iC-96 COMPILER USER'S GUIDE FOR DOS SYSTEMS

Order Number: 481195-004

REV.	REVISION HISTORY	DATE
-001	Original Issue.	11/88
-002	Revised information on compiler controls, library functions, diagnostic messages, environment variables, and installation. Added an application development example.	04/90
-003	Added instruction pin information and vertical windowing support information.	12/90
-004	Added new compiler controls and new diagnostic messages. Added a table of 80C196KR predefined variables. Revised manual for clarity and completeness.	11/91

Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95052-8126

In the United States, additional copies of this manual or other Intel literature may be obtained by writing:

Literature Distribution Center
Intel Corporation
P.O. Box 7641
Mt. Prospect, IL 60056-7641

Or you can call the following toll-free number:

1-800-548-4725

In locations outside the United States, obtain additional copies of Intel documentation by contacting your local Intel sales office. For your convenience, international sales office addresses are printed on the last page of this document.

Intel Corporation makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update nor to keep current the information contained in this document.

Intel Corporation assumes no responsibility for the use of any circuitry other than circuity embodied in an Intel product. No other circuit patent licenses are implied.

Intel software products are copyrighted by and shall remain the property of Intel Corporation. Use, duplication or disclosure is subject to restrictions stated in Intel's Software License Agreement, or in the case of software delivered to the government, in accordance with the software license agreement as defined in FAR 52.227-7013.

No part of this document may be copied or reproduced in any form or by any means without prior written consent of Intel Corporation.

Intel Corporation retains the right to make changes to these specifications at any time, without notice.

Contact your local sales office to obtain the latest specifications before placing your order.

The following are trademarks of Intel Corporation and its affiliates and may be used only to identify Intel products. (Registered trademarks are followed by a superscripted ® .)

376	i287	Intel®	LANPrint®	PRO750
Above	i386	Intel287	LANSelect®	ProSolver
ActionMedia	i387	Intel386	LANShell®	READY-LAN
BITBUS	i486	Intel387	LANSight	Reference Point®
Code Builder	i487	Intel486	LANSpace®	RMX/80
DeskWare	i750®	Intel487	LANSpool®	SatisFAXtion®
Digital Studio	i860	intel inside	MAPNET	SnapIn 386
DVI®	i960	Intellec®	Matched	StorageBroker
EtherExpress	↑	iPSC®	MCS®	SugarCube
ETOX	i®	iRMX®	Media Mail	The Computer Inside
ExCA	ICE	iSBC®	NetPort	Token Express
FaxBACK	iLBX	iSBX	NetSentry	Visual Edge
Grand Challenge	Inboard	iWARP	OpenNET	WYPIWYF

IBM and PC AT are registered trademarks, and PC XT is a trademark of International Business Machines Corporation.

INSTALL is a trademark of Knowledge Dynamics Corporation.

Contents

Chapter 3 Compiler Controls

Chapter 4 Startup Code

Chapter 5 Processor Registers

Chapter 6 Assembly Code Instructions

Chapter 7 Libraries

Chapter 8 Messages and Error Recovery

Chapter 9 Language Implementation

Installation

Glossary

Index

Figures

Tables

Getting Started

This page of the *iC-96 Compiler User's Guide for DOS Systems* tells you where to find the information you need to use the iC-96 compiler. For more information on this manual and related publications, see Chapter 1.

Installing the Compiler on DOS

To install the iC-96 compiler, see the Installation section at the end of this manual. The installation utility on the distribution diskettes leads you through installing the compiler and the utilities on your DOS host system. To automate the compiling and linking processes, configure the environment variables listed in the Installation section, and see Chapter 2 for instructions on how to create a batch or command file.

Running the Compiler on DOS

To learn how to invoke the compiler, read Chapter 2. To learn how each control affects the compilation process, see Chapter 3. Chapter 8 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 iC-96

To learn about the MCS®-96 architecture and the iC-96 data types, calling conventions, and library functions, read Chapters 5 through 9 and the example at the end of Chapter 2.

Contents

1

Overview

Overview 1

This chapter introduces you to the DOS-hosted iC-96 compiler, to the related MCS®-96 utilities, and to this manual. 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.

1.1 iC-96 and the Software Development Process

Figure 1-1 shows a development chart using the iC-96 compiler and other tools.

With the iC-96 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 8096 assembly language or PL/M-96 code, or include in-line source assembly language in your C program. See Chapter 6 for further information on including assembly language code in a C program.

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 2. 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

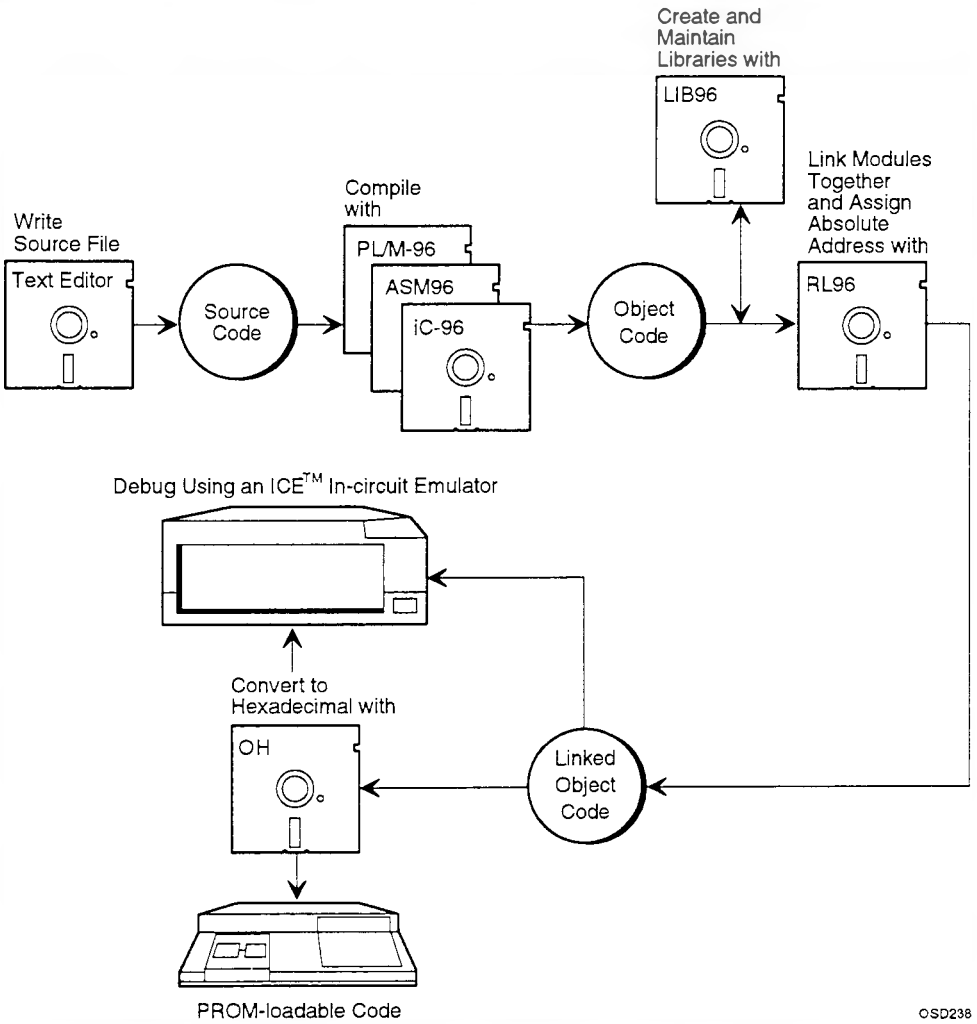

Figure 1-1 The MCS®-96 Application Development Process

messages. You must include symbolic information and line numbers to use symbolic debuggers and emulators. See Chapter 3 for a detailed description of each compiler control. Chapter 8 describes each diagnostic generated by the compiler.

Use RL96 to link object modules from iC-96, PL/M-96, and ASM96 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 iC-96 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 iC-96 package or from your own library. You can create your own libraries using the LIB96 utility.

Use LIB96 to organize frequently used MCS-96 object modules into libraries.

Use OH to convert MCS-96 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 iC-96 implementation of C conforms to the ANSI standard for the C language (x3.159 - 1989), and also supports applications that use features specific to MCS-96 architecture. See Chapter 9 for more information on language implementation.

1.2 Audience Description

To effectively use the iC-96 compiler, you must be familiar with the 8096 architecture and the software development process, as well as with the DOS operating system and the C programming language.

1.3 Related Publications

Table 1-1 lists the literature included in the iC-96 manual package.

Table 1-1 iC-96 Manual Set

Manual Name	Part Number	Contents
iC-96 Compiler User's Guide for DOS Systems	481195	Provides host-independent and DOS operation information.
MCS®-96 Utilities User's Guide	122355	Explains the MCS-96 utilities: RL96, LIB96, and OH.
8096 Floating-point Library Supplement	122365	Explains each iC-96 floating-point function.

C: A Reference Manual by Harbison and Steele, part number 480628, describes the C programming language. If you do not have this manual, return the green card included in this package for a free copy of the manual.

The ASM96 literature includes the floating-point library manual and the utilities user's guide listed above and the *ASM96 Assembler User's Guide for DOS Systems*, order number 122350.

The PL/M-96 literature includes the floating-point library manual and the utilities user's guide listed above and the *PL/M-96 User's Guide for DOS Systems*, order number 122360.

Order the following publications separately:

- *The 8096: Programming, Interfacing, Applications*, by Katz and Boyet, order number 555954
- *The 16-bit Embedded Controller Handbook*, order number 270646
- *The Embedded Control Applications Handbook*, order number 270648
- *8XC196KR User's Manual*, order number 270952
- *8XC196KR User's Guide*, order number 270799

1.4 Problem Reporting

If you need service or assistance with any of the utilities, see the inside back cover for instructions.

Intel values your input. Please suggest changes to the manual using the white reader-comment card at the back of this manual.

