not change any option information specified in an options file. If the new license file specifies a different options file, that information is ignored. If you need to reread the options file, you must shut down the daemon and restart it.

Options

-c *license_file*

Use the specified *license_file*. If no **-c** option is specified, **Imreread** looks for the environment variable **LM_LICENSE_FILE** in order to find the license file to use. If that environment variable is not set, **Imreread** looks for the file `/usr/local/flexlm/licenses/license.dat`.



lmdown(1)

Imstat(1)

Name

Imstat – report status on license manager daemons and feature usage

Synopsis

```
Imstat [ -a ] [ -A ] [ -c license_file ] [ -f [feature] ]
[ -l [regular_expression] ] [ -s [server] ] [ -S [daemon] ] [ -t timeout ]
```

Description

Imstat provides information about the status of the server nodes, vendor daemons, vendor features, and users of each feature. Information can be qualified optionally by specific server nodes, vendor daemons, or features.

Options

- a** Display everything.
- A** List all active licenses.
- c** *license_file* Use the specified *license_file*. If no **-c** option is specified, **Imstat** looks for the environment variable `LM_LICENSE_FILE` in order to find the license file to use. If that environment variable is not set, **Imstat** looks for the file `/usr/local/flexlm/licenses/license.dat`.
- f** [*feature*] List all users of the specified *feature*(s).
- l** [*regular_expression*] List all users of the features matching the given *regular_expression*.
- s** [*server*] Display the status of the specified *server* node(s).
- S** [*daemon*] List all users of the specified *daemon*'s features.
- t** *timeout* Specifies the amount of time, in seconds, **Imstat** waits to establish contact with the servers. The default value is 10 seconds. A larger value may be desirable if the daemons are being run on busy systems or a very heavily loaded network.



lmgrd(1)

4 THE DAEMON LOG FILE

The FLEXlm daemons all generate log files containing messages in the following format:

mm/dd hh:mm (DAEMON name) message

Where:

mm/dd hh:mm Is the month/day hour:minute that the message was logged.

DAEMON name Either “license daemon” or the string from the DAEMON line that describes your daemon.

In the case where a single copy of the daemon cannot handle all of the requested licenses, an optional “_” followed by a number indicates that this message comes from a forked daemon.

message The text of the message.

The log files can be used to:

- Inform you when it may be necessary to update your application software licensing arrangement.
- Diagnose configuration problems.
- Diagnose daemon software errors.

The messages are grouped below into the above three categories, with each message followed by a brief description of its meaning.

4.1 INFORMATIONAL MESSAGES

Connected to node

This daemon is connected to its peer on node *node*.

CONNECTED, master is name

The license daemons log this message when a quorum is up and everyone has selected a master.

DEMO mode supports only one SERVER host!

An attempt was made to configure a demo version of the software for more than one server host.

DENIED: N feature to user (mm/dd/yy hh:mm)

user was denied access to *N* licenses of *feature*. This message may indicate a need to purchase more licenses.

EXITING DUE TO SIGNAL mm

EXITING with code mm

All daemons list the reason that the daemon has exited.

EXPIRED: feature

feature has passed its expiration date.

IN: feature by user (N licenses) (used: d:hh:mm:ss) (mm/dd/yy hh:mm)

user has checked back in *N* licenses of *feature* at *mm/dd/yy hh:mm*.

IN server died: feature by user (number licenses) (used: d:hh:mm:ss) (mm/dd/yy hh:mm)

user has checked in *N* licenses by virtue of the fact that his server died.

License Manager server started

The license daemon was started.

Lost connection to host

A daemon can no longer communicate with its peer on node *host*, which can cause the clients to have to reconnect, or cause the number of daemons to go below the minimum number, in which case clients may start exiting. If the license daemons lose the connection to the master, they will kill all the vendor daemons; vendor daemons will shut themselves down.

Lost quorum

The daemon lost quorum, so will process only connection requests from other daemons.

MASTER SERVER died due to signal mm

The license daemon received fatal signal *mm*.

MULTIPLE xxx servers running. Please kill, and restart license daemon

The license daemon has detected that multiple copies of vendor daemon *xxx* are running. The user should kill all *xxx* daemon processes and re-start the license daemon.

OUT: feature by user (N licenses) (mm/dd/yy hh:mm)

user has checked out *N* licenses of *feature* at *mm/dd/yy hh:mm*

Removing clients of children

The top-level daemon logs this message when one of the child daemons dies.

RESERVE feature for HOST name***RESERVE feature for USER name***

A license of *feature* is reserved for either user *name* or host *name*.

REStarted xxx (internet port mm)

Vendor daemon *xxx* was restarted at internet port *mm*.

Retrying socket bind (address in use)

The license servers try to bind their sockets for approximately 6 minutes if they detect *address in use* errors.

Selected (EXISTING) master node

This license daemon has selected an existing master (node) as the master.

SERVER shutdown requested

A daemon was requested to shut down via a user-generated kill command.

[NEW] Server started for: feature-list

A (possibly new) server was started for the features listed.

Shutting down xxx

The license daemon is shutting down the vendor daemon *xxx*.

SIGCHLD received. Killing child servers

A vendor daemon logs this message when a shutdown was requested by the license daemon.

Started name

The license daemon logs this message whenever it starts a new vendor daemon.

Trying connection to node

The daemon is attempting a connection to *node*.

4.2 CONFIGURATION PROBLEM MESSAGES

hostname: Not a valid server host, exiting

This daemon was run on an invalid hostname.

hostname: Wrong hostid, exiting

The hostid is wrong for *hostname*.

BAD CODE for feature-name

The specified feature name has a bad encryption code.

CANNOT OPEN options file “file”

The options file specified in the license file could not be opened.

Couldn't find a master

The daemons could not agree on a master.

license daemon: lost all connections

This message is logged when all the connections to a server are lost, which often indicates a network problem.

lost lock, exiting

Error closing lock file

Unable to re-open lock file

The vendor daemon has a problem with its lock file, usually because of an attempt to run more than one copy of the daemon on a single node. Locate the other daemon that is running via a **ps** command, and kill it with **kill -9**.

NO DAEMON line for daemon

The license file does not contain a DAEMON line for *daemon*.

No “license” service found

The TCP *license* service did not exist in `/etc/services`.

No license data for “feat”, feature unsupported

There is no feature line for *feat* in the license file.

No features to serve!

A vendor daemon found no features to serve. This could be caused by bad data in the license file.

