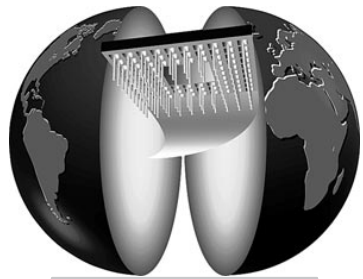
MA006022–00 / 0107

80C196 v6.1

C COMPILER USER'S GUIDE



TASKING
Quality Development Tools Worldwide

A publication of

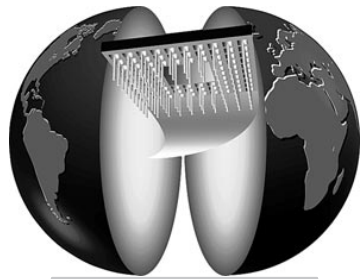
TASKING

Documentation Department

E–mail: support@tasking.com
WWW: http://www.tasking.com

CONTENTS

TABLE OF CONTENTS



TASKING
Quality Development Tools Worldwide

CONTENTS

CONTENTS

LANGUAGE IMPLEMENTATION 10-1

FLEXIBLE LICENSE MANAGER (FLEXLM) A-1

CONTENTS

CONTENTS

MANUAL PURPOSE AND STRUCTURE

PURPOSE

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

INSTALLING THE COMPILER

To install the C196 compiler, see Chapter 1 *Software Installation*. The installation utility on the distribution media leads you through installing the compiler and the utilities on your host system. To automate the compiling 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 COMPILER

To learn how to invoke the compiler, read Chapter 3. To learn how each control affects the compilation process, see Chapter 4. Chapter 9 provides information you can use to interpret a compiler error or warning and including possible causes and suggested actions to recover from the error.

PROGRAMMING IN C196

To learn about the 80C196 architecture and the C196 data types, calling conventions, and library functions, read Chapters 6 through 10 and the example at the end of Chapter 3.

MANUAL STRUCTURE

Related Publications
Conventions Used In This Manual

1. Software Installation
   Describes the installation of the C196 compiler.

2. Overview
   Provides an overview of the TASKING 80C196 toolchain and gives you some familiarity with the different parts of it and their relationship. A sample session explains how to build a 80C196 application from your C file.

3. Compiling and Linking
   Deals with C compiler invocation, output files and describes how to automatically invoke the compiler.

4. Compiler Controls
   Contains an alphabetical list of all compiler controls.

5. Startup Code
   Describes the C startup code.

6. Processor Registers
   Describes the variables declared in the *xx*`_sfrs.h` header files (where *xx* represents the processor as specified with the `model(`*xx*`)` control) for using the SFRs and explains how to use the C196 compiler for efficient register allocation.

7. Assembly Code Instructions
   Describes ways to include assembly language instructions inside your C196 program without requiring a separately written and translated assembly language routine.

8. Libraries
   Contains the library functions supported by the compiler and describes their interface and 'header' files.

9. Messages and Error Recovery
   Describes the error/warning messages of the compiler.

10. Language Implementation
    Concentrates on the approach of the 80C196 architecture and describes the language implementation. The C language itself is not described in this document. We recommend: "The C Programming Language" (second edition) by B. Kernighan and D. Ritchie (1988, Prentice Hall).

APPENDICES

A. Flexible License Manager (FLEXlm)
   Contains a description of the Flexible License Manager.

B. Glossary
   Contains an explanation of terms.

INDEX

RELEASE NOTE

RELATED PUBLICATIONS

- C: A Reference Manual by Harbison and Steele
- The C Programming Language (second edition) by B. Kernighan and D. Ritchie (1988, Prentice Hall)
- ANSI X3.159–1989 standard [ANSI]
- IEEE Standard for Floating–point Arithmetic 754–1985

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

Only the TASKING publications are included in the C196 manual package. Intel publications can be ordered from Intel's Literature Center.

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.

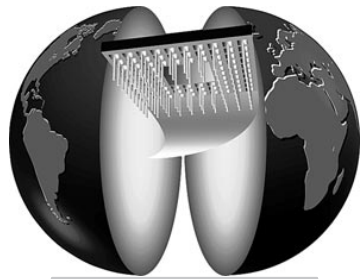
This illustration indicates actions you can perform with the mouse.

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



TASKING
Quality Development Tools Worldwide

1 CHAPTER

1.1 INTRODUCTION

This chapter describes how you can install the TASKING 80C196 C Compiler 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

The C196 compiler recognizes several environment variables that you can use to reduce the amount of typing required for a compiler invocation. These environment variables are as follows:

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 C196 compiler, you need not retype the full pathname each time you invoke the compiler. 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.

C196INC

C196INC is recognized by the compiler as a list of prefix strings, separated with semicolons, that the compiler can use to locate include files. If you specify a filename or partial pathname with the `include` control or the `#include` preprocessor directive, the compiler prepends each string in C196INC in turn and uses each resulting pathname as the name of the include file.

The compiler uses the pathnames formed from the list in C196INC in addition to any you specify with the `searchinclude` control. For example, the following definition of C196INC locates the files `c:\c196\include\stdio.h` and `c:\working\kal4.h` when the `include (stdio.h,kal4.h)` control is specified:

```
set C196INC=c:\c196\include;c:\working
```

C96INIT

C96INIT is recognized by the compiler as a prefix string used to form the pathname of a file named `c96init.h`. For example, setting C96INIT as follows causes the compiler to use the `c96init.h` file in the `c:\working` directory:

```
set C96INIT=c:\working\
```

If C96INIT is not defined or is empty, the compiler searches your current working directory for `c96init.h`.

The compiler always processes `c96init.h`, if it exists, as the first source text to be compiled. You need not specify `c96init.h` in the compiler invocation or in a preprocessor directive.

TMPDIR

The compiler creates temporary work files, which it normally deletes when compilation is complete. If the compilation is interrupted, for example, by your host system losing power, the work files remain. If you see a file with a name that looks like `c`*`num`*`a.dat` then you can simply remove it. The TMPDIR environment variable specifies the directory where the compiler is to put these temporary files. If TMPDIR is not defined or is empty, the compiler uses your current working directory for the temporary files. For example, setting TMPDIR as follows causes the compiler 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 ***
SW006022 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 SW006022 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@elliot
```

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 C196 compiler recognize several environment variables that you can use to reduce the amount of typing required for a compiler 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 C196 compiler, you need not retype the full pathname each time you invoke the compiler. Your UNIX literature explains how to define the PATH environment variable.

C196INC

C196INC is recognized by the compiler as a list of prefix strings, separated with colons, that the compiler can use to locate include files. If you specify a filename or partial pathname with the `include` control or the `#include` preprocessor directive, the compiler prepends each string in C196INC in turn and uses each resulting pathname as the name of the include file.

The compiler uses the pathnames formed from the list in C196INC in addition to any you specify with the `searchinclude` control. For example, the following definition of C196INC locates the files `../include/stdio.h` and `/proj/working/kal4.h` when the `include (stdio.h,kal4.h)` control is specified:

if using the Bourne shell (sh)

```
C196INC=../include:/proj/working
export C196INC
```

or if using the C–shell (csh)

```
setenv C196INC ../include:/proj/working
```

C96INIT

C96INIT is recognized by the compiler as a prefix string used to form the pathname of a file named `c96init.h`. For example, setting C96INIT as follows causes the compiler to use the `c96init.h` file in the `/proj/working` directory:

```
setenv C96INIT /proj/working/
```

If C96INIT is not defined or is empty, the compiler searches your current working directory for `c96init.h`.

The compiler always processes `c96init.h`, if it exists, as the first source text to be compiled. You need not specify `c96init.h` in the compiler invocation or in a preprocessor directive.

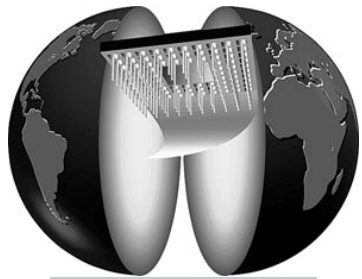
TMPDIR

The compiler creates temporary work files, which it normally deletes when compilation is complete. If the compilation is interrupted, for example, by your host system losing power, the work files remain. If you see a file with a name that looks like `c`*`num`*`a.dat` then you can simply remove it. The TMPDIR environment variable specifies the directory where the compiler is to put these temporary files. If TMPDIR is not defined or is empty, the compiler uses the `/tmp` directory for the temporary files. For example, setting TMPDIR as follows causes the compiler to use the `/tmp` directory for temporary work files:

```
setenv TMPDIR /tmp
```

CHAPTER 2

OVERVIEW



TASKING
Quality Development Tools Worldwide

2 CHAPTER

This chapter introduces you to the C196 compiler and to the related 80C196 utilities. Intended for the new user, this overview helps you understand the general function of the compilation system and directs you to sources of detailed and supplemental information. At the end of this chapter is a sample session.

2.1 C196 AND THE SOFTWARE DEVELOPMENT PROCESS

Figure 2–1 shows a development chart using the C196 compiler and other tools.

With the C196 compiler, you can develop an application using the following techniques:

- Compile and test the application as separate modules, specifying only one or some of the source files in each compilation.
- Use the appropriate compiler controls and preprocessor directives to include source text from several files, or from a set of alternative files, in a single compilation.
- Call functions written in 80C196 assembly language or include in–line source assembly language in your C program. See Chapter 7 for further information on including assembly language code in a C program.

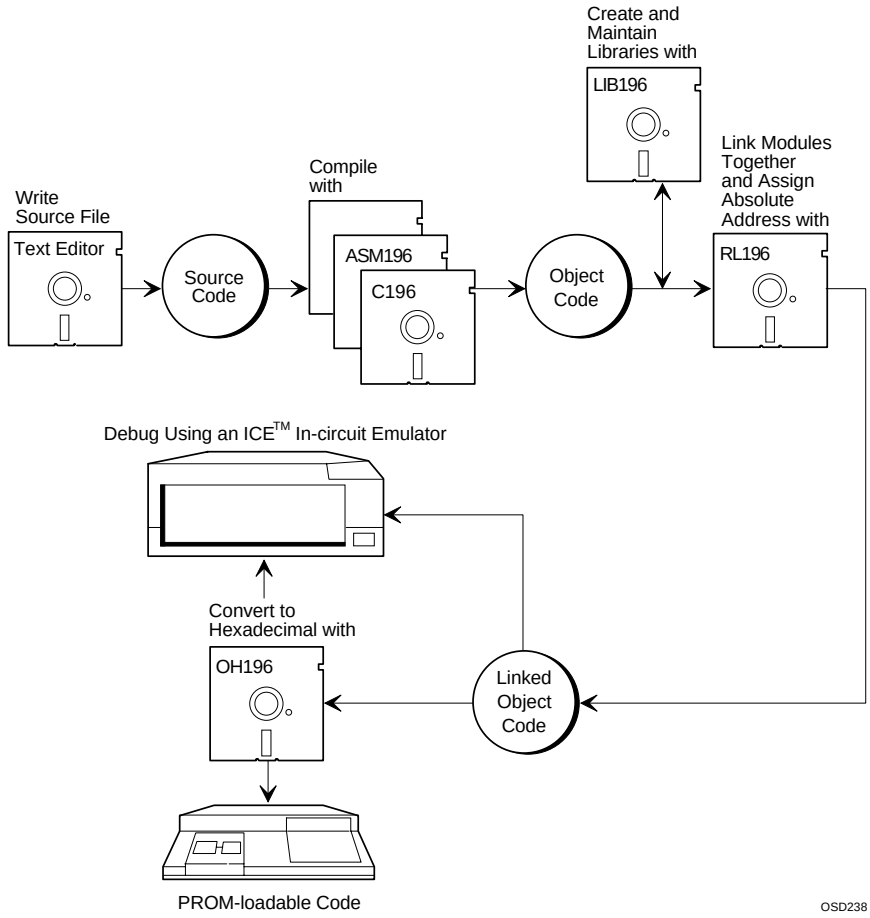


Figure 2–1: 80C196 application development process

To create the source text, use any editor that generates ASCII files. Invoke the compiler to translate the source text into object code, as shown in Chapter 3. You can also use several compiler controls to manipulate the output object file and the output print file. These controls can help you debug your application by including information such as pseudo–assembly language listings, symbolic information, line numbers, and diagnostic messages. You must include symbolic information and line numbers to use symbolic debuggers and emulators.

See Chapter 4 for a detailed description of each compiler control. Chapter 9 describes each diagnostic generated by the compiler.

Use RL196 to link object modules from C196, PL/M–96, and ASM196 and to assign absolute addresses to a module.

If you are linking to an object module from PL/M–96, you must ensure the calling sequence generated by the C196 compiler for the function call matches the fixed parameter–list calling sequence generated by the PL/M–96 translator for the called function. You can declare a fixed–parameter list function in either of the following ways:

- By using the `fixedparams` control or the `#pragma fixedparams` directive.
- By declaring the function with the `alien` keyword. To enable the compiler to recognize this keyword, you must use the `extend` control.

Your application can also call functions from the libraries included in the C196 package or from your own library. You can create your own libraries using the LIB196 utility.

Use LIB196 to organize frequently used 80C196 object modules into libraries.

Use OH196 to convert 80C196 object code into hexadecimal format for PROM programming. You can use an Intel ICE™ in–circuit emulator to debug either object module format code or hexadecimal format code.

The C196 implementation of C conforms to the ANSI standard for the C language (x3.159 – 1989), and also supports applications that use features specific to 80C196 architecture.

See Chapter 10 for more information on language implementation.

2.2 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.

2.2.1 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 C196 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.

2.3 SAMPLE SESSION

The subdirectories of the `examples` directory contain demo programs for the 80C196 toolchain.

In order to debug your programs, you will have to compile, assemble, link and locate them for debugging using the TASKING 80C196 tools. You can do this with EDE, the Embedded Development Environment (which uses a project file and a makefile) or you can call the makefile from the command line.

2.3.1 USING EDE

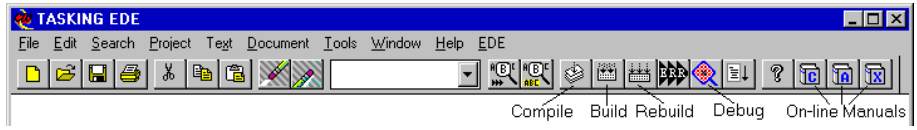
EDE stands for ”Embedded Development Environment” and is the Windows oriented Integrated Development Environment you can use with your TASKING toolchain to design your application.

To use EDE on one of the demo programs in one of the subdirectories in the `examples` subdirectory of the 80C196 product tree follow the steps below. This procedure is outlined as a guide for you to build your own executables for debugging.

How to Start EDE

You can launch EDE by double–clicking on the appropriate icon in the program group created by the installation program. Or you can launch EDE by double–clicking on the EDE shortcut on your desktop.

The EDE screen provides you with a menu bar, a ribbon bar (command buttons) and one or more windows (for example, for source files), a status bar and numerous dialog boxes.

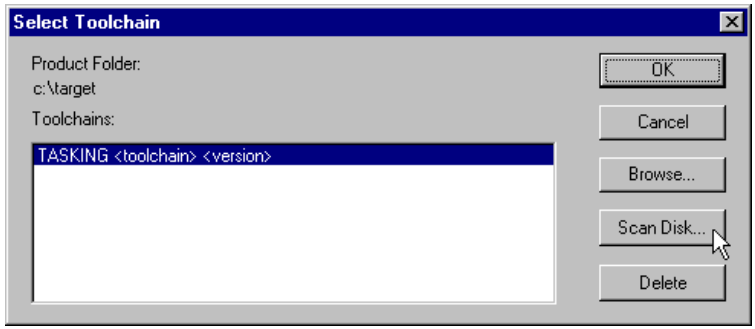


How to Select a Toolchain

EDE supports all the TASKING toolchains. When you first start EDE, the correct toolchain of the product you purchased is selected and displayed in the title of the EDE desktop window.

If you selected the wrong toolchain or if you want to change toolchains do the following:

1. Access the `EDE` menu and select the `Select Toolchain...` menu item. This opens the `Select Toolchain` dialog.
2. Select the toolchain you want. You can do this by clicking on a toolchain in the `Toolchains` list box and press `OK`.



If no toolchains are present, use the `Browse...` or `Scan Disk...` button to search for a toolchain directory. Use the `Browse...` button if you know the installation directory of another TASKING product. Use the `Scan Disk...` button to search for all TASKING products present on a specific drive. Then return to step 2.

How to Open an Existing Project

Follow these steps to open an existing project:

1. Access the `Project` menu and select `Open...`.
2. Select the project file to open and then click `OK`. For the LED light program select the file `ea_examp.pjt` in the subdirectory `ea_examp` in the `examples` subdirectory of the 80C196 product tree. If you have used the defaults, the file `ea_examp.pjt` is in the directory `c:\c196\examples\ea_examp`. If you want to build another example, open the project file in the corresponding subdirectory of the `examples` directory.

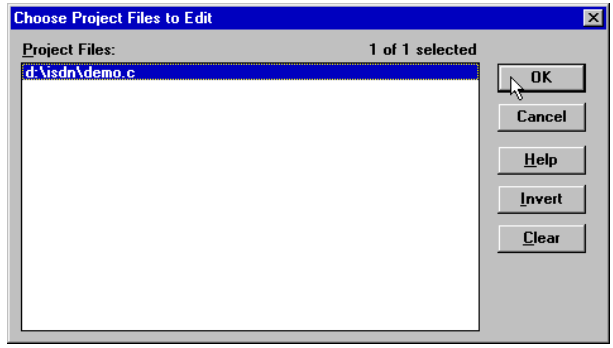
How to Load/Open Files

The next two steps are not needed for the demo program because the source files and `makefile` are already open. To load the file you want to look at.

1. In the `Project` menu click on `Load files...`.

   This opens the `Choose Project Files to Edit` dialog.

2. Choose the file(s) you want to open by clicking on it. You can select multiple files by pressing the `<Ctrl>` or `<Shift>` key while you click on a file. With the `<Ctrl>` key you can make single selections and with the `<Shift>` key you can select everything from the first selected file to the file you click on. Then press the `OK` button.



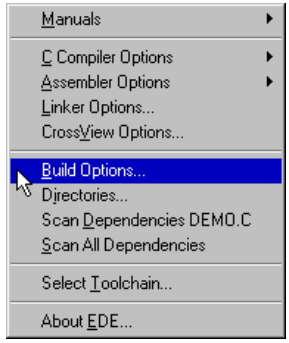
   This launches the file(s) so you can edit it (them).

How to Build the Demo Application

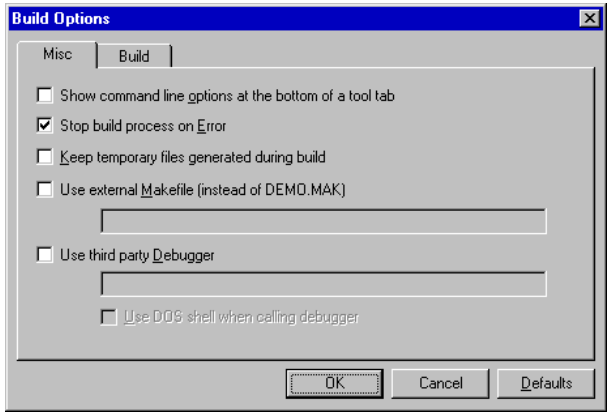
The next step is to compile the file(s) together with its dependent files so you can debug the application.

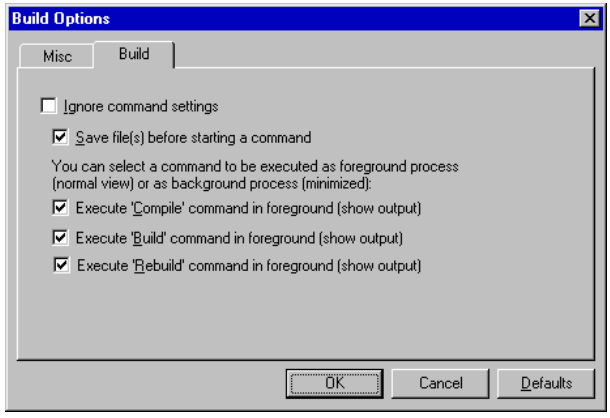
Steps 1 and 2 are optional. Follow these steps if you want to specify additional build options such as to stop the build process on errors and to select a command to be executed as foreground or background process.

1. Access the `EDE` menu and select the `Build Options...` menu item.



This opens the `Build Options` dialog.

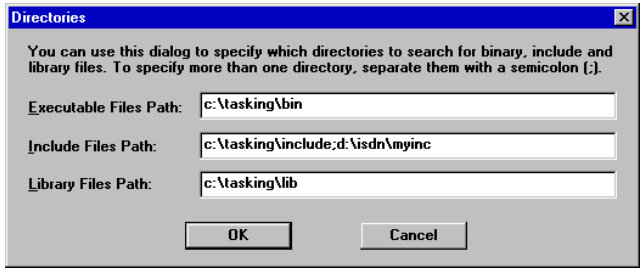




If you set the `Show command line options at the bottom of a tool tab` check box EDE shows the command line equivalent of the selected tool option.

2. Make your changes and press the `OK` button.

3. Select the `EDE | Directories` menu item and check the directory paths for programs, include files and libraries. You can add your own directories here, separated by semicolons.



4. Access the `EDE` menu and select the `Scan All Dependencies` menu item.

5. Click on the `Execute 'Make' command` button. The following button is the execute Make button which is located in the ribbon bar.

If there are any unsaved files, EDE will ask you in a separate dialog if you want to save them before starting the build.

How to View the Results of a Build

Once the files have been processed you can inspect the generated messages.

1. In the `Window` menu select the `Output` menu item.

You can see which commands (and corresponding output captured) which have been executed by the build process in the `Build` tab:

```
c196 lights.c –f c:\tmp\mkdac57a.tmp

C196 Compilation Complete.  0 Remarks, 0 Warnings, 0 Errors

c196 wait_ms.c –f c:\tmp\mkdac57b.tmp

C196 Compilation Complete.  0 Remarks, 0 Warnings, 0 Errors

asm196 –f c:\tmp\mkdac57c.tmp

ASSEMBLY COMPLETED,   NO ERROR(S) FOUND.
rl196 –f c:\tmp\mkdac57d.tmp
RL196 COMPLETED,   0 WARNING(S),   0 ERROR(S)
oh196 lights.abs to lights.hex
80C196 program builder vx.y rz        SN00000000-003 (c) year TASKING, Inc.
```

How to Start the Debugger

Once the files have been compiled, assembled, linked, located and formatted they can be debugged.

To execute the debugger:

1. Click on the `Debug application` button. The following button is the Debug application button which is located in the ribbon bar.

Depending on the project file you selected, either the ChipTools simulator or the Intel AppBuilder is launched. The simulator will automatically download the compiled file for debugging.

How to Start a New Project

When you first use EDE you need to setup a project:

1. Access the `Project` menu and select `New...`.
2. Give your project a name and then click `OK`.

The `Project` dialog box appears.

3. Add all source files and any other files you want associated with your application by locating the appropriate directory and selecting the files and clicking the `Add` button. When all the files you want associated with your application appear in the `Project Files field`, click `OK`.

If you do not have any source files yet, close the Project dialog and create the source files. You can create a new file by selecting `File | New`. Write your source code and select `File | Save As...` to save your source code. Select `Project | Properties...` from the menu and select the tab `Files`. Now go back to step 3.

The new project is now open.

4. Click `Project | Load Files` to open files you want on your EDE desktop.

EDE automatically creates a makefile for the project. EDE updates the makefile every time you modify your project.

2.3.2 USING THE MAKEFILE

The subdirectories in the `examples` directory each contain a `makefile` which can be processed by **mk196**. The `examples` directory contains a `readme.txt` file with a short description of each example.

To build an example follow the steps below. This procedure is outlined as a guide for you to build your own executables for debugging.

1. Make the subdirectory `ea_examp` of the `examples` directory the current working directory.

   This directory contains a makefile for building the LED light example. It uses the default **mk196** rules.

2. Be sure that the directory of the binaries is present in the PATH environment variable.

3. Compile, assemble, link and locate the modules using one call to the program builder **mk196**:

   ```
   mk196
   ```

   This command will build the example using the file `makefile`.

To see which commands are invoked by **mk196** without actually executing them, type:

```
mk196 -n
```

This command produces the following output:

```
80C196 program builder vx.y rz        SN000000-019 (c) year TASKING, Inc.
c:\tmp\mk25495a.tmp:
model(ea-enf) &
type &
debug &
code &
dn(0) &
ot(0) &
ms
c196 lights.c -f c:\tmp\mk25495a.tmp
c:\tmp\mk25495b.tmp:
model(ea-enf) &
type &
debug &
code &
dn(0) &
ot(0) &
ms
c196 wait_ms.c -f c:\tmp\mk25495b.tmp
c:\tmp\mk25495c.tmp:
ea_start.a96 ri model(ea-enf) cmain
asm196 -f c:\tmp\mk25495c.tmp
c:\tmp\mk25495d.tmp:
lights.obj, &
wait_ms.obj, &
ea_start.obj, &
c96.lib &
to lights.abs &
model(ea-enf) ram(01Ah-17Fh) rom(0FF2080h-0FF2FFFh) ss(+6) sfr
rl196 -f c:\tmp\mk25495d.tmp
oh196 lights.abs to lights.hex
```

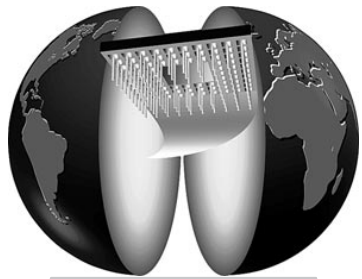
The **debug** control in the makefile (in the CCFLAGS and ASFLAGS macro definition) is used to instruct the C compiler to generate symbolic debug information. This information makes debugging an application written in C much easier to debug.

To remove all generated files type:

```
mk196 clean
```

CHAPTER 3

COMPILING AND LINKING



TASKING
Quality Development Tools Worldwide

CHAPTER 3

3.1 INTRODUCTION

This chapter provides the information you need to invoke the C196 compiler. You can compile a C196 program, without making any modifications to the basic environment, simply by specifying the complete compiler invocation syntax each time you use the compiler. However, setting environment variables, discussed in the *Software Installation* chapter, specifying compiler controls, discussed in detail in Chapter 4, using makefiles, and using batch files and UNIX shell scripts can greatly reduce the complexity of a compiler invocation.

3.2 COMPILER INVOCATION SYNTAX

For the following syntax, the square brackets ([ ]) enclose optional elements for the command line. If you do not specify an optional element, do not use an empty pair of parentheses.

The C196 compiler invocation has the following format:

[*cpath*]**c196** [*spath*]*filename* [*controls*]

Where:

cpath	is the path to the directory that contains the compiler.
spath	is the path to the directory that contains the primary source file.
filename	is the name of the primary source file.
controls	are the compiler controls, separated with spaces. For a complete description of each control, see Chapter 4.

3.2.1 HOW CONTROLS AFFECT THE COMPILATION

Each control affects the compilation in one of three ways:

Source processing controls

specify the names and locations of input files and define macros at compile time.

Object file content controls

specify the contents of the object file.

Listing controls

specify the names, locations, contents, and formats of the output listing files.

3.2.2 WHERE TO SPECIFY CONTROLS

The three types of controls are: primary, general, and invocation–only controls. You can specify these compiler controls in the source text and compiler invocation. The type determines where and how often you can specify any particular control, as follows:

Primary

You can specify this type of control once in the compiler invocation or in a `#pragma` preprocessor directive preceding the first executable statement or data definition statement in the source text. A primary control applies to the entire module and you cannot change or suspend its effects for any part of the source text. To override a primary control specified in a `#pragma`, specify a contradictory control in the invocation. An example of a primary control is the `model` control, which selects the instruction set of one of the processors for the module.

General

You can specify this type of control as often as necessary in the compiler invocation and in `#pragma` preprocessor directives throughout the source text. A general control applies to the subsequent source text or to the arguments of the control. Each specification of a general control adds to or overrides previous specifications of the control. An example of a general control is the `[no]list` control, as follows:

```
#pragma list     /* The following lines are listed */
int f1 (int f)
{
   return(f);
}
```

```
#pragma nolist   /* The following lines are
                    not listed */
long f2 (long i)
{
   return(i);
}
```

Invocation–only

You can specify this type of control as often as necessary in the invocation. An invocation–only control applies to the entire module or to the arguments of the control. Each specification of an invocation–only control adds to or overrides previous specifications of the control. An example of an invocation–only control is the `define` control, which defines a macro.

To save effort, you can put any controls that you use in the compiler invocation into a file named `c96init.h`. The compiler automatically includes this file before the primary source file and before any include file specified in the invocation. See the *Software Installation* chapter under environment variables for more information on the `c96init.h` file.

Table 3–1 lists the controls with descriptions, defaults, precedences, effects, and usage classes. Some controls optionally use one or more arguments, indicated by `[`*a*`]`, where *a* represents the argument list. Some controls require one or more arguments, also indicated by *a*. Case is not significant in the controls but can be significant in arguments to the controls.

Certain controls override other controls even if you explicitly state the overridden controls. Table 3–1 summarizes these precedences.

Control	Abbr.	Description and Default	Effect	Usage
abszero noabszero	az noaz	Tells the compiler to zero uninitialized variables in absolute segments. Default: noabszero.	Object	Primary
bmov nobmov	bm nobm	Tells the compiler to use the bmov instruction to initialize or copy structures or array elements. Default: nobmov.	Object	Primary
case nocase	cs nocs	Tells the compiler to work in a case sensitive manner. Default: nocase.	Object	Primary
ccb	cc	Specifies the initial CCB value.	Object	Primary

Control	Abbr.	Description and Default	Effect	Usage
code nocode	co noco	Generates or suppresses pseudo–assembly code listing in the print file. Default: nocode.	Listing content	General
cond nocond	cd nocd	Includes or suppresses conditionally uncompiled source code in the print file. Default: nocond.	Listing content	General
debug nodebug	db nodb	Includes or suppresses debug information in the object module. Default: nodebug.	Object	Primary
define(*a*)	df	Defines an object–like macro.	Source	Invocation
diagnostic(*a*)	dn	Specifies the level of diagnostic messages.Default: diagnostic level 1.	Listing content	Primary
divmodopt nodivmodopt	dm nodm	Enables generation of efficient DIV instruction. Default: nodivmodopt.	Object	Primary
eject	ej	Inserts a form–feed in the print file. Can only be specified in a #pragma directive.	Listing content	General
extend noextend	ex noex	Recognizes or suppresses Intel extensions to proposed ANSI C. Default: extend.	Source	Primary
extratmp noextratmp	et noet	Allows usage of extra temporary registers TMPREG8 to TMPREG16. Default: noextratmp for non 24–bit models extratmp for 24–bit models.	Object	Primary
farcode nearcode	fc nc	Tells the compiler to generate code for the extended (fc) or compatibility (nc) mode of 24–bit processors. Default: nearcode.	Object	Primary
farconst nearconst	fk nk	Tells the compiler to place constant objects in either the farconst or const segment. Only valid with 24–bit processors. Default: nearconst.	Object	Primary
fardata neardata	fd nd	Tells the compiler to place non–constant, non–register data in either the fardata or data segment. Only valid with 24–bit processors. Default: neardata.	Object	Primary
fastinterrupt nofastinterrupt	fi nofi	Specifies the compiler not to save temporary results on entering the interrupt routine. Default: nofastinterrupt.	Object	General
fixedparams [(*a*)] varparams[(*a*)]	fp vp	Specifies the FPL or VPL function–calling convention. Default: varparams for non–alien functions.	Object	General

Control	Abbr.	Description and Default	Effect	Usage
generatevectors nogeneratevectors	gv nogv	Generate interrupt vectors Default: generatevectors.	Object	Primary
hold nohold	ho noho	Specifies whether the windowing code needs to preserve the HOLD/HOLDA bit in the WSR. Default: nohold	Object	Primary
include(*a*)	ic	Specifies a file to process before the primary source file.	Source	Invocation
init noinit	it noit	Tells the compiler to produce initialization segments and tables. Default: init.	Object	Primary
inst noinst	is nois	Specifies whether the compiler creates vector tables for switch statements. Default: noinst	Source	Primary
interrupt(*a*)	in	Specifies a function to be an interrupt handler.	Object	General
interrupt_piha(*a*) interrupt_pihb(*a*)		Specifies a function to be an interrupt handler in the piha/pihb block.	Object	General
interruptpage(*a*)	ip	Specifies an interrupt page number or base address.	Object	Primary
list nolist	li noli	Includes or suppresses the source text listing in the print file. Default: list. The nolist control overrides cond, listexpand, and listinclude.	Listing content	General
listexpand nolistexpand	le nole	Includes or suppresses macro expansion in the print file. Default: nolistexpand.	Listing content	General
listinclude nolistinclude	lc nolc	Includes or suppresses text from include files in the print file. Default: nolistinclude. The nolistinclude control overrides listexpand and cond for include files.	Listing content	General
locate(*a*,...)	lo	Locates symbols to absolute addresses.Can only be specified in a #pragma directive.	Object	General
mixedsource nomixedsource	ms noms	Specifies to generate mixed assembly source in the print file. Default: nomixedsource.	Listing content	Primary
model(*a*)	md	Selects the processor instruction set. Default: model(kb).	Object	Primary
object [(*a*)] noobject	oj nooj	Generates and names or suppresses the object file. Default: sourcename.obj. The noobject control overrides all object controls except for their effects on the print file.	Object	Primary

Control	Abbr.	Description and Default	Effect	Usage
oldobject nooldobject	oo nooo	Tells the compiler to produce an object file compatible with V2.*x* (the 16–bit only C196 compiler). Default: nooldobject.	Object	Primary
omf(*a*)	omf	Specifies the OMF96 version. Default: omf(2).	Object	Primary
optimize(*a*)	ot	Specifies the level of optimization. Default: optimization level 1.	Object	Primary
overlay	ov	Locates register symbols to absolute addresses in the overlay register segments. (Pragmas only)	Object	General
pagelength(*a*)	pl	Specifies the number of lines per page in the print file. Default: 60 lines per page.	Listing format	Primary
pagewidth(*a*)	pw	Specifies the number of characters per line in the print file. Default: 120 characters per line.	Listing format	Primary
preprint [(*a*)] nopreprint	pp nopp	Generates and names or suppresses the preprint file. Default: nopreprint.	Listing content	Invocation
print [(*a*)] noprint	pr nopr	Generates and names or suppresses the print file. Default: sourcename.lst. The noprint control overrides all listing controls except preprint.	Listing content	Primary
pts(*a*)	pt	Loads a PTS vector with the address of a PTS control block.	Object	General
pts_piha(*a*) pts_pihb(*a*)		Loads a piha/pihab PTS vector with the address of a PTS control block.	Object	General
reentrant [(*a*)] noreentrant [(*a*)]	re nore	Specifies reentrancy or nonreentrancy for a function. Default: reentrant.	Object	General
regconserve [(*a*)] noregconserve	rc norc	Controls whether non–register, file–level, and automatic variables are allocated to registers. Default: noregconserve.	Object	Primary
registers(*a*)	rg	Specifies the number of bytes of register memory available to the module. Default: registers(220).	Object	Primary
relocatabletemps norelocatabletemps	rt nort	Tells the compiler to produce external references to temporary registers symbols. Default: norelocatabletemps.	Object	Primary
searchinclude(*a*) nosearchinclude	si nosi	Specifies the search path for include files. Default: nosearchinclude.	Source	General
signedchar nosignedchar	sc nosc	Causes a char to be treated as a signed char or an unsigned char. Default: signedchar.	Object	Primary

Control	Abbr.	Description and Default	Effect	Usage
speed(*a*)	sp	Tells the compiler to choose between faster code and less code size. Default: speed(0).	Object	Primary
symbols nosymbols	sb nosb	Generates or suppresses the identifier list in the print file. Default: nosymbols.	Listing content.	Primary
tabwidth(*a*)	tw	Specifies the number of characters between tabstops in the print file. Default: 4 characters between tabstops.	Listing format	Primary
title("*a*")	tt	Places a title on each page of the print file. Default: "modulename".	Listing format	Primary
tmpreg(*a*)	tr	Locates the temporary registers at a different memory location. Default: 1CH.	Object	Primary
translate notranslate	tl notl	Completes or stops the compilation after preprocessing. Default: translate. The notranslate control overrides all object and listing controls except preprint.	Object	Invocation
type notype	ty noty	Generates or suppresses type information in the object module. Default: type.	Object	Primary
warning_true_false nowarning_true_false	wt nowt	Enables or disables the 'comparison always returns TRUE' and 'comparison always returns FALSE' warnings Default: warning_true_false.	Object	Primary
win1_32 win1_64	v3 v6	Combination of pragma locate and pragma overlay. Only if WSR1 is present in processor.	Object	General
win32 win64 win128	w3 w6 w1	Combination of pragma locate and pragma overlay.	Object	General
windowram(*a*)	wr	Specifies the area(s) of memory from which to allocate windowed variables.	Object	General
windows[(*a*)] nowindows	wd nowd	Specifies that the whole application uses the vertical windows of the 80C196KC, 80C196KR, or 80C196NT Default: nowindows.	Object	Primary
wordalign nowordalign	wa nowa	Tells the compiler to align long register objects on word boundaries rather than restricting them to longword boundaries. Default: nowordalign.	Object	Primary

Control	Abbr.	Description and Default	Effect	Usage
xref noxref	xr noxr	Adds or suppresses the identifier cross–reference listing in the print file. Default: noxref. The xref control overrides nosymbols.	Listing content	Primary
zero nozero	zr nozr	Tells the compiler to zero uninitialized variables in relocatable segments. Default: zero. (same as init)	Object	Primary

Table 3–1: Compiler controls summary

The following example compiles the primary source file `serial.c` in the `examples` directory, with the `model(kb)`, `optimize(0)`, and `debug` controls specified:

```
c196 examples/serial.c md(kb) ot(0) db
```

You can eliminate repetitive typing of the invocation line by using **mk196**, batch files or command files.

See Section 3.5 for a complete explanation of automating compiler invocations.

3.3 FILENAME CONVENTIONS

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**.

Extension	Description
.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.
.m96	MAP file
.mak	For Makefiles other than 'Makefile' or 'makefile'.
.hex	Hexadecimal output file by **oh196**.

Table 3–2: 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/c196`)

3.4 OUTPUT FILES

The compiler creates and deletes temporary work files during the compilation process and can produce an object file and two listing files, as shown in Table 3–3.

File type	Filename[1]	Contents	Compiler Controls	Defaults
object file	*source*.obj	object module	object or translate	object, translate
preprint file	*source*.i	preprocessed source text	preprint or notranslate	nopreprint, translate
print file	*source*.lst	listings, compilation results	print	print
[1] *source* is the filename of the primary source file, without the filename extension.				

Table 3–3: Compiler output files

Figure 3–1 shows the input and output files of the C196 compiler.

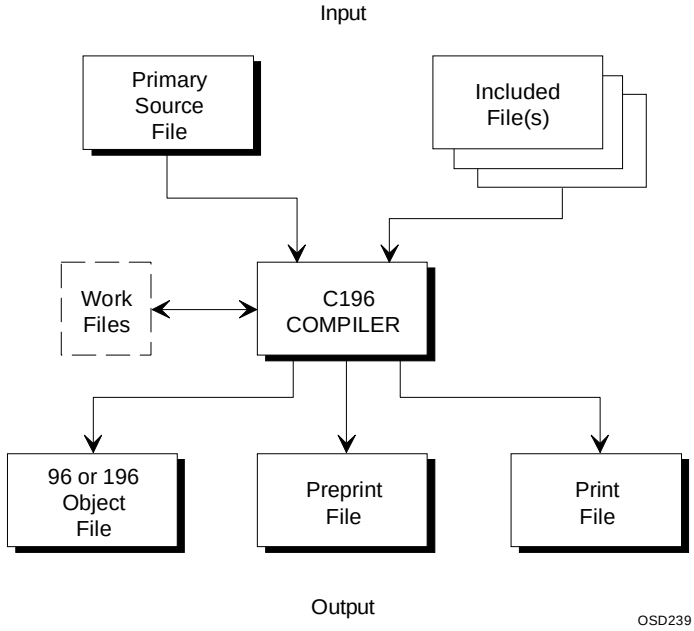


Figure 3–1: C196 input and output files

The two optional listing files produced by the compiler, the preprint file and the print file, embody two aspects of compilation. The preprint file contains the source text after textual preprocessing such as including files and expanding macros. The print file contains information about the results of compiling the source text into an object module. By default, the compiler generates a print file but not a preprint file. The following sections describe the two listing files in detail.

3.4.1 PREPRINT FILE

In generating a preprint file from a source text file, the compiler completes the following operations:

- expands macros by substituting the body, or textual value, of each macro for each occurrence of its name.
- inserts source text from files specified with the `include` compiler invocation control or the `#include` preprocessor directive and inserts the `#line` preprocessor directive to bracket sections of included source text in the preprint file.

- eliminates parts of the source text based on the `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, and `#endif` conditional compilation directives.
- propagates the preprocessor directives `#line`, `#error`, and `#pragma` from the source text to the preprocessed source text.

The preprint file contains the preprocessed source text after all of these operations are completed. The preprint file is especially useful for observing the results of these operations. Compiling the preprint file produces the same results as compiling the source file.

The compiler generates a preprint file only when the `preprint` or `notranslate` control is in effect. The default name for the preprint file is the same as the primary source filename with the `.i` extension substituted for the original extension, as shown in Table 3–3. The compiler places the preprint file by default in the same directory that contains the source file. To override the defaults, use the `preprint` control. If a file with the same name already exists, the compiler writes over it.

3.4.1.1 MACROS

You can see the results of macro expansion in the preprint file. The preprocessor substitutes the body of a macro everywhere a macro name appears in the subsequent source text. To define a macro, use the `define` control or the `#define` preprocessor directive. See the book *C: A Reference Manual*, listed in *Related Publications*, for details on how to use the `#define` preprocessor directive.

The C196 compiler provides several predefined macros for your convenience. Table 3–4 shows these macros and their values.

Name	Value
__LINE__	current source line number
__FILE__	current source filename
__DATE__	date of compilation
__TIME__	time of compilation
__STDC__	conformance to ANSI C 1 indicates conformance
_16_BITS_	defined (set to 1) when using a 16–bit model
_24_BITS_	defined (set to 1) when using a 24–bit model

Name	Value
ARCHITECTURE	processor model compiler uses for object module. See the model() control.
_C196_	always set to 1
DEBUG	level of debug and type information included in object code: 0 if using the nodebug and notype controls 1 if using the nodebug and type controls 2 if using the debug and notype controls 3 if using the debug and type controls
DIAGNOSTIC	level of diagnostics reported: 2 if only errors are reported 1 if warnings and errors are reported 0 if all diagnostics are reported
_FAR_CODE_	operating mode for 24–bit model: 1 if using the farcode control 0 if using the nearcode control
_FAR_CONST_	default placement of constant objects: 1 if using the farconst control 0 if using the nearconst control
_FAR_DATA_	default placement of non–constant objects: 1 if using the fardata control 0 if using the neardata control
_FUNCS_H_	name of processor specific *xx*_funcs.h include file, *xx* is one of the processor models as specified by the model() control.
_HAS_PTS_	only defined as 1 if the processor has a PTS unit
_OMF96_VERSION_	OMF version: 0 V2.0 1 V3.0 2 V3.2
OPTIMIZE	current optimization level as set by the optimize control: 0, 1, 2, or 3
REGISTERS	number of bytes of register memory available for register variable allocation, as specified by the registers control

Name	Value
SIGNEDCHAR	signed or unsigned default of char variables: 1 if using the signedchar control 0 if using the nosignedchar control
_SFR_H_	name of processor specific *xx*_sfrs.h include file, *xx* is one of the processor models as specified by the model() control.

Table 3–4: C196 predefined macros

3.4.1.2 INCLUDE FILES

The preprint file also shows the source text files inserted from file inclusions. To include files in the source text, use the `include` control in the compiler invocation or the `#include` preprocessor directive. The preprocessor inserts the contents of a file included with the `include` control before the first line of the source file. The preprocessor inserts the contents of a file included with the `#include` preprocessor directive into the source text in place of the line containing the `#include` directive.

Paired occurrences of the `#line` preprocessor directive bracket the included text. The compiler inserts the `#line` directive in the preprint listing file at the beginning of the included text and another `#line` directive at the end of the included text. *C: A Reference Manual*, listed in *Related Publications*, provides more information on preprocessor directives.

3.4.1.3 CONDITIONAL COMPILATION

Conditional preprocessor directives delimit sections of source text to be compiled only if the conditional expression evaluates to true. The preprocessor evaluates the conditions and determines which sections of source text are compiled. The source text that is not compiled does not appear in the preprint file.

The conditional directives are `#if`, `#else`, `#elif`, `#endif`, `#ifdef`, and `#ifndef`. The `#if` directive can take a special `defined` operator. See the book *C: A Reference Manual*, listed in *Related Publications*, for explanations on how to write a program using conditional compilation.

3.4.1.4 PROPAGATED DIRECTIVES

The preprocessor propagates the directives `#line`, `#error`, and `#pragma` from the source text to the preprint file to ensure that the preprint text is equivalent to the source text after preprocessing. See the book *C: A Reference Manual*, listed in *Related Publications*, for more information on preprocessor directives.

3.4.2 PRINT FILE

The print file can contain source text and pseudo–assembly code listings, messages, symbolic information, and summary information about the compilation. The compiler generates the print file by default. Use the `noprint` control to suppress the print file.

The default name for the print file is the same as the primary source filename with the `.lst` extension substituted for the original extension, as shown in Table 3–3. The compiler places the print file by default in the directory that contains the source file. If a file with the same name already exists, the compiler writes over it. To override the defaults, use the `print` control.

The following sections describe the print file generated by the compiling phase of the compiler. The print file contains information about the source text read into the compiler and the object code generated by the compiler.

3.4.2.1 PRINT FILE CONTENTS

The print file contains the following sections:

Page header	identifies the compiler, shows the title of the print listing, and gives the date and time of compilation.
Compilation heading	identifies the host operating system, the compiler version, the object module name, and the controls used in the invocation.
Source text listing	lists the C program. Remarks, warnings, and error messages, generated by the compiler, are listed with the source text.

Symbol table and cross–reference	provides symbolic information and cross–reference information.
Pseudo–assembly listing	lists the assembly language object code produced by the compiler. The code does not contain all the assembler directives necessary for a complete assembly language program but shows the instructions generated by the compiler.
Compilation summary	tabulates the size of the output module, the number of diagnostic messages, and the completion status (successful termination or fatal error) of the compilation.

You can use compiler controls and `#pragma` preprocessor directives to produce, suppress, or partially suppress the source text listing, messages, pseudo–assembly listing, and cross–referenced symbol table. The following controls affect the format and contents of the print file:

```
[no]code     [no]listexpand     [no]symbols
[no]cond     [no]listinclude    tabwidth
diagnostic   pagelength         title
[no]list     pagewidth          [no]xref
```

3.4.2.2 PAGE HEADER

Each page of the output listing file begins with a page header. The page header describes the compiler, identifies the module compiled, and shows the date and page number. The page header in Figure 3–2 shows the C196 compiler compiling the module `CTYPE` on 01/29/99 at approximately 2:26 p.m. Page numbers range from 1 to 999, then start over at 0. The example in Figure 3–2 is from the first page of the print file.

```
C196 Compiler  CTYPE                    01/29/99 14:26:30  Page  1
```

Figure 3–2: Print file page header

3.4.2.3 COMPILATION HEADING

The compilation heading is on the first page of the print file. The compilation heading gives the name of the object module, the pathname of the object module file, and the compiler controls specified in the compiler invocation. The heading also identifies the compiler version and host system. Figure 3–3 shows a compilation heading produced by the C196 compiler running on Windows 95.

```
80C196 C compiler vx.y rz SN00000000-004  (c) year TASKING, Inc.
Object module placed in ctype.obj
Compiler invoked by: c:\c196\bin\c196.exe ctype.c sb co le lc xr
```

Figure 3–3: Print file compilation heading

3.4.2.4 SOURCE TEXT LISTING

The source text listing contains a formatted image of the source text, as shown in Figure 3–4. This listing also gives the line number, block nesting level, and include nesting level of each statement in the source text. If a source line is too long to fit on one line, it continues on as many following lines as are needed. Continued lines start with a hyphen (–).

```
Line   Level  Incl
1             #include <ctype.h>
1      1      /* ctype.h
.
.
.
58     1
59     1      #endif /* _ctypeh */
2
3             unsigned char upcx(unsigned char input)
4             {
5      1      if (isascii(input) && input >= 'a' && input <= 'f')
+             if (((unsigned)(input) < 0x80) && input >= 'a' && input <= 'f')
6      1          return input - ('a' - 'A');
7      1      else
8      1          return input;
9      1      }
```

Figure 3–4: Print file source text listing

Line numbers range from 1 to 99999. Each error, warning, and remark message, when present, refers to the line numbers in the source text listing. Line numbers do not always correspond to the sequence of lines in the source text: source text lines that end in a backslash (\) are continued on the following line. The listing's line numbers are not incremented for continuation lines.

The block nesting level ranges from 0, for a statement outside of any function definition, loop, or other control block, to 99. When its value is 0, this field is blank.

The include nesting level describes how many `#include` preprocessor directives or instances of the `include` control the preprocessor encountered to include the statement in the source text. For the primary source file, the nesting depth is 0, and this field is blank. Each nested `#include` preprocessor directive or `include` control increments the include nesting level. The include nesting level column has a value only if the `listinclude` control is in effect. The maximum nesting of include files depends on the number of files open simultaneously during compilation and can vary with the operating system. Table 3–5 shows the compiler controls that affect the source text listing portion of the print file.

Control	Effect
[no]cond	generates or suppresses uncompiled conditional code.
diagnostic	determines class of messages that appear.
[no]list	generates or suppresses source text listing.
[no]listexpand	generates or suppresses macro expansion listing.
[no]listinclude	generates or suppresses text of include files.

Table 3–5: Controls that affect the source text listing

3.4.2.5 REMARKS, WARNINGS, AND ERRORS

Compiler messages indicate fatal errors, errors, warnings, and remarks. The compiler prints each message referring to syntax, such as a misplaced keyword, on a separate line immediately following the offending statement. All messages referring to semantics, such as too many register variables, appear at the end of the source text listing. If the offending statement is not printed, the compiler prints the messages in the listing as the compiler generates them. To suppress the generation of remarks and warning messages, use the `diagnostic` control. Figure 3–5 shows a syntax error message.

See Chapter 9 for a complete list of error messages generated by the compiler.

```
 Line Level  Incl
    1             #include ctype.h
*** Error at line 1 of ctype_x.c: illegal syntax in a directive line
    2
    3       unsigned char upcx(unsigned char input)
    4       {
    5  1      if (isascii(input) && input >= 'a' && input <= 'f')
    6  1          return input - ('a' - 'A');
    7  1      else
    8  1          return input;
    9  1      }
```

Figure 3–5: Print file source text listing with error message

3.4.2.6 SYMBOL TABLE AND CROSS-REFERENCE

The symbol table lists all objects and their attributes from the compiled code. The table includes the name, type, size, and address of each object. The table can optionally include source text cross–reference information. The compiler generates the table in alphabetical order by identifier. A source module may declare a particular identifier more than once, but each object, even if named by a duplicate identifier, appears as a separate entry in the symbol table.

The symbol table shown in Figure 3–6 contains cross–reference information in the `ATTRIBUTES` column. The cross–reference numbers for each symbol are line numbers containing references to the symbol. The line number marked with an asterisk (*) declares the symbol. Use the `[no]symbols` control to generate or suppress the symbol table. Use the `xref` control to add cross–reference information to the symbol table.

```
C196  Compiler   CTYPE                          01/29/99 14:26:30  Page  3
                      Symbol Table
Name   Size   Class  Address  Attributes

input  1      Auto   0        overlayable register unsigned char in
                              function(upcx)
                              *3, 5, 5, 5, 6, 8
upcx          Public          reentrant VPL function returning unsigned char
                              3
```

Figure 3–6: Print file cross–referenced symbol table

3.4.2.7 PSEUDO-ASSEMBLY LISTING

The pseudo–assembly listing, shown in Figure 3–7, is an assembly language equivalent to the object code produced in compilation. The listing shows the object code produced by the compiler and is useful for noticing program variations, such as those that result from changing optimization levels. The assembler cannot assemble the pseudo–assembly language listing because it is not a complete program. The generated pseudo–assembly language lacks the proper assembly directives to define the module and the variables used inside the program. This listing contains a location counter, a source line number, and the equivalent assembly code. The location counter is a hexadecimal value that represents an offset address relative to the start of the object code. Use the `[no]code` control to generate or suppress the pseudo–assembly listing.

```
C196  Compiler   CTYPE                 01/29/99 14:26:30  Page  4
            Assembly Listing of Object Code
                                          cseg
                                          ;   Statement     4
      0000                             upcx:
      0000  C800               R         push   ?OVRBASE
      0002  B3180400           R         ldb    input,4[SP]
                                          ;   Statement     5
      0006  AC001C             R         ldbze  Tmp0,input
      0009  8980001C                     cmp    Tmp0,#80H
      000D  DB10                         bc     @0002
      000F  996100             R         cmpb   input,#61H
      0012  D30B                         bnc    @0002
      0014  996600             R         cmpb   input,#66H
      0017  D906                         bh   @0002
                                          ;   Statement     6
      0019  5920001C           R         subb   Tmp0,input,#20H
      001D  2005                         br     @0001
                                          ;   Statement     7
      001F                           @0002:
                                          ;   Statement     8
      001F  B0001C             R         ldb    Tmp0,input
      0022  2000                         br     @0001
                                          ;   Statement     9
      0024                           @0001:
      0024  CC00               R         pop    ?OVRBASE
      0026  F0                           ret
            ; Function Statistics for: upcx
            ;    Code Size :  39     Parameter Count:   1
            ;    Stack Size:   0     Parameter Size :   2
            ;    OReg Size :   1     Stack Depth    :   2
                                          end
```

Figure 3–7: Print file pseudo–assembly listing

3.4.2.8 COMPILATION SUMMARY

The final line of the compilation summary in the print file is identical to the sign–off message displayed on the screen when the compilation is complete. Before this final line, the compiler lists information about the compiled object module. Figure 3–8 shows a compilation summary from a successful compilation. If the compilation ends with a fatal error, the following line replaces the normal compilation summary:

```
COMPILATION TERMINATED
```

```
Module Information:

     Code Area Size              =   0027H        39D
     Constant Area Size          =   0000H         0D
     Data Area Size              =   0000H         0D
     Static Regs Area Size       =   0000H         0D
     Overlayable Regs Area Size  =   0002H         2D
     Maximum Stack Size          =   0002H         2D

C196 Compilation Complete.      0 Warnings,     0 Errors
```

Figure 3–8: Print file compilation summary

3.4.3 OBJECT FILE

The compiler produces an object file by default, as shown in Table 3–3. The object file contains the relocatable code and data generated by the compiler as a result of a successful compilation. To suppress the object file, you must specify one of the following controls:

notranslate

`notranslate` stops compilation after preprocessing. The compiler can produce a preprint file but no print file. Use `notranslate` to find the effects on the source text of macro expansion, conditional compilation, and file inclusion, without a full compilation.

noobject

`noobject` suppresses the object file, although compilation completes. The compiler can produce both a preprint file and a print file. Use `noobject` to find statement numbers and scope information, any diagnostic messages, symbolic information, and the size of the compiled object code without generating a new object file.

The default name for the object file is the same as the primary source file name with the `.obj` extension substituted for the original extension, as shown in Table 3–3. The compiler places the object file in the directory containing the source file. To override the defaults, use the `object` control. If a file with the same name already exists, the compiler writes over it.

3.5 AUTOMATICALLY INVOKING THE C196 COMPILER

TASKING offers two ways of automatically invoking a series of commands: makefiles, batch files, shell scripts, command procedure files. This section describes how to use these files to automatically invoke the C196 compiler.

3.5.1 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 Chapter *MK196 Make Utility* of the *80C196 Utilities User's Guide*.

3.5.2 USING BATCH FILES

Batch files are a facility within DOS whereby one or more commands can be executed from within a file.

Assume that the following sequence of calls is frequently used:

```
c196 ifile.c
rl196 -f projfile.ltx
```

The files *ifile* and *projfile* may vary from one call to the next. To reduce the number of calls you can make a batch file, for example, `proj.bat`. Note that whatever the batch file is called it must end with the file extension `.bat`. The file should contain:

```
c196 %1.c
rl196 -f %2.ltx
```

On invocation `%1` and `%2` will be replaced by the first and second parameters after the batch file name. Using the name mentioned above (`proj` – note that the file extension `.bat` is not needed for invocation) the call becomes:

```
proj ifile projfile
```

DOS will return on the screen the actual command line executed, with all the parameters expanded to the values used.

3.5.3 USING UNIX SCRIPTS

Scripts are a facility within UNIX whereby one or more commands can be executed from within a file.

Assume that the following sequence of calls is frequently used:

```
c196 ifile.c
rl196 –f projfile.ltx
```

The files *ifile* and *projfile* may vary from one call to the next. To reduce the number of calls you can make a script, for example, `proj`. The file should contain:

```
c196 $1.c
rl196 –f $2.ltx
```

On invocation `$1` and `$2` will be replaced by the first and second parameters after the script file name. Using the name mentioned above (`proj`) the call becomes:

```
sh proj ifile projfile
```

3.6 DEVELOPING A C196 APPLICATION PROGRAM

The C196 compiler supports modular, structured development of applications. You can compile and debug application modules separately, then link them together to create an executable file. Use the RL196 linker and locator utility to combine separately translated object modules into a single program and assign absolute addresses to all relocatable addresses. Use the LIB196 utility to place an object module into a library for later combination, using RL196, into a program. Use the OH196 utility to convert an object module into the standard hexadecimal format that can be loaded into PROM.

Your C196 application programs can contain many separately translated modules. The applications can call functions from a library. The C196 product includes several libraries. You can also create your own libraries using the LIB196 utility.

To create a complete program, the RL196 utility must link all translated code and libraries together. Selecting the correct libraries for linking depends on whether the program does any of the following:

- uses floating–point numbers.

- uses either the `printf` or the `sprintf` function to write floating–point formatted output.
- uses either the `scanf` or the `sscanf` function to read floating–point formatted input.
- uses the `model(nt)` or `farcode` controls.

Figure 3–9 shows how to specify libraries and object files in the correct order for linking with C196 compiled modules.

The following is an example RL196 invocation:

```
rl196 cstart.obj, newmod.obj, my_utils.lib, c96fp.lib,
      c96.lib, fpal96.lib to newmod.out
```

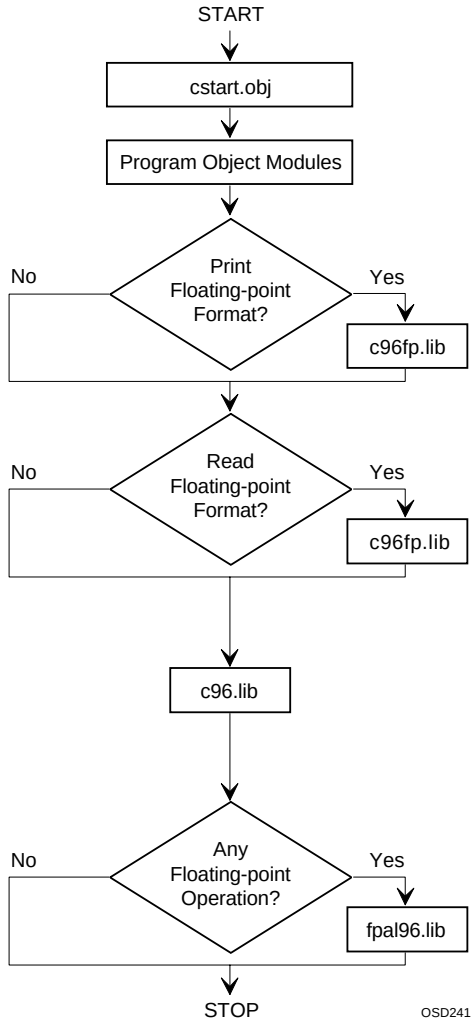


Figure 3–9: Choosing libraries and object files for linking

3.7 COMBINING DIFFERENT OMF96 FORMATS

As of version 5.0, you can specify three different OMF96 formats for the 196 tools. See the `omf` control on how to specify a specific OMF96 format. By default our tools use the OMF96 version 3.2 format. This format contains extra debugging info and support for using initialized global variables. We recommend that you use this default OMF96 format. You can specify the different OMF96 formats with the `omf` control:

`omf(2)` OMF96 version 3.2
`omf(1)` OMF96 version 3.0
`omf(0)` OMF96 version 2.0

3.7.1 GLOBAL INITIALIZATION

It is necessary to use OMF96 version 3.2 if you want to use global initialization. However, it is possible to create an `.abs` file which is OMF 3.0 compatible, but still contains global initialization. This might be necessary for certain third party tools which do not (yet) recognize the new OMF96 format. To do so, you have to use the (default) `omf(2)` for both the compiler and the assembler, and use `omf(1)` for the linker. The resulting `.abs` file has the OMF96 version 3.0 format, but contains all necessary code for global initialization. The same is true for using the libraries. The libraries provided with our tools are compiled with the default `omf(2)` control. If you want to get an OMF96 version 3.0 compatible `.abs` file, just specify `omf(1)` in the linker controls and you can use our default libraries.

A word of caution: if you specify `omf(1)` in your linker controls and if you have any unresolved externals in your application, it is possible that the linker will give a fatal OMF96 error. This is caused by the fact that you have specified OMF96 version 3.0, but the linker needs to write information about the unresolved externals in OMF96 version 3.2 format. You will see a warning about the unresolved externals before you get the fatal omf error. So, do not have any unresolved externals when you convert from OMF96 3.2 format to OMF96 3.0 format.

3.7.2 OMF96 VERSION 3.0 LIMITATIONS

OMF96 version 3.0 has the following limitations compared to OMF96 version 3.2:

- Limited support for functions.
- Limited support for structures.
- Limited support for unions.
- Limited support for bit fields.
- No support for vertical windowing.
- Restricted line number information.

3.8 EXAMPLES

This section contains an example C196 program showing how to use compiler controls and environment variables for:

- compiling interrupt functions.
- finding include and header files in directories other than the current directory.
- debugging the preprocessor directives.
- specifying print file content.
- preparing the object code for symbolic debugging.
- optimizing the object code.

This example also demonstrates the use of the 80C196 processor's special function registers (SFRs) and header file.

3.8.1 SOURCE TEXT

The program used in this example is a digital filter application designed to run on the 80C196KB processor. This program demonstrates the use of the following variables defined in the `kb_sfrs.h` header file for accessing the special function registers (SFRs):

`ad_command` is the command register for the analog to digital converter.

`hso_command`
is the command register for the high–speed output.

`hso_time` is the timer for the high–speed output.

`timer1` is one of the hardware timers.

`ad_result_hi`
is the high–order byte of the analog to digital converter output.

`ad_result_lo`
is the low–order byte of the analog to digital converter output.

`ioport0` is one of the input/output (I/O) ports.

`ioport1` is one of the input/output (I/O) ports.

Figures 3–10 through 3–12 list the contents of the primary source text file, including comments explaining how the program works. In this example, the initialization routine is in `dsfinit.h` and the interrupt and main routines are in `dsf.c`, both in the directory `/project/working`.

The material contained in this chapter on the Digital Filter application is based on Experiment 9–17 of the book *The 16–Bit 8096: Programming, Interfacing, Applications–– 122 Experiments* by R. Katz and H. Boyet (permission granted by the publisher, H. Boyet, 14 E. 8th St., NY, NY).

```
/*****************************************************************************/
/*                                                                           */
/*      This is the initialization code for the digital signal filter.       */
/*                                                                           */
/*****************************************************************************/

#pragma model(kb)   /* Select instruction set for the 80C196KB processor.  */
#pragma interrupt (software_timer = 5)
#pragma interrupt (analog_conversion_done = 1)

#include <kb_sfrs.h> /* Include header file that declares variables to     */
                     /* access analog to digital (A/D) channel, high speed */
                     /* output (HSO), software timer 0, input/output (I/O) */
                     /* ports, and interrupt flags.  This header file also */
                     /* declares enable() to enable interrupts.            */

#define K (unsigned char) 0x71           /* Scaling factor for input    */
#define M (unsigned char) 0x71           /* Scaling factor for history  */
#define MSB (unsigned char) 0x80         /* Mask for low byte of input  */
#define Full_Scale (unsigned char) 0x71  /* Mask for high byte of input */

register unsigned char input;
register unsigned int word_value;

#define byte_value (*((unsigned char *) &word_value + 1))
     /* byte_value is the high-order bit of word_value */
```

```
#define high_byte(x) (*((unsigned char *) & x + 1))

/*****************************************************************************/
```

Figure 3–10: Digital filter source text (initialization) in dsfinit.h

```
/*****************************************************************************/
/*                                                                           */
/*     This program implements a digital signal filter with the following    */
/*     equation:    V(new) = K * input + M * V(old)                          */
/*                                                                           */
/*****************************************************************************/

/*****************************************************************************/
/*                                                                           */
/* software_timer is the interrupt routine for HSO software timer 0:         */
/*                                                                           */
/*****************************************************************************/

void software_timer(void)
{
  ad_command = 8;          /* Start A/D channel 0.                          */
  hso_command = 0x18;      /* Give command to HSO.                          */
  hso_time = timer1 + 64;  /* Set HSO command to occur in 126 microseconds.*/
}

/*****************************************************************************/

/*****************************************************************************/
/*                                                                           */
/* analog_conversion_done is the interrupt routine for A-to-D conversion.    */
/*                                                                           */
/*****************************************************************************/

void analog_conversion_done(void)
{
  input = ad_result_hi;
  if ((ad_result_lo & MSB)  && (ad_result_hi != Full_Scale))
    input++;
  if (ioport0 & MSB)  /* Bypass filter if P0.7 is set. */
    ioport1 = input;
  else
    {
      word_value = input * K + ioport1 * M;
      ioport1 = byte_value;
    }
}

/*****************************************************************************/
```

Figure 3–11: Digital filter source text (interrupt routines) in dsf.c

```
/*************************************************************************/
/*                                                                       */
/* main initializes the system and clears flags between interrupts.      */
/*                                                                       */
/*************************************************************************/

main()
{
  ioport1 = 0;             /* Initialize A/D converter.                  */
  int_mask = 0x22;         /* Enable software timer and A/D conversion   */
                           /*   interrupt routine.                       */
  int_pending = 0;         /* Clear any pending interrupts.              */
  hso_command = 0x18;      /* Give command to HSO.                       */
  hso_time = timer1 + 64;  /* Set HSO command to occur in 126 microseconds.*/

  enable();                /* Enable interrupts.                         */
  while(1);                /* Loop forever, waiting for interrupt.       */
}

/*************************************************************************/
```

Figure 3–12: Digital filter source text (main routine) in dsf.c

3.8.2 SETTING THE WINDOWS ENVIRONMENT

The directory structure of this example is as follows:

- The `c:\c196\bin` directory contains the `c196.exe` compiler.
- The `c:\c196\include` directory contains the `kb_sfrs.h` include file.
- The `c:\project\working` directory contains the following files:
  - The `dsf.c` source text file, used as the primary source file in this example. This file contains the source text shown in Figures 3–11 and 3–12.
  - The `dsfinit.h` source text file, to be included at the beginning of the primary source text by the default initialization file. This file contains the source text shown in Figure 3–10.
  - The `c96init.h` default initialization file, containing the following preprocessor directives used for all compilations of this example:

```
#include <dsfinit.h>
#pragma pagelength(30)
#pragma pagewidth(72)
```

Specify these directories by giving the following DOS commands before invoking the compiler:

```
set C196INC=c:\c196\include;c:\project\working
set C96INIT=c:\project\working
```

Setting the `c196inc` environment variable provides the default search path prefixes that the compiler uses for include files in all subsequent compilations. Setting the `c96init` environment variable provides the path prefix that the compiler uses for the `c96init.h` initialization file in all subsequent compilations. You need to set these environment variables only once each time you reset your host system before compiling the program.

3.8.3 PREPROCESSING

Before compiling the source text into object code, you can check the preprocessing performed in your program to verify all your macro expansions and conditional compilation expressions. Macro expansion, file inclusion, and conditional compilation are all shown in the preprint file. Diagnostic messages appear on your console, not in the preprint file, and you can redirect these messages to a log file. To generate a preprint file without generating object code, redirecting any messages to `dsf_pre.log`, specify the `preprint` and `notranslate` controls, as follows:

```
C:> c196 dsf.c preprint notranslate > dsf_pre.log
```

`#define` macro definitions do not appear in the preprint file. The compiler substitutes the body of the macro wherever the macro name appears in the source text.

Since no errors occurred during preprocessing, `dsf_pre.log` contains only the following sign–on message:

```
80C196 C compiler vx.y rz SN000000-004 (c) year TASKING, Inc.
(C)1980,1990,1992,1993 Intel Corporation
```

When errors occur during preprocessing, `dsf_pre.log` contains, in addition to the sign–on message, lines such as the following:

```
*** Error at line 12 of c:\project\working\dsfinit.h:
illegal constant expression
```

Figure 3–13 shows the resulting `dsf.i` preprint file. Compiling this file has the same result as compiling the `dsf.c` file.

```
#line 1 "c96init.h"

#line 1 "dsfinit.h"

/*************************************************************************/
/*                                                                       */
/*     This is the initialization code for the digital signal filter.    */
/*                                                                       */
/*************************************************************************/

#pragma model(kb)   /* Select instruction set for the 80C196KB processor. */
#pragma interrupt(software_timer = 5)
#pragma interrupt(analog_conversion_done = 1)

#line 1 "c:/c196/kb_sfrs.h"
/* kb_sfrs.h
 *
 * kb_sfrs.h  - declarations for 80C196 SFRs (a superset of
 *              8096 registers) and 80C196-specific library
 *              function declarations
 */

extern volatile register unsigned short r0;         /* at 0x00: r   */
extern volatile register unsigned char ad_command;  /* at 0x02:   w */
extern volatile register unsigned char ad_result_lo;/* at 0x02: r   */
extern volatile register unsigned char ad_result_hi;/* at 0x03: r   */
extern volatile register unsigned char hsi_mode;    /* at 0x03:   w */
extern volatile register unsigned short hso_time;   /* at 0x04:   w */
extern volatile register unsigned short hsi_time;   /* at 0x04: r   */
extern volatile register unsigned char hso_command; /* at 0x06:   w */
extern volatile register unsigned char hsi_status;  /* at 0x06: r   */
extern volatile register unsigned char sbuf;        /* at 0x07: r/w */
extern volatile register unsigned char int_mask;    /* at 0x08: r/w */
extern volatile register unsigned char int_pending; /* at 0x09: r/w */
extern volatile register unsigned char watchdog;    /* at 0x0a:   w */
extern volatile register unsigned short timer1;     /* at 0x0a: r   */
extern volatile register unsigned short timer2;     /* at 0x0c: r   */
extern volatile register unsigned char baud_rate;   /* at 0x0e:   w */
extern volatile register unsigned char ioport0;     /* at 0x0e: r   */
extern volatile register unsigned char ioport1;     /* at 0x0f: r/w */
extern volatile register unsigned char ioport2;     /* at 0x10: r/w */
extern volatile register unsigned char sp_con;      /* at 0x11:   w */
extern volatile register unsigned char sp_stat;     /* at 0x11: r   */
extern volatile register unsigned char ioc0;        /* at 0x15:   w */
extern volatile register unsigned char ios0;        /* at 0x15: r   */
extern volatile register unsigned char ioc1;        /* at 0x16:   w */
extern volatile register unsigned char ios1;        /* at 0x16: r   */
extern volatile register unsigned char pwm_control; /* at 0x17:   w */
```

Figure 3–13: Digital signal filter preprint file

```
/*****************************************************************************/
/*                                                                           */
/* Additional SFRs of the 80C196KB                                           */
/*                                                                           */
/*****************************************************************************/

extern volatile register unsigned char ioc2;        /* at 0x0b:   w */
extern volatile register unsigned char ipend1;      /* at 0x12: r/w */
extern volatile register unsigned char imask1;      /* at 0x13: r/w */
extern volatile register unsigned char wsr;         /* at 0x14: r/w */
extern volatile register unsigned char ios2;        /* at 0x17: r   */

/*****************************************************************************/
/*                                                                           */
/* Additional SFRs of the 80C196KC                                           */
/*                                                                           */
/*****************************************************************************/

extern volatile register unsigned char  ad_time;     /* at 0x03: r/w */
extern volatile register unsigned short ptssel;      /* at 0x04: r/w */
extern volatile register unsigned short ptssrv;      /* at 0x06: r/w */
extern volatile register unsigned char  t2control;   /* at 0x0c: r/w */
extern volatile register unsigned char  pwm1_control;/* at 0x16: r/w */
extern volatile register unsigned char  pwm2_control;/* at 0x17: r/w */

/*****************************************************************************/
/* Define typedefs for PTS Control Blocks of 80C196KC.                       */
/*****************************************************************************/

/*
 * Single Transfer PTS Control Block
 */
typedef struct STran_ptscb_t {
                            unsigned char  ptscount;
                            struct {
                                   unsigned int  di : 1,
                                                 si : 1,
                                                 du : 1,
                                                 su : 1,
                                                b_w : 1,
                                               mode : 3;
                                   } ptscon;
                            void  *ptssrc;
                            void  *ptsdst;
                            int   :16; /* unused */
                            } STran_ptscb;
```

Figure 3–13: Digital signal filter preprint file (continued)

```
/*
 * Block Transfer PTS Control Block
 */
typedef struct BTran_ptscb_t {
                            unsigned char  ptscount;
                            struct {
                                   unsigned int  di : 1,
                                                 si : 1,
                                                 du : 1,
                                                 su : 1,
                                                b_w : 1,
                                               mode : 3;
                                   } ptscon;
                            void  *ptssrc;
                            void  *ptsdst;
                            unsigned char  ptsblock;
                            int   :8; /* unused */
                            } BTran_ptscb;


/*
 * A/D Mode PTS Control Block
 */
typedef struct AD_ptscb_t {
                           unsigned char  ptscount;
                           struct {
                                  unsigned int  const1 : 3,
                                                updt   : 1,
                                                const2 : 1,
                                                mode   : 3;
                                  } ptscon;
                           unsigned int  s_d;
                           unsigned int  reg;
                           int   :16; /* unused */
                           } AD_ptscb;

/*
 *  HSI Mode PTS Control Block
 */
typedef struct HSI_ptscb_t {
                           unsigned char  ptscount;
                           struct {
                                  unsigned int  const1 : 3,
                                                updt   : 1,
                                                const2 : 1,
                                                mode   : 3;
                                  } ptscon;
                           unsigned int  :16;
                           unsigned int  ptsdst;
                           unsigned char ptsblock;
                           int   : 8; /* unused */
                          } HSI_ptscb;
```

Figure 3–13: Digital signal filter preprint file (continued)

```
/*
 * HSO Mode PTS Control Block
 */
typedef struct HSO_ptscb_t {
                          unsigned char  ptscount;
                          struct {
                                 unsigned int  const1 : 3,
                                               updt   : 1,
                                               const2 : 1,
                                               mode   : 3;
                                 } ptscon;
                          unsigned int  ptssrc;
                          unsigned int  :16;
                          unsigned char ptsblock;
                          int   : 8; /* unused */
                         } HSO_ptscb;

/*
 *  PTS A/D Table
 */
typedef struct AD_tab_t {
                         unsigned char  AD_command;
                         unsigned int   AD_result;
                        } AD_tab;

/*
 *  PTS HSI Table
 */
typedef struct HSI_tab_t {
                         unsigned char  HSI_status_lo;
                         unsigned char  HSI_status_hi;
                         unsigned int   HSI_time;
                        } HSI_tab;

/*
 *  PTS HSO Table
 */
typedef struct HSO_tab_t {
                         unsigned char  HSO_command;
                         unsigned int   HSO_time;
                        } HSO_tab;

/*************************************************************************/
/*                                                                       */
/* Additional C96.LIB functions supported by the 80C196 only             */
/*                                                                       */
/*************************************************************************/

void enable(void);
void disable(void);
```

Figure 3–13: Digital signal filter preprint file (continued)

```
void power_down(void);
void idle(void);

void enable_pts(void);
void disable_pts(void);

#line 12 "dsfinit.h"
             /* access analog-to-digital (A/D) channel, high speed */
             /* output (HSO), software timer 0, input/output (I/O) */
             /* ports, and interrupt flags.  This header file also */
             /* declares enable() to enable interrupts.            */

register unsigned char input;
register unsigned int word_value;

     /* byte_value is the high order bit of word_value */

/*****************************************************************************/

#line 2 "c96init.h"
#pragma pagelength(30)
#pragma pagewidth(72)

#line 1 "dsf.c"
/*****************************************************************************/
/*                                                                           */
/*   This program implements a digital signal filter with the following   */
/*   equation:    V(new) = K * input + M * V(old)                           */
/*                                                                           */
/*****************************************************************************/

/*****************************************************************************/
/*                                                                           */
/* software_timer is the interrupt routine for HSO software timer 0.      */
/*                                                                           */
/*****************************************************************************/

void software_timer(void)
{
    ad_command = 8;         /* Start A/D channel 0 */
    hso_command = 0x18;     /* Give command to HSO */
    hso_time = timer1 + 64; /* Set HSO command to occur in 126 microsecs. */
}

/*****************************************************************************/

/*****************************************************************************/
/*                                                                           */
/* analog_conversion_done is the interrupt routine for A-to-D conversion. */
/*                                                                           */
/*****************************************************************************/
```

Figure 3–13: Digital signal filter preprint file (continued)

```
void analog_conversion_done(void)
{
    input = ad_result_hi;
    if (ad_result_lo & (unsigned char)0x80 && ad_result_hi != (unsigned
                                                    char)0x71)
        ++input;
    if (ioport0 & (unsigned char)0x80)    /* Bypass filter if P0.7 is set */
        ioport1 = input;
    else
    {
        word_value = input * (unsigned char)0x71 + ioport1 * (unsigned
                                                    char)0x71;
        ioport1 = (*((unsigned char *)&word_value + 1));
    }
}

/*****************************************************************************/

/*****************************************************************************/
/*                                                                           */
/* main initializes the system and clears flags between interrupts.          */
/*                                                                           */
/*****************************************************************************/

void main(void)
{
    ioport1 = 0;            /* Initialize A/D convertor */
    int_mask = 0x22;        /* Enable software timer and A/D conversion */
                            /*     interrupt routine */
    int_pending = 0;        /* Clear any pending interrupts */
    hso_command = 0x18;     /* Give command to HSO */
    hso_time = timer1 * 64; /* Set HSO command to occur in 126 microsecs. */

    enable();               /* Enable interrupts */
    while (1) ;             /* Loop forever, waiting for interrupt */
}

/*****************************************************************************/
```

Figure 3–13: Digital signal filter preprint file (continued)

3.8.4 CHECKING SYNTAX AND SEMANTICS

You can check your source text for syntax and semantic errors without generating an object file. To generate a print file containing information about the compilation without generating any object code, use the `noobject` control. The same source text listed in the preprint file can be listed in the print file, with additional diagnostic messages that result from the translation. You can also generate a cross–referenced symbol table to verify the symbols defined and referenced in the program.

To generate a print file containing a cross–referenced symbol table such as the one shown in Figure 3–14, invoke the compiler as follows:

```
C:> c196 dsf.c noobject listexpand listinclude xref
```

The `listexpand` and `listinclude` controls expand macros and list include files, respectively, in the source text listing. The `xref` control generates the cross–referenced symbol table.