Please suggest changes and report problems with the utilities using the yellow customer-comment/problem-report card at the back of this manual.

1.5 Trademarks

Intel and MCS are registered trademarks and ICE is a trademark of Intel Corporation.

IBM and PC AT are registered trademarks, and PC XT is a trademark of International Business Machines Corporation.

INSTALL is a trademark of Knowledge Dynamics Corporation.

Contents

2

Compiling and Linking on DOS

Compiling and Linking on DOS

2

This chapter provides the information you need to invoke the iC-96 compiler. You can compile an iC-96 program, without making any modifications to the basic DOS environment, simply by specifying the complete compiler invocation syntax each time you use the compiler. However, setting environment variables, discussed in the Installation section, specifying compiler controls, discussed in detail in Chapter 3, and using DOS batch and command files can greatly reduce the complexity of a compiler invocation.

2.1 Compiler Invocation Syntax

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

The iC-96 compiler invocation has the following format:

```
[cpath]ic96 [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 3.

2.1.1 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 format of the output listing files.

2.1.2 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 8096-90, 8096BH, 80C196KB, 80C196KC, or the 80C196KR processor instruction set 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 = 1);
}

#pragma nolist        /* The following lines
                         are not listed. */
long f2 (long i)
{
   return(i = 0);
}
```

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 Installation section under environment variables for more information on the `c96init.h` file.

Table 2-1 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 2-1 summarizes these precedences.

Table 2-1 Compiler Controls Summary

Control	Abbr.	Description, Default, and Precedence	Effect	Usage
bmov	bm	Tells the compiler to use the bmov instruction to initialize or copy structures or array elements. Only valid with the 80C196KB processor.	Object	Primary
ccb	cc	Specifies the initial CCB value.	Object	Primary
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
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
fixedparams [(*a*)] varparams[(*a*)]	fp vp	Specifies the FPL or VPL function-calling convention. Default: varparams for non-alien functions.	Object	General

Table 2-1 Compiler Controls Summary (continued)

Control	Abbr.	Description, Default, and Precedence	Effect	Usage
include(*a*)	ic	Specifies a file to process before the primary source file.	Source	Invocation
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
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*,...)		Locates symbols to absolute addresses. Can only be specified in a #pragma directive.	Object	General
model(*a*)	md	Selects the processor instruction set. Default: model(bh).	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
optimize(*a*)	ot	Specifies the level of optimization. Default: optimization level 1.	Object	Primary

Table 2-1 Compiler Controls Summary (continued)

Control	Abbr.	Description, Default, and Precedence	Effect	Usage
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
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
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
symbols nosymbols	sb nosb	Generates or suppresses the identifier list in the print file. Default: nosymbols.	Listing content.	Primary

Table 2-1 Compiler Controls Summary (continued)

Control	Abbr.	Description, Default, and Precedence	Effect	Usage
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
windows[(*a*)] nowindows	wd nowd	Specifies that the whole application uses the vertical windows of the 80C196KC or the 80C196KR. Default: nowindows.	Object	Primary
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

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

```
C:>  C:\ic96\ic96 examples\serial.c md(kb) ot(0) db
```

The DOS operating system limits the iC-96 invocation line to 127 characters. If your screen width is less than 127 characters, the command wraps to the next screen line. If you want to continue an invocation line on another line, type the ampersand continuation character (&) at the end of the first line (the & counts as one of the 127 characters), press the Enter key and continue typing on the next screen line at the >> prompt, as shown in the following example:

```
C:> \tools\ic96 &
>>  a:\applix\source\trial.c96 &
>>  object(a:\applix\obj\trial.out) type debug
```

DOS directory and filenames can be no longer than eight characters each. If you enter a name longer than eight characters, DOS truncates the name to eight characters. You can eliminate repetitive typing of the invocation line by using DOS batch or command files. See Section 2.3 for a complete explanation of automating compiler invocations.

2.2 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 2-2.

Table 2-2 Compiler Output Files

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.

Figure 2-1 shows the input and output files of the iC-96 compiler.

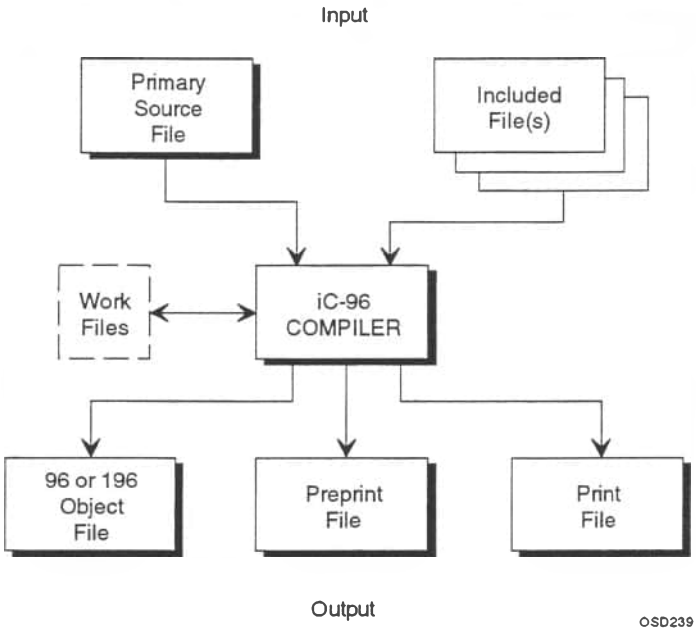

Figure 2-1 iC-96 Input and Output Files

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.

2.2.1 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 2-2. 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.

2.2.1.1 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 Chapter 1, for details on how to use the `#define` preprocessor directive.

The iC-96 compiler provides several predefined macros for your convenience. Table 2-3 shows these macros and their values.

Table 2-3 iC-96 Predefined Macros

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
ARCHITECTURE	instruction set compiler uses for object module: 90 if using the model(90) control for 8096-90 processor 'BH' if using the model(bh) control (default) for 8096-BH processor 'KB' if using the model(kb) control for 80C196KB processor 'KC' if using the model(kc) control for 80C196KC processor 'KR' if using the model(kr) control for 80C196KR processor
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
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
SIGNEDCHAR	signed or unsigned default of char variables: 1 if using the signedchar control 0 if using the nosignedchar control

2.2.1.2 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 Chapter 1, provides more information on preprocessor directives.

2.2.1.3 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 Chapter 1, for explanations on how to write a program using conditional compilation.

2.2.1.4 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 Chapter 1, for more information on preprocessor directives.

2.2.2 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 2-2. 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.

2.2.2.1 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	provide 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
```

2.2.2.2 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 2-2 shows the iC-96 compiler compiling the module `CTYPE_X` on 03/04/91 at approximately 7:04 p.m. Page numbers range from 1 to 999, then start over at 0. The example in Figure 2-2 is from the first page of the print file.

```
iC-96  Compiler   CTYPE_X                  Ø3/Ø4/91 19:Ø4:51  Page   1
```

Figure 2-2 Print File Page Header

2.2.2.3 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 2-3 shows a compilation heading produced by the iC-96 compiler running on DOS 3.3.

```
DOS 3.3Ø (Ø46-N) iC-96 Compiler Vx.y, Compilation of module CTYPE_X
Object module placed in ctype_x.OBJ
Compiler invoked by: C:\IC96\BIN\IC96.EXE ctype_x.c code xref
```

Figure 2-3 Print File Compilation Heading

2.2.2.4 Source Text Listing

The source text listing contains a formatted image of the source text, as shown in Figure 2-4. 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
    .
    .
    .
   63        1
   64        1   #endif /* _ctypeh */
    2            upcx(unsigned char input)
    3            {
    4    1         if (isxdigit(input))
    +              if (((_ctype_)[input] & Øx4Ø))
    5    1           toupper (input);
    6    1       }
```

Figure 2-4 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 2-4 shows the compiler controls that affect the source text listing portion of the print file.

Table 2-4 Controls That Affect the Source Text Listing

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.

2.2.2.5 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 2-5 shows a syntax error message. See Chapter 8 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            upcx(unsigned char input)
    3            {
    4     1        if (isxdigit(input))
    5     1          toupper (input);
    6     1        }
```

Figure 2-5 Print File Source Text Listing With Error Message

2.2.2.6 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 unique 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 2-6 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.

```
iC-96  Compiler   CTYPE_X                  03/04/91 19:31:07  Page   3
                                  Symbol Table

Name      Size  Class     Address  Attributes

_ctype_         Extern             array[] of const signed char
                                   *47, 4
input     1     Static    Ø        overlayable register unsigned char
                                  -in function(upcx)
                                   *2, 4, 5
toupper         Extern             VPL function returning int
                                   *17, 5
upcx            Public             reentrant VPL function returning int
                                   *2
```

Figure 2-6 Print File Cross-referenced Symbol Table

2.2.2.7 Pseudo-assembly Listing

The pseudo-assembly listing, shown in Figure 2-7, 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.

```
iC-96  Compiler   CTYPE_X                  03/04/91 19:31:07  Page   4
                  Assembly Listing of Object Code

                                                ;   Statement     3
          0000                            upcx:
          0000  C800                  R         push  ?OVRBASE
          0002  B3180400              R         ldb   input,4H[SP]
                                                ;   Statement     4
          0006  5D01001C              R         mulub Tmp0,input,#1H
          000A  B31D00001C            E         ldb   Tmp0,_ctype_[Tmp0]
          000F  71401C                          andb  Tmp0,#40H
          0012  981C00                          cmpb  R0,Tmp0
          0015  DF0C                            be    @0002
                                                ;   Statement     5
          0017  AC001C                R         ldbze Tmp0,input
          001A  C81C                            push  Tmp0
          001C  EF0000                E         call  toupper
          001F  65020018                        add   SP,#2H
                                                ;   Statement     6
          0023                            @0002:
          0023  CC00                  R         pop   ?OVRBASE
          0025  F0                              ret
                                                end
```

Figure 2-7 Print File Pseudo-assembly Listing

2.2.2.8 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 2-8 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                = 0026H     38D
     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            = 0006H      6D

iC-96 Compilation Complete.      0 Warnings,     0 Errors
```

Figure 2-8 Print File Compilation Summary

2.2.3 Object File

The compiler produces an object file by default, as shown in Table 2-2. 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` 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` 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 2-2. 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.