UNSUPPORTED FEATURE request: feature by user

The *user* has requested a feature that this vendor daemon does not support. This can happen for a number of reasons: the license file is bad, the feature has expired, or the daemon is accessing the wrong license file.

Unknown host: hostname

The hostname specified on a `SERVER` line in the license file does not exist in the network database (probably `/etc/hosts`).

lm_server: lost all connections

This message is logged when all the connections to a server are lost. This probably indicates a network problem.

NO DAEMON lines, exiting

The license daemon logs this message if there are no `DAEMON` lines in the license file. Since there are no vendor daemons to start, there is nothing to do.

NO DAEMON line for name

A vendor daemon logs this error if it cannot find its own `DAEMON` name in the license file.

4.3 DAEMON SOFTWARE ERROR MESSAGES

accept: message

An error was detected in the accept system call.

ATTEMPT TO START VENDOR DAEMON xxx with NO MASTER

A vendor daemon was started with no master selected. This is an internal consistency error in the daemons.

BAD PID message from mm: pid: xxx (msg)

A top-level vendor daemon received an invalid PID message from one of its children (daemon number xxx).

BAD SCONNECT message: (message)

An invalid “server connect” message was received.

Cannot create pipes for server communication

The pipe call failed.

Can't allocate server table space

A malloc error. Check swap space.

Connection to node TIMED OUT

The daemon could not connect to *node*.

Error sending PID to master server

The vendor server could not send its PID to the top-level server in the hierarchy.

Illegal connection request to DAEMON

A connection request was made to DAEMON, but this vendor daemon is not DAEMON.

Illegal server connection request

A connection request came in from another server without a DAEMON name.

KILL of child failed, errno = mm

A daemon could not kill its child.

No internet port number specified

A vendor daemon was started without an internet port.

Not enough descriptors to re-create pipes

The “top-level” daemon detected one of its sub-daemon’s death. In trying to restart the chain of sub-daemons, it was unable to get the file descriptors to set up the pipes to communicate. This is a fatal error, and the daemons must be re-started.

read: error message

An error in a read system call was detected.

recycle_control BUT WE DIDN'T HAVE CONTROL

The hierarchy of vendor daemons has become confused over who holds the control token. This is an internal error.

return_reserved: can't find feature listhead

When a daemon is returning a reservation to the “free reservation” list, it could not find the listhead of features.

select: message

An error in a select system call was detected.

Server exiting

The server is exiting. This is normally due to an error.

SHELLO for wrong DAEMON

This vendor daemon was sent a “server hello” message that was destined for a different DAEMON.

Unsolicited msg from parent!

Normally, the top-level vendor daemon sends no unsolicited messages. If one arrives, this message is logged. This is a bug.

***WARNING: CORRUPTED options list (o->next == 0)
Options list TERMINATED at bad entry***

An internal inconsistency was detected in the daemon’s option list.

5 FLEXLM LICENSE ERRORS

FLEXlm license error, encryption code in license file is inconsistent

Check the contents of the license file using the license data sheet for the product. Correct the license file and run the **lmreread** command. However, do not change the last (fourth) field of a SERVER line in the license file. This cannot have any effect on the error message but changing it will cause other problems.

license file does not support this version

If this is a first time install then follow the procedure for the error message:

```
FLEXlm license error, encryption code in license file is
inconsistent
```

because there may be a typo in the fourth field of a FEATURE line of your license file. In all other cases you need a new license because the current license is for an older version of the product.

Replace the FEATURE line for the old version of the product with a FEATURE line for the new version (it can be found on the new license data sheet). Run the **lmreread** command afterwards. You can have only one version of a feature (previous versions of the product will continue to work).

FLEXlm license error, cannot find license file

Make sure the license file exists. If the pathname printed on the line after the error message is incorrect, correct this by setting the `LM_LICENSE_FILE` environment variable to the full pathname of the license file.

FLEXlm license error, cannot read license file

Every user needs to have read access on the license file and at least execute access on every directory component in the pathname of the license file. Write access is never needed. Read access on directories is recommended.

FLEXlm license error, no such feature exists

Check the license file. There should be a line starting with:

```
FEATURE SWiiiiii-jj
```

where "iiiiii" is a six digit software code and "jj" is a two digit host code for identifying a compatible host architecture. During product installations the product code is shown, e.g. SW008002, SW019002. The number in the software code is the same as the number in the product code except that the first number may contain an extra leading zero (it must be six digits long).

The line after the license error message describes the expected feature format and includes the host code.

Correct the license file using the license data sheet for the product and run the **lmreread** command. There is one catch: do not add extra SERVER lines or change existing SERVER lines in the license file.

FLEXlm license error, license server does not support this feature

If the LM_LICENSE_FILE variable has been set to the format *number@host* then see first the solution for the message:

```
FLEXlm license error, no such feature exists
```

Run the **lmreread** program to inform the license server about a changed license data file. If **lmreread** succeeds informing the license server but the error message persists, there are basically three possibilities:

1. The license password is incorrect. If this is the case then there must be an error message in the log file of **lmgrd**. Correct the password using the license data sheet for the product. Finally rerun **lmreread**. The log file of **lmgrd** is usually specified to **lmgrd** at startup with the **-l** option or with **>**.
2. Your network has more than one FLEXlm license server daemon and the default license file location for **lmreread** differs from the default assumed by the program. Also, there must be more than one license file. Try one of the following solutions on the same host which produced the error message:

- type:

```
lmreread -c /usr/local/flexlm/licenses/license.dat
```

- set LM_LICENSE_FILE to the license file location and retry the **lmreread** command.
- use the **lmreread** program supplied with the product SW000098, Flexible License Manager. SW000098 is bundled with all TASKING products.

3. There is a protocol version mismatch between **lmgrd** and the daemon with the name "Tasking" (the vendor daemon according to FLEXlm terminology) or there is some other internal error. These errors are always written to the log file of **lmgrd**. The solution is to upgrade the **lmgrd** daemon to the one supplied in SW000098, the bundled Flexible License Manager product.

On the other hand, if **lmreread** complains about not being able to connect to the license server then follow the procedure described in the next section for the error message "Cannot read license file data from server". The only difference with the current situation is that not the product but a license management utility shows a connect problem.