```
C196  Compiler   DSF                        01/29/99 15:14:37  Page   6
                                Symbol Table

 Name             Size    Class  Address Attributes

 ad_command       1       Extern         register volatile unsigned char
                                         *13, 17
 ad_result_hi     1       Extern         register volatile unsigned char
                                         *15, 33, 34
 ad_result_lo     1       Extern         register volatile unsigned char
                                         *14, 34
 analog_conversion_done
                          Public         interrupt function returning void
                                         *9
 enable                   Extern         VPL function returning void
                                         *193, 63
 hso_command      1       Extern         register volatile unsigned char
                                         *19, 18, 60
 hso_time         2       Extern         register volatile unsigned short
                                         *17, 19, 61
 input            1       Public   2     register unsigned char
                                         *23, 33, 35, 37, 40

                                Symbol Table

 int_mask         1       Extern         register volatile unsigned char
                                         *22, 57
 int_pending      1       Extern         register volatile unsigned char
                                         *23, 59
 ioport0          1       Extern         register volatile unsigned char
                                         *28, 36
 ioport1          1       Extern         register volatile unsigned char
                                         *29, 37, 40, 41, 56
 main                     Public         reentrant VPL function returning void
                                         *54
 software_timer           Public         interrupt function returning void
 timer1           2       Extern         register volatile unsigned short
                                         *25, 19, 61
 word_value       2       Public   0     register unsigned int
                                         *24, 40, 41
```

Figure 3–14: Digital signal filter symbol table

3.8.5 SYMBOLIC DEBUGGING

You can configure the object code for type checking and symbolic debugging and you can list the generated code in a format similar to ASM196 source text in the print file. By default, the compiler puts symbolic information for type checking in the object code. The `debug` control generates additional symbolic information for symbolic debugging by in–circuit emulators.

A useful feature of symbolic debuggers is the ability to list the line of source text corresponding to the instruction being executed. However, the optimization that occurs at optimization levels 2 and 3 can rearrange or eliminate code resulting from specific source statements. To ensure that the debugger correctly matches the source text and object code, use the `optimize(0)` control.

The `code` control generates the pseudo–assembly language (code) listing. Figure 3–15 contains the code listing generated by the following compiler invocation:

```
C:> c196 dsf.c debug code optimize(0)
```

```
C196  Compiler   DSF                        01/29/99 15:14:37  Page  13
                  Assembly Listing of Object Code

                                             cseg
                                             ;   Statement    16
          0000                            software_timer:
          0000  F4                           pusha
                                             ;   Statement    17
          0001  B10800             E         ldb    ad_command,#8
                                             ;   Statement    18
          0004  B11800             E         ldb    hso_command,#18H
                                             ;   Statement    19
          0007  4540000000         E         add    hso_time,timer1,#40H
                                             ;   Statement    20
          000C  F5                           popa
          000D  F0                           ret
                                             ;   Statement    32
          000E                            analog_conversion_done:
          000E  F4                           pusha
          000F  C81C                         push   Tmp0
          0011  C81E                         push   Tmp2
                                             ;   Statement    33
          0013  B00002             E         ldb    input,ad_result_hi
                                             ;   Statement    34
          0016  5180001C           E         andb   Tmp0,ad_result_lo,#8
                                -       0H
```

Figure 3–15: Digital signal filter code listing (level 0 optimization)

```
            001A  981C00                        cmpb   R0,Tmp0
            001D  DF07                          be     @0003
            001F  997100            E           cmpb   ad_result_hi,#71H
            0022  DF02                          be     @0003
                                                ;   Statement    35
            0024  1702              R           incb   input
            0026                       @0003:
                                                ;   Statement    36
            0026  5180001C          E           andb   Tmp0,ioport0,#80H
            002A  981C00                        cmpb   R0,Tmp0
            002D  DF05                          be     @0004
                                                ;   Statement    37
            002F  B00200            E           ldb    ioport1,input
                                                ;   Statement    38
            0032  200F                          br     @0005
            0034                       @0004:
                                                ;   Statement    40
            0034  5D71021C          R           mulub  Tmp0,input,#71H

C196  Compiler   DSF                        01/29/99 15:14:37  Page  14
                  Assembly Listing of Object Code

            0038  5D71001E          E           mulub  Tmp2,ioport1,#71H
            003C  441E1C00          R           add    word_value,Tmp0,Tmp2
                                                ;   Statement    41
            0040  B00100            E           ldb    ioport1,word_value+1
            0043                       @0005:
                                                ;   Statement    43
            0043  CC1E                          pop    Tmp2
            0045  CC1C                          pop    Tmp0
            0047  F5                            popa
            0048  F0                            ret
                                                ;   Statement    55
            0049                       main:
                                                ;   Statement    56
            0049  1100              E           clrb   ioport1
                                                ;   Statement    57
            004B  B12200            E           ldb    int_mask,#22H
                                                ;   Statement    59
            004E  1100              E           clrb   int_pending
                                                ;   Statement    60
            0050  B11800            E           ldb    hso_command,#18H
                                                ;   Statement    61
            0053  A0001C            E           ld     Tmp0,timer1
            0056  09061C                        shl    Tmp0,#6
            0059  A01C00            E           ld     hso_time,Tmp0
                                                ;   Statement    63
            005C  EF0000            E           call   enable
                                                ;   Statement    64
            005F  2000                          br     @0008
```

Figure 3–15: Digital signal filter code listing (level 0 optimization) (continued);

```
0061                            @0007:
0061                            @0008:
0061  27FE                              br      @0007
0063                            @0009:
                                        ;   Statement    65
0063  F0                               ret
                                        end
```

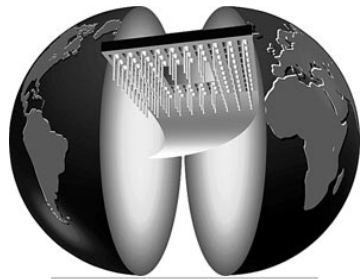
Figure 3–15: Digital signal filter code listing (level 0 optimization) (continued);

3.8.6 OPTIMIZING

When you have finished debugging, you can compile the program for both memory and execution efficiency. By specifying the `notype` control and not specifying the `debug` control, you can eliminate all symbolic information that is not needed for execution. By specifying the highest level of optimization, `optimize(3)`, you can reorganize the object code to occupy less space and to use the fewest instructions.

CHAPTER 4

COMPILER CONTROLS



TASKING
Quality Development Tools Worldwide

4 CHAPTER

This chapter describes the C196 compiler controls. Use compiler controls to specify options such as the location of source text files, the amount of debug information in the object module, and the format and location of the output listings. Since most of the controls have default settings, you need not specify any of the controls if the defaults match your application needs. Table 3–1 lists default settings and a brief description of each control.

The entries in this section describe in detail the syntax and function of each compiler control.

Square brackets ([ ]) enclose optional arguments for controls. If you do not specify optional arguments for a particular control, do not use an empty pair of parentheses either.

Some controls use an optional list of arguments. Separate multiple argument definitions with commas. Brackets surrounding a comma and an ellipsis ([,...]) indicate an optional list.

See the *Conventions Used In This Manual*, listed 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.

abszero

Function

Specifies whether the compiler zeroes uninitialized variables in absolute segments.

Syntax

Select the `EDE | C Compiler Options | Project Options...` menu item. Enable or disable the `Clear unititialized RAM variables in absolute segments` check box in the `Code` tab.

`abszero | noabszero`

Abbreviation

`az | noaz`

Class

Primary control

Default

`noabszero`

Description

Use the `abszero` control/pragma enable the generation of zeroing entries in the initialization tables for absolute segments (variables positioned by either `#pragma locate` or the `_reg` storage class modifier). This control is only valid for the (default) OMF version 3.2.

Use the `noabszero` control/pragma to prevent the generation of zeroing entries in the initialization tables for absolute segments.

`noinit` forces `noabszero`.

`init`
`zero`

bmov

Function

Tells the compiler to use the `bmov` instruction to initialize or copy structures or array elements.

Syntax

Select the `EDE | C Compiler Options | Project Options...` menu item. Enable or disable the `Use the uniterruptable 'bmov' instead of 'bmovi'` check box in the `Object` tab.

`bmov | nobmov`

Abbreviation

`bm | nobm`

Class

Primary control

Default

`nobmov`

Description

Use this control to tell the compiler to use the `bmov` instruction when initializing or copying structures or array elements. This control is valid for all models. Use the `model()` control to specify the specific instruction set.

The compiler generally does not generate the `bmov` instruction because of its interrupt latency. The `bmov` instruction is uninterruptable. See the *Embedded Microcontrollers and Processors Handbook*, listed in *Related Publications*, for more information on the `bmov` and `ebmov` instruction.

Without the `bmov` control the compiler automatically generates the `bmovi` instruction for the same process. The `bmovi` instruction is interruptable.

You can specify the `bmov` control in the compiler invocation or in a `#pragma` preprocessor directive preceding the first line of data definition or executable source text.

`model`

case

Function

Tells compiler to act case sensitive.

Syntax

Select the `EDE | C Compiler Options | Project Options...` menu item. Enable or disable the `Operate in case sensitive mode` check box in the `Code` tab.

`case | nocase`

Abbreviation

`cs | nocs`

Class

Primary control

Default

`case`

Description

Use this control to tell the compiler 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:

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

ccb

Function

Specifies the initial chip configuration byte value.

Syntax

Select the `EDE | C Compiler Options | Options` *`file`*`...` menu item. Enter a single or multiple byte value in the `Specify Chip Configuration (one module only)` field in the `Object` tab.

`ccb(`*`value`*`)`

where:

`value` is a single or multiple byte value.

Abbreviation

`cc`

Class

Primary control

Description

Use this control to initialize the value of up to four chip configuration bytes (CCB), located at `2018H`, or at `0FF2018H`, `0FF201AH`, and `0FF201CH` for the 80C196NT. The 80C196 processor reads the CCB on reset to initialize the value of the chip configuration register (CCR). See the *Embedded Microcontrollers and Processors Handbook*, llisted in *Related Publications*, for a detailed explanation of the contents of the CCR.

To specify more than one byte, the value should be given as a hexadecimal string. The bytes specified will be placed in successive even addresses, as required by the processor, with a byte of `20H` automatically placed in the intervening odd addresses. For example, if you specified `ccb(0x010203)` the compiler would place `0120022003` in addresses `0FF2018H` through `0FF201CH`.

You can specify the `ccb` control in the compiler invocation or in a `#pragma` preprocessor directive preceding the first line of data definition or executable source text.

It is better to use `#pragma` than to use the control on invocation. Can only be used once in the whole application.

code

Function

Generates or suppresses pseudo–assembly language code listing in print file.

Syntax

Select the `EDE | C Compiler Options | Project Options...` menu item. Enable or disable the `Generate pseudo–assembly language` check box in the `Listing` tab.

`code | nocode`

Abbreviation

`co | noco`

Class

General control

Default

`nocode`

Description

Use this control to produce a pseudo–assembly language listing equivalent to the object code generated by the compiler. The compiler places this listing in the print file below the source text listing. Use the default `nocode` control to suppress the pseudo–assembly language listing.

You can use the pseudo–assembly language listing while debugging to view the following:

- The effects of different levels of optimization set by the `optimize` control.
- The difference in code the compiler generates under the various arguments to the `model` control.
- The differences in calling sequences the compiler generates under the `fixedparams`, `varparams`, `noreentrant`, and `reentrant` controls.

The `noprint` and `notranslate` controls suppress the pseudo–assembly language listing specified by the `code` control, but the `noobject` control does not.

You can specify these controls in the compiler invocation and in `#pragma` preprocessor directives throughout the source text. If the `code` or `nocode` control is embedded within the source text, the control only affects the source text that follows the control line until the compiler encounters the opposite control or the end of the source text.

`debug`	`object`	`reentrant`
`extend`	`optimize`	`translate`
`fixedparams`	`print`	`varparams`
`model`		

cond

Function

Includes or suppresses uncompiled conditional code in source text listing.

Syntax

Select the `EDE | C Compiler Options | Project Options...` menu item. Enable or disable the `Include conditionally uncompiled source code` check box in the `Listing` tab.

`cond | nocond`

Abbreviation

`cd | nocd`

Class

General control

Default

`nocond`

Description

Use this control to include in the program listing code that is not compiled because of conditional preprocessor directives. Use the default `nocond` control to suppress listing of source text eliminated by conditional compilation.

Whether you specify the `cond` control or not, the conditional preprocessor directive lines appear in the print file. They only affect the source text listing in the print file.

If you specify `notranslate` or `noprint`, the source text listing is completely suppressed and `cond` has no effect. Also, in any part of the source text listing suppressed by `nolist` or `nolistinclude`, the `cond` control has no effect.

You can specify these controls in the compiler invocation and in `#pragma` preprocessor directives throughout the source text. If the `cond` or `nocond` control is embedded within the source text, the control only affects the source text that follows the control line until the compiler encounters the opposite control or the end of the source text.

```
list                print
listinclude         translate
```

debug

Function

Includes or suppresses debug information in the object module.

Syntax

Select the `EDE | C Compiler Options | Project Options...` menu item. Enable or disable the `Generate symbolic debug information` check box in the `Debug` tab.

```
debug | nodebug
```

Abbreviation

```
db | nodb
```

Class

Primary control

Default

```
nodebug
```

Description

Use this control to place information used by symbolic debuggers, in the object module. Use the default `nodebug` control to suppress symbolic debug information. Suppressing symbolic debug information reduces the size of the object module.

If you specify the `noobject` or `notranslate` control, the compiler does not generate an object module and `debug` has no effect.

Choose one of the following combinations of controls to aid debugging:

`type debug`

for both type checking (by RL196) and symbolic debugging. RL196 also uses the debug information to produce link maps. This combination of controls includes all possible debug and type information in the object code.

`type nodebug`

for type checking by the linker. This combination of controls includes type definition information for external and public symbols only. You can use this combination to reduce the size of the object module when you are not using a symbolic debugger.

`notype nodebug`

to suppress all debug and type information. This combination reduces the size of the object module by omitting information not necessary for execution.

Use `optimize(0)` with `debug` when you use a symbolic debugger. Since higher levels of optimization can result in rearranged or eliminated object code, optimizing can reduce the ability of most symbolic debuggers to accurately correlate debug information to the source text. Even with `optimize(0)` however, some source statements may generate no code.

The predefined macro `_DEBUG_` indicates which of `type`, `notype`, `debug`, or `nodebug` have been specified, as in Table 4–1:

Debug and Type Controls	Value of _DEBUG_
notype nodebug	0
type nodebug	1
notype debug	2
type debug	3

Table 4–1: Values for the _DEBUG_ macro

The `debug` and `nodebug` controls affect the entire object module. You can specify either of these controls in the compiler invocation or in a `#pragma` preprocessor directive preceding the first line of data definition or executable source text. To override a `#pragma debug` or `#pragma nodebug` specified in the source text, specify the opposite control (`nodebug` or `debug`, respectively) in the compiler invocation.

```
object              translate
optimize            type
```

define

Function

Defines a macro

Syntax

Select the `EDE | C Compiler Options | Project Options...` menu item. Define a macro (syntax: *macro*[=*def*]) in the `Define user macros` field in the `Misc` tab. You can define more macros by separating them with commas.

`define(`*`name`* `[=` *`body`*`] [,...])`

where:

`name`	is the name of a macro.
`body`	is the text (that is, value) of the macro. If the body contains spaces or punctuation, surround the entire body with quotation marks (`"`).

Abbreviation

`df`

Class

Invocation

Default

`body` `= 1`

Description

Use this control to create object–like macros in the compiler invocation. The entire module (primary source file and all include files) is within the scope of a macro defined in the compiler invocation. The body of an object–like macro contains no formal parameters. Use the `#define` preprocessor directive in the source text instead of the `define` control for function–like macros. *C: A Reference Manual*, listed in *Related Publications*, describes the `#define` preprocessor directive.

If the definition contains no *body*, the macro expands to the value `1`. The default value of a macro defined with the `define` control differs from that of a macro defined with the `#define` preprocessor directive. A macro defined without a body using `#define` has no value and expands to nothing, although a test for existence of the macro returns a true value.

If you remove the macro definition with a `#undef` preprocessor directive, the macro is no longer defined for source text subsequent to the `#undef` preprocessor directive. You must remove a macro definition before redefining the macro name unless the body of the redefinition exactly matches the body of the original definition. An attempt to redefine a macro without first removing it causes an error.

You can use the `define` control on the invocation line but not in a `#pragma` preprocessor directive. To define a macro within the source text, use the `#define` preprocessor directive. You can abbreviate the `define` control but not the `#define` preprocessor directive.

Examples

1. In this example, using the `define` control in the invocation determines the result of conditional compilation in the source file `ex.c`. The macro `SYS` expands to the value `1` since its definition is in the compiler invocation, so `PATHLENGTH` gets the value `128` and `80C196` is defined with an empty value. Since `80C196` is defined, `NUMINTR` gets the value `16`.

   The invocation is as follows:

   ```
   c196 ex.c define(SYS)
   ```

   The `ex.c` source text contains the following lines:

   ```
   #if SYS
   #define PATHLENGTH 128
   #define 80C196
   #else
   #define PATHLENGTH 45
   #endif

   #ifdef 80C196
   #define NUMINTR 16
   #else
   #define NUMINTR 8
   #endif
   ```

2. The following compiler invocation suppresses the `alien` keyword by defining it as a macro that expands to nothing:

   ```
   c196 ex.c define(alien="") preprint
   ```

   The `ex.c` source text contains:

   ```
   alien int f( );
   ```

   After preprocessing, the `ex.i` preprint file contains:

   ```
   int f( );
   ```

diagnostic

Function

Specifies level of diagnostic messages.

Syntax

Select the `EDE | C Compiler Options | Project Options...` menu item. Select one of the `Diagnostics message level (0–2)` options in the `Listing` tab.

`diagnostic(`*`level`*`)`

where:

`level` is the value 0, 1, or 2. The values correspond to remarks, warnings, and errors, respectively.

Abbreviation

`dn`

Class

Primary control

Default

`diagnostic(1)`

Description

Use this control to specify the level of diagnostic messages that the compiler produces. A remark points out a questionable construct, such as using an undeclared function name. A warning reports a suspicious condition that you might want to change. A warning does not terminate the compilation process. Warnings and remarks usually provide information and do not necessarily indicate a condition affecting the object module. An error also does not terminate the compilation process, but causes the compiler not to produce an object file. A fatal error, on the other hand, terminates the compilation process immediately.

Use the different levels of the `diagnostic` control as follows:

`dn(0)` for the compiler to issue all remarks, warnings, and error messages.

`dn(1)` (the default) for the compiler to issue warnings and error messages but no remarks.

`dn(2)` for the compiler to issue only error messages.

The predefined macro `_DIAGNOSTIC_` has the value specified for the `diagnostic` control.

The compiler also reports the number of remark, warning and error situations in the termination message, according to the diagnostic level which also determines the compiler's exit status. For example, if the diagnostic level is 2, the compiler can issue only error messages and the exit status is zero if no errors occurred, even if warning or remark situations occurred. The diagnostic and termination messages usually appear in the print file. If the print file is suppressed, the messages appear on the console instead.

The `diagnostic` control affects the entire compilation. You can specify this control in the compiler invocation or in a `#pragma` preprocessor directive preceding the first line of data definition or executable source text. To override a diagnostic level set in a `#pragma` preprocessor directive, specify a different diagnostic level in the compiler invocation.

Messages (see Chapter 9)
`print`

divmodopt

Function

Enables generation of efficient DIV instruction.

Syntax

Select the `EDE | C Compiler Options | Project Options...` menu item. Enable or disable the `Generate efficient DIV instruction and don't care about overflow` check box in the `Optimization` tab.

`divmodopt | nodivmodopt`

Abbreviation

`dm | nodm`

Class

Primary control

Default

`nodivmodopt`

Description

Use the `divmodopt` control to enable generatation of an efficient DIV instruction even if there are chances for overflow. Normally, the run–time routine ??DIVL does a better job in case of overflow, but the code is less efficient. If you do not care about the overflow you can now generate the instruction with the `divmodopt` control.

eject

Function

Inserts a form–feed into the print file.

Syntax

```
eject
```

Abbreviation

```
ej
```

Class

Primary control

Description

Use this control to insert a form–feed into the print file. You can only specify the `eject` control in a `#pragma` directive. The page breaks after the control line and the compiler generates a header at the top of the page. This control does not have any effect if the `noprint` or `nolist` control is in effect.

```
list
print
```

extend

Function

Recognizes or suppresses Intel C196 extensions.

Syntax

Select the `EDE | C Compiler Options | Project Options...` menu item. Select the `Enable all language extensions` or `Disable language extensions (strict ANSI mode)` radio button in the `Language` tab.

```
extend | noextend
```

Abbreviation

```
ex | noex
```

Class

Primary control

Default

```
extend
```

Description

Use this control to direct the compiler to accept file–scope `register` variables and the `alien`, `far`, `near`, `reentrant`, and `nonreentrant` keywords in the source text. This control also ensures compatibility between prototype and non–prototype function declarations. Use the `noextend` control to suppress recognition of these extensions. These extensions provide compatibility with earlier versions of C196.

When `extend` is in effect, the `register` storage class and allocation of registers work as follows:

- You can declare file–scope variables with the `register` storage class.
- The `regconserve` and `noregconserve` controls determine whether file–scope non–register variables, as well as block–scope non–register variables, are allocated to registers.

- You can combine the `static` and `extern` storage classes with register declarations at both block and file scope, for example:

```
static register int sri;
extern register int cri;
```

When `noextend` is in effect, the C196 compiler uses ANSI semantics for the `register` storage class. The ANSI semantics allow `register` storage class variables within blocks only, not at the file–scope level, and do not allow combining `static` or `extern` with `register` storage class.

The extension keywords that the compiler recognizes when `extend` is in effect are redundant with some of the compiler controls and are provided for compatibility with earlier versions of C196. The `reentrant`, `nonreentrant`, and `alien` keywords have the same effect as the `reentrant`, `noreentrant`, and `fixedparams` controls, respectively. The `extend` and `noextend` controls have no effect on the `reentrant`, `noreentrant`, and `fixedparams` controls.

The `extend` control also extends the way C196 performs parameter type checking between prototype function declarations and old–style function definitions. The ANSI C standard specifies that, in old–style function definitions, `char` and `short` parameters are promoted to `int`, and `float` parameters are promoted to `double`. When a prototype declaration and an old–style definition exist for a function, the parameters of the prototype must be compatible with the promoted parameters of the old–style definition. With `noextend` in effect, C196 conforms to the ANSI standard. For example, with `noextend` in effect, the following combination causes an invalid redeclaration error for the function `f`:

```
int f(char);        /* prototype declaration */

int f(c)            /* old-style definition: */
char c;             /* char promoted to int  */
{}
```

With `extend` in effect, the compiler allows exact type matching between parameters in a prototype declaration and parameters in the associated old–style definition. The above example is accepted with `extend` in effect.

The `extend` and `noextend` controls affect the entire object module. You can specify either of these controls in the compiler invocation or in a `#pragma` preprocessor directive preceding the first line of data definition or executable source text. To override a `#pragma extend` or `#pragma noextend` specified in the source text, specify the opposite control (`noextend` or `extend`, respectively) in the compiler invocation.

```
alien
fixedparams
regconserve
varparams
```

extratmp

Function

Enable or disable usage of extra temporary registers TMPREG8 to TMPREG16.

Syntax

Select the `EDE | C Compiler Options | Project Options...` menu item. Enable or disable the `Use 16 TMPREG bytes instead of 8 (for 16 bit models)` check box in the `Optimization` tab.

`extratmp | noextratmp`

Abbreviation

`et | noet`

Class

Primary control

Default

`noextratmp` for non 24–bit models
`extratmp` for 24–bit models

Description

The compiler uses a maximum of 8 TMPREG bytes for non 24–bit models, and a maximum of 16 TMPREG bytes for 24–bit models. If you specify one of the controls you get what you specify, ET or NOET. For non 24–bit models the `extratmp` control can be used to let the compiler use more than 8 TMPREG bytes (for example, when the program is too complex). For 24–bit models you can also use `noextratmp` (only recommended for `nearcode/neardata`).

`tmpreg`

farcode

Function

Specifies that the whole application uses the extended addressing mode of 24–bit processors for all functions.

Syntax

Select the `Far Code` radio button in the `EDE | CPU Model...` menu item.

```
farcode
```

Abbreviation

`fc`

Class

Primary control

Default

`nearcode`

Description

Use this control to use the extended addressing mode of 24–bit processors. This control causes the compiler to generate extended calls between modules and make all function pointers four bytes long. In addition to user–defined function pointers, the compiler allows four bytes for switch table entries and return addresses. All executable code will be placed in the `farcode` segment of the object module.

The 24–bit processors are configured by the CCB at reset. One of the settings controlled by the CCB is whether to run in the extended mode or the compatibility mode. Once the CCB is loaded into the chip configuration register, the mode is locked, and all of your code will run in the chosen mode. Therefore, if you use the `farcode` control for one module, you must use it for all modules.

The compiler does not generate extended calls within a module, since such local calls are assumed to be within the 32K range of a normal call. Note that in extended mode, all return addresses are four bytes long, regardless of the type of call instruction used.

The `farcode` control can only be used with 24–bit models.

```
farconst   fardata      model
nearcode   nearconst    neardata
```

farconst

Function

Specifies that the default placement of constant data is the far const segment.

Syntax

Select the `Far Const` radio button in the `EDE | CPU Model...` menu item.

```
farconst
```

Abbreviation

`fk`

Class

Primary control

Default

`nearconst`

Description

Use this control to allow constant data to be placed anywhere in the 24–bit extended address space of the 24–bit processors. This control causes the compiler to place switch table constants in the `farconst` segment of the object module, as well as any user–defined constant data that is not qualified with the `near` keyword. The generated code will use extended addressing to access these constants.

When you link your program module(s) with RL196, you can locate the `farconst` segment anywhere you have ROM.

The `farconst` control can only be used with 24–bit models (NT–CNF or NT–ENF).

```
farcode   fardata     model
nearcode  nearconst   neardata
```

fardata

Function

Specifies that the default placement of variable data is the far data segment.

Syntax

Select the `Far Data` radio button in the `EDE | CPU Model...` menu item.

```
fardata
```

Abbreviation

```
fd
```

Class

Primary control

Default

```
neardata
```

Description

Use this control to allow non–register, non–constant data to be placed anywhere in the 24–bit extended address space of the 24–bit processors. This control causes the compiler to place in the `fardata` segment of the object module all user–defined variable data that has not been assigned to registers, and that you have not qualified with the `near` keyword. The generated code will use extended addressing to access these objects.

When you link your program module(s) with RL196, you can locate the `fardata` segment anywhere you have RAM.

The `fardata` control can only be used with 24–bit models.

```
farcode   farconst    model
nearcode  nearconst   neardata
```

fastinterrupt

Function

Specifies whether the compiler saves temporary registers on entering the interrupt routine.

Syntax

Select the `EDE | C Compiler Options | Project Options...` menu item. Enable or disable the `Fast interrupt code (do not save temporary registers)` check box in the `Object` tab.

```
fastinterrupt | nofastinterrupt
```

Abbreviation

```
fi | nofi
```

Class

Primary control

Default

```
nofastinterrupt
```

Description

Use this control to prevent the C196 compiler from saving temporary registers on entering the interrupt rountine. This results in faster execution of the interrupt.

```
generatevectors
interrupt
interruptpage
```

fixedparams

Function

Specifies fixed–parameter list calling convention

Syntax

`fixedparams[(`*`function`* `[,...])]`

where:

`function` is the name of a function defined in the source text.

Abbreviation

`fp`

Class

General control

Default

`varparams`

Description

Use this control to cause functions to use the fixed parameter list (FPL) calling convention. The variable–parameter list (VPL) calling convention is the default used by the C196 compiler. When calling a PL/M–96 function from a C196 program, specify `fixedparams` for the PL/M–96 function in the C196 compilation.

A function's calling convention dictates the sequence of instructions that the compiler generates to manipulate the stack and registers during a call to the function. Code generated for the FPL calling convention performs the following sequence of operations:

1. The calling function pushes the arguments onto the stack with the leftmost argument pushed first.
2. The calling function transfers control to the called function.
3. The called function executes.
4. The called function removes the arguments from the stack.

5. The called function returns control to the calling function.

See the `varparams` control for more information on how the VPL calling convention differs from the FPL calling convention.

The calling convention specification must precede the function declaration. The first declaration or definition of a function sets the calling convention for that function based on the `fixedparams` or `varparams` control in effect for the function, or based on the `alien` keyword or the comma and ellipsis (`,...`), if specified for the function. The comma and ellipsis indicate that the number of parameters to the function has no limit. In this case, `varparams` is in effect.

The `notranslate` and `noobject` controls suppress the object file, causing `fixedparams` to have no effect. However, if you specify the `code` control with the `noobject` control, the effect of `fixedparams` does appear in the pseudo–assembly code listing.

You can specify `fixedparams` in the compiler invocation and in `#pragma` preprocessor directives throughout the source text. When specified without arguments, this control affects all functions in the subsequent source text and remains in effect until the compiler encounters the opposite control (`varparams`) or the end of the source text. The `fixedparams` control specified with an argument list affects only the functions in the argument list.

More than one explicit calling convention specification for any one function causes a warning. A warning occurs if a function in the source text is explicitly declared with a variable parameter list and is named in the function list for the `fixedparams` control.

```
#pragma fixedparams(x)

int x (int i,...)
{
}
```

In this example, `varparams` is in effect.

Examples

1. The following control in the compiler invocation specifies the default variable parameter list convention (VPL) for all functions in the source text except the `plm_fcn` function:

```
fixedparams(plm_fcn)
```

2. The following `#pragma` preprocessor directive has the same effect as the control in the first example:

```
#pragma fixedparams(plm_fcn)
```

3. The following combination of controls in the compiler invocation specifies the fixed parameter list convention (FPL) for all functions in the source text except the `native` function:

```
fixedparams varparams(native)
```

4. The following `#pragma` preprocessor directives have the same effect as the controls in the above example:

```
#pragma fixedparams
#pragma varparams(native)
```

```
code                translate
extend              varparams
object
```

generatevectors

Function

Generates or suppresses the generation of interrupt vectors.

Syntax

Select the `EDE | C Compiler Options | Project Options...` menu item. Enable or disable the `Generate code for interrupt vector` check box in the `Object` tab.

`generatevectors | nogeneratevectors`

Abbreviation

`gv | nogv`

Class

Primary control

Default

`generatevectors`

Description

Use the `nogeneratevectors` control to specify that the compiler must not generate the interrupt vectors for the interrupt functions.

By default the compiler generates interrupt vectors.

You can specify the `generatevectors` and `nogeneratevectors` controls in the compiler invocation and in `#pragma` preprocessor directives at the beginning of the source text.

`interrupt`
`interruptpage`

hold

Function

Specifies whether the windowing code needs to preserve the HOLD/HOLDA bit in the WSR.

Syntax

Select the `EDE | C Compiler Options | Project Options...` menu item. Enable the `Support Vertical Windowing` check box and enable or disable the `Save/restore HOLDEN bit of WSR at function entry/exit` check box in the `Code` tab.

`hold | nohold`

Abbreviation

`ho | noho`

Class

Primary control

Default

`nohold`

Description

If you are using the HOLD/HOLDA protocol along with vertical windowing, specify the `hold` control. This control causes the compiler to generate additional code to preserve the HOLDEN bit of the Window Select Register (WSR). If you are not using the HOLD/HOLDA protocol, specify the `nohold` control to reduce the amount of overhead code.

This control provides the same function as the `hold` and `nohold` parameter keywords in the `windows` control, but without the need to use the `windows` control.

The `hold` control is not a substitute for the `windows` control. The `hold` control is merely to be used with the windowed parameter method for vertical windows (using the keywords `_reg` and `_win` or `_win1` and the `windowram` control).

The WSR management code allows access to local and static register variables located in the mapped area of the register file and above (from `80H` or `0C0H` or `0E0H` depending on the window size). Public register variables allocated in the register segment are restricted to the registers below the mapped area (below `80H` or `0C0H` or `0E0H` depending on the window size). This allocation scheme allows access to these variables without swapping the `wsr`.

See Section 6.4.3 for more information on vertical windows.

If you specify the `hold` control, the compiler produces the following WSR management code in the prolog:

```
ldbze Tmp0,WSR
push  Tmp0
andb  WSR,#80H     /* to retain hlden in wsr */
orb   WSR,?WSR
```

Otherwise, with the `nohold` control, the following code is produced:

```
ldbze Tmp0,WSR
push  Tmp0
ldb   WSR,?WSR
```

The compiler produces the following code in the epilog, with or without the `hold` parameter:

```
ldb  WSR,[SP]
pop  R0
```

The `hold` control can only be used with processors that support vertical windows. Otherwise the compiler generates a fatal error.

```
model
reentrant
regconserve
registers
windowram
```

include

Function

Inserts text from specified file.

Syntax

Select the `EDE | C Compiler Options | Project Options...` menu item. Enter a filename in the `First #include file(s)` field in the `Misc` tab. You can enter multiple filenames by separating them with commas.

`include(`*`filename`* `[,...])`

where:

`filename`	is the file to be included and compiled before the primary source file.

Abbreviation

`ic`

Class

Invocation control

Description

Use this control to insert and compile text from files other than the primary source file. These files are called include files. The compiler processes include files in the order specified in the *`filename`* list before processing the primary source file.

Files included by the `include` control on the invocation line can use all macros defined by the `define` control on the invocation line, regardless of the order of the controls. Files included by the `include` control on the invocation line precede the scope of macros defined by the `#define` preprocessor directive in source text from the primary source file and from all subsequent include files. Files included by the `#include` preprocessor directive in source text are within the scope of previously defined macros and precede the scope of subsequently defined macros.

You can use the `listinclude` control to include the contents of the include files in the print file. The compiler lists the files specified with the `include` control in the order specified before the first line of source listing. To specify a search path for include files, use the `searchinclude` control. To view names of include files and the order of their inclusion without compilation, use the `preprint` and `notranslate` controls.

You can use the `include` control on the invocation line but not in a `#pragma` preprocessor directive. To include a file from within the source text, use the `#include` preprocessor directive. You can abbreviate the `include` control but not the `#include` preprocessor directive.

Example

This invocation line tells the compiler to include the file `kb_sfrs.h` in its C source:

```
c196 file1.c include(kb_sfrs.h)
```

`listinclude`
`preprint`
`searchinclude`

init

Function

Specifies whether the compiler produces the initialization segments and tables.

Syntax

Select the `EDE | C Compiler Options | Project Options...` menu item. Enable or disable the `Support initialized and cleared RAM variables` check box in the `Code` tab.

```
init | noinit
```

Abbreviation

```
it | noit
```

Class

Primary control

Default

```
init
```

Description

Use the `init` control to allow the compiler to produce the initialization segments and tables. This control is only valid for the (default) OMF version 3.2. At startup (reset), library module `cstart` processes the initialization table: it copies the initial constant data to the corresponding variables, and zeroes the uninitialized variables.

Use the `noinit` control/pragma to prevent the generation of initializing data and tables, even though you have used initializers in your source code (`noinit` also prevents zeroing of uninitialized variables).

```
abszero
zero
```

inst

Function

Specifies whether the compiler generates vector tables for switch statements.

Syntax

Select the `EDE | C Compiler Options | Project Options...` menu item. Enable or disable the `Generate jump table for switch statement` check box in the `Code` tab.

`inst | noinst`

Abbreviation

`is | nois`

Class

Primary control

Default

`noinst`

Description

Use this control to prevent the C196 compiler from generating vector tables for `switch` statements. You must use this control if you are overlapping ROM and RAM memory because the processor reads data from these tables rather than fetching code from them. When `inst` is in effect, the compiler generates a series of compare instructions instead of the vector table.

See the *80C196 Utilities User's Guide*, listed in *Related Publications*, for more information on overlapping ROM and RAM memory.

Note that the `inst` control may be somewhat confusing. It has to do with the fact that in an older version of OMF96 (v2.0) all rommable information (i.e. code and constant data) went into the CODE segment. However, in case the INST pin of the processor is used, the constant data should not be mixed with the code instructions. The Intel iC–96 had the `inst` control to prevent the compiler to generate constant data in the middle of code instructions. With the newer definitions of OMF96 a better solution is available; all constant data in collected in a separate segment. In other words, this `inst` control is only useful if the output format of the compiler is OMF96 v2.0 (see the `oldobject` and `omf` controls).

interrupt

Function

Specifies a function to be an interrupt handler.

Syntax

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

`interrupt(`*`function`*`[=`*`n`*`] [,...])`

where:

`function` is the name of a function defined in the source text.

`n` is the interrupt number or the interrupt address.

Abbreviation

`in`

Class

General control

Description

Use this control to specify a function in the source text to handle some condition signaled by an interrupt. An interrupt function must be of type `void` and cannot take arguments. The interrupt designation must precede the function definition.

You can specify the same interrupt function for multiple interrupts. For example, the following `#pragma` directive is valid:

```
#pragma interrupt(int_log=1, int_log=2,
int_log=0x2006)
```

However, you cannot specify multiple interrupt function handlers for one interrupt. The following example generates a fatal error:

```
#pragma interrupt(int_log=1, rst_func=1)
```

The `interrupt` control causes the compiler to generate prolog and epilog code to save and restore registers. The compiler takes into consideration the differences between the selected processor models when generating the call and return sequences. The exact sequence generated depends on the argument to the `model` control.

Note that the compiler does not automatically saves/restores the state of the floating point library (see `fpsave()` and `fprstor()` in the *80C196 Utilities User's Guide*).

The compiler also creates an interrupt vector entry for each interrupt function. If the code being generated is for the 8096, the interrupt number must be in the range `0` to `7`. If the code being generated is for the 80C196 microcontrollers, the interrupt number must be in the range `0` to `9` or `24` to `31`. The interrupt numbers correspond to positions in the interrupt vector table calculated as follows:

> the interrupt number (n) multiplied by 2 and added to the base of the vector table. For 8096 and most 80C196 parts the base is `2000h`. For 80C196NT/NP the base is `FF2000h`.
>
> With the `interruptpage` control you can override the default base of the vector table.

If you specify an interrupt address instead of an interrupt number, you must check your processor specific manual for valid addresses. It is not necessary to specify the page when you use addresses. For example, address `0x2008` and address `0xFF2008` are the same on an extended model.

The interrupt priority determines interrupt sequencing when several interrupts are pending. You can allow any priority of interrupt to occur by explicitly enabling it using `int_mask` and `imask1`. Since an interrupt function prolog includes either `pushf` (for 8096 code) or `pusha` (for 80C196 code), which disable interrupts, the execution of the interrupt handler cannot be interrupted unless you reenable and unmask interrupts within the interrupt, using `int_mask` and `imask1`. See the processor specific manual for a list of interrupts with their corresponding interrupt numbers.

You can specify the `interrupt` control in the compiler invocation and in `#pragma` preprocessor directives.

In certain cases it is useful to have 'indirect' interrupt vectors.

Two examples. The first example uses an interrupt vector table in register memory. It can be updated at run time. The second example can be used if the 196 vector table and some basic code is in some permanent ROM (at address 2000h), and the interrupt handler and other code is in an EPROM which can be replaced in the target.

First the example with a table in register memory.