2.3 Automatically Invoking the iC-96 Compiler

DOS offers two ways to invoke a series of commands automatically: batch files and command files. This section describes how to use these files to automatically invoke the iC-96 compiler.

2.3.1 Using DOS Batch Files

A DOS batch file contains one or more commands that DOS executes in order. Some batch file commands are valid at the DOS prompt; others are valid only within a batch file. All batch files must have the `.bat` extension.

You can pass arguments to a DOS batch file. The following example is a batch file called `96cl.bat`:

```
c:\ic96\ic96 %1.c
```

To invoke `96cl.bat`, type the pathname of the batch file without its `.bat` extension followed by the name of the primary source file without its `.c` extension. For example, the following command invokes the batch file `96cl.bat` to compile the source file `progl.c`:

```
C:> 96cl progl
```

When `96cl.bat` executes, DOS replaces the `%1` parameter by `progl`, resulting in the command:

```
c:\ic96\ic96 progl.c
```

DOS batch files have several other useful features, such as `if`, `goto`, `for`, and `call` commands.

Consider the following characteristics when developing a batch file for iC-96 on DOS:

- In DOS V3.3 or later, one batch file can use the `call` command to call another batch file and control returns to the original batch file. Previous versions of DOS can use the `command /c` command. If you do not use `call` or `command /c`, the second batch file does not return.

- Batch files can contain command labels and control flow commands such as `if` and `goto`. For example, the following command allows the result of a program execution to determine which command in the current batch file to execute next:

  ```
  if errorlevel n goto label
  ```

  The value of *n* is the error code that the last program or command returned. If the error code is the same or greater than the value of *n*, control transfers to the line immediately after *label*. The *label* is any alphanumeric string significant up to eight characters. Put a label on its own line and prepend a colon, as follows:

  ```
  :label
  ```

- Although a batch file can contain multiple DOS commands, each command must fit on a single line (127 characters). You cannot use continuation lines in batch files. To process a longer line, specify a command to redirect input from a file containing the remainder of the line. The redirected file can contain continuation lines.

The following example shows how to redirect additional input from another file, how to use parameters, and how to call another batch file in DOS 3.3 or later. Figure 2-9 shows the relationships between the `96cl.bat` batch file, the `96cl.ltx` file of filenames, and the `2prom.bat` batch file.

The 96cl.bat DOS batch file compiles, links, and converts to hexadecimal the program in a source text file specified by the %1 parameter:

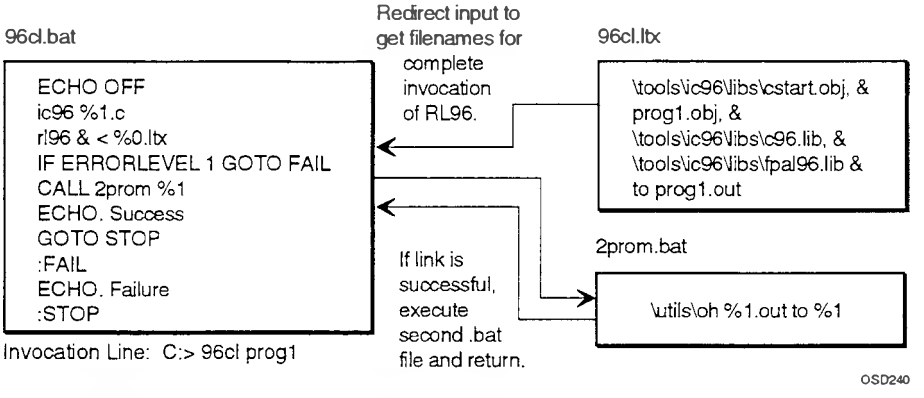

Figure 2-9 Redirecting Input to a Batch File

The DOS batch file `%0` parameter always represents the name of the batch file itself (without the `.bat` extension). In this example, `96cl.bat` and `96cl.ltx` have identical names except for the extension, so `96cl.bat` can refer to `96cl.ltx` using the `%0.ltx` parameter.

To execute the `96cl.bat` batch file and pass `prog1` as an argument, type the following at the DOS command prompt:

```
C:> 96cl prog1
```

When `96cl.bat` executes, iC-96 compiles `prog1.c` then RL96 links the resulting `prog1.obj` object module to another object module and to the file specified in `96cl.ltx`. If the linking is successful, the `2prom.bat` file produces a PROM-loadable module named `prog1`.

2.3.2 Using DOS Command Files

You can invoke the DOS command processor, `command.com`, with input redirected from a file called a command file. A DOS command file contains a sequence of DOS commands and must contain the `exit` command as its final line. Always use the Enter key following the `exit` command, not an `<eof>` character.

For example, the `96linkc.cmd` command file contains the following:

```
ic96 prog0.c
ic96 prog1.c
rl96 cstart.obj, /f link.lst
exit
```

The file `link.lst` is a directive file called by the `/f` linker control. See the *MCS®-96 Utilities User's Guide*, listed in Chapter 1, for more information on the `/f` control. The `link.lst` file contains the following:

```
progxs.obj, prog0.obj, &
prog1.obj, c:\tools\ic96\c96.lib &
to prog0.out
```

To sequentially execute the commands in the command file, redirect `96linkc.cmd` to `command.com` by typing the following at the DOS prompt:

```
C:> command < 96linkc.cmd
```

Consider the following characteristics when developing a command file for iC-96:

- This method of redirecting commands works in a command file containing a fixed sequence of commands only. You cannot pass arguments to `command.com`.
- The flow of control is always sequential, from top to bottom of the command file. Command files must not contain conditional commands.
- You can nest command files. To invoke a second command file, insert a line in the first example such as:

  ```
  command < comfile2.cmd
  ```

  The secondary command file must contain the `exit` command as its final line followed by the Enter key. If it does not, control does not return to the primary command file until you enter `exit` at the DOS prompt. Control returns to the point in the primary file immediately following the point from which the secondary file was invoked.

If you invoke a command file with output redirected to another file, the command-line interpreter records all commands from the first line of the command file through the command `exit` and all console input and output to the file. For example, the following command invokes the `96linkc.cmd` command file and creates a log file named `96linkc.log` containing a record of all commands:

```
C:> command < 96linkc.cmd > 96linkc.log
```

2.4 Developing An iC-96 Application Program

The iC-96 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 RL96 linker and locator utility to combine separately translated object modules into a single program and assign absolute addresses to all relocatable addresses. Use the LIB96 utility to place an object module into a library for later combination, using RL96, into a program. Use the OH utility to convert an object module into the standard hexadecimal format that can be loaded into PROM.

Your iC-96 application programs can contain many separately translated modules. The applications can call functions from a library. The iC-96 product includes several libraries. You can also create your own libraries using the LIB96 utility.

To create a complete program, the RL96 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.

Figure 2-10 shows how to specify libraries and object files in the correct order for linking with iC-96 compiled modules.

The following is an example RL96 invocation:

```
C:> rl96 cstart.obj, newmod.obj, my_utils.lib, &
>> printf.obj, scanf.obj, c96.lib, fpal96.lib &
>> to newmod.out
```

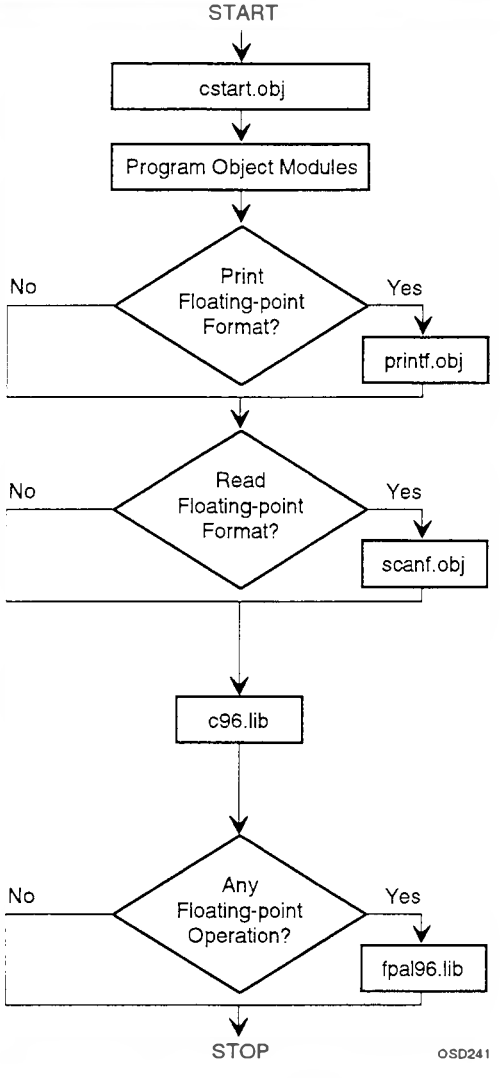

Figure 2-10 Choosing Libraries and Object Files for Linking

2.5 Examples

This section contains an example iC-96 program showing how to use compiler controls and DOS 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.

2.5.1 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 `80C196.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 2-11 through 2-13 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 `c:\project\ic96\working`.