FLEXlm license error, Cannot read license file data from server

This indicates that the program could not connect to the license server daemon. This can have a number of causes. If the program did not immediately print the error message but waited for about 30 seconds (this can vary) then probably the license server host is down or unreachable. If the program responded immediately with the error message then check the following if the `LM_LICENSE_FILE` variable has been set to the format *number@host*:

- is the number correct? It should match the fourth field of a `SERVER` line in the license file on the license server host. Also, the host name on that `SERVER` line should be the same as the host name set in the `LM_LICENSE_FILE` variable. Correct `LM_LICENSE_FILE` if necessary.

In any case one should verify if the license server daemon is running. Type the following command on the host where the license server daemon (**lmgrd**) is supposed to run.

On SunOS 4.x:

```
ps wwax | grep lmgrd | grep -v grep
```

On HP-UX or SunOS 5.x (Solaris 2.x):

```
ps -ef | grep lmgrd | grep -v grep
```

If the command does not produce any output then the license server daemon is not running. See below for an example how to start **lmgrd**.

Make sure that both license server daemon (**lmgrd**) and the program are using the same license data. All TASKING products use the license file `/usr/local/flexlm/licenses/license.dat` unless overruled by the environment variable `LM_LICENSE_FILE`. However, not all existing **lmgrd** daemons may use the same default. In case of doubt, specify the license file pathname with the `-c` option when starting the license server daemon. For example:

```
lmgrd -c /usr/local/flexlm/licenses/license.dat \  
-l /usr/local/flexlm/licenses/license.log &
```

and set the `LM_LICENSE_FILE` environment variable to the `license.dat` pathname mentioned with the `-c` option of **lmgrd** before running any license based program (including **lmreread**, **lmstat**, **lmdown**). If **lmgrd** and the program run on different hosts, transparent access to the license file is assumed in the situation described above (e.g. NFS). If this is not the case, make a local copy of the license file (not recommended) or set `LM_LICENSE_FILE` to the form *number@host*, as described earlier.

If none of the above seems to apply (i.e. **lmgrd** was already running and `LM_LICENSE_FILE` has been set correctly) then it is very likely that there is a TCP port mismatch. The fourth field of a `SERVER` line in the license file specifies a TCP port number. That number can be changed without affecting any license. However, it must never be changed while the license server daemon is running. If it has been changed, change it back to the original value. If you do not know the original number anymore, restart the license server daemon after typing the following command on the license server host:

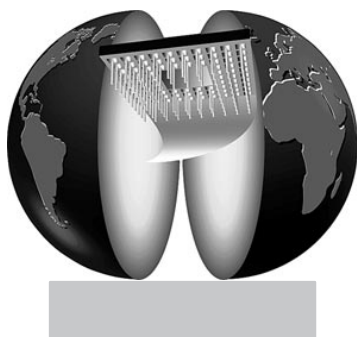
```
kill PID
```

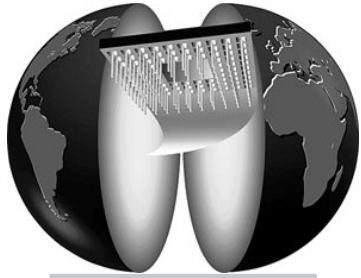
where `PID` is the process id of **lmgrd**.

APPENDIX

B

ASSEMBLER DIRECTIVES OVERVIEW





B

APPENDIX

CSEG

- Category: Location Counter Control
- Syntax: `CSEG [REL | AT base address]`
- Function: Selects a code segment, either relocatable or beginning at absolute address specified.

DCB

- Category: Code Definition
- Syntax: `[label:] DCB {expression | string} [, ...]`
- Function: Specifies a list of zero or more byte values to be inserted sequentially starting at the position indicated by the location counter.

DCL

- Category: Code Definition
- Syntax: `[label:] DCL long-unsigned-number [, ...]`
- Function: Specifies a list of one or more long unsigned numbers to be inserted sequentially starting at the position indicated by the location counter.

DCP

- Category: Code Definition
- Syntax: `[label:] DCP expression [, ...]`
- Function: Specifies a list of one or more 24-bit pointer value to be inserted sequentially starting at the position indicated by the location counter.

DCR

- Category: Code Definition
- Syntax: `[label:] DCR float_number [, ...]`
- Function: Specifies a list of one or more floating point numbers to be inserted sequentially starting at the position indicated by the location counter.

DCW

- Category: Code Definition
- Syntax: `[label:] DCW expression [, ...]`
- Function: Specifies a list of one or more word values to be inserted sequentially starting at the position indicated by the location counter.

DSB

- Category: Storage Reservation
- Syntax: `[label:] DSB expression`
- Function: Reserves storage for byte program variables starting at position of location counter.

DSEG

- Category: Location Counter Control
- Syntax: `DSEG [REL | AT base address]`
- Function: Selects a data segment either relocatable or beginning at absolute address specified.

DSL

Category: Storage Reservation

Syntax: `[label :] DSL expression`

Function: Reserves storage for long word program variables starting at (aligned) position of location counter.

DSP

Category: Storage Reservation

Syntax: `[label :] DSP expression`

Function: Reserves storage for 24-bit pointer variables starting at (aligned) position of location counter.

DSQ

Category: Storage Reservation

Syntax: `[label :] DSQ expression`

Function: Reserves storage for quad word program variables starting at (aligned) position of location counter.

DSR

Category: Storage Reservation

Syntax: `[label :] DSR expression`

Function: Reserves storage of floating point number program variables starting at (aligned) position of location counter.

DSW

Category: Storage Reservation

Syntax: `[label:] DSW expression`

Function: Reserves storage for word program variables starting at (aligned) position of location counter.

ELSE

Category: Conditional Assembly

Syntax: `[ELSE]
[statements]`

Function: Provides alternate in conditional assembly block.

END

Category: Module Level

Syntax: `END`

Function: Marks the end of program.

ENDIF

Category: Conditional Assembly

Syntax: `ENDIF`

Function: Ends conditional assembly block.

EQU

Category: Symbol Definition

Syntax: `symbol name EQU expression [:data type]`

Function: Defines symbols that may not be redefined.

EXTRN

Category: Module Level

Syntax: EXTRN {*symbol name* [:*data type*] } [, ...]

Function: Declares one or more symbols as external.

IF

Category: Conditional Assembly

Syntax: IF *expression*
 [*statements*]

Function: Starts conditional assembly block.

IFEQ, IFNE, IFLT, IFLE, IFGT, IFGE

Category: Conditional Assembly

Syntax: IFxx *expression*
 [*statements*]

Function: Starts conditional assembly block. Comparisons with zero (equal, not equal, less than, less than or equal, greater than, greater than or equal).