```
JMPVEC  MODULE
                RSEG
vector0:        DSW     1
vector1:        DSW     1
vector2:        DSW     1
vector3:        DSW     1

                KSEG at 2000h
                DCW     IntHand0
                DCW     IntHand1
                DCW     IntHand2
                DCW     IntHand3

                CSEG
IntHand0:       PUSHA
                PUSH    #$+5
                BR      [vector0]
                POPA
                RET

IntHand1:       PUSHA
                PUSH    #$+5
                BR      [vector1]
                POPA
                RET

IntHand2:       PUSHA
                PUSH    #$+5
                BR      [vector2]
                POPA
                RET

IntHand3:       PUSHA
                PUSH    #$+5
                BR      [vector3]
                POPA
                RET
```

Of course now you must fill in the table before the first interrupt occurs. If functions in C are called you need to determine if it is necessary to still declare them as 'interrupt'. The example above does not take any precaution to save and restore the TMPREG registers.

The 196 processor requires that the first instruction in the interrupt must be a PUSHA. See the processor User's Manual for details on interrupt servicing.

The next example. The first section (say `2000h–23FFh`) is permanently in ROM, the second section (say `0C000h–0FFFFh`) can be in RAM (downloading) or in some EPROM (that can be replaced).

```
                KSEG at 2000h
                DCW     IntHand0
                DCW     IntHand1
                DCW     IntHand2
                DCW     IntHand3

                CSEG at 0C000h
IntHand0:       PUSHA
                LCALL   IntFunc0
                POPA
                RET

IntHand1:       PUSHA
                LCALL   IntFunc1
                POPA
                RET

IntHand2:       PUSHA
                LCALL   IntFunc2
                POPA
                RET

IntHand3:       PUSHA
                LCALL   IntFunc3
                POPA
                RET
```

Example

The following is an example of a valid `interrupt` control, specified on the invocation line:

```
interrupt(int_handle_1=1, int_handle_24=0x2030)
```

The source text compiled using this control contains the following declarations:

```
void int_handle_1(void)
{...}

void int_handle_2(void)
{...}
```

`fastinterrupt`
`generatevectors`
`interrupt_piha`
interrupt_pihb
`interruptpage`
`model`

interrupt_piha / interrupt_pihb

Function

Specifies a function to be an interrupt handler in the piha block or pihb block respectively.

Syntax

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

```
interrupt_piha(function[=n] [,...])
interrupt_pihb(function[=n] [,...])
```

where:

function	is the name of a function defined in the source text.
n	is the piha/pihb interrupt number or the absolute interrupt address.

Class

General control

Description

These controls are only valid for those models which have a piha and pihb block.

Use this control to specify a function in the source text to handle some condition signaled by an interrupt. An interrupt function must be of type `void` and cannot take arguments. The interrupt designation must precede the function definition.

See the `interrupt` control for more information.

`fastinterrupt`, `generatevectors`, `interrupt`, `interruptpage`, `model`

interruptpage

Function

Specifies an interrupt page or base address.

Syntax

Select the `EDE | C Compiler Options | Project Options...` menu item. Enter a value in the `Specify the Interrupt page (0–0FFH) or base address` field in the `Object` tab.

`interruptpage(`*`num`*`|`*`base`*`)`

where:

`num`	is the page number (`0..0xFF`) for interrupt vectors.
`base`	is the base address.

Abbreviation

`ip`

Class

Primary control

Default

`ip(0x0)`	for non 24–bit models
`ip(0xFF)`	for 24–bit models

Description

Use this control to specify the page number or base address for interrupt vectors. By default the interrupt vectors are put in the `0xFF` page for 24–bit models and in page `0` for other models. Values `0..0xFF` are taken as page numbers (and shifted 16 bits), other values are taken as the base address.

You can specify the `interruptpage` control in the compiler invocation and in `#pragma` preprocessor directives.

Example

The following is an example of a valid `interruptpage` control, specified on the invocation line:

```
interruptpage (0x0F)
```

Here are some `#pragma` examples:

```
#pragma model('nt-e')
#pragma interruptpage(0x0F)
#pragma interrupt(int1=1)
void int1(void){}   /* Vector at 0x0F2002 */

#pragma model('nt-e')
#pragma interruptpage(0xFE0000)
#pragma interrupt(int1=1)
void int1(void){}   /* Vector at 0xFE2002 */

#pragma model(kc)
#pragma interruptpage(0xD000)
#pragma interrupt(int1=1)
void int1(void){}   /* Vector at 0xD002 */
```

`interrupt`
`model`

list

Function

Specifies or suppresses source text listing in print file.

Syntax

`list | nolist`

Abbreviation

`li | noli`

Class

General control

Default

`list`

Description

Use this control to generate a listing of the source text. The compiler places the source listing in the print file. Use the `nolist` control to suppress the source listing.

Several other controls affect the contents of the listing, as follows:

- The `cond` control causes uncompiled conditional code to appear in the listing.
- The `listexpand` control causes macros to be expanded in the listing.
- The `listinclude` control causes text from include files to appear in the listing.

The `noprint` and `notranslate` controls suppress the entire print file, even if `list` is specified. The `nolist` control suppresses the source text listing, even if `cond`, `listexpand`, and `listinclude` are specified.

The `list` and `nolist` controls affect only the subsequent source text and remain in effect until the compiler encounters the opposite control or the end of the source text. You can specify these controls in the compiler invocation and in `#pragma` preprocessor directives throughout the source text.

cond	pagelength	tabwidth
listexpand	pagewidth	title
listinclude	print	translate

listexpand

Function

Includes or suppresses macro expansion in listing.

Syntax

Select the `EDE | C Compiler Options | Project Options...` menu item. Enable or disable the `Include macro expansion` check box in the `Listing` tab.

`listexpand | nolistexpand`

Abbreviation

`le | nole`

Class

General control

Default

`nolistexpand`

Description

Use this control to include the results of macro expansion in the source text listing in the print file. Use the `nolistexpand` control (default) to suppress the results of macro expansion.

The compiler marks the macro expansion lines with a plus (+) in the `Line` column of the source text listing. Macro expansions only appear in the source text listing of compiled code and do not appear in the source text listing of uncompiled code even when you use the `cond` control to list uncompiled conditional code.

If `nolist`, `notranslate`, or `noprint` is specified, the print file is suppressed and `listexpand` has no effect. If `nolistinclude` is in effect, listing of include files is suppressed and `listexpand` has no effect on the included source text.

The `listexpand` and `nolistexpand` controls affect only the subsequent source text and remain in effect until the compiler encounters the opposite control or the end of the source text. You can specify these controls in the compiler invocation and in `#pragma` preprocessor directives throughout the source text.

```
cond                    print
list                    translate
listinclude
```

listinclude

Function

Includes or suppresses text from include files in listing.

Syntax

Select the `EDE | C Compiler Options | Project Options...` menu item. Enable or disable the `Add text from include files` check box in the `Listing` tab.

`listinclude | nolistinclude`

Abbreviation

`lc | nolc`

Class

General control

Default

`nolistinclude`

Description

Use this control to list the text of include files in the source text listing in the print file. Use the default `nolistinclude` control to suppress the listing of include files.

The compiler lists files included with the `include` control in the order they are specified before the first line of source listing and lists the text of files included with the `#include` preprocessor directive after the line with the `#include` directive.

Included files can themselves include files. The nesting level of the included file appears in the `Level` column of the source text listing.

When `nolistinclude` is in effect, diagnostic messages for include files appear in the print file as follows:

- For files included with the `include` control, diagnostic messages precede the first line of source text.

- For files included with the `#include` preprocessor directive, diagnostic messages appear on the lines immediately after the `#include` directive.

If `nolist`, `notranslate`, or `noprint` is specified, the print file is suppressed and `listinclude` has no effect.

The `listinclude` and `nolistinclude` controls affect only the subsequent source text and remain in effect until the compiler encounters the opposite control or the end of the source text. You can specify these controls in the compiler invocation and in `#pragma` preprocessor directives throughout the source text.

```
include             print
list                translate
```

locate

Function

Locates symbols to absolute addresses.

Syntax

`#pragma locate(`*`var1`*`=`*`addr`* `[+|-` *`value`*`],...)`

where:

`var1` is a valid symbol name.

`addr` is a valid absolute address.

`value` is a valid offset value.

Abbreviation

`lo`

Class

General control

Description

Use this pragma control to locate one or more symbols to absolute addresses. Use this control only in a `#pragma` preprocessor directive. This control must follow the declaration of the symbols. For example, the following pragma control line locates `i1` and `i2` to addresses `1F00H` and `1F02H` respectively:

```
int i1, i2;
#pragma locate(i1=0x1F00,i2=0x1F02)
```

You can also use the `#define` preprocessor directive to define the absolute address. Then, you can use the macro symbol as a base address and the `+` and `–` signs to indicate the offset. For example, assume the previous example but with a macro definition:

```
#define abs_addr 0x1F00

int i1, i2;
#pragma locate (i1=abs_addr, i2=abs_addr+2)
```

This example has the same effect as the previous example.

You cannot locate non–static block–scope variables because they are allocated on the stack or in register overlay segments, which are located by RL196 at link time. The following example generates an error:

```
main()
{   int i2;

#pragma locate(i2=0x1F02); /* This line generates an
                              error. */
}
```

mixedsource

Function

Includes or suppresses mixed assembly source in listing.

Syntax

Select the `EDE | C Compiler Options | Project Options...` menu item. Enable or disable the `Merge C-source code with assembly` check box in the `Listing` tab.

`mixedsource | nomixedsource`

Abbreviation

`ms | noms`

Class

Primary control

Default

`nomixedsource`

Description

Use this control to include mixed assembly source text in the print file. Use the `nomixedsource` control (default) to suppress the generation of mixed assembly source in the print file.

By default the compiler lists the C source at the beginning of the print file and generates lines with `; Statement` *`num`* to indicate a C source line. With the `mixedsource` control the compiler does not list the C source at the beginning of the print file but mixes the C source lines with the assembly source. So, the line with `; Statement` *`num`* are replaced by the source line itself.

You can specify the `mixedsource` and `nomixedsource` control in the compiler invocation and in `#pragma` preprocessor directives at the beginning of the source text.

`list`
`listinclude`
`print`

model

Function

Specifies the processor/instruction set.

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`*`)`

where:

`processor` Selects the instruction set the compiler 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`, `np`, `nt` and `nu` arguments of the `model` control also enable the compiler 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 `model` is available for backward compatibility and is equivalent to specifying `kb`. For future compatibility, use the `model(kb)` control specification instead of `model(196)`.

`bh` to select the 8096BH.

`ca` to select the 80C196CA. Specifying `ca` is equivalent to specifying `kr`.

`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 `jq` is equivalent to specifying `kr`.

`jr` to select the 80C196JR. Specifying `jr` is equivalent to specifying `kr`.

`js` to select the 80C196JS. Specifying `js` is equivalent to specifying `kr`.

`jt` to select the 80C196JT. Specifying `jt` is equivalent to specifying `kr`.

`jv` to select the 80C196JV. Specifying `jv` is equivalent to specifying `kr`.

`kb` to select the 80C196KB. Specifying `kb` is equivalent to specifying `196`.

`kc` to select the 80C196KC.

`kd` to select the 80C196KD.

`kl` to select the 80C196KL. Specifying `kl` is equivalent to specifying `kr`.

`kq` to select the 80C196KQ. Specifying `kq` is equivalent to specifying `kr`.

`kr` to select the 80C196KR.

`ks` to select the 80C196KS. Specifying `ks` is equivalent to specifying `kr`.

`kt` to select the 80C196KT. Specifying `kt` is equivalent to specifying `kr`.

`lb` to select the 80C196LB.

`mc` to select the 80C196MC.

`md` to select the 80C196MD.

`mh` to select the 80C196MH.

`np` to select the 80C196NP. This argument can have an extra suffix as described in the note below.

`nt` to select the 80C196NT. This argument can have an extra suffix as described in the note below.

`nu` 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`, `ec`, `np`, `nt` or `nu`. The following six suffixes are possible:

xx`–c` to select the compatible mode and to use near code addressing and near data/near const addressing.

xx`–cnf` to select the compatible mode and to use near code addressing and near data/far const addressing.

xx`–cf` to select the compatible mode and to use near code addressing and far data/far const addressing.

xx`–e` to select the extended mode and to use far code addressing and near data/near const addressing.

xx`–enf` to select the extended mode and to use far code addressing and near data/far const addressing.

xx`–ef` to select the extended mode and to use far code addressing and far data/far const addressing.

The predefined macro _ARCHITECTURE_ has the value 61, 90, 'BH', 'CA', 'CB', 'EA', 'EC', 'KB', 'KC', 'KD', 'KR', 'LB', 'MC', 'MD', 'MH', 'NP', 'NT', or 'NU' depending on the value specified for the model control.

The predefined macro _SFR_H_ has the value 'bh_sfrs.h', 'ca_sfrs.h', 'cb_sfrs.h', 'ea_sfrs.h', 'ec_sfrs.h', 'kb_sfrs.h', 'kc_sfrs.h', 'kd_sfrs.h', 'kr_sfrs.h, 'lb_sfrs.h', 'mc_sfrs.h', 'md_sfrs.h', 'mh_sfrs.h', 'np_sfrs.h', 'nt_sfrs.h', or 'nu_sfrs.h' depending on the value specified for the model control. You can use this predefined macro in your C source instead of the name of the *xx*_sfrs.h include file:

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

The predefined macro _FUNCS_H_ has the value *xx*_sfrs.h depending on the value specified for the model control.

If notranslate or noobject is in effect, the compiler does not generate an object module and model has no effect. However, specifying model with noobject and code can still affect the pseudo–assembly listing in the print file.

The model control affects the entire object module. You can specify this control in the compiler invocation or in a #pragma preprocessor directive preceding the first line of data definition or executable source text. To override a #pragma model(*processor*) preprocessor directive specified in the source text, specify model with a different *processor* in the compiler invocation.

```
farcode              nearcode
farconst             nearconst
fardata              neardata
interrupt            registers
```

nearcode

Function

Specifies that the whole application uses the compatibility addressing mode of 24–bit processors for all functions.

Syntax

Select the `Near Code` radio button in the `EDE | CPU Model...` menu item.

`nearcode`

Abbreviation

`nc`

Class

Primary control

Default

`nearcode`

Description

Use this control to use the compatibility addressing mode of 24–bit processors. This control causes the compiler to generate 16–bit calls between modules and make all function pointers two bytes long. In addition to user–defined function pointers, the compiler allows two bytes for switch table entries and return addresses. All executable code will be placed in the `highcode` segment of the object module.

The 24–bit processors are configured by the CCB at reset. One of the settings controlled by the CCB is whether to run in the extended mode or the compatibility mode. Once the CCB is loaded into the chip configuration register, the mode is locked, and all of your code will run in the chosen mode. Therefore, if you use the `nearcode` control for one module, you must use it for all modules.

The `nearcode` control can only be used with 24–bit models.

`farcode` `farconst` `fardata`
`model` `nearconst` `neardata`

nearconst

Function

Specifies that the default placement of constant data is the constant segment.

Syntax

Select the `Near Const` radio button in the `EDE | CPU Model...` menu item.

```
nearconst
```

Abbreviation

`nk`

Class

Primary control

Default

`nearconst`

Description

Use this control to place constant data in the lowest 64K of the address space of 24–bit processors. This control causes the compiler to place switch table constants in the `const` segment of the object module, as well as any user–defined constant data that is not qualified with the `far` keyword. The generated code will use normal data addressing to access these constants.

When you link your program module(s) with RL196, you must provide sufficient ROM for the `const` segment somewhere within the lowest 64K of the address space.

The `nearconst` control can only be used with 24–bit models (NT–CNF or NT–ENF).

`farcode`	`farconst`	`fardata`
`model`	`nearcode`	`neardata`

neardata

Function

Specifies that the default placement of variable data is the data segment.

Syntax

Select the `Near Data` radio button in the `EDE | CPU Model...` menu item.

```
neardata
```

Abbreviation

`nd`

Class

Primary control

Default

`neardata`

Description

Use this control to place non–register, non–constant data in the lowest 64K of the address space of 24–bit processors. This control causes the compiler to place in the `data` segment of the object module all user–defined variable data that has not been assigned to registers, and that you have not qualified with the `far` keyword. The generated code will use normal data addressing to access these objects.

When you link your program module(s) with RL196, you must provide sufficient RAM for the `data` segment somewhere within the lowest 64K of the address space.

The `neardata` control can only be used with 24–bit models.

```
farcode    farconst     fardata
model      nearcode     nearconst
```

object

Function

Generates and names or suppresses object file.

Syntax

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

`object[(`*`filename`*`)] | noobject`

where:

`filename` is the file, including the path, if necessary, in which the compiler places the object code.

Abbreviation

`oj | nooj`

Class

Primary control

Default

`object`

Description

Use this control to specify a non–default filename or directory for the object file. By default, the compiler places the object file in the directory containing the primary source file. If you do not provide a filename, the compiler composes the default object filename from the primary source filename. For example, the compiler creates an object file named `main.obj` for the primary source file `main.c`.

Use the `noobject` control to suppress creation of an object file. The `notranslate` control suppresses all translation of source text to object code and suppresses the object file and the print file. The `noobject` control does not suppress translation and does not prevent the compiler from producing a print file. The `noobject` control overrides other object file controls except for their effects on the print file.

If a file already exists for either the default or the specified filename, the compiler writes over the existing file with the new object file.

The `object` and `noobject` controls affect the entire compilation. You can specify these controls in the compiler invocation or in a `#pragma` preprocessor directive preceding the first line of data definition or executable source text. To override a `#pragma object(`*`filename`*`)` or `#pragma noobject` preprocessor directive specified in the source text, specify the opposite control (`noobject` or `object` with a different *`filename`*, respectively) in the compiler invocation.

```
code
oldobject
translate
```

oldobject

Function

Produces an object file compatible with the OMF96 V2.*x*.

Syntax

`oldobject | nooldobject`

Abbreviation

`oo | nooo`

Class

Primary control

Default

`nooldobject`

Description

Use this control to produce an object file compatible with earlier versions of the C196 compiler. This control causes the compiler to place all constants, including switch tables, in the `code` segment. No constant segment is produced.

You may need to use the `inst` control when you use `oldobject` if your system overlaps RAM and ROM. Also, if you want your data allocated the same way previous versions of the compiler allocated data, you may need to use the `wordalign` control.

The `oldobject` control is incompatible with 24–bit models.

`inst`
`model`
`wordalign`

omf

Function

Specify OMF96 version.

Syntax

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

`omf( num )`

where:

num is a 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 to produce an object file compatible with a specific OMF96 version.

Specifying `omf(0)` is the same as specifying `oldobject`.

Example

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

```
c196 file1.c omf(0)
```

`oldobject`, `model`

optimize

Function

Specifies the level of optimization.

Syntax

Select the `EDE | C Compiler Options | Project Options...` menu item. Choose an `Optimization level` in the `Optimization` tab.

optimize(*level*)

where:

`level` is `0`, `1`, `2`, or `3`. The values correspond to the levels of optimization; `0` causes the least amount of optimization and `3` causes the most optimization.

Abbreviation

`ot`

Class

Primary control

Default

`optimize(1)`

Description

Use this control to improve the space usage and execution efficiency of a program. Use level `0` when debugging with a symbolic debugger to ensure the closest match between a line of source text and the object code generated for that line. Each optimization level performs all the optimizations of all lower levels. Figure 4–1 summarizes the optimizations performed at each level.

The predefined macro `_OPTIMIZE_` has the value specified for the `optimize` control.

The `optimize` control affects the entire object module. You can specify this control in the compiler invocation or in a `#pragma` preprocessor directive preceding the first line of data definition or executable source text. To override a `#pragma optimize(`*`level`*`)` preprocessor directive specified in the source text, specify `optimize` with a different *`level`* in the compiler invocation.

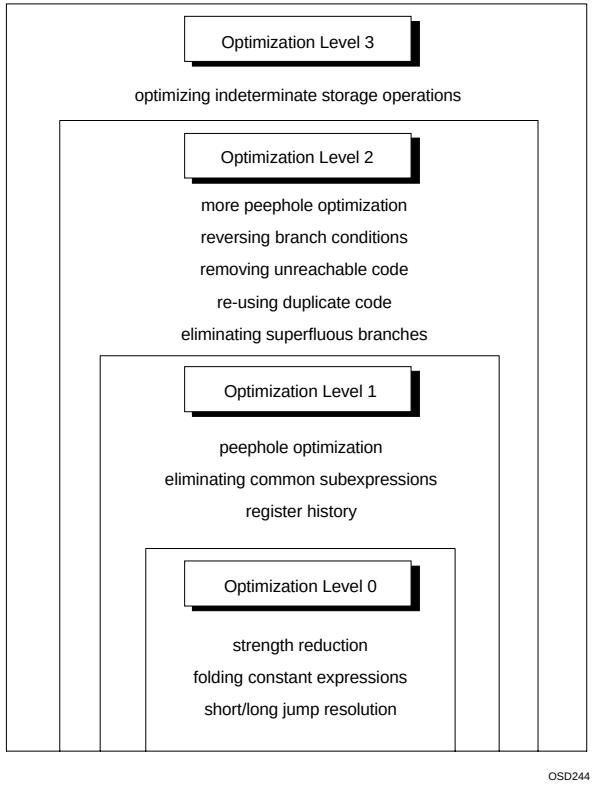


Figure 4–1: Summary of optimization levels

Folding of Constant Expressions at All Levels

The compiler recognizes the operations involving constant operands, then the compiler removes or combines them to save memory space or execution time. Addition of `0`, multiplication by `1` or `0`, and operations on two or more constants fall into this category. For example, the expression `a+2+3` becomes `a+5`.

The following constant operations are detected and reduced for all integral values except unsigned longs, including signed and unsigned bit fields (and, in case you were wondering, a one–bit signed field has the range `–1..0`):

Comparisons involving constants
Comma operators involving constants
Multiplication by zero

There may be some surprises that come from code being eliminated, and also from the warning messages telling you that a comparison always returns TRUE or it always returns FALSE. However, if the part of the expression being eliminated contains a function call, the function will be called but its result will not be used –– a constant will be used instead of the operation involving the function. For example, in the following program:

```
 1  extern unsigned uns_func(void);
 2
 3  void main(void)
 4  {
 5      int i;
 6      unsigned u;
 7
 8      if (i > 84000 || u < 0)
 9          i = 6;
10      if (uns_func() < 0)
11          i = 7;
12      i = uns_func() * 0;
13  }
```

the only code generated for the body of `main` will be the two calls to `uns_func()` (needed because the function might have useful side effects), and the clearing of `i`. This is because an integer can never be greater than `84000`, and an unsigned value can never be less than zero. Here is the generated body of `main`:

```
                            ;   Statement  10
0009  EF0000        E       call    uns_func
                            ;   Statement  12
000C  EF0000        E       call    uns_func
000F  0100          R       clr     i
```

Optimizing Short Jumps and Moves at All Levels

The compiler saves space in the object code by using shorter forms for identical machine instructions.

Reducing Operator Strength at All Levels

The compiler substitutes quick operations for slower ones, such as shifting left by one instead of multiplying by `2`. The substituted instruction requires less space and executes faster.

Eliminating Common Subexpressions at Levels 1, 2, and 3

If an expression reappears in the same basic block of source text, the compiler generates object code to reuse rather than recompute the value of the expression. The generated code saves the intermediate results during expression evaluation in registers and on the stack for later use. The compiler also recognizes commutative forms of subexpressions. For example, in the following block of code, the compiler generates code to compute the value of `c*d/3` for the first expression and to save and retrieve it for the second expression:

```
a = b + c*d/3;
c = e + d*c/3;
```

Eliminating Superfluous Branches at Levels 2 and 3

The compiler combines consecutive or multiple branches into a single branch.

Reusing Duplicate Code at Levels 2 and 3

Duplicate code can be identical code at the ends of two converging paths, or it can be machine instructions immediately preceding a loop identical to those ending the loop. In the first case, the compiler inserts code on only one path and inserts a jump to that path in the other path. In the second case, the compiler generates a branch to reuse the code generated at the beginning of the loop.

Removing Unreachable Code at Levels 2 and 3

The compiler eliminates code that can never be executed. During the second pass of the compiler, the optimization that removes the unreachable code goes through the generated object code and finds areas which can never be reached due to the control structures created in the first pass.

Reversing Branch Conditions at Levels 2 and 3

The compiler optimizes the evaluation of Boolean expressions, so only the shorter of two mutually exclusive conditions is evaluated. For example, in Figure 4–2, the `if` statement on the left has the execution order of its branches reversed as shown on the right:

Original Source Text	Effect of Optimization

`if (!a)`
`{`
`  /* (block 1) */`
`}`
`else`
`{`
`  /* (block 2) */`
`}`
	`if (a)`
	`{`
	`/* (block 2) */`
	`}`
	`else`
	`{`
	`/* (block 1) */`
	`}`

Figure 4–2: Reversing branch conditions

Optimizing Indeterminate Storage Operations at Level 3

The indeterminate storage operations involve pointer indirection. When code assigns a pointer to refer to a variable, it creates an alias for that variable. A variable referenced by a pointer has two aliases: the pointer and the name of the variable itself. Use optimization level 3 only when the compiler need not insert code to guard against aliasing.

The compiler performs this optimization as follows:

- When the code assigns an expression to a variable, the compiler generates code to evaluate the expression and assign the result to the variable. The result also remains in the register used in evaluating the expression.
- When the code subsequently uses the same alias for the variable, the compiler does not generate code to gain access to the variable; instead, it inserts a reference to the register.
- The compiler refers to the same register each time the code uses the alias. This use of registers improves run–time performance since the processor can access the register faster than the variable stored in memory.

This optimization can introduce errors when the code uses multiply aliased variables. The compiler does not insert code to check for intermediate references to a variable using a different alias. If the code modifies a variable using a different alias, the value in the variable is not necessarily the same as the value in the register referenced by the compiler. For example, in the following code under optimization level 3, `y` erroneously acquires the value `1` instead of `2`. If the optimization level is less than 3, the compiler codes the assignment correctly:

```
int x,y;
int *a = &x;    /*  *a is aliasing x    */
x = 1;          /*  put a value in x    */
*a = 2;         /*  x now has value 2   */
y = x;          /*  TROUBLE at level 3! */
```

Use the `volatile` modifier to prevent the compiler from optimizing any reference to a variable.

`volatile`

overlay

Function

Locates register symbols to absolute addresses in the overlay register segment.

Syntax

```
#pragma overlay(var1=addr [+|– value],...)
```

where:

var1 is a valid symbol name.

addr is a valid absolute address.

value is a valid offset value.

Abbreviation

ov

Class

General control

Description

Use this pragma control to locate one or more register symbols to absolute addresses, and mark them as overlayable. This control must follow the declaration of the symbols. For example, the following pragma control line locates both `int1` and `long1` to address `C0H`:

```
int int1;
long long1;
#pragma overlay(int1=0xC0,long1=0xC0)
```

You can also use the `#define` preprocessor directive to define the absolute address. Then, you can use the macro symbol as a base address and the + and – signs to indicate the offset. For example, assume the previous example, but with a macro definition:

```
#define WIN_BASE 0xC0
int int1;
long long1;
#pragma overlay(int1=WIN_BASE,long1=WIN_BASE)
```

This example has the same effect as the previous example.

If you are handling the WSR yourself, you can use this control to arrange your data in a vertical window (you should not use the `windows` control). Since the specified symbols are marked overlayable, the linker will not issue warnings about more than one of these symbols overlapping. You must still allocate space under a different name for the windowed data, and locate it with the `locate` control.

Only `register` variables can be located with this control.

`locate`
`windows`

pagelength

Function

Specifies lines per page in the print file.

Syntax

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

`pagelength(` *`lines`*`)`

where:

`lines`	is the length of a page in lines. This value can range from `10` to `32767`.

Abbreviation

`pl`

Class

Primary control

Default

`pagelength(60)`

Description

Use this control to specify the maximum number of lines printed on a page of the print file before a form feed is printed. The number of lines on a page includes the page headings.

The `noprint` and `notranslate` controls suppress the print file, causing the `pagelength` control to have no effect.

The `pagelength` control affects the entire print file. You can specify this control in the compiler invocation or in a `#pragma` preprocessor directive preceding the first line of data definition or executable source text. To override a `#pragma pagelength(`*`lines`*`)` specified in the source text, specify `pagelength` with a different *`lines`* in the compiler invocation.

```
pagewidth          title
print              translate
tabwidth
```

pagewidth

Function

Specifies line width in the print file.

Syntax

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

pagewidth(*chars*)

where:

`chars` is the line length in number of characters. This value can range from `72` to `255`.

Abbreviation

`pw`

Class

Primary control

Default

`pagewidth(120)`

Description

Use this control to specify the maximum width, in characters, of lines in the print file.

The `noprint` and `notranslate` controls suppress the print file, causing the `pagewidth` control to have no effect.

The `pagewidth` control affects the entire print file. You can specify this control in the compiler invocation or in a `#pragma` preprocessor directive preceding the first line of data definition or executable source text. To override a `#pragma pagewidth(`*`chars`*`)` specified in the source text, specify `pagewidth` with a different *`chars`* in the compiler invocation.

```
pagelength          title
print               translate
tabwidth
```

preprint

Function

Generates or suppresses a preprocessed source text listing file.

Syntax

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

`preprint[(`*`filename`*`)] | nopreprint`

where:

`filename` is the filename, including a device name and directory name or pathname, if necessary, in which the compiler places the preprint information.

Abbreviation

`pp | nopp`

Class

Invocation control

Default

`nopreprint`

Description

Use this control to create a file containing the text of the source after preprocessing. Use the default `nopreprint` control to suppress creation of a preprint file. Preprocessing includes file inclusion, macro expansion, and elimination of conditional code. The preprint file is the intermediate source text after preprocessing and before compilation. This file is not related to the print file created by the `print` control.

The preprint file is useful for observing the results of macro expansion, conditional compilation, and the order of include files. If the preprint file contains no errors, compiling the preprint file produces the same results as compiling the primary source file and any files included in the compiler invocation.

By default, the compiler places the preprint file in the directory containing the source file. If you do not provide a filename, the compiler composes the default preprint filename from the source filename with the `.i` extension. For example, the compiler creates a preprint file named `proto.i` for the source file `proto.c`.

The `preprint` and `nopreprint` controls affect the entire source text. You can specify one of these controls once in the compiler invocation. Do not use these controls in a `#pragma` preprocessor directive.

`translate`

print

Function

Generates or suppresses the print file.

Syntax

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

`print[(`*`filename`*`)] | noprint`

where:

`filename` is the file, including a device name and directory name or pathname, if necessary, in which the compiler places the print information.

Abbreviation

`pr`

Class

Primary control

Default

`print`

Description

Use this control to produce a text file of information about the source and object code. The print file is not the same as the preprint file. By default, the compiler places the print file in the directory containing the source file. If you do not provide a filename, the compiler composes the default print filename from the source filename with the `.lst` extension. For example, the compiler creates a print file named `main.lst` for the source file `main.c`.

The `noprint` control suppresses the print file. The compiler then displays all diagnostic messages at the console. The `noprint` control overrides all other listing controls. Only the `notranslate` control can override the `print` control.

The `print` and `noprint` controls affect the entire source text. You can specify either of these controls in the compiler invocation or in a `#pragma` preprocessor directive preceding the first line of data definition or executable source text. To override a `#pragma print` or `#pragma noprint` in the source text, specify the opposite control, `noprint` or `print`, respectively, in the compiler invocation.

`code`	`listinclude`	`title`
`cond`	`pagelength`	`translate`
`diagnostic`	`pagewidth`	`xref`
`list`	`symbols`	
`listexpand`	`tabwidth`	

pts

Function

Loads a PTS vector with the address of a PTS control block.

Syntax

`#pragma pts(`*`struct-name`*`=`*`vector`*`)`

where:

`struct-name` is a name assigned to the control block.

`vector` is an interrupt vector number or an interrupt vector address.

Abbreviation

`pt`

Class

General control

Description

Use this pragma control, combined with the `locate` pragma control, to load the peripheral transaction server (PTS) vectors with the addresses of the PTS control blocks. You must use the `locate` pragma control to locate the PTS control blocks in internal RAM space (`1AH–1FFH`) at an address evenly divisible by eight (8). The *`xx`*`_funcs.h` header files, where *`xx`* is one of the processor models (for example `kc_funcs.h`), contain type definitions for the various PTS control blocks.

Example

The following example demonstrates the use of the `pts` pragma control.

```
#pragma model(kc)
#include <stdio.h>
#include _SFR_H_
#include _FUNCS_H_

STran_ptscb single;

#pragma locate(single=0x50)   /* Address divisible by 8. */
#pragma pts(single=0)         /* Assign control block to
                                 vector 0*/
#pragma interrupt(timer1=0)   /* Interrupt vector zero. */
```

```
int count;
const char src[]   = ”This is a pts test.”;
      char dst[20];

main()
{
      unsigned char save_wsr;

      init_serio();
      count = 0;
      strcpy( dst, ”This should not be.” );
      single.ptscount = 20;
      single.ptscon.di = 1;
      single.ptscon.si = 1;
      single.ptscon.du = 1;
      single.ptscon.su = 1;
      single.ptscon.b_w = 1;
      single.ptscon.mode = 4;
      single.ptssrc = (void *) src;
      single.ptsdst = (void *) dst;
      save_wsr = wsr;
      wsr = 1;           /* Hwindow 1 */
      ptssel = 1;        /* Enable pts timer overflow. */
      wsr = save_wsr;
      int_mask = 0x01;   /* Enable timer overflow. */
      ioc1 = 0x04;       /* Enable timer1 overflow interrupt. */
      enable();
      asm epts;          /* Enable PTS. */
      while (count < 1); /* Wait for timer ovfl interrupt. */
      printf(”src = %s\n\r”, src);
      printf(”dst = %s\n\r”, dst);
      }

      void timer1(void)  /* Interrupt Handler for vector 0. */
      {
         count ++;
      }
```

interrupt
locate
pts_piha
pts_pihb

pts_piha / pts_pihb

Function

Loads a piha/pihb PTS vector with the address of a PTS control block.

Syntax

```
#pragma pts_piha(struct-name=vector)
#pragma pts_pihb(struct-name=vector)
```

where:

struct-name is a name assigned to the control block.

vector is an interrupt vector number or an interrupt vector address.

Class

General control

Description

These controls are only available for those models which support the piha and pihb interrupt blocks.

Use these pragma controls, combined with the `locate` pragma control, to load the peripheral transaction server (PTS) vectors with the addresses of the PTS control blocks. You must use the `locate` pragma control to locate the PTS control blocks in internal RAM space (`1AH–1FFH`) at an address evenly divisible by eight (8). The *xx*`_funcs.h` header files, where *xx* is one of the processor models (for example `kc_funcs.h`), contain type definitions for the various PTS control blocks.

`interrupt`
`locate`
`pts`

reentrant

Function

Specifies attributes for called functions.

Syntax

`reentrant[(`*`function`* `[,...])] | noreentrant`

where:

`function` is the name of a function declared in the source text.

Abbreviation

`re | nore`

Class

General control

Default

`reentrant`

Description

Use this control to define functions in the module as reentrant. A reentrant function can call itself or be called again through a call loop so the function is activated more than once simultaneously. When the `reentrant` control is in effect, the compiler generates additional code in a function's prolog and epilog to save and restore registers modified by the function. Since registers are preserved, functions can reuse the same locations in register memory even if multiple instances of the functions are active simultaneously.

Specifying the `reentrant` control for a function has the same effect as defining the function with the `reentrant` keyword. The `reentrant` keyword is available for compatibility with earlier versions of C196. The default `extend` control must be in effect for the compiler to recognize the `reentrant` keyword.

You cannot reactivate a nonreentrant function if it is currently active. Use the `noreentrant` control to define the functions in the module as nonreentrant. In this case, the compiler allocates a set of registers for the local variables of the function. After the function exits, the compiler then reuses the same register space for another nonreentrant function depending on the call graph. The compiler generates no additional code to save and restore the registers modified by the function.

Specifying the `noreentrant` control for a function has the same effect as defining the function with the `nonreentrant` keyword. The `nonreentrant` keyword is available for compatibility with earlier versions of C196. The `extend` control must be in effect for the compiler to recognize the `nonreentrant` keyword.

The `noobject` and `notranslate` controls suppress the object file, causing `reentrant` and `noreentrant` to have no effect. However, if you specify `code` with `noobject`, the effects of `reentrant` and `noreentrant` appear in the pseudo–assembly listing of the print file.

The reentrancy specification for a function must precede the function declaration. The first declaration or definition of a function sets the reentrancy specification for that function based on the `[no]reentrant` control in effect for the function or based on the `[non]reentrant` keyword, if specified for the function.

You can specify `reentrant` and `noreentrant` in the compiler invocation and in `#pragma` preprocessor directives throughout the source text. When specified without arguments, these controls affect all functions in the subsequent source text and remain in effect until the compiler encounters the opposite control (`noreentrant` or `reentrant`, respectively) or the end of the source text. Either of these controls specified with an argument list affects only the functions in the argument list.

`extend`
`registers`

regconserve

Function

Disallows file–scope and automatic non–register variables in registers.

Syntax

Select the `EDE | C Compiler Options | Project Options...` menu item. Enable or disable the `Allow allocation of block–scope variables to registers` and/or `Allow allocation of file–scope variables to registers` check box in the `Optimization` tab.

`regconserve[(`*`scope`*`,...)] | noregconserve`

where:

`scope` can be `bscope`, indicating block–scope variables or `fscope`, indicating file–scope variables.

Abbreviation

`rc | norc`

Class

Primary control

Default

`noregconserve`

Description

Use this control to specify whether the compiler can allocate file–scope and automatic (block–scope) non–register variables to registers. If unused register memory remains after all explicit `register` variables have been allocated, the compiler can put frequently used non–register variables in the unused register locations.

You can prevent the compiler from using the remaining register memory for file–scope, block–scope, or all non–register variables. Specifying `regconserve` without arguments keeps all non–register variables out of register memory. The `bscope` argument restricts block–scope non–register variables to the stack and the `fscope` argument restricts file–scope non–register variables to the data segment. Table 4–2 lists where non–register variables can be allocated for each variation of `[no]regconserve`.

The `regconserve` and `noregconserve` controls affect the entire object module. You can specify either of these controls in the compiler invocation or in a `#pragma` preprocessor directive preceding the first line of data definition or executable source text. To override a `#pragma regconserve` or `#pragma noregconserve` specified in the source text, specify a different `[no]regconserve(`*`scope`*`)` control in the compiler invocation.

Control	File–scope Variables	Block–scope Variables
regconserve	data segment	stack
regconserve(bscope, fscope)	data segment	stack
regconserve(bscope)	data segment or registers	stack
regconserve(fscope)	data segment	stack or registers
noregconserve	data segment or registers	stack or registers

Table 4–2: Allocation of non–register variables to registers

The `registers(all)` control conflicts with the `regconserve` control. The use of these two controls results in a fatal error because the compiler cannot both conserve registers and allocate all program variables to registers. In conserving registers, the compiler does not allocate non–register variables to registers.

`registers`

registers

Function

Allocates register space for variables.

Syntax

Select the `EDE | C Compiler Options | Options` *`file`*`...` menu item. Enter the maximum number of bytes for registers in the `Module limit for register memory` field in the `Object` tab.

`registers(`*`num`*`)`

where:

`num` is a number from `0` to `220` or the keyword `all`.

Abbreviation

`rg`

Class

Primary control

Default

`registers(220)`

Description

Use this control to limit the number of bytes of register memory the module can use. 80C196 microcontrollers have 256 bytes of register space except for the 80C196KC that has an additional 256 bytes of registers, and the 80C196KR and 80C196NT/NP that have an additional 512 bytes of registers.

The argument to the `registers` control can be a number from `0` to `220` or the `all` keyword. For example, `registers(145)` limits the module being compiled to 145 bytes of register memory for register variables allocated in the register segment and the overlayable register segment. If you specify `registers(all)`, the compiler uses the on–chip registers only and allocates all variables in the module to register memory. The `registers(all)` control is not the same as the `registers(220)` control. The predefined macro `_REGISTERS_` has the value specified for the `registers` control.

The C196 compiler does not use the additional register space of the 80C196KC or the 80C196KR even if you compile with the `registers(all)` control. The compiler only allows a module to use up to 220 bytes of register space. So in order to use the additional register space, you must have multiple modules and your modules must have enough register variables to occupy the additional register space. The compiler then accesses the additional register space through the use of vertical windowing.

See Section 6.4.3 for additional information on vertical windows.

If you declare more register variables than available registers, the compiler issues a diagnostic message, as follows:

`error` if too many file–scope registers are requested or if `registers(all)` is specified and the number of program variables is greater than the size of the register file.

`warning` if too many block–scope registers are requested.

This error or warning condition can occur, for example, if you specify `registers(all)` and your module contains more than 220 register variables.

The `registers` control affects the entire object module. You can specify this control in the compiler invocation or in a `#pragma` preprocessor directive preceding the first line of data definition or executable source text. To override a `#pragma registers(`*num*`)` specified in the source text, specify `registers` with a different argument in the compiler invocation.

The overlay segment the compiler generates for the module is word–aligned (at least). The compiler adds one more byte to the size of the overlay segment if it has an odd number of bytes. If you specified a limit to the number of registers the module can use, the compiler can use one more byte than what you have specified because of the additional byte.

The `registers(all)` control conflicts with the `regconserve` control. The use of these two controls results in a fatal error because the compiler cannot both conserve registers and allocate all program variables to registers. In conserving registers, the compiler does not allocate non–register variables to registers.