NOTE

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). The book is listed in the *Intel Literature Guide* (available from any Intel sales office) and in Chapter 1 of this manual, and can be purchased through Intel Literature Sales, P.O. Box 58130, Santa Clara, CA 95052-8130.

```
/*************************************************************************/
/*                                                                       */
/*      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 <80C196.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) 0x71         /* 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 2-11 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 2-12 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 2-13 Digital Filter Source Text (Main Routine) in dsf.c

2.5.2 Setting the DOS Environment

The directory structure of this example is as follows:

- The `c:\ic96\bin` directory contains the `ic96.exe` compiler.
- The `c:\ic96\include` directory contains the `80C196.h` include file.
- The `c:\project\ic96\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 2-12 and 2-13.
  - 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 2-11.

- 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:

```
C:> set c96inc=c:\ic96\include;c:\project\ic96\working
C:> set c96init=c:\project\ic96\working
```

Setting the `c96inc` 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.

2.5.3 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:> ic96 dsf.c preprint notranslate > dsf_pre.log
```

NOTE

`#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:

```
DOS 3.30 (046-N) iC-96 Compiler Vx.y

Copyright 1980, 89, 90, 91 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\ic96\working\dsfinit.h:
illegal constant expression
```

Figure 2-14 shows the resulting `dsf.i` preprint file. Compiling this file has the same result as compiling the `dsf.c` file.

```
#line 1 "c:\project\ic96\working\c96init.h"

#line 1 "c:\project\ic96\working\dsfinit.h"
/*****************************************************************************/
/*                                                                           */
/*      This is the initialization code for the digital signal filter.       */
/*                                                                           */
/*****************************************************************************/

#pragma model(kb)   /* Select instruction set for the 80C196-KB processor.  */
#pragma interrupt (software_timer = 5)
#pragma interrupt (analog_conversion_done = 1)


#line 1 "c:\ic96\include\80C196.h"
/*****************************************************************************/
/*                                                                           */
/* 80C196.h - definition of symbolic named registers for the I/O registers   */
/*            of the 80C196 (a superset of the 8096 registers)               */
/*                                                                           */
/*****************************************************************************/


extern volatile register unsigned short r0;         /* at 0x00: r   zero */
extern volatile register unsigned char ad_command;  /* at 0x02:   w */
extern volatile register unsigned char ad_result_lo;/* at 0x02: r   */
```

Figure 2-14 Digital Signal Filter Preprint File

```
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 wd timer*/
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 */

/*****************************************************************************/
/*                                                                           */
/* Additional I/O registers of the 80C196                                    */
/*                                                                           */
/*****************************************************************************/

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   */

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

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

Figure 2-14 Digital Signal Filter Preprint File (continued)

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

#line 10 "c:\project\ic96\working\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 of word_value */

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

#line 2 "c:\project\ic96\working\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 microseconds. */
}
/*****************************************************************************/
```

Figure 2-14 Digital Signal Filter Preprint File (continued)

```
/*****************************************************************************/
/*                                                                           */
/* 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 & (unsigned char) 0x71)  && (ad_result_hi != (unsigned
char) 0x71))
    input++;
  if (ioport0 & (unsigned char) 0x71)  /* 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.          */
/*                                                                           */
/*****************************************************************************/

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 2-14 Digital Signal Filter Preprint File (continued)

2.5.4 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 2-15, invoke the compiler as follows:

```
C:> ic96 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.

```
iC-96  Compiler   DSF                          Ø3/Ø5/9Ø 13:33:35  Page  12
                                  Symbol Table

  Name              Size    Class    Address    Attributes

  ad_command        1       Extern               register volatile unsigned
                                                -char
                                                 *12, 18
  ad_result_hi      1       Extern               register volatile unsigned
                                                -char
                                                 *14, 33, 34
  ad_result_lo      1       Extern               register volatile unsigned
                                                -char
                                                 *13, 34
  analog_conversion_done
                            Public               interrupt function returni
                                                -ng void
                                                 *2Ø
  enable                    Extern               VPL function returning voi
                                                -d
                                                 *5Ø, 62
```

Figure 2-15 Digital Signal Filter Symbol Table

```
                                Symbol Table

hso_command      1       Extern              register volatile unsigned
                                            - char
                                             *18, 19, 59
hso_time         2       Extern              register volatile unsigned
                                            - short
                                             *16, 20, 60
input            1       Public   2          register unsigned char
                                             *24, 33, 35, 37, 40
int_mask         1       Extern              register volatile unsigned
                                            - char
                                             *21, 56
int_pending      1       Extern              register volatile unsigned
                                            - char
                                             *22, 58
ioport0          1       Extern              register volatile unsigned
                                            - char
                                             *27, 36
ioport1          1       Extern              register volatile unsigned
                                            - char
                                             *28, 37, 40, 41, 55
main                     Public              reentrant VPL function ret
                                            -urning int
                                             *53
software_timer           Public              interrupt function returni
                                            -ng void
                                             *20
timer1           2       Extern              register volatile unsigned
                                            - short
                                             *24, 20, 60
word_value       2       Public   0          register unsigned int
                                             *25, 40, 41
```

Figure 2-15 Digital Signal Filter Symbol Table (continued)

2.5.5 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 ASM96 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 Intel's in-circuit emulators.

A useful feature of symbolic debuggers such as Intel's ICE™-196 in-circuit emulator 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 2-16 contains the code listing generated by the following compiler invocation:

```
C:> ic96 dsf.c debug code optimize(0)
```

```
iC-96  Compiler   DSF                          03/05/90 14:03:40  Page   6
                  Assembly Listing of Object Code

                                               ;   Statement    17
          0000                              software_timer:
          0000  F4                                pusha
          0001  C81C                              push  Tmp0
                                               ;   Statement    18
          0003  B10800              E             ldb   ad_command,#8H
                                               ;   Statement    19
          0006  B11800              E             ldb   hso_command,#18H
                                               ;   Statement    20
          0009  454000001C          E             add   Tmp0,timer1,#40H
          000E  A01C00              E             ld    hso_time,Tmp0
                                               ;   Statement    21
          0011  CC1C                              pop   Tmp0
          0013  F5                                popa
          0014  F0                                ret
                                               ;   Statement    32
          0015                              analog_conversion_done:
          0015  F4                                pusha
          0016  C81C                              push  Tmp0
          0018  C81E                              push  Tmp2
                                               ;   Statement    33
```

Figure 2-16 Digital Signal Filter Code Listing (Level 0 Optimization)

```
                Assembly Listing of Object Code

          001A  B00002            E         ldb   input,ad_result_hi
                                            ;   Statement    34
          001D  5171001C          E         andb  Tmp0,ad_result_lo,#71H
          0021  981C00                      cmpb  R0,Tmp0
          0024  DF0A                        be    @0003
          0026  B1711C                      ldb   Tmp0,#71H
          0029  98001C            E         cmpb  Tmp0,ad_result_hi
          002C  DF02                        be    @0003
                                            ;   Statement    35
          002E  1702              R         incb  input
                                            ;   Statement    36
          0030                    @0003:
          0030  5171001C          E         andb  Tmp0,ioport0,#71H
          0034  981C00                      cmpb  R0,Tmp0
          0037  DF05                        Be    @0004
                                            ;   Statement    37
          0039  B00200            E         ldb   ioport1,input
                                            ;   Statement    38
          003C  2011                        br    @0005
          003E                    @0004:
                                            ;   Statement    40
          003E  5D71021C          R         mulub Tmp0,input,#71H
          0042  5D71001E          E         mulub Tmp2,ioport1,#71H
          0046  641E1C                      add   Tmp0,Tmp2
          0049  A01C00            R         ld    word_value,Tmp0
                                            ;   Statement    41
          004C  B00100            E         ldb ioport1,word_value+1H
                                            ;   Statement    42
          004F                    @0005:
                                            ;   Statement    43
          004F  CC1E                        pop   Tmp2
          0051  CC1C                        pop   Tmp0
          0053  F5                          popa
          0054  F0                          ret
                                            ;   Statement    54
          0055                    main:
                                            ;   Statement    55
          0055  1100              E         Clrb  ioport1
```

Figure 2-16 Digital Signal Filter Code Listing (Level 0 Optimization) (continued)

```
iC-96  Compiler   DSF                    03/05/90 14:03:40  Page   8
               Assembly Listing of Object Code

                                                ;   Statement    56
          0057  B12200              E          ldb   int_mask,#22H
                                                ;   Statement    58
          005A  1100                E          clrb  int_pending
                                                ;   Statement    59
          005C  B11800              E          ldb   hso_command,#18H
                                                ;   Statement    60
          005F  454000001C          E          add   Tmp0,timer1,#40H
          0064  A01C00              E          ld    hso_time,Tmp0
                                                ;   Statement    62
          0067  EF0000              E          call  enable
                                                ;   Statement    63
          006A                        @0008:
          006A  27FE                           br    @0008
          006C                        @0007:
                                                ;   Statement    64
          006C  F0                             ret
                                               end
```

Figure 2-16 Digital Signal Filter Code Listing (Level 0 Optimization) (continued)

2.5.6 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.

Contents

3

Compiler Controls

Compiler Controls 3

This chapter describes the iC-96 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 parenthesis 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 Notational Conventions, listed on the inside front cover, for special meanings of type styles used by this manual.

bmov

Primary control: tells the compiler to use the bmov instruction to initialize or copy structures or array elements
80C196KB processor only

Syntax

```
bmov
```

Abbreviation

```
bm
```

Discussion

Use this control to tell the compiler to use the `bmov` instruction when initializing or copying structures or array elements. This control is valid only for the 80C196KB processor. Use the `model(kb)` control to specify the 80C196KB instruction set.

The compiler generally does not generate the `bmov` instruction because of its interrupt latency. The `bmov` instruction is uninterruptable. See the *ASM96 Assembler's User's Guide for DOS Systems*, listed in Chapter 1, for more information on the `bmov` instruction.

If you specify the `model(kc)` or the `model(kr)` control, meaning you are using the 80C196KC or the 80C196KR processor respectively, the compiler automatically generates the `bmovi` instruction for the same process. The `bmovi` instruction is interruptable. Do not specify the `bmov` control in this case.

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.

Cross reference

```
model(kb)
```

Syntax

```
ccb(value)
```

Where:

value is a byte value.

Abbreviation

```
cc
```

Discussion

Use this control to initialize the value of the chip configuration byte (CCB), located at 2018H. The MCS®-96 processor reads the CCB on reset to initialize the value of the chip configuration register (CCR). See the *16-Bit Embedded Controller Handbook*, listed in Chapter 1, for a detailed explanation of the contents of the CCR.