IFDEF, IFNDEF

Category: Conditional Assembly

Syntax: IF[N]DEF *symbol*
 [*statements*]

Function: Starts conditional assembly block. Test if *symbol* is defined or not.

IFB, IFNB

Category: Conditional Assembly

Syntax: `IF[N]B string`
`[statements]`

Function: Starts conditional assembly block. Test if *string* is empty or not.

IFIDN, IFNIDN

Category: Conditional Assembly

Syntax: `IF[N]IDN <str1>, <str2>`
`[statements]`

Function: Starts conditional assembly block. Test if *str1* is equal to *str2* or not.

KSEG

Category: Location Counter Control

Syntax: `KSEG [REL | AT base address]`

Function: Selects a constant segment either relocatable or beginning at absolute address specified.

MODULE

Category: Module Level

Syntax: `module name MODULE [attr[,attr]]`
 where *attr* can be either MAIN or STACKSIZE(*n*)

Function: Defines a module and its attributes.

ODSEG

Category: Location Counter Control

Syntax: `ODSEG [REL | AT base address]`

Function: Selects an overlayable data segment either relocatable or beginning at absolute address specified.

ORG

Category: Location Counter Control

Syntax: `ORG expression`

Function: Sets value of location counter.

OSEG

Category: Location Counter Control

Syntax: `OSEG [REL | AT base address]`

Function: Selects an overlayable register segment either relocatable or beginning at absolute address specified.

PUBLIC

Category: Module Level

Syntax: `PUBLIC symbol name [, ...]`

Function: Declares one or more symbols public.

RSEG

- Category: Location Counter Control
- Syntax: `RSEG [REL | AT base address]`
- Function: Selects a non-overlayable register segment either relocatable or beginning at absolute address specified.

SET

- Category: Symbol Definition
- Syntax: `symbol name SET expression [:data type]`
- Function: Defines symbols that may be redefined.

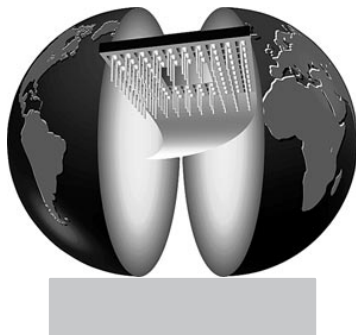
SSEG

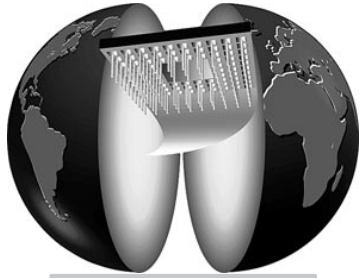
- Category: Location Counter Control
- Syntax: `SSEG [REL | AT base address]`
- Function: Selects a stack segment either relocatable or beginning at absolute address specified.

APPENDIX

C

ASSEMBLER CONTROLS TABLE





C

APPENDIX

Table C-1 provides the following four types of information about each assembler control:

- The Control Name column gives the full name of each control. You can suppress many of the control functions by prefacing the control or its abbreviation with NO.
- The Abbreviation column shows the two-letter abbreviation for each control. If the control can be prefaced with NO, the two-letter abbreviation can also be prefaced with NO.
- The Default column gives the default setting the assembler assigns for each control.
- The Type column indicates the type of each control (primary or general).

Control Name	Abbreviation	Default	Type
case	cs	cs	Primary
cmain	cm	nocm	Primary
cond	co	co	General
copyattr	ca	noca	Primary
debug	db	nodb	Primary
directaddr	da	noda	Primary
eject	ej	n/a	General
error(<i>'string'</i>)	er	n/a	General
errorprint	ep	noep	Primary
extra_mnem	em	noem	Primary
gen	ge	noge	General
include(<i>pathname</i>)	ic	n/a	General
limit_bitno	lb	nolb	Primary
linedebug	ld	nold	Primary
list	li	li	General
model(<i>processor</i>)	md	md(kb)	Primary
optimize	ot	noot	Primary
nearcode/farcode	nc/fc	nc	Primary
nearconst/farconst	nk/fk	nk	Primary
neardata/fardata	nd/fd	nd	Primary
object	oj	oj(<i>src.obj</i>)	Primary



Control Name	Abbreviation	Default	Type
omf(<i>number</i>)	omf	omf(2)	Primary
optionalcolon	oc	nooc	Primary
pagelength(<i>number</i>)	pl	pl(60)	Primary
pagewidth(<i>number</i>)	pw	pw(120)	Primary
print	pr	pr(<i>src.lst</i>)	Primary
relaxedif	ri	nori	Primary
save/restore	sa/rs	n/a	General
searchinclude(<i>pathname</i>)	si	nosi	General
set/reset	se/re	n/a	General
signedoper	so	no	Primary
source	sc	nosc	General
subtitle('string')	st	nost	General
symbols	sb	sb	Primary
title('string')	tt	tt(<i>modname</i>)	General
xref	xr	noxr	Primary

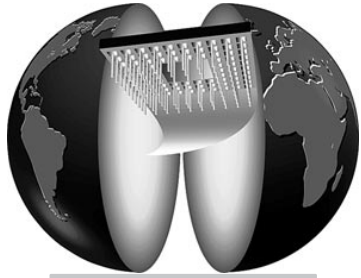
Table C-1: Assembler controls

APPENDIX

ASM196 RESERVED WORDS

D





D

APPENDIX

* This reserved word is specific to 24-bit models.