```
reentrant
regconserve
windows
```

relocatabletemps

Function

Tells the compiler to produce external references to temporary register symbols.

Syntax

`relocatabletemps | norelocatabletemps`

Abbreviation

`rt | nort`

Class

Primary control

Default

`norelocatabletemps`

Description

Use this control to tell the compiler to produce external references to temporary register symbols, instead of absolute addresses.

Use the `relocatabletemps` control/pragma to allow the placing of the temporary registers at a non–standard location. References to these registers will then be resolved by the linker.

The size of your object file can increase significantly when you use the control, since there are a great many references to the temporary registers in the generated code. Once resolved b the linker, though, your executable file should be the same size as it would be without this control.

If you use the `tmpreg` control in conjunction with the `relocatabletemps` control, only the names of the temporary registers and frame pointer are changed, and the names corresponding to the `tmpreg` argument will be generated as external references for resolution by the linker.

The argument to the `tmpreg` control must still give an address that is a multiple of four, since the temporary registers must be longword aligned, and the compiler will report an error if this requirement is not met in the `tmpreg` argument. However, when the `relocatabletemps` control is used, the argument to the `tmpreg` control does not have to be accurate, as it otherwise would.

`tmpreg`

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`

where:

`pathprefix` is a string of characters that the compiler 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 compiler 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 compiler issues an error.

An include file is a source text file specified with the `include` control in the compiler invocation or with the `#include` preprocessor directive in the source text. The contents of each include file are inserted into the source text during preprocessing.

The order in which the compiler uses the `searchinclude` and default path prefixes depends on how the include file is specified. When searching for a file specified with the `include(`*`filename`*`)` control or with the `#include "`*`filename`*`"` preprocessor directive, the compiler tests the prefixes in the following order:

1. The source directory.
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 **c196** binary. For example, **c196** is installed in `/usr/local/c196/bin`, then the directory searched for include files is `/usr/local/c196/include`.
5. The current directory (no prefix).

When searching for a file specified with the `#include <`*`filename`*`>` preprocessor directive, the compiler tests the prefixes in the following order:

1. The directories specified by the `searchinclude` list.
2. The directory in the `C196INC` environment variable, if defined.
3. The `include` directory, one directory higher than the directory containing the **c196** binary.
4. The source directory.
5. The current directory (no prefix).

The `searchinclude` and `nosearchinclude` controls affect only the subsequent source text and remain in effect until the compiler 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 in the compiler invocation and in `#pragma` preprocessor directives throughout the source text.

Example

This example demonstrates the paths searched by the compiler 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 compiler invocation as follows:

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

The source text contains the following preprocessor directive:

```
#include "t_locate.h"
```

The source file is in the directory `\proj001\src` for PC (`/proj001/src` for UNIX). The compiler is invoked in the root directory and executed from `/usr/local/c196/bin`. The compiler searches for filenames in the following order (UNIX notation is used):

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

`include`

signedchar

Function

Sign–extends or zero–extends promoted chars.

Syntax

Select the `EDE | C Compiler Options | Project Options...` menu item. Enable or disable the `Treat 'char' variables as unsigned` check box in the `Language` tab.

`signedchar | nosignedchar`

Abbreviation

`sc | nosc`

Class

Primary control

Default

`signedchar`

Description

Use this control to specify that objects declared to be the `char` data type are treated as if declared to be the `signed char` data type. The compiler sign–extends these objects when they are converted to a data type that occupies more memory than the `char` data type.

Use the `nosignedchar` control to specify that objects declared as the `char` data type are treated as if they were declared as the `unsigned char` data type. The compiler zero–extends these objects when they are converted to a data type that occupies more memory than the `char` data type.

The `signedchar` and `nosignedchar` controls do not affect the interpretation of objects specifically declared as either `signed char` or `unsigned char` data types.

The predefined macro `_SIGNEDCHAR_` has the value `1` if `signedchar` is specified and `0` if `nosignedchar` is specified.

If `notranslate` or `noobject` is in effect, the compiler does not generate an object module, so `signedchar` and `nosignedchar` have no effect. However, specifying `signedchar` or `nosignedchar` with `noobject` and `code` can still affect the pseudo–assembly listing in the print file.

The `signedchar` and `nosignedchar` controls affect the entire object module. You can specify either of these controls in the compiler invocation or in a `#pragma` preprocessor directive preceding the first line of data definition or executable source text. To override a `#pragma signedchar` or `#pragma nosignedchar` preprocessor directive, specify the opposite control (`nosignedchar` or `signedchar`, respectively) in the compiler invocation.

```
object
translate
```

speed

Function

Choose between faster code and less code size

Syntax

Select the `EDE | C Compiler Options | Project Options...` menu item. Choose a `Speed level` in the `Optimization` tab.

`speed( level )`

where:

`level`	is the value 0, 1, or 2. The values correspond to no fast code, faster code and fastest code respectively.

Abbreviation

`sp`

Class

Primary control

Default

`speed(0)`

Description

Use this control to tell the compiler to choose between faster code and less code size.

symbols

Function

Generates or suppresses identifier list in print file.

Syntax

Select the `EDE | C Compiler Options | Project Options...` menu item. Enable or disable the `Include identifier list` check box in the `Listing` tab.

`symbols | nosymbols`

Abbreviation

`sb | nosb`

Class

Primary control

Default

`nosymbols`

Description

Use this control to include in the print file a table of all identifiers and their attributes from the source text. Use the default `nosymbols` control to suppress the table.

The `xref` control causes the compiler to generate a cross–referenced symbol table even if the `nosymbols` control is specified. If `noprint` or `notranslate` is in effect, the compiler does not generate a print file and `symbols` has no effect.

The `symbols` and `nosymbols` controls affect the entire object module. You can specify either of these controls in the compiler invocation or in a `#pragma` preprocessor directive preceding the first line of data definition or executable source text. To override a `#pragma symbols` or `#pragma nosymbols` specified in the source text, specify the opposite control (`nosymbols` or `symbols`, respectively) in the compiler invocation.

`print`, `translate`, `xref`

tabwidth

Function

Specifies the number of characters per tab stop in the print file.

Syntax

`tabwidth(`*`width`*`)`

where:

`width` is a value from `1` to `80`. This value is the number of characters from tab stop to tab stop in the print file.

Abbreviation

`tw`

Class

Primary control

Default

`tabwidth(4)`

Description

Use this control to specify the number of characters between tab stops in the print file.

The `noprint` and `notranslate` controls suppress the print file, causing the `tabwidth` control to have no effect.

The `tabwidth` control affects the entire source text. You can specify this control in the compiler invocation or in a `#pragma` preprocessor directive preceding the first line of data definition or executable source text. To override a `#pragma tabwidth(`*`width`*`)` specified in the source text, specify `tabwidth` with a different *`width`* in the compiler invocation.

`pagelength` `title`
`pagewidth` `translate`
`print`

title

Function

Specifies the print file title.

Syntax

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

`title(`*`"string"`*`)`

where:

`string` is the print file title.

Abbreviation

`tt`

Class

Primary control

Default

`title(`*`"primary_source_filename"`*`)`

Description

Use this control to specify the print file title. A title can be up to 60 characters long. To specify no title, use at least one blank space character in the title string. Do not use the null string.

The compiler uses the primary source filename, without the filename extension as the title. For example, if `myprog.c` is the primary source file, `myprog` is the print file title.

The compiler places the title at the top of each page of the print file. A narrow page width can cause the compiler to truncate a long title.

The `noprint` and `notranslate` controls suppress the print file, causing the `title` control to have no effect.

The `title` control affects the entire print file. You can specify this control in the compiler invocation or in a `#pragma` preprocessor directive preceding the first line of data definition or executable source text. To override a `#pragma title("string")` specified in the source text, specify `title` with a different *`string`* in the compiler invocation.

`pagelength`	`tabwidth`
`pagewidth`	`translate`
`print`	

tmpreg

Function

Locates the temporary registers.

Syntax

`tmpreg(`*`addr`*`)`

where:

`addr` is a valid absolute address in decimal or hexadecimal format.

Abbreviation

`tr`

Class

Locating control

Default

`tmpreg(1CH)`

Description

Use this control to locate the temporary registers, namely `TMPREG0`, at a different address.

See Chapter 6 for more information on `TMPREG0` and see Chapter 10 for more information on the `?FRAME01` variable.

By default, the temporary registers are located at address `1CH`. To relocate these registers, specify an address in the *`addr`* parameter using decimal or hexadecimal format. For example, the address `44` in decimal is equivalent to `2CH` or `0x2C` in hexadecimal format. The address you specify must be on a double–word boundary.

If you specify this control, `TMPREG0` appears as `TMPR`*`xx`* and `?FRAME01` appears as `?FRAME`*`xx`* in the listing file. The placeholder *`xx`* stands for the hexadecimal address where the registers are currently located. For example, if you locate the temporary registers to `2CH`, `TMPREG0` appears as `TMPR2C` and `?FRAME01` appears as `?FRAME2C`.

To correctly use this feature, you must link to your application, using RL196, a module containing a declaration that reserves eight bytes of memory space (sixteen bytes for `model(nt)`) at the address specified by *`addr`*. You can create this module with ASM196. The compiler uses these eight bytes as the new temporary registers. Name the variable `TMPR`*`xx`* where *`xx`* is the hexadecimal address specified by *`addr`*. For example, if you want to locate the temporary registers to `2CH`, the variable name must be `TMPR2C`. See the example section for instructions on how to create this module.

This control is particularly useful for multi–tasking applications. The control allows each task to have its own set of temporary registers.

Examples

As mentioned in the discussion, you must reserve an eight–byte (or sixteen–byte) memory space to be used as the new temporary registers, so that no other module attempts to use these eight bytes. The following example shows how to declare this variable in assembly language. This example also explains how to assemble, compile, and link the module to your application.

Create an assembly module called `tmpreg.a96`, for this example, with the declaration shown below. This example locates the temporary registers at location `2CH` and allocates a relocatable register for the frame pointer.

```
public TMPR2C
rseg at 2CH
TMPR2C  equ  $
dsl 2

rseg
?FRAME2C equ $
dsw 1
end
```

Assemble this module. See the *80C196 Assembler User's Guide*, for the ASM196 assembly invocation syntax. Compile your C196 programs with the `tmpreg(2CH)` or `tmpreg(0x2C)` control. This control tells the compiler that the temporary registers are now located at `2CH`. During the link phase, link the ASM196 object module with your C196 object modules, as follows:

```
rl196 cprg1.obj, cstart.obj, cprg2.obj, tmpreg.obj, c96.lib
```

TMPREG0 can be pulled in by using MUL, DIVL etc. If that is not what you want, use the following example:

```
   TMPR2C  equ  $
   TMPREG0 equ  $
```

```
?FRAME01
extratmp control
locate control
relocatabletemps control
TMPREG0
```

translate

Function

Compiles or suppresses compilation after preprocessing.

Syntax

`translate | notranslate`

Abbreviation

`tl | notl`

Class

Invocation control

Default

`translate`

Description

Use this control to cause compilation to continue after preprocessing. Use the `notranslate` control to cause compilation to cease after preprocessing. Translation includes parsing the input, checking for errors, generating code, and producing an object module.

The `notranslate` control suppresses the print and object files, causing all object controls and all listing controls, except for `preprint`, to have no effect. If `notranslate` is in effect, preprocessing diagnostic messages appear at the console.

The `translate` and `notranslate` controls affect the entire compilation. You can specify either of these controls in the compiler invocation.

```
object
preprint
```

type

Function

Generates or suppresses type information in the object module.

Syntax

Select the `EDE | C Compiler Options | Project Options...` menu item. Enable or disable the `Generate type information` check box in the `Debug` tab.

`type | notype`

Abbreviation

`ty | noty`

Class

Primary control

Default

`type`

Description

Use this control to include type information for public and external symbols in the object module. Type information can be useful to other tools in the application development process. A linker uses type information to perform type checking across modules. A debugger or an emulator uses type information to display symbols according to their attributes.

To include all possible information for symbolic debugging, use `type` with the `debug` control, as described in the `debug` entry in this chapter.

Use the `notype` control to suppress type information, reducing the size of the object module.

The `noobject` and `notranslate` controls suppress the object file, causing `type` and `notype` to have no effect.

The `symbols` and `xref` controls are the print file counterparts to the `type` control. The `symbols` control puts a listing of all identifiers and their types into the print file. The `xref` control adds line–number cross–reference information to the symbol table listing.

The `type` and `notype` controls affect the entire object module. You can specify either of these controls in the compiler invocation or in a `#pragma` preprocessor directive preceding the first line of data definition or executable source text. To override a `#pragma type` or `#pragma notype` specified in the source text, specify the opposite control (`notype` or `type`, respectively) in the compiler invocation.

```
debug                   translate
object                  xref
symbols
```

varparams

Function

Specifies variable–parameter list calling convention.

Syntax

`varparams[(`*`function`* `[,...])]`

where:

`function` is the name of a function defined in the source text.

Abbreviation

`vp`

Class

General control

Default

`varparams`

Description

Use this control to cause the specified functions to use the variable parameter list (VPL) calling convention. The VPL calling convention provides more flexibility than the FPL calling convention. See the `fixedparams` control for more information on the FPL calling convention. Use the VPL calling convention for functions that take a variable number of parameters.

A function's calling convention dictates the sequence of instructions that the compiler generates to manipulate the stack and registers during a call to the function. Code generated for the VPL calling convention performs the following sequence of operations:

1. The calling function pushes the arguments onto the stack with the rightmost argument pushed first.
2. The calling function transfers control to the called function.
3. The called function executes.

4. The called function returns control to the calling function.
5. The calling function removes the arguments from the stack.

The calling convention specification must precede the function declaration. The first declaration or definition of a function sets the calling convention for that function based on the `fixedparams` or `varparams` control in effect for the function, or based on the `alien` keyword or the comma and ellipsis (`,...`), if specified for the function. The comma and ellipsis indicate that the number of parameters to the function has no limit. In this case, `varparams` is in effect.

The `notranslate` and `noobject` controls suppress the object file, causing `varparams` to have no effect. However, if you specify the `code` control with the `noobject` control, the effect of `varparams` does appear in the pseudo–assembly code listing.

You can specify `varparams` in the compiler invocation and in `#pragma` preprocessor directives throughout the source text. When specified without arguments, this control affects all functions in the subsequent source text and remains in effect until the compiler encounters the opposite control (`fixedparams`) or the end of the source text. The `varparams` control specified with an argument list affects only the functions in the argument list.

See the `fixedparams` control for more information on how the FPL calling convention differs from the VPL calling convention.

More than one explicit calling convention specification for any one function causes a warning. A warning occurs if a function in the source text is explicitly declared with a variable parameter list and is named in the function list for the `fixedparams` control.

```
#pragma fixedparams(x)

int x (int i,...)
{
}
```

In this example, `varparams` is in effect.

Examples

1. The following control in the compiler invocation specifies the default variable parameter list convention (VPL) for all functions in the source text except the `plm_fcn` function:

```
fixedparams(plm_fcn)
```

2. The following `#pragma` preprocessor directive has the same effect as the control in the above example:

```
#pragma fixedparams(plm_fcn)
```

3. The following combination of controls in the compiler invocation specifies the fixed parameter list convention (FPL) for all functions in the source text except the `native` function:

```
fixedparams varparams(native)
```

4. The following `#pragma` preprocessor directives have the same effect as the controls in the above example:

```
#pragma fixedparams
#pragma varparams(native)
```

```
code                object
extend              translate
fixedparams
```

warning_true_false

Function

Enables the 'comparison always returns TRUE' and 'comparison always returns FALSE' warnings.

Syntax

`warning_true_false | nowarning_true_false`

Abbreviation

`wt | nowt`

Class

Primary control

Default

`warning_true_false`

Description

Use this control to generate the 'comparison always returns TRUE' or the 'comparison always returns FALSE' warnings. These warnings appear for instance when comparing two constants or when comparing a negative number with an unsigned integer.

Use the `nowarning_true_false` control to suppress these warnings.

win1_32, win1_64

Function

Combination of a pragma `locate` and pragma `overlay`. Only if WSR1 is present.

Syntax

```
#pragma win1_32(var1=addr,regvar1)
#pragma win1_64(var1=addr,regvar1)
```

where:

var1 is a valid symbol name.

addr is a valid absolute address.

regvar1 is a valid register symbol name.

Abbreviation

`v3 / v6`

Class

General control

Description

Use this pragma control to locate one or more register symbols to absolute addresses, and mark them as overlayable. Use this control only if WSR1 is present in the selected processor, otherwise use the `win32`, `win64` or `win128` control. The `win1_32` and `win1_64` control must follow the declaration of the symbols. For example, the following pragma control line locates `var1` to address `220H` and `regvar1` at address `060H`:

```
int var1;
register int regvar1;
#pragma win1_64(var1=0x220,regvar1)
```

`locate`
`overlay`
`windows`

win32, win64, win128

Function

Combination of a pragma `locate` and pragma `overlay`.

Syntax

```
#pragma win32(var1=addr,regvar1)
#pragma win64(var1=addr,regvar1)
#pragma win128(var1=addr,regvar1)
```

where:

var1 is a valid symbol name.

addr is a valid absolute address.

regvar1 is a valid register symbol name.

Abbreviation

```
w3 / w6 / w1
```

Class

General control

Description

Use this pragma control to locate one or more register symbols to absolute addresses, and mark them as overlayable. This control must follow the declaration of the symbols. For example, the following pragma control line locates `var1` to address `220H` and `regvar1` at address `0E0H`:

```
int var1;
register int regvar1;
#pragma win64(var1=0x220,regvar1)
```

```
locate
overlay
windows
```

windowram

Function

Specifies the area(s) of memory from which to allocate windowed variables.

Syntax

Select the `EDE | C Compiler Options | Options` *`file`*`...` menu item. Enter one or more memory areas in the `Specify the memory area(s) of windowed variables for this module` field in the `Code` tab.

`windowram(` *`startaddr`* `–` *`endaddr`* `[, ...] )`

where:

`startaddr` is a valid absolute address.

`endaddr` is a valid absolute address.

Abbreviation

`wr`

Class

General control

Description

Use this control to specify the area(s) of memory from which to allocate windowed variables. Any number of ranges can be specified within the parentheses, and the `windowram` control and/or pragma may be specified any number of times. The ranges specified must not overlap, and must be within the range of mappable memory for the selected model.

When the compiler allocates an object from these ranges, it first tries to select an area from the beginning of a properly aligned range. If it cannot find a properly aligned range, it will take an area from within the first range with enough space remaining.

Example:

```
#pragma windowram(0x100–0x17F)
```

See Section 6.4.3 for more information on vertical windows and the use of the special keywords `_reg`, `_win` and `_win1`.

```
hold
model
```

windows

Function

Specifies that the whole application uses vertical windows.

Syntax

`windows[( [no]hold )] | nowindows`

Abbreviation

`wd | nowd`

Class

Primary control

Default

`nowindows`

Description

Use this control to use the additional registers of the processors that support vertical windows through the vertical windowing feature of these microcontrollers. This control causes the compiler to generate instructions to save and set the `wsr` register in the prolog and restore the `wsr` register in the epilog of all functions, except for `static` and public functions which have no local register variables and no calls to other functions. If you are using the HOLD/HOLDA protocol along with vertical windowing, specify the `hold` parameter. This parameter causes the compiler to generate additional code to preserve the HOLDEN bit of the Window Select Register (WSR). Specifying `windows` without any parameter is equivalent to specifying `windows(hold)`. If you are not using the HOLD/HOLDA protocol, specify the `nohold` parameter to reduce the amount of overhead code.

The WSR management code allows access to local and static register variables located in the mapped area of the register file and above (from `80H` or `0C0H` or `0E0H` depending on the window size). Public register variables allocated in the register segment are restricted to the registers below the mapped area (below `80H` or `0C0H` or `0E0H` depending on the window size). This allocation scheme allows access to these variables without swapping the `wsr`.

See Section 6.4.3 for more information on vertical windows.

If you specify the `hold` parameter, the compiler produces the following WSR management code in the prolog:

```
ldbze Tmp0,WSR
push  Tmp0
andb  WSR,#80H     /* to retain hlden in wsr */
orb   WSR,?WSR
```

Otherwise, with the `nohold` parameter, the following code is produced:

```
ldbze Tmp0,WSR
push  Tmp0
ldb   WSR,?WSR
```

The compiler produces the following code in the epilog, with or without the `hold` parameter:

```
ldb  WSR,[SP]
pop  R0
```

Your application must consist of several modules to take advantage of the vertical windowing feature. You can then determine the register windowing requirement by adding the sum of the overlayable register bytes from the end of every print file.

See Section 6.4.2 for more information on how to calculate the number of register bytes needed by a module.

If your application only consists of one module, your application does not use the extra register space since a module at most only uses 220 register bytes.

The `windows` control can only be used with processors that support vertical windows. Otherwise the compiler generates a fatal error.

`model`
`reentrant`
`regconserve`
`registers`
Vertical Windowing in Section 6.4.3

wordalign

Function

Specifies that no longword alignment be done.

Syntax

`wordalign | nowordalign`

Abbreviation

`wa | nowa`

Class

Primary control

Default

`nowordalign`

Description

Use this control to prevent the compiler from aligning objects to longword boundaries. This control causes the compiler to place all objects requiring word–alignment or longword–alignment on word boundaries, but not necessarily on longword boundaries. Using this control will allocate your data in the same order that it was allocated by C196 version 2.*x*.

`oldobject`

xref

Function

Specifies symbol table cross–reference in listing.

Syntax

Select the `EDE | C Compiler Options | Project Options...` menu item. Enable or disable the `Include identifier cross–reference` check box in the `Listing` tab.

`xref | noxref`

Abbreviation

`xr | noxr`

Class

Primary control

Default

`noxref`

Description

Use this control to add cross–reference information to the symbol table listing in the print file. Use the default `noxref` control to suppress the cross–reference information.

The print file lists the cross–reference line numbers on the far right with the data or function type under the `ATTRIBUTES` column in the symbol table listing. The cross–reference line numbers refer to the line numbers in the source text listing in the print file. An asterisk (`*`) indicates the line where the object or function is declared.

Specifying `noprint` or `notranslate` suppresses the print file, causing `xref` to have no effect. If the print file is produced, specifying `xref` generates a cross–referenced symbol table even if `nosymbols` is specified.

The `xref` and `noxref` controls affect the entire source text. You can specify either of these controls in the compiler invocation or in a `#pragma` preprocessor directive preceding the first line of data definition or executable source text. To override a `#pragma xref` or `#pragma noxref` specified in the source text, specify the opposite control (`noxref` or `xref`, respectively) in the compiler invocation.

```
print
symbols
translate
```

zero

Function

Specifies whether the compiler zeroes uninitialized variables in relocatable data segments.

Syntax

Select the `EDE | C Compiler Options | Project Options...` menu item. Enable or disable the `Clear unititialized RAM variables in relocatable segments` check box in the `Code` tab.

`zero | nozero`

Abbreviation

`zr | nozr`

Class

Primary control

Default

`zero`

Description

Use the `zero` control to allow the compiler to zero uninitialized variables in relocatable segments. The default setting matches the setting of the `init` control. This control is only valid for the (default) OMF version 3.2. At startup (reset), library module `cstart` processes the initialization table: it copies the initial constant data to the corresponding variables, and zeroes the uninitialized variables.

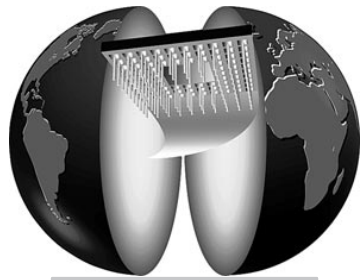
Use the `nozero` control/pragma to prevent the generation of zeroing entries in the initialization tables for relocatable segments (ordinary variables).

`noinit` forces `nozero`.

`abszero`
`init`

CHAPTER 5

STARTUP CODE



TASKING
Quality Development Tools Worldwide

CHAPTER 5

This chapter describes the startup files (cstart.a96 and _main.c) which are supplied with your C196 compiler.

5.1 CONTENTS OF CSTART.A96

When you link your program, you need to include the file cstart.obj as one of your input files. For each model the corresponding cstart.obj is provided in each subdirectory of the lib directory. If you like, you can customize the source file, cstart.a96 (nt_start.196 for model NT, np_start.196 for model NP), which is provided in the src subdirectory of the lib directory. It looks something like this:

```
STARTUP       MODULE  CMAIN

      RSEG
SP    EQU     018H:WORD

      CSEG    AT 2080H

      EXTRN   _main:NULL

cstart:
      PUBLIC  cstart
      LD      SP,#STACK

      LJMP    _main  ; _main calls the others

_exit:
      PUBLIC  _exit
      BR      _exit
      END
```

The C language treats each routine as an ordinary function, including main(). For that reason, you can use the startup code as the main module for your C196 modules. When you link the cstart.obj file with your modules, the linker creates an absolute code segment, which becomes the main module segment, containing a long jump to the _main (see 5.2) routine.

You can tailor the cstart.a96 file according to your specific needs and the environment under which your application executes. For example, if you do not need the initializations done in _main(), you can substitute the long jump to _main with a long jump to main(). This will directly call your main() routine.

The models NT and NP have their own specific startup files, called `nt_start.196` and `np_start.a96` respectively. The comments in these files explain how to create the `cstart.obj` file.

5.2 CONTENTS OF _MAIN.C

The module `_main.c` contains the routine `_main()`. This routine is used to initialize different variables before their first use. The `_main.obj` is included in both `c96.lib` and `c96fp.lib`. The module `_main.c` is provided in the `src` subdirectory of the `lib` directory.

Depending on your application, the routine `_main()` calls the following subroutines:

init_serio()

Initialization routine to initialize the serial port. This routine is only necessary if you use `putch()` to write to the serial port or `getch()` to read from the serial port. If you use any third party vendors which include their own `putch()` which do not use the serial port, you will not need this call.

_imain()

Initialization routine for initialization of global variables. This routine is not needed if you do not have any initialized global variables.

main()

The `main()` routine from your application.

exit()

The ANSI–C compatible `exit()` routine. This includes the support of `atexit()` and closes all open streams.

__exit()

This routine is the `exit()` routine without support of `atexit()` and without stream support.

If you cannot use the default `omf(2)` control, it might be necessary for you to also call some of the following subroutines. Note that you should also recompile several library routines for `omf(1)`.

fpinit()

Initialization routine for floating point operations. The call is only needed if you use floating point calculations. This call is only needed in the `_main()` routine which is provided in `c96fp.lib`.

init_stdio()

Initialization routine to set up the streams 'stdin', 'stdout' and 'stderr'. These streams are used for the `printf()` routines and the `scanf()` routines.

init_atexit()

Initialization routine to set up the `atexit()` routine. This is only necessary if you use the ANSI–C compatible `exit()` routine.

init_malloc()

Initialization routine for dynamic memory allocation. This call is only needed if you use dynamic memory allocation.

The `_main()` routine can be tailored to your specific needs by adding `#define` statements. The `_main()` routine as is supplied within the libraries contains the minimum required calls to execute an application. It only calls `_imain()`, `main()`and `__exit()`.

5.3 WRITING YOUR OWN STARTUP CODE

You can write your own startup code using the ASM196 assembly language. You must declare your module to be the main module by using the `cmain` attribute. Load the stack pointer with the address of the stack. Initialize any other registers you need, call any initialization routines you need, then do a long call (`lcall`) to your main C function. Your ASM196 main module must contain at least the following lines:

```
cstart  module  cmain

sp      equ 18H:word

cseg    at 2080H
extrn   main

ld      sp,#stack
lcall   main
rst     ; reset the processor if program returns
        ; to cstart
end
```

Assemble the file and then link it with your C object files. Note that this example startup code does not use the `_main()` routine.

5.4 WRITING YOUR OWN _MAIN ROUTINE

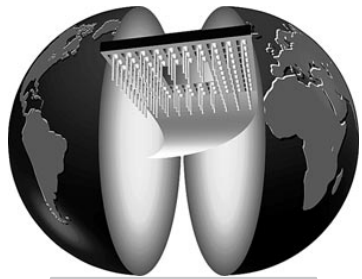
You can also write your own `_main()` routine. This routine must contain at least a call to `main()` and a call to `__exit`:

```
extern void     main(void)
extern void     __exit(int)

void _main(void)
{
     main();

     __exit(0);
}
```

Compile this file and either link it with your C object files, or use the **lib196** tool to replace the `_main.obj` in the library files with your own `_main.obj`.

CHAPTER 6

PROCESSOR REGISTERS



TASKING
Quality Development Tools Worldwide

6 CHAPTER

The 80C196 family of microcontrollers contains special function registers (SFRs) for processor hardware manipulation and a register file for faster operand access. This chapter describes the variables declared in the *xx*_sfrs.h header files (where *xx* represents the processor as specified with the model(*xx*) control) for using the SFRs and explains how to use the C196 compiler for efficient register allocation.

6.1 REGISTER MEMORY

Figure 6–1 shows the register memory layout of the 80C196KB processor. This layout is the same as the register memory layout of the 8096–90 and 8096BH. Not shown in the figure is the additional register space of the 80C196KC, the 80C196KR, and the 80C196NT microcontrollers. The 80C196KC, 80C196KR, and 80C196NT have 256 bytes of additional registers from 100H through 1FFH or higher. The C196 compiler tries to allocate variables to the register memory as much as possible, if the registers(all) control is in effect, so that instructions can be more compact and can execute faster. Some of these locations have dedicated or default uses, as follows

- Special function registers (SFRs) are defined in *xx*_sfrs.obj. For an explanation of the structure and use of the SFRs, see the *80C196 Assembler User's Guide* or the *Embedded Microcrontrollers and Processors Handbook*, listed in *Related Publications*.
- The stack pointer (SP), in locations 18H and 19H, indicates the address of the top of the stack.
- Temporary registers, in locations 1CH through 23H (or 2BH), are used for intermediate calculations and for returning the value of a typed function. The compiler treats this section of memory as the TMPREG0 (and TMPREG8, if needed) register variable. You can use the tmpreg control to change the location of the temporary registers.

See Chapter 4 for more information about the tmpreg control.

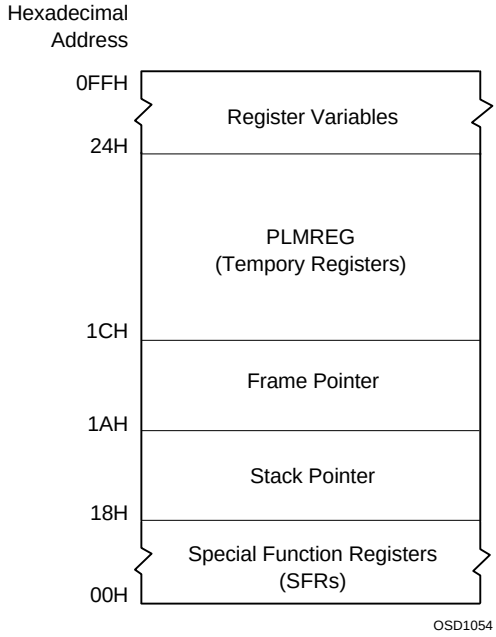


Figure 6–1: 80C196KB register memory

The 80C196 processors contain on–chip peripherals, listed in Table 6–1, controlled by the special function registers (SFRs) located in the first 24 (18H) bytes of the register file. The C196 header files and libraries define symbols, macros, and functions to read and write the SFRs.

I/O Function	Description
high–speed input (HSI)	Automatically records events; records the line that had an event and the time when the event occurred.
high–speed output (HSO)	Automatically triggers events and real–time interrupts; sends messages to turn on, turn off, start processing, or .reset devices
pulse width modulation (PWM)	Outputs signals to drive motors or analog circuits; replaces an analog output signal.
A–to–D converter	Provides a 10–bit analog–to–digital converter that can use any one of eight input channels.
watchdog timer	Resets the processor if not written to within the designated time.

I/O Function	Description
serial port	Provides one synchronous mode with rates up to 1.5M baud or three asynchronous modes with rates up to 187.5K baud.
standard I/O lines	Provide interfaces to the external world when other special features are not needed.

Table 6–1: Major I/O functions

6.2 ACCESSING SPECIAL FUNCTION REGISTERS

The *xx*_sfrs.h header files declare variables that you can use to access the SFRs.

To manipulate the program status word (PSW), you must write an assembly language routine to get the value of the PSW. In this example, the register variable `flags` is the destination of the value. Define `flags` as a register integer variable.