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.

code

General control: generates or suppresses pseudo-assembly language code listing in print file
default: nocode

Syntax

```
code | nocode
```

Abbreviation

```
co | noco
```

Discussion

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.

Cross-references

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

cond

General control: includes or suppresses uncompiled conditional code in source text listing
default: nocond

Syntax

```
cond | nocond
```

Abbreviation

```
cd | nocd
```

Discussion

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.

Cross-references

```
list                    print
listinclude             translate
```

debug

Primary control: includes or suppresses debug information in the object module
default: nodebug

Syntax

```
debug | nodebug
```

Abbreviation

```
db | nodb
```

Discussion

Use this control to place information used by symbolic debuggers, such as an ICE™ in-circuit emulator, 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 RL96) and symbolic debugging. RL96 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.

debug (continued)

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.

The predefined macro `_DEBUG_` indicates which of `type`, `notype`, `debug`, or `nodebug` have been specified, as in Table 3-1:

Table 3-1 Values for the _DEBUG_ Macro

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

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.

Cross-references

`object`
`optimize`
`translate`
`type`

Syntax

```
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
```

Discussion

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 Chapter 1, 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:

   ```
   ic96 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:

```
ic96 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

Primary control: specifies level of diagnostic messages
default: diagnostic(1)

Syntax

```
diagnostic(level)
```

Where:

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

Abbreviation

```
dn
```

Discussion

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:

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

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

`diagnostic(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.

Cross-references

Messages (see Chapter 8)
`print`

eject

General control: inserts a form-feed into the print file

Syntax

```
eject
```

Abbreviation

```
ej
```

Discussion

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.

Cross-references

```
list
print
```

extend

Primary control: recognizes or suppresses Intel iC-96 extensions
default: extend

Syntax

```
extend | noextend
```

Abbreviation

```
ex | noex
```

Discussion

Use this control to direct the compiler to accept file-scope `register` variables and the `alien`, `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 iC-96.

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 iC-96 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 iC-96. 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 iC-96 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, iC-96 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.

Cross-references

```
alien
fixedparams
regconserve
varparams
```

fixedparams

General control: specifies
fixed-parameter list calling convention
default: varparams

Syntax

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

Where:

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

Abbreviation

```
fp
```

Discussion

Use this control to require functions to use the fixed-parameter list (FPL) calling convention. The PL/M-96 compiler generates object code for function calls using the FPL calling convention. The variable-parameter list (VPL) calling convention is the default used by the iC-96 compiler. When calling a PL/M-96 function from an iC-96 program, specify `fixedparams` for the PL/M-96 function in the iC-96 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 compiler pushes the arguments onto the stack with the leftmost argument pushed first before transferring control to the called function.
2. If the called function is reentrant, the compiler saves any of the calling function's registers that can be modified by the called function after transferring control to the called function.

3. If the called function is reentrant, the compiler restores any of the calling function's registers that were saved before returning control to the calling function.
4. The compiler then removes the arguments from the stack before returning 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.

NOTE

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.

fixedparams (continued)

```
#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)
   ```

Cross-references

`code`
`extend`
`object`
`translate`
`varparams`

Syntax

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

Where:

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

Abbreviation

```
ic
```

Discussion

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 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 file, use the `searchinclude` control. To view names of include files and the order of their inclusion without compilation, use the `preprint` and `notranslate` controls.

include (continued)

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.

Cross-references

`listinclude`
`preprint`
`searchinclude`

inst

Primary control: specifies whether the compiler generates vector tables for switch statements
default: noinst

Syntax

```
inst | noinst
```

Abbreviation

```
is | nois
```

Discussion

Use this control to prevent the iC-96 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 *MCS®-96 Utilities User's Guide*, listed in Chapter 1, for more information on overlapping ROM and RAM memory.

interrupt

General control: specifies a function to be an interrupt handler

Syntax

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

Where:

function	is the name of a function defined in the source text.
n	is the interrupt number.

Abbreviation

```
in
```

Discussion

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=3)
```

However, you cannot specify multiple interrupt function handlers to one interrupt. The following example generates a warning, or a fatal error if the control is placed in the invocation line:

```
#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 8096 and 80196 microcontrollers when generating the call and return sequences. The exact sequence generated depends on the argument to the `model` control.

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 and identify the interrupts.

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`. Table 3-2 lists the interrupts corresponding to the valid 8096 and 80C196 interrupt numbers. Table 3-3 lists the interrupts specific to the 80C196.

Table 3-2 8096 and 80C196 Interrupt Numbers

Interrupt Type	Interrupt Number
timer overflow	0
A/D conversion complete	1
high speed input (HSI) data available	2
high speed outputs (HSO)	3
HSI.0 Pin	4
software timer	5
serial port	6
EXTINT	7

Table 3-3 80C196-only Interrupt Numbers

Interrupt Type	Interrupt Number
trap	8
undefined opcode	9
transmit	24
receive	25
HSI FIFO 4th entry	26
timer2 capture	27
timer2 overflow	28
EXTINT pin	29
HSI FIFO full	30
non-maskable	31

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

Example

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

```
interrupt (int_handle_1=1, int_handle_24=24)
```

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

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

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

Cross-reference

`model`

list

General control: specifies or suppresses source text listing in print file
default: list

Syntax

```
list | nolist
```

Abbreviation

```
li | noli
```

Discussion

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.

Cross-references

cond	pagelength	tabwidth
listexpand	pagewidth	title
listinclude	print	translate

listexpand

General control: includes or suppresses
macro expansion in listing
default: nolistexpand

Syntax

```
listexpand | nolistexpand
```

Abbreviation

```
le | nole
```

Discussion

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.

listexpand (continued)

Cross-references

```
cond                    print
list                    translate
listinclude
```

listinclude

General control: includes or suppresses
text from include files in listing
default: nolistinclude

Syntax

```
listinclude | nolistinclude
```

Abbreviation

```
lc | nolc
```

Discussion

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.

listinclude (continued)

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.

Cross-references

`include`
`list`
`print`
`translate`

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
```

Discussion

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=0xF02)
```

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 RL96 at link time. The following example generates an error:

```
main()
{   int i2;

#pragma locate(i2=0xF02);     /* This line generates an
                                 error. */
}
```

model

Primary control: specifies the instruction set
default: model(bh)

Syntax

```
model(processor)
```

Where:

processor Selects the instruction set the compiler uses in generating code for the 8096-90, 8096-BH, 80C196KB, 80C196KC, or the 80C196KR processor.

Abbreviation

```
md
```

Discussion

Use this control to select the instruction set for the processor.

Specify the *processor* as one of the following:

`90` to select an instruction set that executes on all MCS-96 processors.

`196` to select the 80C196KB instruction set. 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 an instruction set that executes on the 8096BH processor but does not allow for the 8096-90 processor functional differences or additional 80C196 functionality.

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

`kc` to select the 80C196KC instruction set. You must specify this model type if you want to use the vertical windowing feature of the 80C196KC. Failure to do so results in a fatal error.

`kr` to select the 80C196KR instruction set. Similar to the `kc` argument, you must specify this model type if you intend to use the vertical windowing feature of the 80C196KR. Failure to do so results in a fatal error.

Code compiled for the 8096 processor is upward compatible and can execute on the 80C196 processor. However, code compiled for the 80C196 processor cannot execute on the 8096 processor because the 80C196 processor has several additional instructions and special function registers (SFRs). Code compiled for the 8096BH processor might not execute correctly on the 8096-90 processor because of differences in processor operation. Table 3-4 shows how code compiled for one MCS-96 processor can or cannot execute on a different processor:

Table 3-4 Compatibility Between Processors

Compiled for	Executes on 8096BH	Executes on 8096-90	Executes on 80C196KB	Executes on 80C196KC/KR
8096BH	yes	no	yes	yes
8096-90	yes	yes	yes	yes
80C196KB	no	no	yes	yes
80C196KC	no	no	no	yes
80C196KR	no	no	no	yes

The predefined macro `_ARCHITECTURE_` has the value `90`, `'bh'`, `'kb'`, `'kc'`, or `'kr'` 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 `#pragma` preprocessor directives 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.

Cross-references

```
interrupt
registers
```

object

Primary control: generates and names or suppresses object file
default: object

Syntax

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

Where:

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

Abbreviation

```
oj | nooj
```

Discussion

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.

CAUTION

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.

Cross-references

`code`
`translate`

optimize

Primary control: specifies the level of optimization
default: optimize(1)

Syntax

```
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
```

Discussion

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 3-1 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.

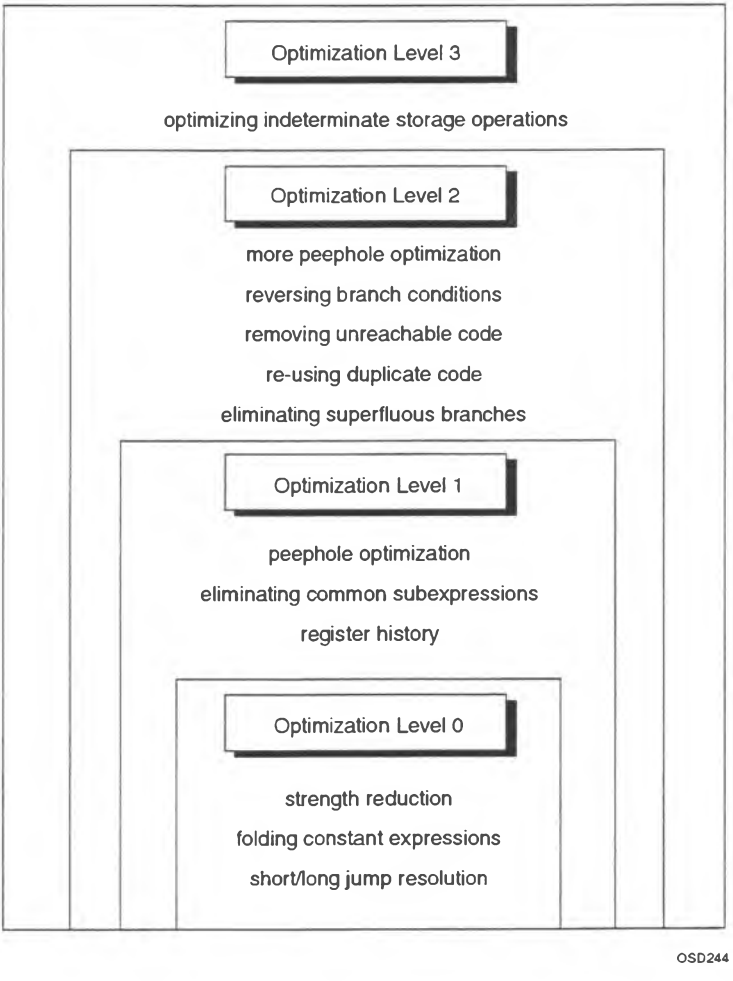

Figure 3-1 Summary of Optimization Levels

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`, and operations on two or more constants fall into this category. For example, the expression `a+2+3` becomes `a+5`.