** This reserved word is specific to 80C196.

Assembler / Macro Directives

.STRLEN.	EQU	MACRO
AT	EXITM	MAIN
BYTE	EXTRN	MODULE
CSEG	FAR*	NEAR*
DCB	IF	NULL
DCL	IFB	ODSEG
DCP	IFDEF	ORG
DCR	IFEQ	OSEG
DCW	IFGE	POINTER
DSB	IFGT	PUBLIC
DSEG	IFIDN	QUAD
DSL	IFLE	REAL
DSP	IFLT	REL
DSQ	IFNB	REPT
DSR	IFNDEF	RSEG
DSW	IFNE	SET
DWORD	IFNIDN	SSEG
ELSE	IRP	STACK
END	IRPC	STACKSIZE
ENDIF	KSEG	WORD
ENDM	LOCAL	
ENTRY	LONG	

Generic Instructions

BBC	BNVT	EBLE*
BBS	BST	EBLT*
BC	BV	EBNC*
BE	BVT	EBNE*
BGE	CALL	EBNH*
BGT	DBNZ	EBNST*
BH	DBNZW	EBNV*
BLE	EBBC*	EBNVT*
BLT	EBBS*	EBST*
BNC	EBC*	EBV*
BNE	EBE*	EBVT*
BNH	EBGE*	EDBNZ*
BNST	EBGT*	EDBNZW*
BNV	EBH*	

Other Reserved Words

ADD	CMPB	EBR*
ADDB	CMPL**	ECALL*
ADDC	DEC	EI
ADDCB	DECB	EJMP*
AND	DI	ELD*
ANDB	DIV	ELDB*
BMOV**	DIVB	EPTS**
BMOVI**	DIVU	EQ
BR	DIVUB	EST*
CLR	DJNZ	ESTB*
CLRB	DJNZW**	EXT
CLRC	DPTS**	EXTB
CLRVT	EBMOV*	GE
CMP	EBMOVI*	GT

RESERVED WORDS

HIGH	LOW	SHL
IDLPD**	LSW	SHLB
INC	LT	SHLL
INCB	MOD	SHR
JBC	MSW	SHRA
JBS	MUL	SHRAB
JC	MULB	SHRAL
JE	MULU	SHRB
JGE	MULUB	SHRL
JGT	NE	SJMP
JH	NEG	SKIP
JLE	NEGB	ST
JLT	NOP	STB
JNC	NORML	SUB
JNE	NOT	SUBB
JNH	NOTB	SUBC
JNST	NUL	SUBCB
JNV	OR	TIJMP**
JNVT	ORB	TRAP
JST	POP	UGE
JV	POPA**	UGT
JVT	POPF	ULE
LCALL	PUSH	ULT
LD	PUSHA**	XCH**
LDB	PUSHF	XCHB**
LDBSE	RET	XOR
LDBZE	RST	XORB
LE	SCALL	
LJMP	SETC	

Additional Mnemonics

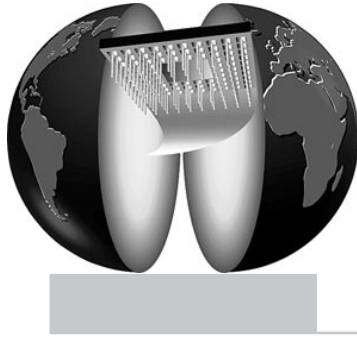
BGES	EBGES*	JGES
BGEU	EBGEU*	JGEU
BGTS	EBGTS*	JGTS
BGTU	EBGTU*	JGTU
BLES	EBLES*	JLES
BLEU	EBLEU*	JLEU
BLTS	EBLTS*	JLTS
BLTU	EBLTU*	JLTU
BNZ	EBNZ*	JNZ
BZ	EBZ*	JZ

INDEX

INDEX



INDEX



Symbols

.strlen. directive, 7-19
 ; (semicolon), 3-7
 : (colon), 3-4
 " (double quotation marks), 4-47, 4-49
 ' (Apostrophe), 5-15
 ' (single quotation marks), 4-47, 4-49
 & (ampersand), 7-24
 \$ (dollar sign), 3-7, 5-15, 5-17
 % (percent sign), 7-27
 - (minus sign), 3-4
 * (asterisk), 3-18
 ^ (caret), 3-18
 = (equal sign), 3-4
 \ (backslash), 3-4
 _ (underscore), 3-18
 _16_BITS_, 5-8
 _24_BITS_, 5-9
 _BOTTOM_OF_STACK_, 5-33
 _FAR_CODE_, 5-8
 _FAR_CONST_, 5-8
 _FAR_DATA_, 5-8
 _MODEL_xx_, 5-8
 _OMF96_VERSION_, 5-9
 _SFR_INC_, 5-8
 _TOP_OF_STACK_, 5-33

A

Absolute expression, 5-17, 5-22, 6-24
 Absolute object file, 2-3, 3-11
 Absolute segment, 6-24
 Absolute segments, 5-32
 Absolute values, 5-5
 Addressing modes
 direct, 4-11
 extended-indexed, 5-14
 long/short-indexed, 5-14
 mixed, 5-14
 register direct, 5-14

Alignment

Expressions, 6-12
Floating point number, 6-11
Label, 6-14
Program variables, 6-14
Unsigned number, 6-9

Alignment attribute , 5-32

ASM196 assembler, 2-3

Assembler

Controls, 3-7
controls, 4-1
in software development process, 2-3
Invocation, 3-1
Listing file, 2-3
Options, 3-4