```
register int flags;
```

Using in–line assembly code, the assembly language source text must include the following instructions:

```
asm pushf;           /* push contents of PSW onto stack */
asm ld flags, [sp];  /* load PSW value from stack into flags */
asm popf;            /* and restore all flags */
```

Six functions in `c96.lib` in the processor specific `lib` directory (with their function prototypes in *xx*_funcs.h) directly manipulate the processor hardware, as follows:

`enable` enables interrupts.

`disable` disables interrupts.

`enable_pts` enables PTS interrupts.

`disable_pts` disables PTS interrupts.

`idle` puts the processor into idle state (80C196 processor only).

`powerdown` puts the processor into power–down state (80C196 processor only).

The *Embedded Microcontrollers and Processors Handbook*, listed in *Related Publications*, describes the processor idle and powerdown modes.

6.3 TMPREG0

The `TMPREG0` variable, defined as a two long–word variable in the `c96.lib` library, is used to hold the following:

- Intermediate results during computation.
- Return values of typed (non–void) functions.

The compiler assigns the name `TMPREG0` to the address `1CH` in the register segment, by default, and gives `TMPREG0` a `null` attribute. You can change the location of `TMPREG0` by using the `tmpreg` control, as described in Chapter 4. The `null` attribute allows any function to use `TMPREG0` without having to specify a data type, as described in the *80C196 Assembler User's Guide*, listed in *Related Publications*.

If the compiler needs more than eight bytes of work registers (this can only happen with 24–bit models), it will use the next eight bytes, normally at address `24H`, and assign the name `TMPREG8` to them.

`TMPREG0` is declared in assembly language as follows:

```
public TMPREG0
rseg    at     1CH
TMPREG0 EQU    $
dsl     2
end
```

6.4 REGISTER VARIABLES

You can use the `register` attribute in a variable declaration to allocate a variable in register memory. The compiler allocates automatic register variables in the overlayable register segment and allocates register variables with static duration in the register segment. If the windows control is in effect, the compiler also allocates static register variables in the overlayable register segment. A `register` variable can be any data type and is read or written using 8–bit addressing instead of 16–bit addressing.

6.4.1 USING THE EXTEND CONTROL

If you specify the `extend` control, the compiler allows more flexibility in the operation of the `register` attribute, as follows:

- You can declare file–scope variables with the `register` storage class. That is, you can declare `register` variables outside of any block.
- The compiler uses register memory to optimize data access for variables not explicitly declared with the `register` keyword, allocating variables to registers in the following order:

1. All variables explicitly declared with `register` are allocated first. If it runs out of register memory before all the explicitly declared `register` variables have been allocated, the compiler generates an error message.

2. If register memory remains after all the explicitly declared `register` variables have been allocated, the compiler can allocate frequently used variables to registers as specified by the `regconserve` and `registers` controls. See Chapter 4 for the description of each control.

6.4.2 ALLOCATING AND OVERLAYING REGISTERS

The maximum number of registers available for variable allocation for a module is 220 bytes. You can further limit this number by specifying the `registers` control. However, you can declare more register variables in a program than the number of registers available in the processor hardware. The C196 compiler can reuse the registers used by the local register variables of one function for another function, provided the functions are never simultaneously active. This process of reusing registers is called overlaying. The C196 compiler overlays registers within each module. The RL196 relocator and linker can also be used to overlay registers between modules through the use of the `regoverlay` control.

The compiler generates prolog and epilog code, and it overlays registers differently for reentrant and nonreentrant functions. The two functions differ as follows:

- Reentrant functions contain overhead code and use a smaller number of registers because the functions share the same register space. The prolog and epilog of a reentrant function contain code that saves and restores the values of registers used by the function. The compiler can then overlay (reuse) the preserved registers. For example, if functions `f`, `a`, and `b` are reentrant, the compiler can overlay all the registers used by `f`, `a`, and `b`.
- Local variables of a function become undefined once the function finishes its execution. The C196 compiler allocates a set of registers specifically for the function's local register variables, so the compiler does not need to generate the code to preserve the register values in the prolog and epilog. The compiler attempts overlaying by using the critical–path analysis call graph to determine which functions are active simultaneously and which are not. For example, if function `f` calls functions `a` and `b`, and `a` and `b` do not call each other, the compiler can overlay the registers used by `a` and `b` but `f` must use its own separate registers.

You can specify a function to be reentrant either by using the `reentrant` storage class in the function declaration or by specifying the `reentrant` control. Since the `reentrant` storage class is a non–ANSI Intel extension to the C language, the `reentrant` control is recommended for writing portable programs. Similarly, you can specify a function to be nonreentrant either by using the non–ANSI `nonreentrant` attribute in the function declaration or by specifying the `noreentrant` control.

Since an interrupt function can be active at any time, simultaneously with any other function, the compiler treats interrupt functions as reentrant functions.

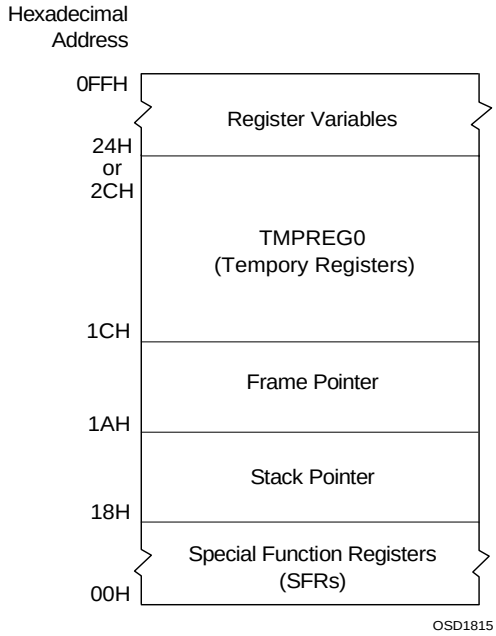


Figure 6–2: Calculating register memory requirements

You can calculate the number of bytes of register memory needed by a module as in the following example, illustrated in Figure 6–2:

- The compiler allocates two bytes of register memory for the `A` function. These two bytes are locations `24H` and `25H`.
- The compiler allocates six bytes of register memory for the `B` function. Since `A` and `B` are never simultaneously active, `B` can use the same two bytes that `A` uses. The `B` function uses locations `24H` through `29H`.
- The compiler allocates six bytes of register memory for the `C` function. Since `A` calls `C`, `C` cannot use register locations allocated for `A`. However, since `C` is never active at the same time as function `B`, `C` can reuse locations used by `B`. The `C` function uses locations `26H` through `2BH`.
- The compiler allocates nine bytes of register memory for the `D` function. Since both `A` and `B` call `D`, `D` cannot use register locations allocated for `A` or `B`. The `D` function uses locations `2AH` through `32H`.

- The compiler allocates three bytes of register memory for the E function. Since B calls E, E cannot use register locations allocated for B. Since E is not active at the same time as either C or D, E can reuse locations used by these functions. The E function uses locations 2AH through 2CH.
- The compiler allocates four bytes of register memory for the F function. Since A calls C and C calls F, F cannot use register locations allocated to either C or A. Also, since A and B call D and D calls F, F cannot use register locations allocated to either D, B, or (again) A. The F function uses locations 33H through 36H.
- The compiler allocates three bytes of register memory for the G function. Since B calls E and E calls G, G cannot use register locations allocated to B or E. The G function uses locations 2DH through 2FH.

If the module represented in Figure 6–2 is the only code running in the processor, the module uses locations 24H through 36H of register memory, that is, 19 of the maximum 220 bytes allowed by the processor. If a different module is already located in that part of register memory, the module in Figure 6–2 occupies the same number of bytes (19) but in different locations. The registers used by any given module are not necessarily contiguous.

6.4.3 SUPPORT FOR VERTICAL WINDOWS

Many of the 80C196 processors have 256 bytes of additional registers or more. Register windowing enables the compiler to access the additional registers using the 8–bit direct–addressing mode instead of the 16–bit addressing mode. This 8–bit addressing mode results in faster and tighter code generation. The available two types of windows are Horizontal Windows (HWindows) and Vertical Windows (VWindows). This section focuses on Vertical Windows. See the *Embedded Microcontrollers and Processors Handbook*, listed in *Related Publications*, for more information on register windowing.

The 80C196 processor family provides vertical windowing so that you can use the additional bytes of RAM as general–purpose registers using the 8–bit direct–addressing mode. VWindows differ from HWindows in that you can still access these registers through 16–bit addressing using indexed or indirect–addressing mode since VWindows reside in the same address space. You can use VWindows to map sections of the register file as 32–, 64–, or 128–byte windows onto the top 32–, 64–, or 128–byte portion of the register file. Use the Window Select Register (`WSR`) to switch between windows.

The C196 compiler uses the additional registers for the block–scope and static register variables allocated in overlay segments. Block–scope variables are variables declared within non–reentrant functions. Figure 6–3 shows the register allocation scheme that the linker uses to locate register and overlay segments on the 80C196KC processors.

There are two distinct methods provided by the compiler for using vertical windows: one using the `windows` control, and one using windowed parameters.

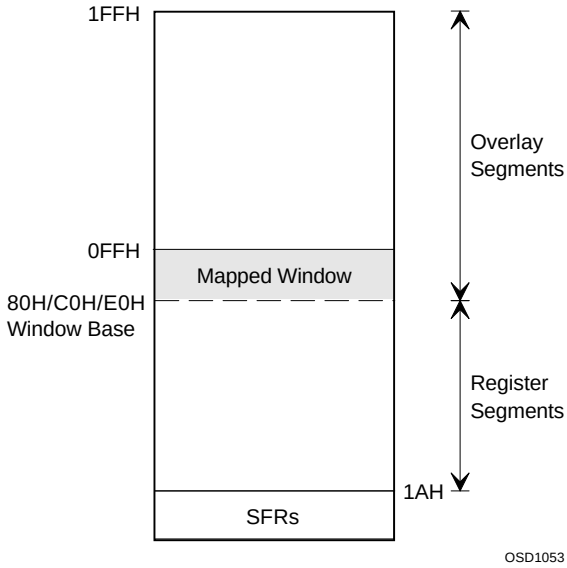


Figure 6–3: 80C196KC register allocation scheme

6.4.3.1 USING THE WINDOWS CONTROL

To read or write to the local register variables, the C196 compiler generates the WSR management code in the prolog and epilog of all public functions compiled with the `windows` control, except for functions that do not contain local register variables, do not access static variables, and do not call another function. The `windows` control must be in effect in order for the WSR management code to be generated.

If your application is using the HOLD/HOLDA protocol along with vertical windowing, specify the `windows` control with the `hold` parameter. Specifying `windows` without any parameter is equivalent to specifying `windows(hold)`. The compiler then generates the WSR management code, which saves the HLDEN bit in the WSR in the prolog, as follows:

```
ldbze Tmp0, WSR
push  Tmp0
andb  WSR, #80h       /* to retain HLDEN in WSR */
orb   WSR, ?WSR
```

If you are not using the HOLD/HOLDA protocol, specify the `windows` control with the `nohold` parameter. The compiler then produces a reduced amount of overhead code as follows:

```
ldbze Tmp0, WSR
push  Tmp0
ldb  WSR,?WSR
```

For the epilog, the compiler produces the following code:

```
ldb  WSR,[SP]
pop  R0
```

For more compact code, declare functions as `static` when appropriate. This declaration suppresses the generation of the WSR management code in the prolog and epilog of these functions.

The linker first locates the global variables allocated in register segments below the window base selected, in the lower 256 registers, during link–time. This scheme enables access to a global variable without regard to the WSR. Then, the linker locates the overlay segments after all register segments are located. If there are gaps between register segments, the linker attempts to fill the gaps with overlay segments of the right size. The linker selects the window size based on the last (highest) address space occupied by the last register segment. The last occupied address must fall below `80H` (the 128–byte window) or `0C0H` (the 64–byte window) or `0E0H` (the 32–byte window). Otherwise, the linker sets the WSR to zero, takes no action on the additional registers, and generates a warning stating that there are too many registers.

When linking modules together, specify the range of the registers available to the application with the RL196 `registers` control and the desired window size through the RL196 `windowsize` control. See the *80C196 Utilities User's Guide*, listed in *Related Publications*, for more information on these controls.

To efficiently use VWindows, your program must meet the following requirements:

- The size of all but one of the overlay segments must be smaller than or equal to the window size. The one overlay segment can be bigger than the window size providing the register segment does not reach the window base address (`80H/0C0H/0E0H`). The RL196 linker locates this overlay segment below `0FFH`.
- Your program must have enough overlay segments to occupy the additional registers.
- The total number of global registers must fit, at most, below the 32–byte window base (`0E0H`). Otherwise, the linker issues a warning and your overlay segment must fit below `0FFH` (vertical windowing is not used). See the *80C196 Utilities User's Guide*, listed in *Related Publications*, for a complete list of warning messages the linker generates and explanations of their causes.
- Specify `nonreentrant` and `static` storage class to functions whenever appropriate.

If you are linking ASM196 modules together with C196 modules and you want the overlay segment from your ASM196 module to use the vertical windowing done in C, declare `?wsr` as an external byte variable in your ASM196 module and add the WSR management code to the prolog (use the address of `?wsr`, which is `#?wsr`) and to the epilog of local routines where appropriate. The following example shows how to write your ASM196 module. This example assumes that the ASM196 module is called by `main()`.

Your C196 module contains the following line:

```
void func(void);

main()
{
     func();
}
```

Your ASM196 module must contain the following lines:

```
example module
$include(_SFR_INC_)

oseg
     var1: dsw 1

cseg
     public func
func:
     push wsr
     andb wsr, #80h
     orb wsr,#?wsr
     .
     .
     .
     ld var1, #10
     .
     .
     .
     ldb wsr,[sp]
     add sp,#2
     ret
end
```

If you have a specific use for vertical windows and do not want the C196 compiler to allocate windows for your application, do not compile with the `windows` control. Move the desired value to the `wsr` register to switch to the desired window inside the desired function. You must restore the original window before exiting that function. Do not link with the `registers` and `windowsize` controls.

6.4.3.2 USING WINDOWED PARAMETERS

Structures can be placed at locations that can be mapped into one of the vertical windows, and the window can then be used by a function to access the fields of the structure using the best possible addressing mode (direct register access).

You can use this feature by inserting a few new keywords (extensions to the language) into the declarations of certain structures and into the function declarations in which a pointer to one of these structures is passed. The compiler will then handle the setup and restoration of the vertical windowing register(s).

The storage class keywords used in the structure declarations assure proper placement of the structures for use with this feature, and the qualifier keywords used in the formal parameter lists of functions activate the special handling of these parameters (pointers to these structures).

To specify the area(s) of memory from which to allocate windowed variables use the `windowram` control (abbreviation `wr`). Any number of ranges can be specified, and the `windowram` control and/or pragma may be specified any number of times. The ranges specified must not overlap, and must be within the range of mappable memory for the selected model.

When the compiler allocates an object from these ranges, it first tries to select an area from the beginning of a properly aligned range. If it cannot find a properly aligned range, it will take an area from within the first range with enough space remaining.

Use the `hold` or `nohold` control to specify whether the windowing code needs to preserve the HOLD/HOLDA bit in the WSR.

You can use the `_reg` storage class keyword in variable declarations, to indicate that the object should be allocated from the memory identified by a `windowram` range.

There are two type qualifier keywords, `_win` and `_win1`, you can use (for certain models) in parameter declarations, to indicate that the parameter being defined should reference a particular window in the register area, controlled by the WSR special function register (`_win`) or by the WSR1 special function register (`_win1`). The parameter must be a pointer. No more than one parameter in a function may use the same qualifier (both `_win` and `_win1` may be used together.

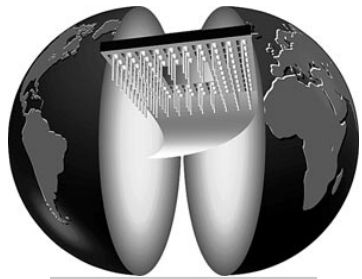
Example:

```
#pragma windowram(0x100–0x17F)

typedef struct _wheel_struct {
    int nflags;
    int oflags;
    /* ... whatever */
} WheelStruct;

_reg WheelStruct Wheel_LF;
_reg WheelStruct Wheel_RF;
_reg WheelStruct Wheel_LR;
_reg WheelStruct Wheel_RR;

void ProcessWheel(WheelStruct *_win Wheel, int f)
{
    Wheel–>oflags = Wheel–>nflags;
    Wheel–>nflags = f;
    /* ... whatever */
}

void main(void)
{
    int flags;

    /* ... whatever */
    ProcessWheel(&Wheel_LF, flags);
    ProcessWheel(&Wheel_RF, flags);
    ProcessWheel(&Wheel_LR, flags);
    ProcessWheel(&Wheel_RR, flags);
    /* ... whatever */
}
```

Do not use the `windows` control when using windowed parameters.

CHAPTER 7

ASSEMBLY CODE INSTRUCTIONS



TASKING
Quality Development Tools Worldwide

1 CHAPTER

This chapter describes ways to include assembly language instructions inside your C196 program without requiring a separately written and translated assembly language routine.

7.1 IN-LINE ASSEMBLY CODE SYNTAX

An additional reserved word, `asm`, is provided to identify in–line assembly instructions. To insert an in–line assembly statement, begin the statement with the `asm` keyword and terminate the statement with a semicolon. To indicate a block of statements, insert an open curly brace ({) after the `asm` keyword, and a close curly brace (}) after the last statement. The syntax is as follows:

```
asm pseudo_asm_inst;     /* Single line */
```

or

```
asm {                    /* In–line assembly block. */
     pseudo_asm_inst;
     .
     .
     .
}
```

Where:

`asm`	is the keyword indicating an assembly instruction follows.
`pseudo_asm_inst`	is assembly instruction (followed by a semicolon).

You can place an in–line assembly statement anywhere a valid C statement can be placed. C–style comments can be included as desired (enclosed with a slash–asterisk (`/*`) and an asterisk–slash (`*/`)). Assembler–style comments, beginning with a semicolon, are not allowed.

The `extend` control must be in effect in order to use the in–line assembly feature. The control allows the compiler to recognize the `asm` keyword and the in–line assembly instruction following it.

7.2 PSEUDO-ASSEMBLY INSTRUCTION INTERPRETATION

The pseudo–assembly instruction statement follows the same format as any regular ASM196 instruction. The syntax is:

```
operation [operand [,...]]
```

Where:

operation contains a machine instruction mnemonic code. It names the instruction to be executed.

operand specifies a register or value on which the operation is to be performed.

See the *80C196 Assembler User's Guide*, listed in *Related Publications*, for more information on assembly source program statement format.

The operand or operands of the operation can reference C variables, constants, and labels. You can use register variables wherever a register is a legal operand. The compiler interprets the pseudo–assembly instruction, replacing the C identifiers, as necessary, with assembly language equivalents, and translates the instruction into object code. The compiler also performs dead–code elimination, branch, and peephole optimizations.

Only machine instructions and the `dcb`, `dcw`, and `dcl` code definition directives are supported. See Section 7.3 for information on defining constant tables. The following types of instructions are not currently supported by the compiler:

- Labels.
- Assembly directives:
  - Module level directives, such as `module` and `public`.
  - Location counter control directives, such as `cseg` and `rseg`.
  - Symbol definition directives, such as `set` and `equ`.
  - Code definition directives other than `dcb`, `dcw`, and `dcl`, such as `dcr` and `dcp`.
  - Storage definition directives, such as `dsb`, `dsw`, `dsl`, and `dsr`.
  - Conditional assembly directives, such as `if`, `else`, and `endif`.
- Macro support directives, such as `macro`, `local`, `rept`, and `exitm`.

See Section 7.4 for a complete list of supported assembly instructions and the *80C196 Assembler User's Guide*, listed in *Related Publications*, for more information on labels and directives.

The following restrictions apply to the interpretation of the in–line assembly instructions:

- You cannot define new symbols with in–line assembly code.
- You must use C notation for non–decimal numbers. For example, C notation for the hexadecimal value 10H is `0x10` or `0X10`. Assembly notation equivalents `10H` and `10h` are invalid.
- You can enter instruction mnemonics in uppercase or lowercase.
- You must specify the `model` control to allow the use of the processor specific instruction sets.
- You cannot use numeric expressions. Expressions must consist of simple numbers.
- For `dcb`, `dcw`, and `dcl` directives: you can only specify one operand for each instruction.
- For generic conditional branch, unconditional branch, bit branch, and iterative branch instructions, the code address you specify for the branch must be a C label.
- For the generic `call` instruction, the code address you specify for the call must be a C function name.
- To access a specific element of an array, enclose the constant index in parentheses. The compiler scales the index to produce the equivalent offset. For example, the following code shows how to load the fifth element of an array into a register:

```
const int a[10] = { 2,3,5,7,11,13,17,19,23,29 };

asm ld wreg, a(5);   /* Loads the fifth element of
                        a into wreg. */
```

7.3 CONSTANT TABLE DECLARATION

When building word–aligned tables using the `dcw` directive, you must ensure that any label to be associated with the start of the table is also aligned. To do this, precede the label with a `dcw` directive, as shown below:

```
        asm dcw 0;      /* force word alignment. */
tbl:
        asm dcw lbl1;
        asm dcw lbl2;
        asm dcw lbl3;
```

If you omit the `dcw` directive preceding the `tbl` label, the compiler assigns the label to the current location counter value, which might not be word–aligned. The `dcw` directive following `tbl` forces the location counter to a word boundary, possibly incrementing the location counter. Thus, a one–byte gap can be placed between the `tbl` label and the word constant. Placing the `dcw` on the same line as the label does not alleviate this problem because the compiler processes the label separately from any in–line assembly instructions. See example 2 in Section 7.6 for an application of this process.

7.4 ASSEMBLY INSTRUCTIONS

The following assembly instructions are supported by the C196 compiler:

Arithmetic, Logical, and Memory Transfer Instructions

add	eldb	pop
addb	est	push
addc	estb	st
addcb	ld	stb
and	ldb	sub
andb	ldbse	subb
cmp	ldbze	subc
cmpb	mul	subcb
div	mulb	xch
divb	mulu	xchb
divu	mulub	xor
divub	or	xorb
eld	orb	

Special Register Instructions

```
clr     extb    negb
clrb    inc     not
dec     incb    notb
decb    neg     skip
ext
```

Shift Instructions

```
norml   shr     shral
shl     shra    shrb
shlb    shrab   shrl
shll
```

Generic Branch Instructions

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

The C196 compiler supports a pseudo–instruction, `call` *`register`*, which implements the indirect call by means of a code sequence containing a `br [indirect]` instruction.

Zero–operand Instructions

```
clrc    epts    pushf
clrvt   nop     ret
di      popa    rst
dpts    popf    setc
ei      pusha
```

Extended Instructions

```
bmov    cmpl    idlpd
bmovi   ebmovi  tijmp
```

7.5 UNSUPPORTED INSTRUCTIONS

The following assembly instructions are not supported:

Non-generic Branch Instructions

```
djnz            jgt             jnvt
djnzw           jh              jst
ecall           jle             jv
ejmp            jlt             jvt
jbc             jnc             lcall
jbs             jne             ljmp
jc              jnh             scall
je              jnst            sjmp
jge             jnv
```

Module-level Directives

```
end             module          public
extrn
```

Location Counter Control Directives

```
cseg            kseg            oseg
dseg            org             rseg
```

Symbol Definition Directives

```
equ             set
```

Code Definition Directives

```
dcp             dcr
```

Storage Reservation Directives:

```
dsb             dsp             dsw
dsl             dsr
```

Conditional Assembly Directives:

```
if              else            endif
```

Macro Support Directives:

```
endm            irpc            macro
exitm           local           rept
irp
```

For more information on these instructions and directives, see the *80C196 Assembler User's Guide* listed in *Related Publications*.

7.6 EXAMPLES

1. The following example shows how you can add in–line assembly code to C196 source text:

```
#include <stdio.h>

register int ri;

main()
{
    int i;

    init_serio();
    asm
    {   ld ri, #10;
        st ri, i;
        add ri, i;
    }
    printf("ri = %d\r\n", ri);
}
```

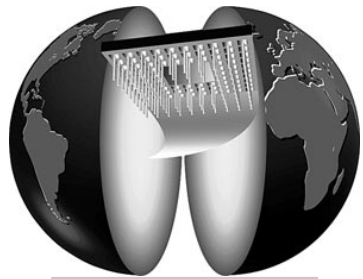
2. The following is an example of the use of the `tijmp` instruction.

```
#pragma model(kc)

int func(unsigned char k)
{
    unsigned char *ip = &k;

    goto around;

    lbl1:  return(1);
    lbl2:  return(2);
```

```
    lbl3:  return(3);
    lbl4:  return(4);

    asm dcw 0;        /* force word alignment. */
tbl:
    asm {
          dcw lbl1;
          dcw lbl2;
          dcw lbl3;
          dcw lbl4;
        }

around:
    asm {
          ld  0x22,ip;
          ld  0x20,#tbl;
          tijmp 0x20,[0x22],#3;
        }

    return (0);
}
main()
{
    unsigned char i,j;

    for (i = 0; i < 3; i++)
        j = func(i);
    return;
}
```

CHAPTER 8

LIBRARIES



TASKING
Quality Development Tools Worldwide

8 CHAPTER

This chapter describes ways to include assembly language instructions inside your C196 program without requiring a separately written and translated assembly language routine.

The C196 libraries provide the ANSI standard C library functionality and some additional functionality specific to 80C196 architecture. This chapter describes C196 functions and macros that are implementation–specific or that do not conform to the 1989 ANSI standard for C. Functions and macros that do conform to ANSI C are described in *C: A Reference Manual*, listed in *Related Publications*.

8.1 LIBRARY FILES

You can link your application with any of the C196 libraries, as well as with any libraries that you define. This section explains how to select and use the libraries and header files for your application.

The C196 Compiler includes the following library files:

`c96.lib` Defines all standard functions and extensions. One version present in each processor dependent subdirectory of `lib`.

`c96fp.lib` Defines all math functions that require floating point arithmetic. Examples are: `sin`, `cos`, `fabs`, `strtod`. Also contains a version of `printf` and `scanf` that recognize the `%f`, `%g`, and `%e` format
One version present in each processor dependent subdirectory of `lib`.

`fpal96.lib` Defines all floating–point operations. One version present in each processor dependent subdirectory of `lib`.

Special Function Registers (SFRs) are defined in the following object files in the `lib` directory:

xx`_sfrs.obj` Defines the SFRs, where *xx* represents a processor model.

The C196 product also includes the following object file in each subdirectory of the `lib` directory:

`cstart.obj` Defines the startup code.

8.1.1 LIBRARY DIFFERENCES AND HEADER FILE CORRELATIONS

In the subdirectory `lib` you will find subdirectories with all the libraries supplied with the C196 compiler. You choose one of the library files to link library code from, according to the model (and the execution mode, if the model is 24–bit). The library files are listed in Table 8–1.

Lib Directories	Library	Code Model	Data/Const Addressing	Segment	Code Addressing	Remarks
nt_c / np_c	cstart.obj c96.lib c96fp.lib fpal96.lib	note[1]	16–bit	HighCode	16–bit	Compatibility Mode
nt_e / np_e	cstart.obj c96.lib c96fp.lib fpal96.lib	note[1]	16–bit	FarCode	24–bit	Extended Mode
nt_cnf / np_cnf	cstart.obj c96.lib c96fp.lib fpal96.lib	note[1]	16/24–bit	HighCode	16–bit	Compatibility Mode
nt_enf / np_enf	cstart.obj c96.lib c96fp.lib fpal96.lib	note[1]	16/24–bit	FarCode	24–bit	Extended Mode
nt_cf / np_cf	cstart.obj c96.lib c96fp.lib	note[1]	24–bit	HighCode	16–bit	Compatibility Mode
nt_ef / np_ef	cstart.obj c96.lib c96fp.lib	note[1]	24–bit	FarCode	24–bit	Extended Mode
all other	cstart.obj c96.lib c96fp.lib fpal96.lib	Any	16–bit	Code	16–bit	

[1] `nt`–libraries are used by the models NT, CB, EA and EC; `np`–libraries are used by the models NP and NU.

Table 8–1: Library/Object Files

The objects and libraries in the subdirectories `nt_c` and `np_c` of the `lib` directory are suitable only for use with the 80C196NT/CB/EA/EC or 80C196NP/NU respecively, running code compiled with the `nearcode` control in effect. They use only near code linkage, but all data pointers are 16 bits wide.

The objects and libraries in the subdirectory `nt_e` and `np_e` of the `lib` directory are suitable only for use with the 80C196NT/CB/EA/EC or 80C196NP/NU respecively, running code compiled with the `farcode` control in effect. They use only far code linkage, and all data pointers are 16 bits wide, and all const pointers are 32 bits wide.

The objects and libraries in the subdirectories `nt_cnf` and `np_cnf` of the `lib` directory are suitable only for use with the 80C196NT/CB/EA/EC or 80C196NP/NU respecively, running code compiled with the `nearcode` control in effect. They use only near code linkage, but all data pointers are 16 bits wide, and all const pointers are 32 bits wide.

The objects and libraries in the subdirectory `nt_enf` and `np_enf` of the `lib` directory are suitable only for use with the 80C196NT/CB/EA/EC or 80C196NP/NU respecively, running code compiled with the `farcode` control in effect. They use only far code linkage, and all data pointers are 16 bits wide.

The objects and libraries in the subdirectory `nt_cf` and `np_cf` of the `lib` directory are suitable only for use with the 80C196NT/CB/EA/EC or 80C196NP/NU respecively, running code compiled with the `nearcode` control in effect. They use only near code linkage, but all data pointers are 32 bits wide.

The objects and libraries in the subdirectory `nt_ef` and `np_ef` of the `lib` directory are suitable only for use with the 80C196NT/CB/EA/EC or 80C196NP/NU respecively, running code compiled with the `farcode` control in effect. They use only far code linkage, and all data pointers are 32 bits wide.

The objects and libraries in the other subdirectories of the `lib` directory are suitable for use with a processor specific model respectively.

The header files supplied with the C196 compiler are sensitive, where necessary, to the model and mode you select. Function prototypes involving pointers declare `far` pointers when the selected model is 24–bit; otherwise the prototypes declare `near` pointers. The resulting function prototype declarations match the actual functions that you will link from the appropriate library file.

8.1.2 LINKING LIBRARY FILES

The RL196 linker searches through the library files to resolve external references to library functions. It uses the first instance of a function that it encounters, and skips any function with the same name in any subsequent library. The linker only makes one pass through each library file and tries to resolve as many external references as it can. The linker does not reopen previously searched library files if it encounters more external references later on in the process. For this reason, you must link all library files last, to ensure that all external references are known before the linker searches through each library. RL196 searches through a list of files to find an input file. The contents of the list depends on the `model()` control. Specify the libraries and object files for linking in the following order:

1. startup code; choose the object file according to the model and the execution mode: `cstart.obj`
2. program modules
3. user–defined libraries, if any
4. the floating point C library file, containg math functions and floating–point versions of `printf` and `scanf`, if you are performing floating–point formatted input and output: `c96fp.lib`
5. the C library file; choose the library file according to the model and the execution mode: `c96.lib`
6. the FPAL96 floating–point library, if you are using floating–point functions; choose the library files according to the model and the execution mode: `fpal96.lib`

If an object file or library is specified with a relative pathname (or without any path), then RL196 will search through a list of directories to find the file. See the description of RL196 in the *80C196 Utilities User's Guide*.

8.2 HEADER FILES

You can write your own external declaration for any library function or variable, but doing so does not guarantee an exact match. The supplied header files contain C196 source text to declare the library function prototypes and macros. The function declarations in the C196 header files are prototyped, to ensure an appropriate match between definition and use of the functions. Use the `#include` preprocessor directive to include a header file.

Some functions declared with prototypes in the header files are also defined as macros in the same header files. To use the library function rather than the macro, simply use `#undef` to remove the macro definition before specifying the function call in your source text.

Table 8–2 lists the names and functionality of the C196 library header files and the manuals which describe each header file.

Filename	Contents	Described In
ctype.h	character–handling utilities	this chapter
float.h	floating–point limits	*C: A Reference Manual*
limits.h	fixed–point limits	*C: A Reference Manual*
math.h	absolute value function prototype	*C: A Reference Manual*
setjmp.h	non–local jump function prototypes	*C: A Reference Manual*
stdarg.h	variable argument list utilities	*C: A Reference Manual*
stddef.h	common definitions	*C: A Reference Manual*
stdio.h	input/output (I/O) utilities	*C: A Reference Manual*
stdlib.h	general utilities	*C: A Reference Manual*
string.h	string handling utilities	this chapter
*xx*_funcs.h	processor specific functions	this chapter
*xx*_sfrs.h	processor special facilities as they are described in the processor specific *User's Manual*.	this chapter

Table 8–2: Header files

xx_funcs.h

Function

Processor–specific functions
non–ANSI

Description

The *xx*_funcs.h header files contain function prototypes of the functions enable, disable, enable_pts, disable_pts, power_down, idle and possible type definitions for the various PTS control blocks. *xx* represents a processor as specified to the model(*xx*) control.

The compiler initializes the predefined macro _FUNCS_H_ with the name of the processor specific *xx*_funcs.h.

Example

The following example includes the file nt_funcs.h:

```
#pragma model(nt_ef)

#include _FUNCS_H_
```

The same result can be obtained by:

```
#include <nt_funcs.h>
```

xx_sfrs.h

Function

Processor–specific facilities
non–ANSI

Description

The *xx*_sfrs.h header files define variables to access the Special Function Registers (SFRs) and declare functions to manipulate the processor hardware. *xx* represents a processor as specified to the model(*xx*) control. See the processor specific *User's Manual* for details on SFRs.

The compiler initializes the predefined macro _SFR_H_ with the name of the processor specific *xx*_sfrs.h.

Example

The following example includes the file nu_sfrs.h:

```
#pragma model(nu_ef)

#include _SFR_H_
```

The same result can be obtained by:

```
#include <nu_sfrs.h>
```

ctype.h

Function

Character handling
ANSI

Description

The `ctype.h` header file contains macros and function prototypes useful for testing and mapping characters. These character–handling utilities operate as described in *C: A Reference Manual*, listed in *Related Publications*.

The `ctype.h` header file provides both function–like macros and function prototypes for some ANSI character query and conversion functions. Include `ctype.h` if your program calls any of the following functions:

```
isalnum   isdigit   isprint   isupper
isalpha   isgraph   ispunct   isxdigit
iscntrl   islower   isspace
```

The `ctype.h` header file also provides both function–like macros and function prototypes for some non–ANSI character query and conversion functions. Include `ctype.h` if your program calls any of the following functions:

```
isascii   _tolower  _toupper
```

If you do not want to use the function–like macros, use the `#undef` control to remove the macro definition and the compiler calls the actual function.

Examples

The `ctype.h` header file contains a function prototype for `toupper` and both a function prototype and a macro definition for `isxdigit`. The following examples show the differences in the code generated by the compiler when you use the function prototype or the macro definition of `isxdigit`.

1. The following source text uses the macro definition of `isxdigit` and is compiled with the `listexpand` control:

```
#pragma listexpand
#include <ctype.h>
int upcx(unsigned char input)  /* Use the prototypes*/
{                              /* and macros in the */
  if (isxdigit(input))         /* ctype.h header    */
    return(toupper (input));   /* file.             */
  return input;
}
```

The compiler generates the source file listing shown in Figure 8–1.

```
C196 Compiler CYTPE_X                    01/29/99  11:29:19  Page  1
80C196 Compiler Vx.y Rz SN (C)1993 Tasking BV, Compilation of module CTYPE_X
(C)1980,1990,1992,1993 Intel Corporation
Object module placed in CTYPE_X.obj
Compiler invoked by: c:\c196\bin\C196.EXE CTYPE_X.c code

 Line Level  Incl

    1            #pragma listexpand
    2            #include <ctype.h>
    3            int upcx(unsigned char input)
    4            {
    5     1         if (isxdigit(input))
    +               if ((((unsigned)(input) < 0x80) ?
                          (_ctype_)[input] & 0x40 : 0))
    6     1           return(toupper (input));
    7     1         return input;
    8     1       }
```

Figure 8–1: Example using the macro definition

2. The following source text undefines the macro definition of `isxdigit` and is compiled with the `listexpand` control:

```
#pragma listexpand
#include <ctype.h>
#undef isxdigit     /* Undefine the      */
                    /* macro definition. */
                    /* Use the function  */
                    /* definition. */
int upcx(unsigned char input)
{
  if (isxdigit(input))
    return(toupper (input));
  return input;
}
```

The compiler generates a source text listing shown in Figure 8–2:

```
C196  Compiler   CTYPE_XU                01/29/99 12:41:49  Page  1


80C196 Compiler Vx.y Rz SN (C)1993 Tasking BV, Compilation of module CTYPE_XU
(C)1980,1990,1992,1993 Intel Corporation
Object module placed in CTYPE_XU.obj
Compiler invoked by: c:\c196\bin\C196.EXE CTYPE_XU.c code

 Line Level  Incl

    1             #pragma listexpand
    2             #include <ctype.h>
    3             #undef isxdigit
    4
    5
    6
    7             int upcx(unsigned char input)
    8             {
    9     1         if (isxdigit(input))
   10     1            return(toupper (input));
   11     1         return input;
   12     1       }
```

Figure 8–2: Example using the function prototype

string.h

Function

Character array manipulation
ANSI

Description

The ANSI contents of `string.h` are described in *C: A Reference Manual*, listed in *Related Publications*. In addition, `string.h` defines the following non–ANSI functions:

`cstr`	converts a length–prefixed string to a null–terminated string
`udistr`	converts a null–terminated string to a length–prefixed string

8.3 FUNCTIONS

This section provides descriptions of the C196 library functions that are not covered in *C: A Reference Manual*.

Each entry in this section is organized as follows:

Function — contains a short description of the function.

Prototype Declaration — lists the prototype provided in the header file.

Header File — indicates which header file contains the prototypes, macros, and type definitions relevant to the function.

Description — explains the operation and use of the function.

Returns — describes the values returned by the function on successful completion or (where relevant) on error.

cstr

Function

Converts a UDI string to a C–type string.

Prototype Declaration

```
char *cstr (char *c_ptr, const char *udi_ptr);
```

where:

`c_ptr` points to a buffer large enough to contain the converted string.

`udi_ptr` points to a length–prefixed string.

Header File

`string.h`

Description

Use this function to convert the length–prefixed string (UDI string) to a null–terminated string (C–type string).

The *`c_ptr`* argument must point to a buffer large enough to contain the C–type string. The length of a C–type string is one byte more than the number of characters in the string.

The two pointer arguments normally point to separate string buffers. If the arguments point to the same location, the `cstr` function overwrites the original UDI string with the new C–type string.

Returns

The `cstr` function returns a pointer to the converted string. This return value is the same as the value passed in via the *`c_ptr`* parameter.

disable

Function

Disables the processor's interrupts.

Prototype Declaration

```
void disable (void);
```

Header File

*xx*_funcs.h

xx represents a processor model.

Description

Use this function to disable the processor's interrupts.

Returns

The `disable` function does not return a value.

disable_pts

Function

Disable the peripheral transaction server's interrupts.

Prototype Declaration

```
void disable_pts (void);
```

Header File

*xx*_funcs.h

xx represents a processor model.

Description

Use this function to disable the peripheral transaction server's (PTS) interrupts. This function is valid only for the 80C196KC and higher processors.

Returns

The `disable_pts` function does not return a value.

enable

Function

Enable the processor's interrupts.

Prototype Declaration

```
void enable (void);
```

Header File

*xx*_funcs.h

xx represents a processor model.

Description

Use this function to enable the processor's interrupts.

Returns

The `enable` function does not return a value.

enable_pts

Function

Enable the peripheral transaction server's interrupts.

Prototype Declaration

```
void enable_pts (void);
```

Header File

*xx*_funcs.h

xx represents a processor model.

Description

Use this function to enable the peripheral transaction server's (PTS) interrupts. This function is valid only for the 80C196KC and higher processors.

Returns

The `enable_pts` function does not return a value.

fpinit

Function

Initializes floating–point library.

Prototype Declaration

```
void fpinit (void);
```

Header File

```
fpal96.h
```

Description

Use this function to perform the following necessary initializations for the functions in the FPAL96 libraries:

- Set rounding flag in control word to round–to–nearest.
- Mask all exceptions in control word.
- Set floating–point accumulator to indicate signalling Not–a–Number (sNaN).
- Set `stat` field to indicate sNaN and clear error byte of status word.
- Attach a dummy error handler.

A program must call the `fpinit` function before performing any floating–point operation. See the *80C196 Utilities User's Guide*, listed in *Related Publications*, for more information on floating–point numbers and initialization.

Returns

The `fpinit` function does not return a value.

idle

Function

Enters a power–saving mode.

Prototype Declaration

```
void idle (void);
```

Header File

xx`_funcs.h`

xx represents a processor model.

Description

Use this function to place the 80C196 processor in the power–saving idle mode. The `idle` function is available only on the 80C196 processor. See the *Embedded Microcontrollers and Processors Handbook*, listed in *Related Publications*, for more information on the idle mode of the 80C196 processors.

The processor enters the following state during idle mode:

- The CPU stops executing.
- All internal clocks assume logic state zero.
- Peripheral clocks and the `CLKOUT` pin remain active.
- All peripherals and the interrupt controller continue to function.
- If the watchdog timer was enabled, after a reset it continues to operate.
- All RAM is preserved.

You can release the CPU from idle mode with an interrupt or a hardware reset.

Returns

The `idle` function does not return a value.

power_down

Function

Enters a power–saving mode.

Prototype Declaration

```
void power_down (void);
```

Header File

*xx*_funcs.h

xx represents a processor model.

Description

Use this function to place the 80C196 processor in powerdown mode. The `power_down` function is available only on the 80C196 processor. See the *Embedded Microcontrollers and Processors Handbook*, listed in *Related Publications*, for more information on the powerdown mode of the 80C196 processors.

All peripherals must be idle before the program calls the `power_down` function. In powerdown mode, the state of the processor has the following characteristics:

- The CPU stops executing.
- All internal clocks assume logic state zero.
- The oscillator is turned off. The 80C196 processor cannot detect oscillator failure in powerdown mode.
- The watchdog timer is disabled on reset and becomes enabled on the first write operation to it. The 80C196 processor cannot time out the watchdog timer in powerdown mode.
- All internal RAM is preserved.

You can exit out of powerdown mode with an external interrupt on the pin mapped to `INT7` or with a hardware reset.

Returns

The `power_down` function does not return a value.

printf, sprintf

Function

Formats output.

Prototype Declaration

```
int printf (const signed char *format_ptr,...);

int sprintf ( signed char *buf_ptr,
              const signed char *format_ptr,...);
```

where:

format_ptr points to the output format specification.

buf_ptr points to a memory output buffer.

,... indicates variables containing values to be written.

Header File

stdio.h

Description

Use these functions to perform formatted output: `printf` to the output serial port, and `sprintf` to a memory buffer. For guidelines on how these functions operate, see *C: A Reference Manual*, listed in *Related Publications*. The `printf` and `sprintf` functions do not support the `%p` floating–point conversion operations.

Before using `printf` for the first time, you must call the `init_serio` function once after a reset or exit from the powerdown mode to ensure correct operation of subsequent calls to `putch`. The `printf` function calls the `putch` function. The `init_serio` function initializes a static variable used to hold the serial–port status. This function sets the TI bit in the static variable, thereby initializing the mechanism used by the `putch` function. The `putch` function then waits for the TI bit to be set, indicating that the previous character has been transmitted, before writing the character argument to the serial port. If you do not call `init_serio` before calling `putch`, the `putch` function can wait indefinitely for the TI bit to be set.

The `sp_stat` and `sbuf` variables are defined in the *`xx`*`_sfrs.h` header files. If you redefine `putch` to write to a different destination, you can use `printf` to write formatted output to locations other than the serial port. The program must then ensure the new destination is enabled as appropriate.

Before using `printf` or `sprintf` with floating–point numbers, you must call the `fpinit` function to initialize floating–point capability. You must also specify one of the `c96fp.lib` libraries and one of the FPAL96 libraries when you link your program, to provide floating–point support.

Returns

The `printf` function returns the number of characters actually transmitted. If an I/O error occurs, the return value is negative.

The `sprintf` function returns the number of characters written into the memory buffer. This return value does not include the terminating null character.

scanf, sscanf

Function

Formats input.

Prototype Declaration