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. The addition of identical subexpressions may also generate left shift instructions.

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 3-2, 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 3-2 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.

Cross-reference

`volatile`

pagelength

Primary control: specifies lines per page in the print file
default: pagelength(60)

Syntax

```
pagelength(lines)
```

Where:

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

Abbreviation

```
pl
```

Discussion

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.

Cross-references

```
pagewidth          title
print              translate
tabwidth
```

pagewidth

Primary control: specifies line width
in the print file
default: pagewidth(120)

Syntax

```
pagewidth(chars)
```

Where:

chars is the line length in number of characters. This value can range from 72 to 132.

Abbreviation

```
pw
```

Discussion

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.

Cross-references

```
pagelength          title
print               translate
tabwidth
```

preprint

Invocation control: generates or suppresses
a preprocessed source text listing file
default: nopreprint

Syntax

```
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
```

Discussion

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 differs from 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`.

preprint (continued)

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.

Cross-reference

`translate`

print

Primary control: generates or suppresses the print file
default: print

Syntax

```
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
```

Discussion

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.

print (continued)

Cross-references

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

Syntax

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

Where:

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

vector is an interrupt vector number.

Abbreviation

```
pt
```

Discussion

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 `80c196.h` header file contains 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 <80c196.h>

STran_ptscb single;

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

pts (continued)

```
#pragma interrupt(timer1=0)       /* Interrupt vector zero. */

int count;
const char src[]   = "This is a pts test.";
      char dst[20] = "This should not be.";

main()
{
unsigned char save_wsr;

init_putchar();
count = 0;
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. */
iocl = 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 ++;
}
```

Cross-references

```
interrupt
locate
```

Syntax

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

Where:

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

Discussion

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 iC-96. 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 iC-96. 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 effect 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.

Cross-references

`extend`
`registers`

regconserve

Primary control: disallows file-scope and automatic non-register variables in registers.
default: noregconserve

Syntax

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

Where:

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

Abbreviation

```
rc | norc
```

Discussion

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 3-5 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.

Table 3-5 Allocation of Non-register Variables to Registers

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

NOTE

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.

Cross-reference

`registers`

registers

Primary control: allocates register space for variables
default: registers(220)

Syntax

```
registers(num)
```

Where:

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

Abbreviation

```
rg
```

Discussion

Use this control to limit the number of bytes of register memory the module can use. MCS-96 microcontrollers have 256 bytes of register space except for the 80C196KC that has an additional 256 bytes of registers, and the 80C196KR that has 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 iC-96 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 5.4.3 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.

NOTE

The overlay segment the compiler generates for the module is word-aligned. 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.

Cross-references

```
reentrant
regconserve
windows
```

searchinclude

General control: specifies or suppresses search paths for include files
default: nosearchinclude

Syntax

```
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 DOS expects in a path prefix.

Abbreviation

```
si | nosi
```

Discussion

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 `c96inc` DOS environment variable, if defined.
4. 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 `c96inc` DOS environment variable, if defined.
3. The source directory.
4. 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 `c96inc` environment variable is defined and the `searchinclude` control is specified.

The `c96inc` environment variable is defined at the DOS prompt as follows:

```
C:> set c96inc=\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`. The compiler is invoked in the root (`\`) directory. The compiler searches for filenames in the following order:

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 `c96inc`: `\proj001\t_locate.h`
5. From `c96inc`: `\proj001\headers\t_locate.h`
6. The current directory: `\t_locate.h`

Cross-reference

`include`

signedchar

Primary control: sign-extends or zero-extends promoted chars
default: signedchar

Syntax

```
signedchar | nosignedchar
```

Abbreviation

```
sc | nosc
```

Discussion

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.

signedchar (continued)

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.

Cross-references

`object`
`translate`

symbols

Primary control: generates or suppresses identifier list in print file
default: nosymbols

Syntax

```
symbols | nosymbols
```

Abbreviation

```
sb | nosb
```

Discussion

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.

Cross-references

```
print
translate
xref
```

tabwidth

Primary control: specifies the number
of characters per tab stop
in the print file
default: tabwidth(4)

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
```

Discussion

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.

Cross-references

```
pagelength          title
pagewidth           translate
print
```

title

Primary control: specifies the print file title
default: title("primary_source_filename")

Syntax

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

Where:

string is the print file title.

Abbreviation

```
tt
```

Discussion

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.

Cross-references

```
pagelength          tabwidth
pagewidth           translate
print
```

Syntax

```
tmpreg(addr)
```

Where:

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

Abbreviation

```
tr
```

Discussion

Use this control to locate the temporary registers, namely PLMREG, at a different address. See Chapter 5 for more information on PLMREG and see Chapter 9 for more information on the ?FRAMEØ1 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 Ø*x*2C in hexadecimal format. The address you specify must be on a double-word boundary.

If you specify this control, PLMREG appears as PLMR*xx* and ?FRAMEØ1 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, PLMREG appears as PLMR2C and ?FRAMEØ1 appears as ?FRAME2C.

To correctly use this feature, you must link to your application, using RL96, a module containing a declaration that reserves eight bytes of memory space at the address specified by *addr*. You can create this module with ASM96 or iC-96. The compiler uses these eight bytes as the new temporary registers. Name the variable PLMR*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 PLMR2C. 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 memory space to be used as the new temporary registers, so that no other module attempts to use these eight bytes. The following examples show how to declare this variable in assembly language and C. These examples also explain how to assemble, compile, and link the module to your application.

1. For ASM96, create a 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 PLMR2C
rseg at 2CH
PLMR2C  equ  $
dsl 2

rseg
?FRAME2C equ $
dsw 1
end
```

   Assemble this module. See the *ASM96 Assembler User's Guide for DOS Systems*, for the ASM96 assembly invocation syntax. Compile your iC-96 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 ASM96 object module with your iC-96 object modules, as follows:

```
> rl96 cprg1.obj, cstart.obj, cprg2.obj, tmpreg.obj, &
  c96.lib
```

2. For iC-96, create a module called `tmpreg.c` with the following declaration:

```
register long PLMR2C[2];          /* Temp registers. */
#pragma locate(PLMR2C=Øx2c);      /* Locate to 2CH.  */
register int ?FRAME2C;
```

The `locate` pragma directive locates the `PLMR2C` variable to address `2CH`. This module also allocates a 16-bit register for the frame pointer, `?FRAME2C`. Compile this module with the `notype` control. This control suppresses any type information to be generated for this module. Compile the rest of your iC-96 programs with the `tmpreg(2CH)` control. The `tmpreg` control informs the compiler that the temporary registers are located at `2CH`. During the link phase, link the `tmpreg.obj` file to the rest of your iC-96 object modules, for example,

```
> rl96 cprg1.obj, cstart.obj, cprg2.obj, tmpreg.obj, &
  c96.lib
```

Cross-references

?FRAMEØ1
locate control
PLMREG

translate

Invocation control: compiles or suppresses compilation after preprocessing
default: translate

Syntax

```
translate | notranslate
```

Abbreviation

```
tl | notl
```

Discussion

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.

Cross-references

```
object
preprint
```

type

Primary control: generates or suppresses type information in the object module
default: type

Syntax

```
type | notype
```

Abbreviation

```
ty | noty
```

Discussion

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 symbol 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.

type (continued)

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.

Cross-references

`debug`
`object`
`symbols`
`translate`
`xref`

varparams

General control: specifies variable-parameter list calling convention
default: varparams

Syntax

```
varparams[(function [,...])]
```

Where:

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

Abbreviation

```
vp
```

Discussion

Use this control to require 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 compiler pushes the arguments onto the stack with the rightmost argument pushed first before transfering control to the called function.
2. If the called function is reentrant, the compiler saves any of the calling function's registers that can be modified by the called function after transfering control to the called function.

3. If the called function is reentrant, the compiler restores any of the calling function's registers that were saved before returning control to the calling function.
4. The compiler then removes the arguments from the stack after returning control to the calling function.

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.

NOTE

The PL/M-96 compiler generates object code for function calls using the fixed parameter list (FPL) calling convention. 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)
   ```

Cross-references

`code`
`extend`
`fixedparams`
`object`
`translate`

windows

Primary control: specifies that the whole application uses the vertical windows of the 80C196KC or the 80C196KR
default: nowindows

Syntax

```
windows[( [no]hold )] | nowindows
```

Abbreviation

```
wd | nowd
```

Discussion

Use this control to use the additional registers of the 80C196KC and the 80C196KR through the vertical windowing feature of both 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 register variables located in the mapped area of the register file and above (from `80H` or `0C0H` or `0E0H` depending on the window size). Global 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 5.3.4 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]+
inc  SP
```

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 5.4.2 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 the `model(kc)` or `model(kr)` control, otherwise the compiler generates a fatal error.

Cross-references

`model`
`reentrant`
`regconserve`
`registers`

xref

Primary control: specifies symbol table cross-reference in listing
default: noxref

Syntax

```
xref | noxref
```

Abbreviation

```
xr | noxr
```

Discussion

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.