Assembler controls, overview, C-1

Assembler directives, 6-1

overview, B-1

Assembly language components, 5-3

Character set, 5-3

Delimiters, 5-7

Numbers, 5-3

Reserved words, 5-8

Strings, 5-15

Symbols, 5-9

Assembly language elements, 5-1

Assigning a listing filename, 4-36

Assigning a page header, in listing file,
 4-47, 4-49

Assigning an object filename, 4-30

Attributes, 5-18

Assigning to module, 5-28

Defining symbols, 6-17

Location, 5-23

Relocatability, 5-21

Type, 5-18

Value, 5-21

Automating the assembly process, 3-22

B

Batch file commands

goto, 3-24

if, 3-24

Batch files, 3-22

Argument passing, 3-23, 3-24

call command, 3-23

Invocation, 3-23

Parameters, 3-24

Binary operations, Type attribute, 5-20

Bit numbers, 5-25

Byte constants, Defining, 6-7

C

C196INC environment variable, 4-41

Calling a macro, 7-20

Carriage return, 5-27

case control, 4-4

Case sensitivity, 4-4

Reserved words, 5-8

Symbols, 5-9

Case significance, 5-16

cmain control, 4-6

Code segment, 5-32, 6-5

Command files, 3-22

Comment lines, 5-27

Statement structure, 5-27

Complex expression, 3-17

cond control, 4-7

Conditional-assembly directives, 6-3,
6-20

Constant segment, 5-32

Constant-definition directives, 6-3,
6-7, 6-10

Constants

Defining, 6-7, 6-10

Expressions, 5-24

Continuation lines, 3-4, 5-27

Control line, 3-7, 5-27

Control Z, 5-28

Controls

Abbreviations, 3-8

case, 4-4

Case sensitivity, 3-8

cmain, 4-6

cond, 4-7

copyattr, 4-8

debug, 4-10

Defaults, 3-8

directaddr, 4-11

eject, 4-12

error, 4-13

errorprint, 4-14

extra_mnem, 4-15

farcode, 4-27

farconst, 4-28

fardata, 4-29

gen, 4-17

General, 3-8

include, 4-19

limit_bitno, 4-20

linedebug, 4-21

list, 4-22

model, 4-23

nearcode, 4-27

nearconst, 4-28

neardata, 4-29

Negating, 3-8

object, 4-30

omf, 4-31

optimize, 4-32

optionalcolon, 4-33

pagelength, 4-34

pagewidth, 4-35

Parameters, 3-8

Primary, 3-8

print, 4-36

Processing, 3-10

relaxedif, 4-38

save/restore, 4-39, 4-43

searchinclude, 4-40

- signedoper*, 4-44
- source*, 4-45
- Specifying*, 3-7
- subtitle*, 4-47
- symbols*, 4-46
- Table*, 3-9
- title*, 4-49
- xref*, 4-50
- Conventions, 2-6
- copyattr control, 4-8
 - Segment type*, 5-18
- cseg directive, 6-4
- Customer comments, 2-7
- Customer service hotline, 2-7

D

Data representation

- Alignment*, 6-9, 6-11
- Floating point number*, 6-11
- Numbers*, 5-3
 - Pages*, 5-5
- Operands*, 5-17
- Strings*, 5-15
- Unsigned number*, 6-9
- Word values*, 6-12
- Data segment, 5-30, 6-5
- Data structures
 - Macros*, 7-5
 - Routines*, 7-5
- Data types, 6-17, 6-19
 - byte*, 3-21, 6-17, 6-19
 - dword*, 6-17, 6-19
 - entry*, 3-21, 6-17, 6-19
 - long*, 3-21, 6-17, 6-19
 - null*, 3-21, 6-17, 6-19
 - pointer*, 3-21, 6-17, 6-19
 - real*, 3-21, 6-17, 6-19
 - word*, 3-21, 6-17, 6-19
- dcb directive, 6-7
- dcl directive, 6-9
- dcp directive, 6-10

- dcr directive, 6-11
- dcw directive, 6-12
- debug control, 4-10
- Debugging with emulators, 4-10
- Declaring external symbols, 6-19
- Declaring public symbols, 6-25
- Declaring segments, 6-4
- Defining byte constants, 6-7
- Defining floating point number
 - constants, 6-11
- Defining long constants, 6-9
- Defining macros, 7-6
- Defining pointer constants, 6-10
- Defining program symbols, 6-13
- Defining word constants, 6-12
- Delimiters, 5-7, 7-20
 - Table*, 5-7
- Diagnostic messages, 8-1
- directaddr control, 4-11
- Directives
 - Conditional assembly*, 6-3, 6-20
 - Constant definition*, 6-3, 6-7, 6-10
 - Location counter control*, 6-3
 - Module level*, 6-3, 6-23
 - Segment selection*, 6-3
 - Storage definition*, 6-3
 - Storage reservation*, 6-13
 - Symbol definition*, 6-3

- Displaying lines, in listing file, 4-22

- Dollar sign (\$), 5-17

- dsb directive, 6-13
- dseg directive, 6-4
- dsl directive, 6-13
- dsp directive, 6-13
- dsq directive, 6-13
- dsr directive, 6-13
- dsw directive, 6-13

E

Efficiency

- Macros*, 7-3, 7-5

Routines, 7-5
 eject control, 4-12
 end directive, 5-27, 6-16
 End-of-file directive, 6-16
 End-of-file marker, 5-28
 End-of-macro, 7-8
 endm directive, 7-8
 environment variable, 1-4, 1-8
 LM_LICENSE_FILE, 1-7, A-6
 PATH, 1-4, 1-8
 TMPDIR, 1-4, 1-8
 equ directive, 6-17
 Segment type, 5-18
 error control, 4-13
 Error Lines, 3-4, 3-18
 Error messages, 8-4
 Argument, 8-6
 Console output, 8-3
 Fatal error messages, 8-4
 I/O, 8-7
 Memory, 8-7
 Redirecting, 4-14
 Source file error messages, 8-9
 User-defined, 4-13
 Warning messages, 8-8
 Error recovery, 8-4
 errorprint control, 4-14
 Errorprint file, 3-22
 errors, FLEXlm license, A-25
 exitm directive, 7-9
 Expressions, 5-16
 Absolute, 5-22
 Complex, 5-21
 External, 5-21
 External bit numbers, 5-25
 Relocatable, 5-21, 5-24
 Type of, 5-21
 Extensions, 2-5
 External bit numbers, 5-25
 External reference, 3-17
 External symbols, Declaring, 6-19
 extra_mnem control, 4-15

extrn directive, 6-19

F

FAR segments, 5-33
 farcode control, 4-27
 farconst control, 4-28
 fardata control, 4-29
 Fatal error messages, 8-4
 Filename extensions, 2-5
 .bat, 3-23
 .tmp, 2-5
 Files
 Identified, 3-15
 Include, 4-19
 Listing, 4-36
 Object, 4-30
 Source, 4-19
 Fixup indicator, in listing file, 3-17
 Flexible License Manager, A-1
 FLEXlm, A-1
 daemon log file, A-17
 daemon options file, A-6
 license administration tools, A-8
 license errors, A-25
 user commands, A-11
 Floating point number constants,
 Defining, 6-11
 Floating point numbers, 5-4
 Format, Source program, 5-27

G

gen control, 4-17
 Generic instructions
 call, 5-12
 in asm196, 5-11
 in listing file, 3-17

H

HIGH Code segments, 5-33

I

if/else/endif directive, 6-20
 ifeq/ifne/ift/ifle/iftgt/iftge, 6-20
 include control, 4-19
 Include files, 4-40
 Examples, 4-41
 Search path, 4-40, 4-41
 Include indicator, in listing file, 3-17
 Including a cross-reference listing, in
 listing file, 4-50
 Including expansion lines, in listing
 file, 4-17
 Including external files, in assembly
 process, 4-19
 Including symbol table listing, in
 listing file, 4-46
 Inline code, 7-5
 Installation
 UNIX, 1-5
 Windows 95, 1-3
 Windows 98, 1-3
 Windows NT, 1-3
 Installation procedure, 1-1
 Instructions
 Additional mnemonics, 5-13
 Generic, 5-11
 Invocation syntax, 3-3
 irp directive, 7-10
 irpc directive, 7-12