```
int scanf (const signed char *format_ptr,...);

int sscanf ( signed char *buf_ptr,
             const signed char *format_ptr,...);
```

where:

format_ptr points to the output format specification.

buf_ptr points to a memory input buffer.

,... indicates any number of pointers to variables to which the input values are assigned.

Header File

stdio.h

Description

Use these functions to perform formatted input: `scanf` from standard input and `sscanf` from a character string in memory. For guidelines on how these functions operate, see *C: A Reference Manual*, listed in *Related Publications*. The `scanf` and `sscanf` functions do not support the `%p` pointer formatting specification.

If conversion terminates because of a conflict between an input character and the corresponding format specifier, the offending character remains unread. Trailing white space (including a newline character) in a format specification can match optional white space in the input field.

Before using `scanff` for the first time, you must call the `init_ungetc` function once after a reset or exit from the powerdown mode to ensure correct operation of subsequent calls to `ungetc`. The `scanf` function calls the `ungetc` function. The `init_ungetc` function initializes a static variable used to hold the serial–port status. This function sets the TI bit in the static variable, thereby initializing the mechanism used by the `ungetc` function. The `ungetc` function then waits for the TI bit to be set, indicating that the previous character has been transmitted, before reading the character argument from the serial port. If you do not call `init_ungetc` before calling `ungetc`, the `ungetc` function can wait indefinitely for the TI bit to be set.

Before using `scanf` or `sscanf` with floating–point numbers, you must call the `fpinit` function to initialize floating–point capability. To provide floating point support, You must also specify one of the `c96fp.lib` libraries and one of the FPAL96 libraries when you link your program.

Returns

The `scanf` and `sscanf` functions return the number of successfully read input values.

udistr

Function

Converts a C–type string to UDI form.

Prototype Declaration

```
char *udistr (char *udi_ptr, const char *c_ptr);
```

where:

udi_ptr points to a buffer large enough to contain the converted string.

c_ptr points to a null–terminated string.

Header File

`string.h`

Description

Use this function to convert a null–terminated string (C–type string) to a length–prefixed string (UDI–type string).

The *udi_ptr* argument must point to a buffer large enough to contain the UDI–type string. You can use the `strlen` function on the C–type string to determine the required length of the buffer. *C: A Reference Manual*, listed in *Related Publications*, describes how to use `strlen`. The length of the buffer must be one byte longer than the value returned by the `strlen` function. The behavior of the `udistr` function for strings longer than 255 bytes is unpredictable.

The two pointer arguments normally reference separate string buffers. If the arguments point to the same location, the `udistr` function overwrites the original C string with the new UDI string.

Returns

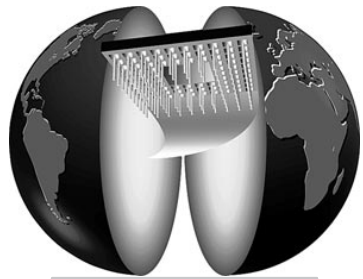
The `udistr` function returns a pointer to the converted string. This return value is the same as the value passed in via the *udi_ptr* parameter.

8.4 DYNAMIC MEMORY ALLOCATION

In order to use the library routines for dynamic memory allocation `malloc()`, `realloc()`, `calloc()` and `free()` it is necessary to reserve RAM space for the dynamic memory. This RAM space, called HEAP, is reserved by the linker. See the linker controls `heap` and `ram` on how to specify this HEAP space.

CHAPTER 9

MESSAGES AND ERROR RECOVERY



TASKING
Quality Development Tools Worldwide

CHAPTER 6

9.1 INTRODUCTION

The C196 compiler can issue the following types of messages:

- Sign–on and sign–off messages (discussed in Section 9.2)
- Fatal errors (discussed in Section 9.3)
- Errors (discussed in Section 9.4)
- Warnings (discussed in Section 9.5)
- Remarks (discussed in Section 9.6)

All messages, except fatal error messages, are reported in the print file. Fatal error messages appear on the screen; the compiler aborts compilation and produces no object module. Other errors do not abort compilation but no object module is produced. Warnings and remarks usually provide information and do not necessarily indicate a condition affecting the object module.

Messages relating to syntax and most messages relating to semantics are interspersed in the listing at the point of error. Some messages relating to semantics appear at the end of the source text listing and refer to the statement number on which the error occurred.

9.2 SIGN-ON AND SIGN-OFF MESSAGES

The compiler writes information to the screen at the beginning and the end of compilation. On invocation, the compiler displays the following message:

```
80C196 C compiler vx.y rz    SN00000000–004 (c) years TASKING, Inc.
(C)years Intel Corporation
```

where:

v*x.y*	identifies the version of the compiler.
r*z*	identifies the revision of the compiler.
years	identifies the copyright years.

On normal completion, the compiler displays a message similar to the following:

```
C196 Compilation Complete. x Remark[s], y Warning[s], z Error[s]
```

where:

x indicates the number of remarks that the compiler generated.

y indicates the number of warning messages that the compiler generated.

z indicates the number of non–fatal errors that the compiler generated.

You can use compiler controls to specify the contents of this message, as follows:

`diagnostic(0)` displays the entire message.

`diagnostic(1)` suppresses the number of remarks.

`diagnostic(2)` suppresses the numbers of remarks and warnings.

`notranslate` suppresses the `Compilation Complete`.

The defaults of these controls are `diagnostic(1)` and `translate`.

If the compilation ends because of a fatal error, the compiler displays the following message:

```
C196 FATAL ERROR
COMPILATION TERMINATED
```

The print file lists the error that ended the compilation. If the `noprint` control is in effect, all diagnostics (restricted by the `diagnostic` control) that the compiler generates appear on the screen.

9.3 FATAL ERROR MESSAGES

Fatal error messages have the following syntax:

```
C196 FATAL ERROR – message
```

Following is an alphabetic list of fatal error messages.

argument not allowed for** control **control

This message indicates an attempt to pass arguments to a control that accepts none. Improper argument passing is a fatal error if it occurs in the compiler invocation, but the preprocessor only issues a warning if the improper argument occurs in a `#pragma` directive.

argument not allowed for negated** control **control

Negated controls, except for the `noreentrant` control, do not accept arguments. If you specified an argument for a negated control in the compiler invocation, the compiler generates this error. However, if you specified the argument for a negated control in a `#pragma` directive line, the preprocessor only issues a warning.

***argument out of range for** control **control:** arg*

This message indicates an attempt to use an argument value that is out of the valid range. An out–of–range argument is a fatal error if it occurs in the compiler invocation, but the preprocessor only issues a warning if the improper argument occurs in a `#pragma` directive.

argument required for** control **control

A missing required argument is a fatal error if it occurs in the compiler invocation, but the preprocessor only issues a warning if the missing argument occurs in a `#pragma` directive.

argument too long for** control **control

The length of the argument to the control exceeds the maximum number of characters allowed by the compiler.

BMOV only valid for model KB

The `bmov` control is valid only if you specified the `model(kb)` control.

compiler error

This message follows internal compiler error messages. If you receive this message, you should contact TASKING customer service. See the Service Information on the inside back cover.

control control cannot be negated

You cannot use the `no` prefix with this compiler control. Improper negating is a fatal error if it occurs in the compiler invocation, but the preprocessor only issues a warning if it occurs in a `#pragma` directive.

duplicate** control **control

A primary control that must not be specified more than once was specified more than once. Only the following controls can be specified more than once:

```
define          reentrant
fixedparams     regconserve
include       searchinclude
interrupt       varparams
```

If you specify a compiler control both in the compiler invocation and in a `#pragma` preprocessor directive, the compiler invocation specification takes precedence. A duplicate control is a fatal error if it occurs in the compiler invocation but the preprocessor only issues a warning if it occurs in a `#pragma` directive.

duplicate argument** argument **for** control **control

An argument for *`control`* was specified more than once; for example: specifying more than one handler for one interrupt number.

expression too complex

A complex expression exhausted an internal structure in the compiler. Break the expression down into simpler components.

FARCODE conflicts with NEARCODE

These two controls determine the segment into which all code is generated. They are mutually exclusive, but both controls were explicitly specified. Eliminate the one you don't want.

FARCODE control invalid for the component

The `farcode` control is valid only for 24–bit models. Add a 24–bit `model()` control or remove the offending control.

FARDATA conflicts with NEARDATA

These two controls determine the default location of non–constant, non–register data. They are are mutually exclusive, but both controls were explicitly specified. Eliminate the one you don't want.

FARDATA control invalid for the component

The `fardata` control is valid only for 24–bit models. Add a 24–bit `model()` control or remove the offending control.

function call nesting limit exceeded

The nesting of function calls within an expression exceeded 32.

HOLD is not effective when WINDOWS is off

The `HOLD` control can only be used when the compiler saves/restores `WSR` in the function prolog/epilog, that is when the `WINDOWS` control is on.

illegal macro definition: *macro_name*

An invalid macro was defined on the command line with the `define` control.

input pathname is missing

A primary source file pathname was not specified in the compiler invocation.

insufficient memory for macro expansion

An internal structure was exhausted during macro expansion. Two causes of this error are: the macro or the actual arguments are too complex, or the macro's expansion is too deeply nested. Also see the related error message, `macro expansion too nested`.

internal limit exceeded – block too large: *statement_number*

The block being processed exceeds the internal buffer size. To resolve this error, break the block into two functions or introduce a label.

internal limit exceeded – call nesting too deep: *statement_number*

Calls within an expression are nested more than the internal limit of 20. To resolve the problem, split the expression such that call nesting does not exceed 20.

internal limit exceeded – expression too complex

The compiler ran out of temporary working registers to allocate to a computation. Use explicit variables to hold intermediate results.

internal limit exceeded – program too complex

The table of compiler–generated labels was exhausted, usually because the program flow is too complex. Reduce the complexity or break down the module.

internal limit exceeded – stack too deep: *statement_number*

The stack requirement of the function exceeded the internal limit of 128 bytes. This error can be caused by an expression that is too complex or a large structure or union that appears as an argument to a call.

internal limit exceeded – statement too complex: *statement_number*

The statement being processed is too complex and exceeded the internal buffer size. Split the statement into less complex statements.

invalid argument for *control* ***control***

The argument specified for `control` is not valid; for example: the argument specified for `model` is invalid.

invalid control: *control*

A control not supported by the compiler was specified. Check the spelling of the control. An invalid control is a fatal error if it occurs in the compiler invocation but the preprocessor only issues a warning if the invalid control occurs in a `#pragma` directive.

invalid decimal argument: *value*

Non–decimal characters were found in an argument that must be a decimal value. An improper argument is a fatal error if it occurs in the compiler invocation, but the preprocessor only issues a warning if the improper argument occurs in a `#pragma` directive line.

invalid identifier: *identifier*

An identifier does not follow the rules for forming identifiers in C. An invalid identifier is a fatal error if it occurs in the compiler invocation, but the preprocessor only issues a warning if the invalid identifier occurs in a `#pragma` directive.

invalid syntax for *control* ***control***

The compiler control contained a syntax error. Invalid control syntax is a fatal error if it occurs in the compiler invocation, but the preprocessor only issues a warning if the invalid syntax occurs in a `#pragma` directive line.

no more free space

The internal structure used to hold macros is exhausted. Use fewer macros in your program.

null argument for *control* ***control***

Null arguments for compiler controls are not allowed. For example, the following is illegal:

```
varparams(f1,,f2)
```

A null argument is a fatal error if it occurs in the compiler invocation, but the preprocessor only issues a warning if the null argument occurs in a `#pragma` directive.

out of memory

The internal memory buffer used to hold macros was exhausted. Use fewer macros in your program.

previous errors prevent further compilation

The compiler was unable to recover from previous errors in the compilation. Correct the errors reported thus far, then recompile.

regconserve conflicts with registers(all)
registers(all) conflicts with regconserve

The `registers(all)` control specifies that the compiler is to allocate all program variables to registers, including variables declared without the explicit `registers` attribute (`register` variables). The `regconserve` control specifies that the compiler is to conserve registers, placing only `register` variables (and, optionally, a subset of the non–`register` variables) in registers. The compiler cannot resolve these conflicting directions. This error is fatal if both controls are specified in the compiler invocation, but the preprocessor only issues a warning if the conflict occurs in `#pragma` directives.

switch table overflow

Too many active `case`s exist in a `switch` statement that has not yet been completed.

symbol table overflow

Too many symbols are defined in the module. Remove unused definitions or break the module into two or more smaller modules.

too many directories are specified for search – *pathname*

Too many directories are specified in the compiler invocation with the control `searchinclude`. The *`pathname`* is the directory at which the error occurred, that is, the first directory over the limit.

too many include files

Too many include files have been specified. Combine include files or break the module into two or more smaller modules.

syntax error

An unrecoverable syntax error has occurred. Two situations that can cause this error are:

- The `alien`, `reentrant`, or `nonreentrant` keyword is present with the `noextend` control in effect.
- An identifier is present in function context but does not have a body, for example,

```
int f1()   /* syntax error missing semicolon */
int f2();  /* valid */
```

type table full

Too many symbols with non–standard data types are defined in the module. Remove unused definitions, or break down the module.

whiles, fors, etc. too deeply nested

The statement nesting structure of the module exhausted an internal structure in the compiler. A possible solution is to make a function out of the more deeply nested control structures, and call that function.

9.4 ERROR MESSAGES

Error messages have the following syntax:

```
*** Error at line nn of filename: message
```

where:

filename	is the name of the primary source file or include file in which the error occurred.
nn	is the source line number where the error is detected.
message	is the explanation.

Following is an alphabetic list of error messages.

 # operator missing macro argument operand

The # operator must be followed by a macro argument.

operator occurs at beginning or end of macro body

The ## (token concatenation) operator is used to paste together adjacent preprocessing tokens, so it cannot be used at the beginning or end of a macro body.

address out of range

The constant expression used as the absolute address is greater than `0xFFFF` (`0xFFFFFF` for a 24–bit model). This error can only occur if you are dereferencing a constant expression; for example, the following code generates the error:

```
*( (char *) 0x10000) = 5;   /* 0x10000 is > 0xFFFF. */
```

anonymous parameter

An argument in a function definition is prototyped but not named.

arguments not allowed

Arguments were passed to a function that does not accept arguments.

array too large

This error occurs when the size of an array exceeds 64 kilobytes.

call not to a function

A call is made to a symbol which is not a function.

cannot initialize

The type or number of initializers does not match the initialized variable or the variable was not declared with the `const` qualifier. With `omf(0)` and `omf(1)` the C196 compiler supports static initialization of only `const` objects in file scope and only `static const` objects in block scope. For example:

```
const int i = 10;    /* supported */
int j = 10;          /* not supported */
static int a = 10;   /* not supported */

void f(void)
{
     int i = 10;               /* supported */
     static const int b = 10;  /* supported */
     const int c = 10;         /* supported */
     static int j = 10;        /* not supported */
};
```

Use `omf(2)` (or higher).

cannot initialize extern in block scope

An external declaration cannot be initialized in any scope other than file scope. The following example is an invalid external declaration:

```
f()
{ extern const int i = 1;
}
```

cannot take the address of asm register operand

The address of a register variable was accessed using the ampersand (&) address operator, after the variable was used as a register operand in an in–line code assembly statement; for example,

```
register int a;
int *p;

asm ld a,#0A0H;
p = &a;     /* This statement generates the error. */
```

case not in switch

A case was specified, but not within a `switch` statement.

code segment too large

The size of the code segment, which includes the program's code and constant objects, exceeds 64 kilobytes.

conditional compilation directive is too nested

The nesting of conditional compilation directives exceeded 16 levels.

constant expected

A non–constant expression appears when a constant expression is expected (e.g., a non–constant expression as array bounds or as the width of a bit field).

constant value must be an int

The constant specified must be representable as the data type `int`.

data segment too large

The data segment, which includes the program's variables and can include some constants, exceeds 64 kilobytes.

declaration exceeded 64K

The size of a declared object exceeded 64 kilobytes, thus exceeding the space available for the data segment.

default not inside switch

A `default` label was specified outside of a `switch` statement.

division by 0

Evaluation of an expression resulted in division by a `0` value.

duplicate case in switch, *number*

The same value, *`number`*, was specified in more than one `case` in the same `switch` statement.

duplicate default in switch

More than one `default` label was specified within the same `switch` statement.

duplicate label

A label was defined more than once within the same function.

duplicate parameter name

The same identifier was found more than once in the identifier list of a function declarator. For example, the following code contains a duplicate `a` identifier:

```
int f(a, a) {}
```

duplicate parameter name in macro

Two arguments in the definition of a macro are identical. Every argument must be unique in the macro definition.

duplicate tag

A `struct`, `union`, or `enum` tag was defined more than once within the same scope.

empty character constant

A character constant should include at least one character or escape sequence.

expression not within range

A register specified 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 `dcb` constant is not in the range of `–128` to `+127`.

FAR qualifier cannot be applied to function

The `far` qualifier was used in the declaration of a function pointer. Only the `farcode` and `nearcode` controls determine whether a function is placed in the farcode segment. All functions in all modules must reside in the same segment. Remove or relocate the `far` qualifier.

FAR qualifier cannot be applied to function result

The `far` qualifier was used in the declaration of a function. The result of a function is a value, and has no address. Remove or relocate the `far` qualifier.

FAR qualifier cannot be applied to member

The `far` qualifier was used in the declaration of a component of a structure or union. Only the entire aggregate object may be so qualified. Remove or relocate the `far` qualifier.

floating point operand not allowed

An operand is non–integral, but the operator requires integral operands. That is, `~`, `&`, `|`, `^`, `%`, `>>`, and `<<` all require integral operands.

function body for non–function

A function body was supplied for an identifier that does not have function type; for example:

```
int i {}
```

This error message can also appear when mismatched braces appear in the source code preceding the identified line.

function declaration in bad context

A function is defined (i.e., appears with a formal argument list), but not at module–level. Or, a function declarator with an identifier list, which is legal only for function definitions, was encountered within a function, as in this example:

```
int main(void)
{
  int f(a);
}
```

function level error

This internal error can be caused by an earlier syntax error.

function redefinition

More than one function body has been found for a single function, as in this example:

```
int f() {}
int f() {}
```

illegal array element reference

In–line code assembly statements cannot access stack–based array variables. These variables are declared as `auto` variables. You can only access arrays when they are declared globally or declared as `static` within the function block.

illegal assignment to const object

Constants cannot be modified.

illegal break

A `break` statement appears outside of any `switch`, `for`, `do`, or `while` statement.

illegal character in header name: *hex_value* ***(hex)***

An illegal character was found in the header name of an `#include < >` preprocessor directive.

illegal constant expression

The expression within an `#if` or `#elif` is not built correctly.

illegal constant suffix

The suffix of a number is not `L` or `U`, in either uppercase or lowercase, or a legal combination of the two.

illegal continue

A `continue` statement appears, but not within any `for`, `do`, or `while` statement.

illegal #elif directive

An `#elif` directive is encountered after an `#else` directive.

illegal #else directive

An `#else` directive is encountered after an initial `#else` directive.

illegal field size

Legal field size is 1 to 16 bits for a named field.

illegal floating point constant in exponent

A floating–point exponent must be an integer.

illegal function declaration

Internal error; can be caused by an earlier syntax error.

illegal hex constant

A hexadecimal constant contains non–hexadecimal characters or is without a `0x` or `0X` prefix.

illegal macro redefinition

A macro can be redefined only if the body of the redefined macro is exactly the same as the body of the originally defined macro.

illegal syntax – left parenthesis is expected

The name of a macro that accepts arguments is specified with no argument list, or the argument list is not properly delimited with parentheses.

illegal syntax in a directive line

A syntax error is encountered in a preprocessor directive.

illegal syntax in a directive line – newline expected

A preprocessor directive line is not terminated with a newline.

illegal syntax in an argument list

An argument list in a macro contains misplaced or illegal characters.

illegal use of FAR qualifier

The `far` qualifier was used in the declaration of an automatic (block scope) object. Only file scope and static objects may contain the `far` qualifier. Remove the qualifier or make the object static.

illegal use of NEAR qualifier

The `near` qualifier was used in the declaration of an automatic (block scope) object. Only file scope and static objects may contain the `near` qualifier. Remove the qualifier or make the object static.

incompatible types

The two operands of a binary operator have incompatible types, for example, assigning a non–zero integer to a pointer.

incomplete static object: *name*

The type of an object with static storage class must be complete by the end of the module. For example:

```
static int i[][10][20];  /* is an incomplete static */
                         /* object type */
static int i[5][10][20]; /* completes the type */
```

incomplete type

The compiler detected a variable whose type is incomplete, such as the following example declaration where the type of `s` is not complete if the program contains no previous declaration defining the tag `S`.

```
int f(struct S s)
{ ... }
```

incorrect void usage

The `void` attribute was specified in conflict with another attribute. For example:

```
int f(void, ...);
```

invalid mnemonic

The assembly mnemonic specified after the `asm` keyword is not valid. See Section 7.3 for a list of supported assembler instructions. Also see the *80C196 Assembler User's Guide* for a complete list of assembler instructions.

invalid instruction for model specified

The instruction is not valid for the model specified by the `model` control. See the *80C196 Assembler User's Guide* for a detailed explanation of each instruction.

invalid addressing mode

The indexed addressing mode is not valid in the first operand position of a two– or three–operand instruction or in the second position of a three–operand instruction. See *80C196 Assembler User's Guide* for descriptions of valid operands for each instruction.

invalid attribute for: *function*

The source program attempted to set multiple and conflicting attributes for a function. For example, a `varparams` or `fixedparams` control appears for a function whose calling convention has already been established by use, definition, declaration, or a previous calling–convention control. For another example, a function identifier appears as an argument to an `interrupt` control which appeared in a previous `varparams`, `fixedparams`, or `interrupt` control, or the function identifier has been previously used, defined, or declared.

invalid cast

The following are examples of invalid casts:

- casting to or from `struct` or `union`
- casting a `void` expression to any type other than `void`

invalid dereference

A dereference (the `*` operator) is applied to an expression other than a pointer.

invalid field definition

A field definition appears outside a structure definition or is attached to an invalid type.

invalid function reference, address–of assumed

An expression that evaluates to a function reference was used in any context other than call. For example, `f(*b)`, where `b` is a pointer to a function, generates this error.

invalid index

The identifier specified with an index register is not a file–scope aggregate object (array, structure, union).

invalid interrupt handler

Since interrupt handlers take no arguments and return no value, they must be declared as `void` *`irf`*`(void)`, where *`irf`* is the name of the interrupt function.

invalid label

The destination code address of the instruction must be a C label.

invalid member name

The member name (that is, the right operand of a `.` or `->` operator) is not a member of the corresponding structure or union.

invalid number of arguments

The number of arguments passed to a function does not match the number of parameters defined in the prototype of that function.

invalid number of operands

The number of operands specified in the instruction is incorrect. See the *80C196 Assembler User's Guide*, listed in Chapter 2, for the syntax of each instruction.

invalid object type

A variable declaration specified an invalid data type; for example: a variable of `void` type.

invalid operand: *operand_number*

`operand_number` is a decimal value stating which operand is invalid in the instruction. Probable causes are a byte register was specified where a word register is expected, a constant was not specified where an immediate value is expected, a word–aligned register variable or register number was not specified for the base register, or an operand of the call instruction was not a C function name.

invalid pointer arithmetic

The only arithmetic allowed on pointers is adding an integral value and a pointer, subtracting an integral value from a pointer, or subtracting two pointers of the same type. Any other arithmetic operation is illegal.

invalid redeclaration *name*

An object or function is being redeclared, but not with the same type. For example, a function reference implicitly declares the function as a function returning an `int`. If the actual definition that follows is different, an error results.

invalid register operand

The C variable used as a register operand in the instruction is not valid because it was not declared as a register variable. Declare the C variable with the `register` storage class. This error can also occur if you accessed the address of the register variable with the ampersand (&) address operator in a C statement, then used the variable as a register operand in an in–line code assembly statement. For example, the following in–line assembly code statement generates the error:

```
register int a;
int *p;

p = &a;
asm ld a,#0AH  /* This statement generates the error. */
```

invalid recursive call to nonreentrant function

You cannot recall a nonreentrant function within itself or call it again through a call loop so that the function is activated more than once simultaneously. Make sure that the `reentrant` control is in effect or precede the function name with the `reentrant` keyword.

invalid storage class

The storage class is invalid for the object declared; for example: a module–level object cannot be `auto` or `register`, however, the `register` storage class is valid if the `extend` compiler control is in effect.

invalid storage class combination

You cannot have more than one storage class specifier in a declaration with `noextend` in effect. With `extend` in effect, you can specify `extern register`, `static register`, and (in block scope) `auto register` storage classes.

invalid structure reference

The left operand of a `.` operator is not a structure or a union; or the left operand of a `->` operator is not a pointer to a structure or a pointer to a union. This error message also occurs if an assignment is made from one structure to another of a different type.

invalid type

An invalid combination of type modifiers was specified.

invalid type combination

An invalid type was specified, for example, a function returning an array.

invalid use of void expression

An expression of data type `void` was used in an expression.

left operand must be lval

The left operand of an assignment, or the operand of a `++` or `--` operator must be an lvalue; that is, it must have an address.

macro expansion buffer overflow

Insufficient memory exists for expansion of a macro; the macro is not expanded.

macro expansion too nested

The maximum nesting level of macro expansion has been exceeded. Macro recursion, direct or indirect, can also cause this error.

NEAR qualifier cannot be applied to function

The `near` qualifier was used in the declaration of a function pointer. Only the `farcode` and `nearcode` controls determine whether a function is placed in the farcode segment. All functions in all modules must reside in the same segment. Remove or relocate the `near` qualifier.

NEAR qualifier cannot be applied to function result

The `near` qualifier was used in the declaration of a function. The result of a function is a value, and has no address. Remove or relocate the `near` qualifier.

NEAR qualifier cannot be applied to member

The `near` qualifier was used in the declaration of a component of a structure or union. Only the entire aggregate object may be so qualified. Remove or relocate the `near` qualifier.

nesting too deep

One of the nesting limits described in Chapter 10 has been exceeded.

newline in string or char constant

The new–line character can appear in a string or character constant only when it is preceded by a backslash (\). For example, the following line generates this error:

```
printf("Hello
```

no body for static function = *function_name*

The *`function_name`* function is declared as a static function and is called but is not defined in the module.

no more room for macro body

Argument substitution in the macro has increased the number of characters to more than maximum allowed.

non addressable operand

The `&` operator is used illegally, such as, to take the address of a register or of an expression.

non–constant case expression

The expression in a `case` is not a constant.

nothing declared

A data type without an associated object or function name is specified.

number of arguments does not match number of parameters

The number of arguments specified for the macro expansion does not match the number of arguments specified in the macro definition.

operand stack overflow
operand stack underflow

An illegal constant expression exists in a preprocessor directive line.

operand too large

Constant specified in shift count, `dcb` operand, bit count, etc. is too large.

operator not allowed on pointer

An operand is a pointer, and the operator requires non–pointer operands; for example: `&`, `|`, `^`, `*`, `/`, `%`, `>>`, `<<`).

operator stack overflow
operator stack underflow

An illegal constant expression appears in a preprocessor directive line.

parameter list cannot be inherited from typedef

A function body was supplied for an identifier that has function type, but whose type was specified with a `typedef` identifier, as in the following example:

```
typedef void func(void);

func f {}
```

parameters can't be initialized

An attempt was made to initialize the arguments in a function definition.

respecified storage class

A storage class specifier is duplicated in a declaration.

respecified type

A type specifier is duplicated in a declaration.

respecified type qualifier

A type qualifier is duplicated in a declaration.

sizeof invalid object

An implicit or explicit `sizeof` operation references an object with an unknown size. Examples of invalid implicit `sizeof` operations are `*`*`fp`*`++`, where *`fp`* is a pointer to a function, or `struct` *`sigtype siga`*, when *`sigtype`* is not yet completely defined.

stack segment too large

The estimated or requested stack size is greater than 64 kilobytes.

string too long

A string of over 1024 characters is being defined.

syntax error

An error is discovered in the syntax of an assembly instruction.

syntax error near 'string'

A syntax error occurred in the program. The *`string`* information attempts to identify the error more precisely.

too many parameters for one function

The number of arguments specified for one function has exceeded the compiler limit.

too many parameters for one macro

The number of arguments specified for one macro has exceeded the compiler limit.

too many characters in a character constant

A character constant can include one to two characters. The effect of this error on the object code is that the character constant value remains undefined.

too many functions

The number of functions declared has exceeded the compiler limit.

too many initializers

An array is initialized with more items than the number of elements specified in the array definition.

too many macro arguments

The number of arguments specified for a macro has exceeded the compiler limit.

too many nested struct/unions

The lexical nesting of `struct` and `union` member lists has exceeded the compiler limit.

too many public register variables

The number of public variables explicitly declared as `register` variables is greater than the number of register locations available to the module.

too many register variables

The number of variables explicitly declared as `register` variables is greater than the number of register locations available to the module.

unable to recover from syntax error

An unrecoverable syntax error has occurred. Check the list file to see where the compiler found the error.

unbalanced conditional compilation directive

Conditional compilation directives are improperly formed. For example, the program contains too many `#endif` preprocessor directives, or an `#else` preprocessor directive without a matching `#if` preprocessor directive.

undefined identifier: *name*

The program contains a reference to an identifier that has not been previously declared.

undefined label: *label*

A label has been referenced in the function, but has never been defined.

undefined or not a label

An identifier following a `goto` must be a label; the identifier was declared otherwise, or the label was not defined.

undefined parameter

The argument being defined did not appear in the formal parameter list of the function.

unexpected EOF

The input source file or files ended in the middle of a token, such as a character constant, string literal, or comment.

unit string literal too long; truncated

The maximum length of a string is 1024 characters.

variable reinitialization

An initializer for this variable was already processed.

void function cannot return value

A return with an expression is encountered in a function that is declared as type `void`. In `void` functions, all returns must be without a value.

9.5 WARNINGS

Warnings have the following syntax:

```
*** Warning at line nn of filename: message
```

where:

filename	is the name of the file in which the error occurred.
nn	is the source line number where the error is detected.
message	is the explanation.

Following is an alphabetic list of warnings.

a #endif directive is missing

At least one `#endif` preprocessor directive is missing at the end of an input source file. The `#if` and `#endif` preprocessor directives are not balanced.

argument not allowed for** control **control

This message indicates an attempt to pass arguments to a control that accepts none. Improper argument passing is a fatal error if it occurs in the compiler invocation, but the preprocessor only issues a warning if the improper argument occurs in a `#pragma` directive.

argument not allowed for negated** control **control

Negated controls, except for the `noreentrant` control, do not accept arguments. An improper argument for a negated control is a fatal error if it occurs in the compiler invocation, but the preprocessor only issues a warning if the improper argument occurs in a `#pragma` directive line.

***argument out of range for** control **control:** arg*

This message indicates an attempt to use an argument value that is out of the valid range. An out–of–range argument is a fatal error if it occurs in the compiler invocation, but the preprocessor only issues a warning if the improper argument occurs in a `#pragma` directive.

argument required for** control **control

A missing required argument is a fatal error if it occurs in the compiler invocation, but the preprocessor only issues a warning if the missing argument occurs in a `#pragma` directive.

argument too long for** control **control

The length of the argument to the control exceeds the maximum number of characters allowed by the compiler.

bad octal digit:** hex_value **(hex)

An octal number contains a non–octal character. The *hex_value* is the ASCII value of the illegal character.

comment extends across the end of a file

A comment started in a file is not closed before the end of the file.

comparison of signed and unsigned value

This warning is generated when one of the operands in an `<`, `>`, `<=`, or `>=` operation has a signed type, and the other operand has an unsigned type, but only if the unsigned value is at least as wide as the signed value. The usual conversions are done before the comparison, if needed.

const declaration made non–register

The `const` qualifier was specified in an initialized data declaration in file scope that otherwise would have placed the declared object in the register segment. The constant object will instead be placed in the constant segment.

control control cannot be negated

You cannot use the `no` prefix with this compiler control. Improper negating is a fatal error if it occurs in the compiler invocation, but the preprocessor only issues a warning if it occurs in a `#pragma` directive.

control ***control not allowed in pragma***

The compiler encountered either a `define` or an `include` control in a `#pragma` preprocessor directive.

different enum types

An attempt was made to assign one `enum` type to a different `enum` type.

directive line too long

The line length limit for `#pragma` preprocessor directives was exceeded.

division by 0

Evaluation of an expression resulted in division by a `0` value.

duplicate** control **control

A primary control that must not be specified more than once was specified more than once. Only the following controls can be specified more than once:

```
define          reentrant
fixedparams     regconserve
include       searchinclude
interrupt       varparams
```

If you specify a compiler control both in the compiler invocation and in a `#pragma` preprocessor directive, the compiler invocation specification takes precedence. A duplicate control is a fatal error if it occurs in the compiler invocation but the preprocessor only issues a warning if it occurs in a `#pragma` directive.

duplicate argument** argument **for** control **control

An argument for *`control`* was specified more than once; for example: specifying more than one handler for one interrupt number.

escape sequence value overflow

The value of an octal or hexadecimal escape sequence does not fit in one byte.

***extra characters in pragma ignored:** string*

The *`string`* represents characters that the compiler cannot process as part of the current `#pragma`.

FAR qualifier requires 24–bit model

The `far` qualifier was used in a declaration, but a 24–bit `model()` control was not specified. The `far` qualifier is valid only for 24–bit models. Add a 24–bit `model()` control or remove the offending qualifier.

FARCODE conflicts with NEARCODE

These two controls determine the segment into which all code is generated. They are mutually exclusive, but both controls were explicitly specified. Eliminate the one you don't want.

FARCODE control invalid for the component

The `farcode` control is valid only for 24–bit models. Add a 24–bit `model()` control or remove the offending control.

FARDATA conflicts with NEARDATA

These two controls determine the default location of non–constant, non–register data. They are are mutually exclusive, but both controls were explicitly specified. Eliminate the one you don't want.

FARDATA control invalid for the component

The `fardata` control is valid only for 24–bit models. Add a 24–bit `model()` control or remove the offending control.

filename too long; truncated

The filename length exceeded the limit of the operating system.

***fixedparams attribute ignored for** function*

The `fixedparams` control was specified for the function, or the function was declared with the `alien` keyword, but the prototype contained the `,...` construct. The compiler has changed the calling convention to `varparams`, on the assumption that the variable parameter list will be used. Specify the function in a `varparams` control or don't use the `,...` construct.

***fixedparams attribute ignored for:** function*

This function has been specified in a `fixedparams` control or in a `#pragma` directive line, but its parameter list ends with comma and ellipsis (`,...`), for example, `func(a,b,c,...)`. The function uses the `varparams` calling convention.

function exits without returning a value

Be sure to use a return statement when a function requires one.

illegal character: *hex_value* ***(hex)***

The character with the ASCII value `hex_value` is not part of the C196 character set.

illegal escape sequence

The sequence following the backslash is not a legal escape sequence. The compiler ignores the backslash and prints the sequence.

illegal macro definition: *macro_name*

An invalid macro was defined on the command line with the `define` control.

illegal syntax in a directive line – newline expected

A preprocessor directive line is not terminated with a new–line character.

incomplete definition of *name****, one element assumed***

No completing definition of *name* was found in the module. For example, somewhere in the program, the following declaration exists:

```
int xyz[];              /* No size was declared
                           inside the []'s. */
```

or

```
void (*xyz[]) (void)    /* Same. */
```

The compiler issues the warning, at the end of the file, when it does not find another declaration of the array declaring its true size.

indirection to different types

A pointer to one data type was used to reference a different data type.

invalid argument for *control* ***control***

The argument specified for `control` is not valid. For example, the argument specified for `model` is invalid.

invalid control: *control*

A control not supported by the compiler was specified. Check the spelling of the control. An invalid control is a fatal error if it occurs in the compiler invocation but the preprocessor only issues a warning if the invalid control occurs in a `#pragma` directive.

invalid decimal argument: *value*

Non–decimal characters were found in an argument that must be a decimal value. An improper argument is a fatal error if it occurs in the compiler invocation, but the preprocessor only issues a warning if the improper argument occurs in a `#pragma` directive.

invalid identifier: *identifier*

An identifier does not follow the rules for forming identifiers in C. An invalid identifier is a fatal error if it occurs in the compiler invocation, but the preprocessor only issues a warning if the invalid identifier occurs in a `#pragma` directive.

invalid syntax for *control* ***control***

The compiler control contained a syntax error. Invalid control syntax is a fatal error if it occurs in the compiler invocation, but the preprocessor only issues a warning if the invalid syntax occurs in a `#pragma` directive.

null argument for *control* ***control***

Null arguments for compiler controls are not allowed. For example, the following argument is illegal:

```
varparams(f1,,f2)
```

A null argument is a fatal error if it occurs in the compiler invocation, but the preprocessor only issues a warning if the null argument occurs in a `#pragma` directive.

missing left brace

An aggregate initializer list must be enclosed in braces; for example:

```
const int i[] = {1,2,3};
```

no body for static function = *function_name*

The *`function_name`* function is declared as a static function but is neither defined nor called in the module.

pragma ignored

An entire `#pragma` preprocessor directive was ignored as a result of an error. Whenever an error is found in a `#pragma` preprocessor directive, the diagnostic is followed by either this message or `remainder of pragma ignored`, whichever is appropriate.

predefined macros cannot be deleted/redefined

The predefined macros (e.g., `__LINE__` or `__FILE__`) cannot be deleted or redefined by the preprocessor directives `#define` or `#undefine`.

qualifier ignored for bit fields

You cannot use a type qualifier with bit field members of a structure or union.

redefined attribute ignored for: *function*

Calling convention (`varparams` or `fixedparams`) or reentrancy for the function name has already been established with a declaration, definition, or compiler control.

regconserve conflicts with registers(all)
registers(all) conflicts with regconserve

The `registers(all)` control specifies that the compiler is to allocate all program variables to registers, including variables declared without the explicit `registers` attribute (`register` variables). The `regconserve` control specifies that the compiler is to conserve registers, placing only `register` variables (and, optionally, a subset of the non–`register` variables) in registers. The compiler cannot resolve these conflicting directions. This error is fatal if both controls are specified in the compiler invocation, but the preprocessor only issues a warning if the conflict occurs in `#pragma` directives.

register declaration too large

The number of `register` variables declared with block scope is greater than the number of registers available to the module. The `register` storage class is ignored for some variables.

remainder of pragma ignored

This message indicates that a `#pragma` preprocessor directive is partially ignored as a result of an error. Whenever an error is found in a `#pragma` preprocessor directive, the message is followed by either this message or `pragma ignored`, whichever is appropriate.

shift count out of range

The number of shifts you specified exceeds the number of bits in the register operand, for example, `asm shl wreg, #17`. This example issues a warning because a word operand only contains 16 bits.

token too long; ignored from character: *hex_value* ***(hex)***

The length of a character sequence, such as an identifier or a macro argument, has exceeded the compiler limit.

too many register variables

The number of variables explicitly declared as `register` variables has exceeded the compiler limit. This limit is either the processor limit or is imposed by the `registers` control. Use a different argument for the `registers` control or declare fewer variables as `register` variables.

undefined tag

A tag was used before its definition was completed.

zero or negative subscript

The value of an array subscript must be a positive integer.

9.6 REMARKS

Remarks have the following syntax:

```
*** Remark at line nn of filename: message
```

Following is an alphabetic list of remark messages.

a constant in a selection statement

A constant is encountered in the expression of a selection statement such as an `if`, `else`, or `switch` statement.

comparison of signed and unsigned value

This remark is generated when one of the operands in an `==` or `!=` operation has a signed type, and the other operand has an unsigned type, but only if the unsigned value is at least as wide as the signed value. The usual conversions are done before the comparison, if needed.

interrupt pragma should precede the function definition of: *name*

In the source module the compiler expects to find the interrupt designation before the actual function definition.

invalid number of parameters

The actual number of arguments in a function call do not agree with the number of parameters in a function definition that is not a prototype.

Name hides usage of variable with same name

This remark is generated when a variable at an inner block has the same name as a variable at an outer block. In this case the inner block variable hides the other variable.

NEAR qualifier requires 24-bit model

The `near` qualifier was used in a declaration, but a 24–bit `model()` control was not specified.

Precision lost in cast

A cast expression `long` to pointer (non 24–bit) or pointer to `int` loses precision.

return statement has no expression

A return statement with no return expression is encountered in a function definition which returns an expression other than `void`.

reuse of interrupt function: *func_name* ***for interrupt:*** *dec_value*

The same interrupt handler had been assigned to handle another interrupt, represented by *`dec_value`*.

tag scope ends in current block

A tag is defined either in a formal parameter list or at block scope and will go out of scope at the end of the block containing the definition.

the characters /* are found in a comment

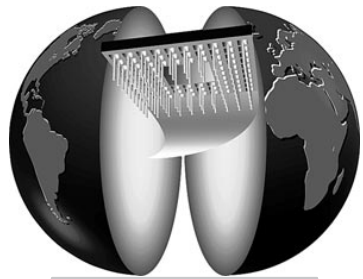
A comment–start delimiter (`/*`) occurs within a comment.

value overflows field

The specified initial value is too large to be contained in the corresponding bit field. One or more high order bits of the value have been truncated.

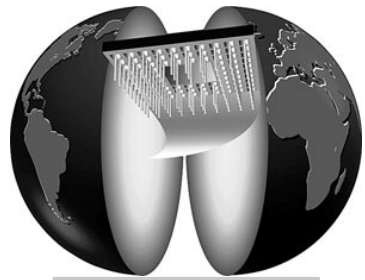
CHAPTER 10

LANGUAGE IMPLEMENTATION



TASKING
Quality Development Tools Worldwide

CHAPTER 10



This chapter describes compatibility issues regarding data types and calling conventions when linking modules written in other languages for the 80C196 processor with C196 modules. It also describes C196 conformance to ANSI C and explains how C196 implements some characteristics of the C language.

10.1 DATA REPRESENTATION

A large application can consist of many separate modules. Linking combines the modules before execution to satisfy references to external symbols. Although other modules can be written in PL/M–96, ASM196, or an older version of Intel C Compiler for the MCS®–96 processor, variables referenced by external symbols must be represented in memory in a format compatible with C196 data type representations, as described in this chapter.

10.1.1 DATA TYPES

The C196 compiler supports all ANSI data types except wide characters. *C: A Reference Manual*, listed in *Related Publications*, describes the ANSI data types. Floating–point data types in C196 are always 32 bits.

Table 10–1 shows the scalar data types for the 80C196 processor, the amount of memory occupied by the data type, the arithmetic format, and the range of accepted values.

Data Type	Size in Bytes	Format	Range
char[1]	1	integer or two's–complement integer	0 to 255 (unsigned char) or –128 to 127 (signed char)
unsigned char	1	integer	0 to 255
signed char	1	two's–complement integer	–128 to 127
unsigned int	2	integer	0 to 65,535
int	2	two's–complement integer	–32,768 to 32,767
unsigned short		same as unsigned int	
short		same as signed int	
unsigned long	4	integer	0 to 4,294,967,295
long	4	two's–complement integer	–2,147,483,648 to 2,147,483,647

Data Type	Size in Bytes	Format	Range
float	4	single–precision floating–point	8.43×10^{-37} to 3.37×10^{38} (approximate absolute value)
double		same as float	
long double		same as float	
bit field[2]	1 to 16 bits	integer or two's complement integer	depends on number of bits
near pointer	2	address	64 kilobytes
far pointer	4	address	16 megabytes
enum	2	two's complement	–32768 to 32,767

[1] unsigned char if the nosigned char control is in effect, or signed char if the signedchar control is in effect
[2] occurs only as a member of a structure or union aggregate data type

Table 10–1: 80C196 processor scalar data types

A character constant can contain up to two characters and is stored in character format, one byte per character. The rightmost character in the constant occupies the low–order byte. A character constant operates as an `unsigned char` data type.

10.1.2 CONTIGUITY

Variables reside in memory from low–order to high–order bytes within a word and from low address to high address across multiple bytes. The address of a variable is the location of the low–order byte of the variable. Scalar variables longer than one byte and aggregate variables that contain word–aligned members are word–aligned, starting on even byte addresses and occupying consecutive words in memory. Scalar variables shorter than one word (`char`, `signed char`, and `unsigned char` variables) and aggregate variables that contain only unaligned members are byte–aligned, starting on any byte address and occupying consecutive bytes in memory. Register variables that are a multiple of four bytes in length are longword–aligned, unless the wordalign control is in effect.

The alignment of variables affects the amount of memory space occupied by the program's data. The compiler attempts to realign data items to optimize the memory space used. This realignment can result in an arrangement of the declared items in memory different from the arrangement of the declarations in the source text. Figure 10–1 shows an example of the memory allocation corresponding to a set of declarations. The variables occupy the low–order byte first, starting from bit `0`.

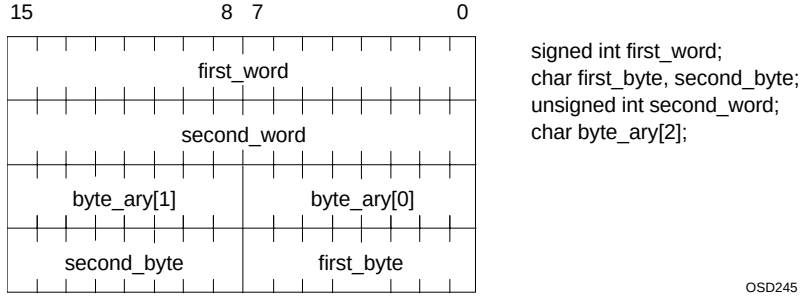
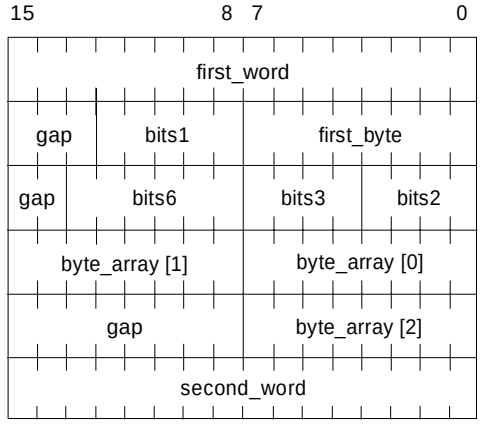


Figure 10–1: Contiguity of variables

10.1.3 ALIGNMENT

Members of an aggregate variable occupy contiguous storage in the order specified in the declaration. Byte gaps are introduced as needed for alignment. Figure 10–2 shows the memory allocation of a structure. The compiler places the structure at a word–aligned location since the structure contains members that must be aligned. A gap appears between the last byte of `byte_array` and the following integer variable (`second_word`) because `second_word` must start on an even byte address.



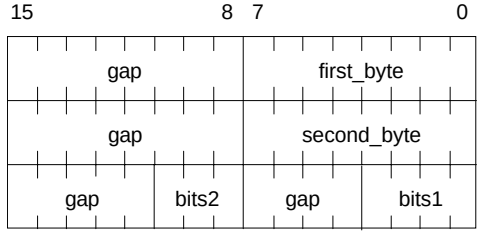

```
struct gaps
{
  int first_word;
  char first_byte;
  unsigned bits1 : 5;
  unsigned bits2 : 4;
  unsigned bits3 : 4;
  unsigned bits6 : 6;
  char byte_array [3];
  int second_word;
};
```

OSD246

Figure 10–2: Alignment of structure members

Bit fields (members of structures or unions) are not necessarily aligned on byte or word boundaries. A bit field cannot span a word boundary, but it can span a byte boundary. The compiler allocates two or more adjacent bit fields to a single word whenever possible.

You can use bit fields for padding to force a structure to conform to an externally imposed format. If no field name precedes the field–width expression, the compiler allocates an unnamed field of the specified number of bits. An unnamed field with a length of zero creates a gap until the next word boundary. Figure 10–3 shows a structure allocation using bit fields for padding.




```
struct pad
{
  char first_byte;
  unsigned :0;
  char second_byte;
  unsigned : 0;
  unsigned bits1 : 4;
  unsigned : 4;
  unsigned bits2 : 3;
};
```

OSD247

Figure 10–3: Alignment of Structure Members With Padding

The overlay segment the compiler generates is word–aligned. The compiler adds one byte to the size of an odd–size overlay segment. This additional byte can cause the compiler to use one more byte of registers than what you have specified in the `registers` control, if any.

10.2 CALLING CONVENTIONS

This section describes the four sections of object code (shown in Figure 10–4 that the compiler generates to handle a function call, as follows:

setup — is code in the calling function that the processor executes just before control transfers to the called function.

cleanup — is code in the calling function that the processor executes just after control returns from the called function.

prolog — is code in the called function that the processor executes first when control has transferred from the calling function.

epilog — is code in the called function that the processor executes just before control returns to the calling function.

The calling convention determines the contents of each of these four sections of code.

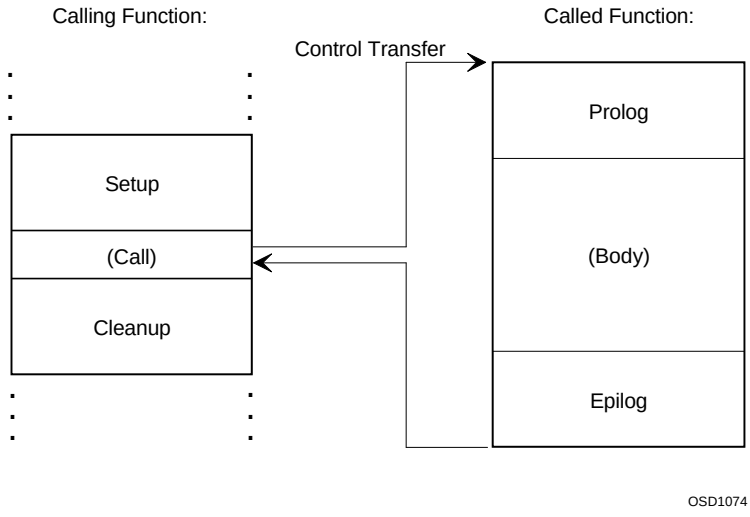


Figure 10–4: The four sections of code for a function call

The C196 compiler supports two calling conventions: fixed–parameter list (FPL) and variable–parameter list (VPL). The object code for the calling function and for the called function must use the same convention; otherwise, incorrect execution can occur. The C196 compiler uses VPL as its default calling convention. To specify FPL for a function, you can use either the `fixedparams` control or the `alien` keyword. Use FPL for external functions defined in a PL/M–96 module.

10.2.1 PASSING ARGUMENTS

The calling convention determines the order in which arguments occupy the stack. In both VPL and FPL, the setup code of the calling function pushes all arguments onto the stack using pass–by–value. Each argument on the processor stack occupies a multiple of two bytes and is pushed from the higher address to the lower address. If the size of the argument is less than two bytes, the compiler zero–extends or sign–extends to two bytes depending on the data type of the argument.The compiler allocates space on the stack as follows:

- A floating–point value occupies two words (32 bits).
- A non–floating–point, 32–bit, scalar value occupies two words.
- A 16–bit scalar value occupies one word.
- An aggregate value occupies the same number and sequence of words on the stack that it does in memory, extended to the next higher whole word if necessary.

In the VPL convention, the calling function pushes the rightmost argument in the function call first and the leftmost argument last. Therefore, the first argument in the function call occupies the lowest memory location of all the arguments on the stack. The cleanup code of the calling function pops all the arguments off the processor stack after the called function returns control.

In the FPL (PL/M–96) convention, the calling function pushes the leftmost argument in the function call first and the rightmost argument last. Therefore, the first argument in the list occupies the highest memory location of all the arguments on the stack for this function call. The epilog of the called function pops all the arguments off the processor stack before returning control to the calling function.

10.2.2 RETURNING A VALUE

In both the VPL and FPL calling conventions, the epilog of the called function returns a scalar value in the global double–word register, `TMPREG0`. For aggregate return values, `TMPREG0` contains a pointer to a temporary aggregate variable.

10.2.3 LOCAL VARIABLES

The prolog allocates space on the stack for local variables. This space is commonly called a frame. The `?FRAME01` variable is a relocatable word register that points to the beginning of the frame and is commonly called the frame pointer. A module named `FRAM01`, in the `c96.lib` library, defines `?FRAME01` and allocates a word register for it. The pseudo–assembly language listing in the print file, produced by the compiler, shows how the compiler uses `?FRAME01`. To find the address of the `?FRAME01` variable, examine the map file produced by RL196.

If your C196 function calls or is called by an ASM196, you must know the possible differences in stack usage for local variables. For example, since the C196 compiler does not support nested function definitions, it uses only one frame pointer.

The following example demonstrates the pseudo–assembly listing generated by compiling a function that has three local variables declared as integers. Since each integer occupies a word, the frame size is 3 words or 6 bytes long. The prolog of the called function uses a frame and frame pointer as follows:

```
sub  SP,#6          ;Allocates space for local variables of
                    ;the called function.
push ?FRAME01       ;Saves the frame pointer of the calling
                    ;function, to allow for reentrancy.
ld   ?FRAME01,SP    ;Loads the stack pointer of the called
                    ;function into the frame pointer.
```

Figure 10–5 contains the print file of a compiled program (named `exfrm` in this example) that uses a frame and frame pointer. In this program, the variables `a`, `b`, and `c` are represented as follows:

```
[?FRAME01]          ; The old frame pointer saved on the stack
2[?FRAME01]         ; The local variable 'a'
4[?FRAME01]         ; The local variable 'b'
6[?FRAME01]         ; The local variable 'c'
```

The compiled program uses the frame pointer as follows:

```
sub   SP,#6H              ;These first three instructions set up the
push  ?FRAME01            ;stack frame.  ?FRAME01 is set up as
ld    ?FRAME01,SP         ;a pointer to local variables a, b, and c.

ld    Tmp0,4[?FRAME01]    ;Load the contents of the variable
                          ;'b' into a temporary register.

add   Tmp0,6[?FRAME01]    ;Add the variable 'c' to the variable
                          ;'b' which was stored in TMP0,
                          ;then store the sum in TMP0.

st    Tmp0,2[?FRAME01]    ;Store the result of the
                          ;addition into the variable 'a'.

pop   ?FRAME01            ;Restore the old frame pointer.
add   SP,#6H              ;Free the local variable space on
                          ;the stack.
ret                       ;Return to the calling procedure.
```

The epilog restores the previous value of the frame pointer and deallocates the space allocated for the frame on the stack. Control then returns to the calling function with the stack as it was when the called function began execution.

The map file, produced by RL196, lists the address of `?FRAME01`. For example, linking the `exfrm.obj` object file with the following RL196 invocation produces the `exfrm.m96` map file:

```
rl196 cstart.obj,exfrm.obj,c96.lib to exfrm.abs print(exfrm.m96)
```

Figure 10–6 contains a section of the `exfrm.m96` map file. The address of `?FRAME01`, `1AH` in this example, appears in the `VALUE` column, beside the `?FRAME01` entry in the `NAME` column. The local variables reside on the stack immediately following the top of the frame.

```
C196  Compiler   EXFRM                   01/29/99 17:56:19  Page   1

80C196 Compiler Vx.y Rz SN (C)1993 Tasking BV, Compilation of module EXFRM
(C)1980,1990,1992,1993 Intel Corporation

Object module placed in EXFRM.obj
Compiler invoked by: c:\c196\bin\C196.EXE EXFRM.c code xref

 Line Level  Incl

      1            main()
      2            {
      3      1        int a,b,c;
      4      1        a = b + c;
      5      1     }


                                 Symbol Table

  Name        Size   Class  Address       Attributes

  a                  2      Auto   2      int in function(main)
                                           *3, 4
  b                  2      Auto   4      int in function(main)
                                           *3, 4
  c                  2      Auto   6      int in function(main)
                                           *3, 4
  main               Public        reentrant VPL function
                                           returning int

                     Assembly Listing of Object Code
                                           ;   Statement     2
               0000                     main:
               0000  69060018                    sub   SP,#6H
               0004  C800           E      push  ?FRAME01
               0006  A01800         E      ld    ?FRAME01,SP
                                           ;   Statement     4
               0009  A300041C             E      ld    Tmp0,b[?FRAME01]
               000D  6700061C             E      add   Tmp0,c[?FRAME01]
               0011  C300021C             E      st    Tmp0,a[?FRAME01]
                                           ;   Statement     5
               0015  CC00           E      pop   ?FRAME01
               0017  65060018                    add   SP,#6H
               001B  F0                          ret
```

Figure 10–5: Print file

```
ATTRIBUTES    VALUE  NAME
----------    -----  ----

                     PUBLICS:
CODE ENTRY    2083H    MAIN
REG  WORD     001AH    ?FRAME01
REG  NULL     001CH    TMPREG0
NULL NULL     002EH    MEMORY
NULL NULL     1FD2H    ?MEMORY_SIZE
```

Figure 10–6: Map file illustrating frame pointer

10.2.4 REENTRANT FUNCTIONS

The prolog of a reentrant function includes code, if necessary, to save all registers that are to be used by the function. The epilog includes code to restore the saved registers to the values used by the calling function.

The prolog of each reentrant public function contains a statement or statements pushing a term. The number of `push` statements depends upon the number of overlayable registers (local register variables) defined inside the function. The compiler uses the symbol `?OVRBASE` to keep track of the offset into the relocatable overlay segment created during compilation. The RL196 linker locates this overlay segment during linkage. The compiler pushes the `?OVRBASE` values onto the stack to preserve the overlayable registers so the function can use the same locations in the register memory even if the functions are active simultaneously. The compiler includes the `?OVRBASE` variable in your output object file, for example, for the following code:

```
foo1()
{
   register char a, b, c;
   a = b = c;
}
```

The compiler produces the following code:

```
foo1:
     push  ?OVRBASE
     push  ?OVRBASE+2H
     ldb   b,c
     ldb   a,b
     pop   ?OVRBASE+2H
     pop   ?OVRBASE
     ret
```

10.2.5 INTERRUPT FUNCTIONS

A call to an interrupt function always results in more object code than a call to an equivalent non–interrupt function. First, the compiler generates code to preserve the Program Status Word (PSW). Since all interrupt functions are assumed to be reentrant, the compiler also generates code to save and restore registers used by the interrupt function.

In the prolog of the interrupt function, the first instruction pushes the PSW onto the stack and clears the PSW. This action sets the interrupt mask to zero and disables interrupts. Code for the 8096 processor uses a `pushf` instruction; code for the 80C196 processors use a `pusha` instruction.

In the epilog of the interrupt function, the last instruction pops the saved PSW off the stack. This action restores the processor state to what it was before the interrupt. Code for the 8096 processor uses a `popf` instruction; code for the 80C196 processors use a `popa` instruction.

The compiler pushes/pops all temporary registers when an interrupt function calls any other function. This has been done for savety reasons. If possible it will be more efficient not to call any function at all. In that case only the used registers will be saved/restored.

10.3 STACK SIZE CALCULATION

For each function the C196 compiler calculates its stack size requirements. The results of these calculations can be seen in the 'Function Statistics' as they are printed for each function in the pseudo–assembly listing (see section 3.4.2.7).

The C196 compiler also calculates the total stack size required by each module. This result can be found in the 'Module Information' in the list file (see section 3.4.2.8). The stack size calculated is correct as long as there are no recursive function calls in the source. If there are any recursive function calls, the compiler will generate a warning and the user should reserve additional room for the stack while linking. This can be done with the `ss` control (see the linker documentation for that).

10.4 IMPLEMENTATION-DEPENDENT C196 FEATURES

This section provides information about how C196 implements some characteristics of the C language as specified by the ANSI C standard. The `__STDC__` macro, defined as `0`, indicates that the compiler does not conform strictly to the ANSI C standard.

10.4.1 CHARACTERS

The C196 source character set is 7–bit ASCII, except in comments and strings, where it is 8–bit ASCII. The execution character set is 8–bit ASCII. The compiler maps characters one–to–one from the source to the execution character set. You can represent all character constants in the execution character set.

10.4.2 IDENTIFIERS

The C196 compiler supports 40–character significance in external and internal names. The compiler forces external names to uppercase. Case is significant in internal names.

10.4.3 EXTENDED SEMANTICS AND SYNTAX

The C196 compiler supports the `alien`, `reentrant`, and `nonreentrant` keywords, and allows file–level `register` variables when the `extend` control is in effect. These C196 storage–class specifiers operate as follows:

`alien` has the same effect as specifying the `fixedparams` control for the function.

`reentrant` has the same effect as specifying the `reentrant` control.

`nonreentrant` has the same effect as specifying the `noreentrant` control.

The C196 compiler allows an extended syntax for type qualifiers that does not conflict with ANSI C.

In C196, a qualifier can follow a left parenthesis or comma. For example, the following line is not valid in ANSI C:

```
int (const i), volatile j;
```

However, the C196 compiler recognizes this line as equivalent to the following:

```
int const i;
int volatile j;
```

This extension does not affect the semantics of any source text that follows the rules of ANSI C but does cause an asymmetry. For example, the first of the following two declarations causes `x`, `y`, and `z` all to be read–only variables. The second declaration causes only `y` to be read–only; `x` and `z` are both modifiable:

```
int const x, y, z;      /* valid for ANSI C */
int x, const y, z;      /* C196 extended syntax */
```

10.4.4 INITIALIZATION

The C196 compiler supports initialization of object at both file and block scope with the new OMF version 3.2. If you specify either `omf(0)` or `omf(1)` on the invocation line, initialization of non–constant file scope objects is not allowed.

Examples of constant file–scope (global) initializations are:

```
const int i = 1;

long l;
long *const lp = &l;

const struct {  int i, j; } s1 = { 1, 2 };
```

These examples can be used with both the `omf(1)` and the `omf(2)` control.

The C196 compiler also supports initialized variable data in RAM. This feature affects only non–constant data. Constant data, of course, has always been initialized. This feature is only available with OMF level 2 or above.

The following examples of file–scope initializations are only valid with the default OMF version 3.2. If you use `omf(0)` or `omf(1)` the following examples are invalid:

```
int si = 1;

struct { const int i;
         int j;
} s2 = {3};

static register int sri = 1;

const register int cri = 1;
```

The following code is an example of block–scope initialization:

```
void foo(void)
{
    int ai = 0;               /* automatic object */
    int * aip = &ai;
    char ach[2] = {"ab"};

    const long al = 1;

    const struct {
        int i1, i2;
    } as1 = {1 ,2};


struct {  const int ai;
          int aj;
} as2 = {2, 3};

static const register int cri = 1;  /* valid, constant
                                        object */

static int si = 2;            /* invalid, non–const
                                 object with static
                                 storage duration */
static register int sri = 1;  /* cannot be
                                 initialized.  */
}
```

Character string initializers within a character array are null–terminated unless the array is shorter than the initializing string. For example:

```
str1[ ]="test";  /* value is "test\0" (null-terminated)   */
str2[5]="test";  /* value is "test\0" (null-terminated)   */
str3[4]="test";  /* value is "test" (not null-terminated) */
```

The compiler produces additional register, overlay, data and far data segments for the initialized variables, along with corresponding const segments which provide the initial values.

In addition to the new segments, the compiler also produces table entries that the linker then combines to build an initialization table in the final absolute module.

At startup (reset), library module `_main` processes the initialization table: it copies the initial constant data to the corresponding variables, and zeroes the uninitialized variables. You may modify the `_main` module if you do not need the initialization and wish to save space in ROM by eliminating the initialization routine.

There are up to five data segments with initial values, one for each of the register, overlay, data and far data segments, and one for all the absolute segments (produced by the `locate` pragma and the `_reg` storage class modifier).

The initialization segments corresponding to the four relocatable segments are congruent to those segments, and need only one table entry each. The single initializaton segment for all the absolute segments requires one table entry for each such variable, since they can be scattered all over the place.

The following pragmas/controls can be used for data initialization:

init | ***noinit***

allows the compiler to produce the initialization segments and tables. The default setting is `init`. `init` can be abbreviated `it`.

Use the `noinit` control/pragma to prevent the generation of initializing data and tables, even though you have used initializers in your source code (`noinit` also prevents zeroing of uninitialized variables).

zero | ***nozero***

allows the compiler to zero uninitialized variables in relocatable data segments. The default setting matches the setting of the `init` control. `noinit` forces `nozero`. `zero` can be abbreviated `zr`.

Use the `nozero` control/pragma to prevent the generation of zeroing entries in the initialization tables for relocatable segments (ordinary variables).

abszero* | *noabszero

tells the compiler to zero uninitialized variables in absolute segments. The default setting is `noabszero`. `noinit` forces `noabszero`. `abszero` can be abbreviated `az`.

Use the `abszero` control/pragma to enable the generation of zeroing entries in the initialization tables for absolute segments (variables positioned by either `#pragma locate` or the `_reg` storage class modifier).

10.4.5 DATA TYPE CONVERSION

An unsigned integer is sufficient to hold the maximum size of an array and can hold the difference between two pointers to members of the same array.

The result of casting a pointer to an integer data type is as follows:

- Casting a near pointer to an `int` or `short` preserves the bit representation. If cast to a signed integer, the result can be negative.
- Casting a near pointer to a `long` or `char` is not supported.
- Casting a far pointer (24–bit model) to a `long` preserves the bit representation.
- Casting a far pointer (24–bit model) to an `int` or `char` is not supported.

The result of casting an integer data type to a pointer is as follows:

- Casting a signed `char` to a pointer sign–extends.
- Casting an unsigned `char` to a pointer zero–extends.
- Casting a `char` to a pointer sign–extends or zero–extends, depending on whether `signedchar` or `nosignedchar` is in effect.
- Casting an `int` or `short` to a near pointer preserves the bit representation.
- Casting a `long` to a near pointer is not supported.
- Casting a signed `int` or `short` to a far pointer (24–bit model) sign–extends.

- Casting an unsigned `int` or `short` to a far pointer (24–bit model) zero–extends.
- Casting a `long` to a far pointer (24–bit model) preserves the bit representation.

The result of casting one size pointer to the other size pointer (24–bit model) is as follows:

- Casting a near pointer to a far pointer zero–extends.
- Casting a far pointer to a near pointer is not supported.

The compiler represents enumeration types as `int`.

The `[no]signedchar` control determines whether the compiler considers a `char` that is declared without the `signed` or `unsigned` keywords to be signed or unsigned.

10.4.6 BIT FIELDS

You must declare bit fields as `signed int`, `unsigned int`, or just `int`; otherwise, the compiler issues an error.

The allocation of bit fields in a word is from low address to high address.

Bit fields are not necessarily allocated on word boundaries; if a bit field is short enough, it occupies the space between the end of the previous bit field and the beginning of the next word.

10.4.7 DIVISION/REMAINDER OPERATORS

The binary operator `/` indicates division. Operands may be of any arithmetic type. The type of the result is that of the converted operands.

For integral operands, if the mathematical quotient is not an exact integer, then the result will be one of the two integers closest to the mathematical quotient of the operands. Of those integers, the one closer to 0 is chosen if both operands are positive. If either operand is negative, the C196 implementation also choses the one closer to 0.

The binary operator `%` computes the remainder when the first operand is divided by the second. Operands may be of any integral type. The type of the result is that of the converted operands.

It is always true that `(a/b)*b + a%b` is equal to `a` if `b` is not 0. When both operands are positive, the remainder operation will always be equivalent to the mathematical "mod" operation. If either operand is negative, the C196 implementation is defined in a manner corresponding to the integer division. The sign of the remainder will be the same as the sign of the first operand.

Examples

```
int a,b,d,r;

a = 14;
b = 4

d = a/b;     /* Division  is  3 */
r = a%b;     /* Remainder is  2 */
d = a/-b;    /* Division  is -3 */
r = a%-b;    /* Remainder is  2 */
d = -a/b;    /* Division  is -3 */
r = -a%b;    /* Remainder is -2 */
d = -a/-b;   /* Division  is  3 */
r = -a%-b;   /* Remainder is -2 */
```

10.4.8 VOLATILE OBJECTS

Access to a `volatile` object constitutes a load and a store reference when the object is one of the following:

- An operand of a prefix or postfix increment or decrement; for example, `counter++`.
- A left operand of a compound assignment operator; for example, `counter += 100`.

The compiler does not perform any optimization on objects declared as `volatile`.

10.4.9 EXTENDED ADDRESSING

The compiler treats alike all processor models that provide more than 16 address bits. This family of processors is known as 24–bit models. Code compiled for 24–bit model processors allows for the extra address bits by means of a 32–bit pointer.

When both the extend control and a 24–bit `model()` control (for example, `model(nt)`) are in effect, the language is extended by the `far` and `near` type qualifier keywords. You use `far` and `near` just like the `const` and `volatile` type qualifiers.

10.4.9.1 FAR AND NEAR DATA

Far data can be located anywhere within the entire extended address space of the processor. Near data must reside in the lower 64K, because it is accessed with 16–bit addressing. You may use a `fardata` or a `neardata` control to set the default for non–constant, non–register data. You can override the default by using the `far` or `near` type qualifiers in your declarations.

Any pointer that points to a far data object will be 32 bits wide. An example of a declaration for such a pointer is:

```
far int *near_ptr_to_far_int;
```

The pointer itself can be a far object. It can even point to a near object. Assuming the `neardata` control (the default) is in effect, here are some other ways you could declare a pointer:

```
int *far far_ptr_to_near_int;
far int *far far_ptr_to_far_int;
int *near_ptr_to_near_int;
```

10.4.9.2 FAR AND NEAR CODE

Far code can be located anywhere within the entire extended address space of the processor. Near code must reside in the upper 64K, because it is accessed with 16–bit addressing. You choose between far or near code solely by the `farcode` and `nearcode` controls. The `far` and `near` keywords may not be used to qualify functions.

All separately compiled modules must use the same choice of far or near code, since the selected mode is determined by the processor upon reset, and is not changeable by software. The processor's Configuration Control Register (CCR) controls whether the processor will use the extended addressing mode, and you must set it to match your choice of far or near code (see the `ccb` compiler control). Using only near code is more efficient than using far code.

When the `farcode` control is in effect:

- Extended call instructions are used to invoke external functions, but ordinary call instructions are used to invoke functions defined in the same module.
- All function pointers are 32 bits wide.
- Return addresses on the stack are 32 bits wide.
- Any jump tables generated for `switch` statements are built with 32–bit table entries.

10.5 COMPILER LIMITS

The values listed in Table 10–2 represent the maximum size or number of each item that the compiler can process. Exceeding any of these can produce a diagnostic message or result in incorrect execution.

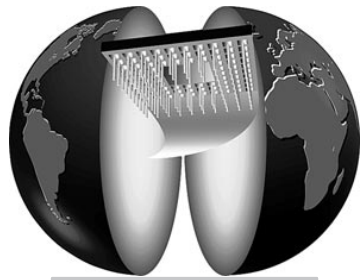
Item	Maximum
number of conditional compilation directives	16
nesting level of macro invocations	64
number of arguments in a macro invocation	31
length (in characters) of a #pragma preprocessor directive	1024
number of search–path prefixes, including prefixes for the searchinclude control and prefixes defined in the C196INC environment variable	19
number of filenames in the include control if c96init.h is present	18
number of filenames in the include control if c96init.h is not present	19
length of a pathname (in characters)	128
number of case values in a switch statement	255
nesting level of functions specified in function argument lists	20

Item	Maximum
number of functions defined in a module	255
number of external references in a module	65536
statement nesting level	32
number of arguments in a function call	31
nesting level of structures/unions	32
maximum size of structure returned from a function (in bytes)	127

Table 10–2: Compiler limits

APPENDIX A

FLEXIBLE LICENSE MANAGER (FLEXLM)



TASKING
Quality Development Tools Worldwide

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:

- 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 *server*.

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), lmreread(1)

lmgrd(1)

Name

lmgrd – flexible license manager daemon

Synopsis

lmgrd [**–c** *license_file*] [**–l** *logfile*] [**–t** *timeout*] [**–s** *interval*]

Description

lmgrd 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, **lmgrd** looks for the environment variable `LM_LICENSE_FILE` in order to find the license file to use. If that environment variable is not set, **lmgrd** 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 **lmgrd** logs the time in the log file.

lmdown(1), lmstat(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.

lmremove(1)

Name

lmremove – remove specific licenses and return them to license pool

Synopsis

lmremove [**–c** *license_file*] *feature user host* [*display*]

Description

lmremove 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.

Options

–c *license_file*

Use the specified *license_file.* If no **–c** option is specified, **lmremove** looks for the environment variable `LM_LICENSE_FILE` in order to find the license file to use. If that environment variable is not set, **lmremove** looks for the file `/usr/local/flexlm/licenses/license.dat`.

lmstat(1)

lmreread(1)

Name

lmreread – tells the license daemon to reread the license file

Synopsis

lmreread [**–c** *license_file*]

Description

lmreread 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.

lmreread 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 **lmreread** 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.

lmreread 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, **lmreread** looks for the environment variable `LM_LICENSE_FILE` in order to find the license file to use. If that environment variable is not set, **lmreread** looks for the file `/usr/local/flexlm/licenses/license.dat`.

lmdown(1)

lmstat(1)

Name

lmstat – report status on license manager daemons and feature usage

Synopsis

lmstat [**–a**] [**–A**] [**–c** *license_file*] [**–f** [*feature*]]
[**–l** [*regular_expression*]] [**–s** [*server*]] [**–S** [*daemon*]] [**–t** *timeout*]

Description

lmstat 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, **lmstat** looks for the environment variable `LM_LICENSE_FILE` in order to find the license file to use. If that environment variable is not set, **lmstat** 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, **lmstat** 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 nnn
EXITING with code nnn

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 nnn

The license daemon received fatal signal *nnn*.

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 nnn)

Vendor daemon *xxx* was restarted at internet port *nnn*.

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 nnn: 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 = nnn

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.

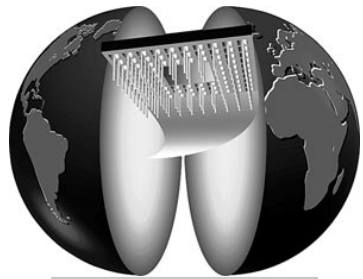
If none of the above seems to apply (i.e. **lmgrd** was already running and `LM_LICENSE_FILE` has been set correctly) then it is very likely that there is a TCP port mismatch. The fourth field of a SERVER line in the license file specifies a TCP port number. That number can be changed without affecting 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

GLOSSARY



TASKING
Quality Development Tools Worldwide

B APPENDIX

A

aggregate data type. Block of memory containing a group of values.

ANSI. American National Standards Institute.

application. The entire system designed by the user.

application program. Software for the user's application.

argument. Value or location passed to a function or macro.

asm196. 80C196 assembler.

B

branch optimization. Compiler process to combine consecutive or multiple branches into a single branch.

buffer. Contiguous block of memory treated as a simple array or character string.

byte. 8 bits.

C

calling convention. Object code inserted by the compiler to handle function calls.

cleanup. Code in the calling function that the processor executes just after control returns from the called function.

console. The user's workstation.

C–type string. Null–terminated string.

c196. 80C196 C compiler.

D

dead–code optimization. Compile process that eliminates code that can never be executed.

E

environment variable. A variable set by the user to configure the host operating system.

epilog. Code in the called function that the processor executes just before control returns to the calling function.

F

file–level variable. A variable defined outside of any function.

FPAL96. Floating–point arithmetic library.

FPL. Fixed–parameter list calling convention.

frame. A space in the stack allocated for a local variable.

frame pointer. A relocatable word register that points to the beginning of the frame.

G

gap. In memory, one or more bits located between aligned variables and containing undefined values.

global variable. Variable that exists independently of any block or function.

H

header file. Source text file containing variable declarations, function prototypes, in–line assembly language functions, and macro definitions.

I

idle mode. Power–saving processor mode in which all peripherals and the watchdog timer can continue to operate but all other features are disabled or turned off.

include file. Source text files named in an `include` compiler control or in a `#include` preprocessor directive.

in–line assembly code. Source text, embedded in a c196 program, that is assembled as ASM196 source text rather than compiled as C196 source text.

instruction set. The set of machine codes recognized by the 80C196 processor.

integral types. Types that include all forms of integers, characters, and enumerations.

L

length–prefixed string. Character string beginning with a value that indicates how many characters long it is.

lib196. 80C196 library utility.

local variable. Variable that exists only while the block or function in which it is defined is executing, and that is redefined every time the block or function is re–executed.

longword. 32 bits; 4 bytes.

M

MCS®–96. 8096 microcontroller system: 8096–90, 8096–BH, 80C196CA, 80C196CB, 80C196EA, 80C196KB, 80C196KC, 80C196KD, 80C196KR, 80C196NP, 80C196NT, 80C196NU, 80C196MC, 80C196MD and more.

multiply–aliased. Having more than one name.

N

Not–a–Number. Value in floating–point format that does not represent any real number.

null–terminated string. Character string ending with a null (`0`) value.

O

OH196. 80C196 object code to hexadecimal conversion utility.

old–style. Function declaration format that is not a prototype, that is, with the parameter data types not specified in the function declaration's parameter list.

overlaying registers. Allocating the same registers to more than one function.

P

padding. User–defined gaps.

parameter. Variable defined in a function or macro to receive an argument.

peephole optimization. Compiler process that examines generated code and attempts to combine or eliminate instruction sequences to reduce overall code size.

portable. Not dependent on the target environment.

powerdown mode. Power–saving processor mode in which RAM is preserved but all other features are disabled or turned off.

primary source file. Source text file named in the compiler invocation, outside of any control, as the file to be compiled.

program modules. Separately compiled sections of an application program.

prolog. Code in the called function that the processor executes first when control has transferred from the calling function.

PROM. Programmable read–only memory.

promoting. Casting a data type to a longer data type.

pseudo–assembly. A language similar to assembly language used to represent object code in a humanly readable format.

R

RAM. Random access memory.

reentrant. A function that calls itself or gets called again in a call loop.

register file. 80C196 on–chip memory used for high–speed data access and for hardware control; also called register memory.

registers. Bytes in the register file.

rl196. 80C196 relocation and linking utility.

ROM. Read–only memory.

run–time. During execution.

S

scalar data type. Block of memory containing a single value.

search path. The list of directories that the compiler or the host system can search to find a filename.

setup. Code in the calling function that the processor executes just before control transfers to the called function.

SFRs. Special function registers: part of the register file used for hardware control.

sign–extend. In promoting a data type, filling the bits of the unused high–order part of the longer data type with the value of the shorter data type's sign bit.

startup code. Instructions that initialize the processor.

T

target. System on which the application program executes.

U

UDI string. Length–prefixed string.

V

vector table. table containing addresses pointing to code.

VPL. variable–parameter list calling convention.

W

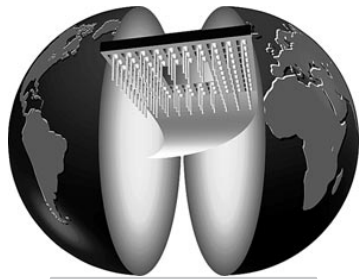
word. 16 bits; 2 bytes.

Z

zero–extend. In promoting a data type, filling the bits of the unused high–order part of the longer data type with zeroes.

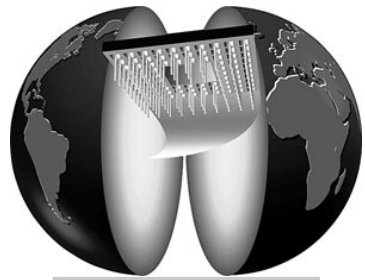
INDEX

INDEX



TASKING
Quality Development Tools Worldwide

INDEX



Symbols

Numbers

A

B

C

INDEX

D

E

F

G

H

I

J

L

M

N

O

P

R

S

INDEX

T

U

V

W

X

Z

RELEASE NOTE

INDICATOR	:	Customer Information Software
INDICATOR NR.	:	CIS9927
CONCERNS	:	TK006022–00 80C196 C Compiler Release 6.1
ISSUE DATE	:	May '99
SUPERSEDES	:	CIS9828
TO BE FILED IN	:	80C196 C Compiler User's Guide

SUMMARY

A new release of the 80C196 C Compiler 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_c.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_c.hlp`). The information is also available as HTML (`readme_c.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.