Cross-references

```
print
symbols
translate
```

Contents

4

Startup Code

4 Startup Code

This chapter describes the startup code (`cstart.obj`) supplied with your iC-96 compiler.

4.1 Contents of cstart.obj

The `cstart.p96` file is a PL/M-96 program containing a call to a procedure called `main`. This line of code calls the `main()` function in your iC-96 source text. The startup program is not written in assembly language because `main` is a reserved word in ASM96. The `cstart.p96` program contains the following lines:

```
cstart: do;

main: procedure external;
end;

   call main;       /* Call the main() function */

end cstart;
```

This source text contains no safety procedure to handle any accidental return to the `cstart` module. In such cases, the processor executes whatever is in memory after the call-to-main instruction, which often contains uninitialized memory. You must ensure that `main()` never returns to this module.

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 iC-96 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 `cstart` routine.

The `cstart` routine then loads the stack pointer (SP) and makes a long call to `main()`. The following code is produced:

```
          ljmp cstart
cstart:   ld sp, #stack
          lcall main
```

The startup source file contains the minimum required statements to execute your application. You can tailor it according to your specific needs and the environment under which your application executes. For example, you can add a call to the `init$real$math$unit` floating-point initialization procedure if you are using floating-point functions. To call the initialization procedure, add the following line before the call to `main`:

```
call init$real$math$unit;  /* Calls the FPAL96          */
                           /* initialization procedure; */
                           /* you need to do this before */
                           /* using any floating-point   */
                           /* operations.                */
```

4.2 Writing Your Own Startup Code

You can write your own startup code using the ASM96 assembly language. You must declare your module to be the main module by using the `main` directive. Load the stack pointer with the address of the stack. Initialize any other registers you need, then do a long call (`lcall`) to your main C function. You cannot call your main C function as `main()`. You must use another name, such as `_main()`. Your ASM96 main module must contain at least the following lines:

```
cstart  module  main

sp equ 18H:word

cseg at 2080H
extrn _main

ld sp,#stack
lcall _main
rst             ; resets the processor if program returns
                ; to cstart
end
```

Assemble the file and then link it with your C object files.

Contents

5

Processor Registers

5 Processor Registers

The MCS®-96 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 `8Ø96.h`, `8ØC196.h`, `kr.h` and `auto_kr.h` header files for using the SFRs and explains how to use the iC-96 compiler for efficient register allocation.

5.1 Register Memory

Figure 5-1 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 and the 80C196KR microcontrollers. The 80C196KC and 80C196KR have 256 bytes of additional registers from `1ØØH` through `1FFH`. The iC-96 compiler tries to allocate variables to the register memory as much as possible, if `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 the `c96.lib` library. For an explanation of the structure and use of the SFRs, see the *ASM96 Assembler User's Guide for DOS Systems* or the *16-bit Embedded Controller Handbook*, listed in Chapter 1.
- 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`, are used for intermediate calculations and for returning the value of a typed function. The compiler treats this section of memory as the `PLMREG` register variable. You can use the `tmpreg` control to change the location of the temporary registers. See Chapter 3 for more information about the `tmpreg` control.

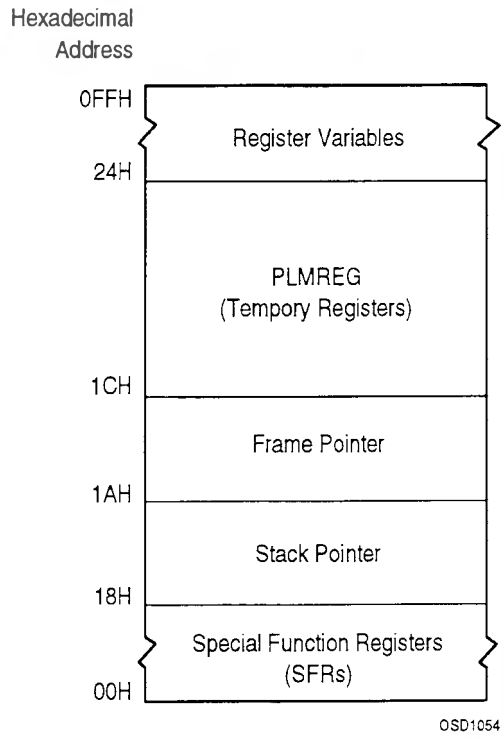

Figure 5-1 80C196KB Register Memory

The MCS-96 processors contain on-chip peripherals, listed in Table 5-1, controlled by the special function registers (SFRs) located in the first 24 bytes of the register file. The iC-96 header files and libraries define symbols, macros, and functions to read and write the SFRs.

Table 5-1 Major I/O Functions

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.
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.

5.2 Accessing Special Function Registers

The `8Ø96.h`, `8ØC196.h`, `kr.h`, and `auto_kr.h` header files declare variables that you can use to access the SFRs. Table 5-2 lists the variables useful for both the 8096 and 80C196 processor SFRs. Table 5-3 lists the variables useful for the additional 80C196 processor SFRs. Table 5-4 lists the variables useful for the additional 80C196KR SFRs.

Table 5-2 8096 SFR Predefined Variables

Name	Data Type	Read/Write	Operation
ad_command	unsigned char	write only	Controls the A/D converter.
ad_result_hi	unsigned short	read only	Contains the high-order result of the A/D converter.
ad_result_lo	unsigned short	read only	Contains the low-order result of the A/D converter.
baud_rate	unsigned char	write only	Determines the baud rate.
hsi_mode	unsigned char	write only	Sets the mode of the high-speed input (HSI) unit.
hsi_status	unsigned char	read only	Indicates the state of the HSI pins detected at the time in hsi_time.
hsi_time	unsigned short	read only	Contains the time at which the high speed input (HSI) unit was triggered.
hso_command	unsigned char	write only	Determines the action taken at the time loaded into hso_time.
hso_time	unsigned short	write only	Sets the time or count for the high-speed output (HSO) unit to execute the command in the command register.
int_mask	unsigned char	read and write	Enables and disables individual interrupts.
int_pending	unsigned char	read and write	Indicates that an interrupt signal has occurred and has not been serviced.
ioc0	unsigned char	write only	I/O control register 0; controls the alternate functions of the HSI pins, Timer 2 reset sources, and Timer 2 clock sources.
ioc1	unsigned char	write only	I/O control register 1; controls the alternate functions of the Port 2 pins, timer interrupts, and HSI interrupts.
ioport0	unsigned char	read only	Reports levels on pins of port 0.
ioport1	unsigned char	read and write	Reads from or writes to port 1.
ioport2	unsigned char	read and write	Reads from or writes to port 2.

Table 5-2 8096 SFR Predefined Variables (continued)

Name	Data Type	Read/Write	Operation
ios0	unsigned char	read only	I/O status register 0; contains information on the status of the HSO.
ios1	unsigned char	read only	I/O status register 1; contains information on the status of the timers and HSI.
pwm_control	unsigned char	write only	Pulse width modulation (PWM) control register; sets the duration of the PWM pulse.
r0	unsigned short	read and write	Zero register; always contains 0.
sbuf	unsigned char	read and write	Operates as the transmit and receive buffer for the serial port.
sp_con	unsigned char	write only	Sets the mode of the serial port.
sp_stat	unsigned char	read only	Indicates the status of the serial port.
timer1	unsigned short	read only	Contains the current value of Timer 1.
timer2	unsigned short	read only	Contains the current value of Timer 2.
watchdog	unsigned char	write only	When written to, prevents automatic reset every 64K clock cycles.

Table 5-3 80C196 Additional Register Predefined Variables

Name	Data Type	Read/Write	Operation
ad_time	unsigned char	read and write	A/D time; determines A/D conversion time (80C196KC and 80C196KR only).
imask1	unsigned char	read and write	Interrupt mask register for the 8 80C196-specific interrupts.
ioc2	unsigned char	write only	I/O control register 2; controls the 80C196-specific features.
ios2	unsigned char	read only	I/O status register 2; contains information on the HSO events.
ipend1	unsigned char	read and write	Interrupt pending register for the 8 80C196-specific interrupts.
ptssel	unsigned short	read and write	PTS select register; individually enables PTS channels (80C196KC and 80C196KR only).
ptssrv	unsigned short	read and write	PTS serve register; holds the interrupt pending flags for end-of-PTS requests (80C196KC and 80C196KR only).
pwm1_control	unsigned char	read and write	PWM1 control register; sets the duration of the PWM1 pulse (80C196KC only).
pwm2_control	unsigned char	read and write	PWM2 control register; sets the duration of the PWM2 pulse (80C196KC only).
t2cap	unsigned short	read	Timer2 capture register; holds the value of Timer2 on the rising edge of P2.7 (80C196KB and 80C196KC only).
t2control	unsigned char	read and write	New feature control register; enables the new features of the 80C196KC (e.g., Timer2 internal clocking, PWM1 and PWM2)
wsr	unsigned char	read and write	Window select register (WSR); selects register window 0 or 15. Registers that are read-only when WSR=0 are write-only when WSR=15, and vice versa. 8096 register operation is available to the 80C196 when WSR=0. Register operation changes when WSR=15 or WSR=0FH.

Table 5-4 80C196KR Additional Predefined Variables

Name*	Data Type	Read/Write	Operation
ad_result	unsigned short	read only	Contains the result of the A/D converter.
ad_test	unsigned char	read and write	Enables conversions on ANGND and VREF and specifies adjustments for D.C. offset errors.
comp0_con comp_control0	unsigned short	read and write	Controls the operation of compare channel 0.
comp0_time comp_time0	unsigned short	read and write	Contains the time an event is to occur in compare channel 0.
comp1_con comp_control1	unsigned short	read and write	Controls the operation of compare channel 1.
comp1_time comp_time1	unsigned short	read and write	Contains the time and event is to occur in compare channel 1.
epa_mask	unsigned short	read and write	Contains the interrupt mask bits for all EPA channel overruns and EPA channels 4 through 9.
epa_mask1	unsigned short	read and write	Contains the interrupt mask bits for timer overruns and compare modules.
epa_pend	unsigned short	read and write	Indicates all EPA channel overruns and EPA channel 4 through 9 events.
epa_pend1	unsigned short	read and write	Indicates Timer overruns and both EPA compare module events.
epa0_con epa_control0	unsigned short	read and write	Controls the operation of EPA channel 0.