L

Label
 Definition, 5-26, 7-14
 Definition of, 6-14

Macros, 7-14
 Statement structure, 5-26
 Labels, 5-27
 Macros, 7-6
 Scope, 7-6
 limit_bitno control, 4-20
 Line feed, 5-27
 Line numbers
 in listing file, 3-17
 in object file, 4-21
 linedebug control, 4-21
 Linking multiple files, 3-24
 list control, 4-22
 Listing file, 3-3
 Body, 3-16
 Fixup indicator, 3-17
 Generic instruction indicator, 3-17
 Include indicator, 3-17
 Line numbers, 3-17
 LOC field, 3-16
 Macro expansion indicator, 3-17
 Object code field, 3-17
 Set/Equ field, 3-17
 Source line, 3-18
 Header, 3-15
 Including expansion lines, 4-17
 Printing all source lines, 4-7
 printing on a new page, 4-12
 LM_LICENSE_FILE, 1-7, A-6
 lmdown, A-11
 lmgrd, A-12
 lmhostid, A-13
 lmremove, A-14
 lmread, A-15
 lmstat, A-16
 Loading memory, 2-3, 3-11
 LOC field, in listing file, 3-16
 local directive, 7-14
 Location attribute, 5-23
 Location counter, 3-16, 5-15, 6-14
 Location-counter control directives,
 6-3
 Log file, 3-24

Logging the assembly process, 3-24
 Long constants, 5-4
 Defining, 6-9
 LSW operator, 5-23

M

Machine language instructions, 2-3
 Macro calls, 7-20
 Nested, 7-27
 macro directive, 7-16
 Macro directives, 7-6, 7-7
 .strlen., 7-19
 endm, 7-6, 7-8
 exitm, 7-6, 7-9
 irp, 7-6, 7-10
 irpc, 7-6, 7-12
 local, 7-6, 7-14
 macro, 7-6, 7-16
 macro call, 7-6
 rept, 7-6, 7-18
 Macro expansion, 7-27
 in listing file, 3-17
 Macro processing, 7-1
 Advantages, 7-3
 Macro processing language, 2-6
 Macros
 _16_BITS_, 5-8
 _24_BITS_, 5-9
 _FAR_CODE_, 5-8
 _FAR_CONST_, 5-8
 _FAR_DATA_, 5-8
 _MODEL_xx_, 5-8
 _OMF96_VERSION_, 5-9
 _SFR_INC_, 5-8
 Arguments, 5-22
 Calling, 7-26
 Definition, 5-22, 7-6
 Empty arguments, 7-23
 Expansion, 7-27
 Label, 7-14
 narg symbol, 7-23

Nesting, 7-26
Null, 7-29
Number of arguments, 7-23
Operators, 5-22
Predefined, 5-8
Special operators, 7-24
 Make Utility MK196, 3-22
 Messages, 8-1
 Diagnostics, 8-1
 Mnemonics, Additional, 4-15, 5-13
 model control, 4-23
 module directive, 5-27, 6-23
 Module-level directives, 6-3, 6-23
 MSW operator, 5-23

N

Name
 Attributes, 5-26
 Statement structure, 5-26
 Names
 Assembler-generated, 5-10
 Character set, 5-9
 Macros, 5-10
 Reserved words, 5-8
 Symbols, 5-10
 Names case sensitivity, 5-8, 5-9
 Naming the listing file, 4-36
 narg symbol, 7-23
 NEAR segments, 5-33
 nearcode control, 4-27
 nearcode/farcode, 4-27
 nearconst control, 4-28
 nearconst/farconst, 4-28
 neardata control, 4-29
 neardata/fardata, 4-29
 Negating controls, 3-8
 Nested macro calls, 7-27
 Nesting macros, 7-26
 nocase control, 4-4
 nocmain control, 4-6
 nocond control, 4-7

nocopyattr control, 4-8
 nodebug control, 4-10
 nodirectaddr control, 4-11
 noerrorprint control, 4-14
 noextra_mnem control, 4-15
 nogen control, 4-17
 nolimit_bitno control, 4-20
 noline debug control, 4-21
 nolist control, 4-22
 Non-overlayable segment, 5-30
 noobject control, 4-30
 nooptimize control, 4-32
 nooptionalcolon control, 4-33
 noprint control, 4-36
 norelaxedif control, 4-38
 nosearchinclude control, 4-40
 nosignedoper control, 4-44
 nosource control, 4-45
 nosubtitle control, 4-47
 nosymbols control, 4-46
 noxref control, 4-50
 Null macros, 7-29

O

Object code field, in listing file, 3-17
 object control, 3-3, 4-30
 Object file, 2-3, 3-3, 3-11
 Absolute, 2-3, 3-11
 Including line numbers, 4-45
 Including the symbol table information, 4-10
 Relocatable, 2-3, 3-11
 odseg directive, 6-4
 omf, _OMF96_VERSION_ macro, 5-9
 omf control, 4-31
 Opcode listing, 3-17
 Operand
 Relocatability, 5-21
 Type attribute, 5-18
 Value, 5-21

Operands, 5-16, 5-17
 Absolute, 5-23
 Definition, 5-17
 Expression structure, 5-17
 Operation, Statement structure, 5-26
 Operators
 Delimiters, 5-23
 Expression structure, 5-17
 Macro, 5-22
 Macros, 7-24
 MSW/LSW, 5-23
 Order of precedence, 5-22
 Prefix unary, 5-17
 Relational, 5-22, 5-23
 Unary, 5-22
 Operators Infix, 5-17
 optimize control, 4-32
 optionalcolon control, 4-33
 Options
 Specifying, 3-4
 Table, 3-5
 Turning off/on, 3-5
 org directive, 6-24
 oseg directive, 6-4
 Output files, 3-3
 Absolute object module, 2-4
 Hex file, 2-4
 Listing file, 2-3, 3-3
 Map file, 2-4
 Object file, 2-3, 3-3, 3-11
 Quasi-absolute object module, 2-4
 Overlay segment, 5-30
 Overview, 2-1

P

pagelength control, 4-34
 pagewidth control, 4-35
 Parameters
 Macros, 7-4, 7-6, 7-16, 7-24
 Re-evaluation, 7-27

Scope, 7-6

Special characters, 7-24

PATH, 1-4, 1-8

Path prefixes, 4-40

Percent sign (%), 7-27

Pointer constants, Defining, 6-10

Pointer data type, 3-21, 6-17, 6-19

Predefined macros, 5-8

print control, 3-3, 4-36

Printing all source lines, in listing file,
4-7

Printing on a new page, 4-12

Processing controls, 3-10

Processor models, 4-23

8096-90, 4-23

8096-BH, 4-24

80C196CA, 4-24

80C196CB, 4-24

80C196EA, 4-24

80C196EC, 4-24

80C196JQ, 4-24

80C196JR, 4-24

80C196JS, 4-24

80C196JT, 4-24

80C196JV, 4-24

80C196KB, 4-24

80C196KC, 4-24

80C196KD, 4-24

80C196KL, 4-24

80C196KQ, 4-24

80C196KR, 4-25

80C196KS, 4-25

80C196KT, 4-25

80C196LB, 4-25

80C196MC, 4-25

80C196MD, 4-25

80C196MH, 4-25

80C196NP, 4-25

80C196NT, 4-25

80C196NU, 4-25

Program format, 5-27

public directive, 6-25

Public symbols, Declaring, 6-25

R

Redirecting error messages, 4-14

Register segment, 5-30

Register segment , 6-5

relaxedif control, 4-38

Relocatability attribute, 5-21

Relocatable expression, 5-17, 6-24

Relocatable object file, 2-3, 3-11

Relocatable reference, 3-17

Relocatable segment, 6-24

Relocatable segments, 5-32

Repeating macros, 7-18

Repeating macros indefinitely, 7-10

Repeating macros per character, 7-12

rept directive, 7-18

Reserved words, 5-8, D-1

restore/save control, 4-39, 4-43

Restoring save control condition, 4-39

RL196 linker, 2-3, 3-11

rseg directive, 6-4

S

save/restore control, 4-39, 4-43

Saving current control condition, 4-39

Scope

Labels, 7-6

Macros, 7-6

Parameters, 7-6

Search path, 4-40

searchinclude control, 4-40

Segment, Type attribute, 4-8, 5-18

Segment selection directives, 6-3

Segment type

equ/set, 5-18

extrn, 5-18

Segment-selection directives, 6-4

cseg, 6-4

dseg, 6-4

odseg, 6-4

- oseg*, 6-4
- rseg*, 6-4
- sseg*, 6-4
- Segments, 5-30
 - Code*, 5-30
 - Data*, 5-30
 - Declaring*, 6-4
 - Non-overlayable register*, 5-30
 - Register*, 5-30
 - Stack*, 5-30
- set* directive, 6-17
 - Segment type*, 5-18
- Set/Equ field, in listing file, 3-17
- Setting the environment
 - UNIX*, 1-8
 - Windows*, 1-4
- Sign-off message, 8-3
- Sign-on message, 8-3
- signedoper control, 4-44
- source control, 4-45
- Source file, 3-3
- Source file error messages, 8-9
- Source listing, 3-18
- Spaces, 5-27
- Special characters, Macro parameters, 7-24
- sseg* directive, 6-4
- Stack
 - _BOTTOM_OF_STACK_*, 5-33
 - _TOP_OF_STACK_*, 5-33
 - Symbol*, 5-17
- Stack overflow, 5-33
- Stack segment, 5-31
 - user defined*, 5-31
- Statement format, 5-26
- Statement structure, 5-26
 - Delimiters*, 5-7, 5-15
 - Expressions*, 5-16
 - Operands*, 5-16, 5-17
 - Operators*, 5-16
 - Strings*, 5-15
 - Symbols*, 5-9
- Storage-reservation directives, 6-3, 6-13
 - dsb* directive, 6-13
 - dsl* directive, 6-13
 - dsp* directive, 6-13
 - dsq* directive, 6-13
 - dsr* directive, 6-13
 - dsw* directive, 6-13
- String, Defining, 6-7
- String length, 7-19
- Strings, Expressions, 5-17
- subtitle control, 4-47
- Suffix rules, 2-5
- Summary of controls, 3-9
- Summary of options, 3-5
- Symbol definition directives
 - equ*, 6-17
 - set*, 6-17
- Symbol table, 3-18
 - Attributes field*, 3-21
 - Cross-references*, 3-18, 3-22
 - Header*, 3-20
 - Information in object file*, 4-10
 - Name field*, 3-21
 - Value field*, 3-21
- Symbol-definition directives, 6-3
- Symbols
 - Assembler-generated*, 5-10
 - Expressions*, 5-24
 - External*, 5-24
 - External bit numbers*, 5-25
 - Macros*, 7-3, 7-6
 - Relocatable*, 5-24
 - Scope*, 7-6
- symbols control, 4-46
- symbols control , 3-18

T

Temporary files, 1-4, 1-8

Terminating macros, 7-9
title control, 4-49
TMPDIR, 1-4, 1-8
Toggling controls, 4-39
Type attribute, 5-18

U

Using a log file, 3-24
Using batch files, 3-22
Utilities
 OH196 converter, 2-4
 RL196, 2-3, 3-11

V

Value attribute, 5-21

W

Warning messages, 8-8
Word constants, Defining, 6-12
Work files, 1-4, 1-8

X

xref control, 4-50

RELEASE NOTE

INDICATOR : Customer Information Software
INDICATOR NR. : CIS9926
CONCERNS : TK006020-00
80C196 Assembler
Release 6.1

ISSUE DATE : May '99
SUPERSEDES : CIS9827
TO BE FILED IN : 80C196 Assembler User's Guide

SUMMARY

A new release of the 80C196 Assembler has been made: Release 6.1.

The main reasons for this update are:

- Solving of reported problems
- New style manuals
- PDF and HTML versions of on-line manuals

ON-LINE MANUALS

For Windows 95/98 and Windows NT the complete set of manuals is available as Windows on-line help files (in the `etc` directory). The manuals are also available as HTML files for Web browsers (in the `html` directory) and in PDF format (in the subdirectory `pdf`) for viewing with Adobe's Acrobat Reader.

SOLVED / KNOWN PROBLEMS

The distribution contains the file `readme_a.txt` with information about solved problems, known problems and additional notes. For Windows 95/98 and Windows NT the readme is available as on-line help (`readme_a.hlp`). The information is also available as HTML (`readme_a.html`). And there are other `read*. *` files with information about previous releases. They could be of interest to you if you have been using iC-96 before.