* The symbol names are first listed as they appear in the kr.h header file. The auto_kr.h-header-file version of the symbol name appears below that symbol, if the auto_kr.h version differs from the kr.h version.

Table 5-4 80C196KR Additional Predefined Variables (continued)

Name*	Data Type	Read/Write	Operation
epa0_time epa_time0	unsigned short	read and write	When EPA channel 0 is in capture mode, this register is loaded with the time an event occurred. When the channel is in compare mode, this register contains the time an event is to occur.
epa1_con epa_control1	unsigned short	read and write	Controls the operation of EPA channel 1.
epa1_time epa_time1	unsigned short	read and write	When EPA channel 1 is in capture mode, this register is loaded with the time an event occurred. When the channel is in compare mode, this register contains the time an event is to occur.
epa2_con epa_control2	unsigned short	read and write	Controls the operation of EPA channel 2.
epa2_time epa_time2	unsigned short	read and write	When EPA channel 2 is in capture mode, this register is loaded with the time an event occurred. When the channel is in compare mode, this register contains the time an event is to occur.
epa3_con epa_control3	unsigned short	read and write	Controls the operation of EPA channel 3.
epa3_time epa_time3	unsigned short	read and write	When EPA channel 3 is in capture mode, this register is loaded with the time an event occurred. When the channel is in compare mode, this register contains the time an event is to occur.
epa4_con epa_control4	unsigned short	read and write	Controls the operation of EPA channel 4.

* The symbol names are first listed as they appear in the kr.h header file. The auto_kr.h-header-file version of the symbol name appears below that symbol, if the auto_kr.h version differs from the kr.h version.

Table 5-4 80C196KR Additional Predefined Variable (continued)

Name*	Data Type	Read/Write	Operation
epa4_time epa_time4	unsigned short	read and write	When EPA channel 4 is in capture mode, this register is loaded with the time an event occurred. When the channel is in compare mode, this register contains the time an event is to occur.
epa5_con epa_control5	unsigned short	read and write	Controls the operation of EPA channel 5.
epa5_time epa_time5	unsigned short	read and write	When EPA channel 5 is in capture mode, this register is loaded with the time an event occurred. When the channel is in compare mode, this register contains the time an event is to occur.
epa6_con epa_control6	unsigned short	read and write	Controls the operation of EPA channel 6.
epa6_time epa_time6	unsigned short	read and write	When EPA channel 6 is in capture mode, this register is loaded with the time an event occurred. When the channel is in compare mode, this register contains the time an event is to occur.
epa7_con epa_control7	unsigned short	read and write	Controls the operation of EPA channel 7.
epa7_time epa_time7	unsigned short	read and write	When EPA channel 7 is in capture mode, this register is loaded with the time an event occurred. When the channel is in compare mode, this register contains the time an event is to occur.
epa8_con epa_control8	unsigned short	read and write	Controls the operation of EPA channel 8.

* The symbol names are first listed as they appear in the kr.h header file. The auto_kr.h-header-file version of the symbol name appears below that symbol, if the auto_kr.h version differs from the kr.h version.

Table 5-4 80C196KR Additional Predefined Variables (continued)

Name*	Data Type	Read/Write	Operation
epa8_time epa_time8	unsigned short	read and write	When the EPA channel 8 is in capture mode, this register is loaded with the time an event occurred. When the channel is in compare mode, this register contains the time an event is to occur.
epa9_con epa_control9	unsigned short	read and write	Controls the operation of EPA channel 9.
epa9_time epa_time9	unsigned short	read and write	When the EPA channel 9 is in capture mode, this register is loaded with the time an event occurred. When the channel is in compare mode, this register contains the time an event is to occur.
epaipv	unsigned char	read and write	Contains the encoded value for the highest priority pending EPA interrupt.
p0_pin p0pin	unsigned char	read only	Returns the current state of the Port 0 pins.
p1_dir p1io	unsigned char	read and write	Determines whether the Port is used for input (or open-drain output) or for output.
p1_mode p1ssel	unsigned char	read and write	Controls the mode of operation for I/O Port 1.
p1_pin p1pin	unsigned char	read only	Returns the current state of the Port 1 pins.
p1_reg p1reg	unsigned char	write only	Writes data out to I/O port 1 pins.
p2_dir p2io	unsigned char	read and write	Determines whether Port 2 is used for input (or open-drain output) or for output.

* The symbol names are first listed as they appear in the kr.h header file. The auto_kr.h-header-file version of the symbol name appears below that symbol, if the auto_kr.h version differs from the kr.h version.

Table 5-4 80C196KR Additional Predefined Variables (continued)

Name*	Data Type	Read/Write	Operation
p2_mode p2ssel	unsigned char	read and write	Controls the mode of operation for I/O Port 2.
p2_pin p2pin	unsigned char	read only	Returns the current state of the Port 2 pins.
p2_reg p2reg	unsigned char	read and write	Writes data out to I/O port 2 pins.
p3_pin p3pin	unsigned char	read only	Returns the current state of the Port 3 pins.
p3_reg p3reg	unsigned char	read and write	Writes data out to I/O port 3 pins.
p4_pin p4pin	unsigned char	read only	Returns the current state of the Port 4 pins.
p4_reg p4reg	unsigned char	read and write	Writes data out to I/O port 4 pins.
p5_dir p5io	unsigned char	read and write	Determines whether Port 5 is used for input (or open-drain output) or for output.
p5_mode p5ssel	unsigned char	read and write	Controls the mode of operation for I/O Port 5.
p5_pin p5pin	unsigned char	read only	Returns the current state of the Port 5 pins.
p5_reg p5reg	unsigned char	read and write	Writes data out to I/O port 5 pins.
p6_dir p6io	unsigned char	read and write	Determines whether Port 6 is used for input (or open-drain output) or for output.
p6_mode p6ssel	unsigned char	read and write	Controls the mode of operation for I/O Port 6.

* The symbol names are first listed as they appear in the kr.h header file. The auto_kr.h-header-file version of the symbol name appears below that symbol, if the auto_kr.h version differs from the kr.h version.

Table 5-4 80C196KR Additional Predefined Variables (continued)

Name*	Data Type	Read/Write	Operation
p6_pin p6pin	unsigned char	read only	Returns the current state of the Port 6 pins.
p6_reg p6reg	unsigned char	read and write	Writes data out to I/O port 6 pins.
sbuf_rx	unsigned char	read only	Operates as receive buffer for the serial port.
sbuf_tx	unsigned char	write only	Operates as transmit buffer for the serial port.
slp_cmd slpcmd	unsigned char	read only	Written to by the master processor when the 80C196KR processor is in slave mode.
slp_con slpfunreg	unsigned char	read and write	Controls the SLPINT pin operation, SLP_ADDR source, and Slave Port enable and disable.
slp_stat slpstat	unsigned char	read only	Contains the Slave Port status bits.
sp_baud	unsigned short	read and write	Determines the serial port baud rate.
sp_status	unsigned char	read only	Contains the six status bits indicating the current status of the serial input and output buffers, sbuf_tx and sbuf_rx.
ssio_baud	unsigned char	read and write	Determines the baud rate of the synchronous serial I/O.
ssio0_buf ssio_stb0	unsigned char	read and write	In transmit mode, writing a byte to this register causes the start of a transmission. In receiving mode, you can read incoming data from this register.
ssio0_con ssio_stcr0	unsigned char	read and write	Determines how ssio0_buf operates.

* The symbol names are first listed as they appear in the kr.h header file. The auto_kr.h-header-file version of the symbol name appears below that symbol, if the auto_kr.h version differs from the kr.h version.

Table 5-4 80C196KR Additional Predefined Variables (continued)

Name*	Data Type	Read/Write	Operation
ssio1_buf ssio_stb1	unsigned char	read and write	In transmit mode, writing a byte to this register causes the start of a transmission. In receiving mode, you can read incoming data from this register.
ssio1_con ssio_stcr1	unsigned char	read and write	Determines how ssio1_buf operates.
t1control timer1_control	unsigned char	read and write	Controls the operation of TIMER1.
usfr	unsigned char	read and write	UPROM special function register; specifies which bits are available for the ROM and EPROM versions of the 80C196KR processor.
zero_reg	unsigned short	read and write	Always contains 0.

* The symbol names are first listed as they appear in the kr.h header file. The auto_kr.h-header-file version of the symbol name appears below that symbol, if the auto_kr.h version differs from the kr.h version.

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` 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 *16-bit Embedded Controller Handbook*, listed in Chapter 1, describes the processor idle and powerdown modes.

5.2.1 Name Collision with the 80C196KR Registers

Some of the SFRs of the 80C196KR processor have the same names as the SFRs in the 80C196KC processor, but are located at different addresses. The RL96 linker does not allow multiple `public` symbols within a library. Therefore, the duplicate names have been changed by appending `_kr` or `_okr` to these symbols in the `kr.h` and `auto_kr.h` header file, respectively. Continue to access these SFRs without the appended characters (`_kr` or `_okr`) in your C program. The `kr.h` and `auto_kr.h` header files contain `#define` statements that redefine these registers to the 80C196KR version of the symbols. The only time you see these symbols with the appended characters is when you debug your program using an emulator.

5.3 PLMREG

The `PLMREG` 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 `PLMREG` to the address `1CH` in the register segment, by default, and gives `PLMREG` a `null` attribute. You can change the location of PLMREG by using the `tmpreg` control, as described in Chapter 3. The `null` attribute allows any function to use `PLMREG` without having to specify a data type, as described in the *ASM96 Assembler User's Guide for DOS Systems*, listed in Chapter 1.

PLMREG is declared in assembly language as follows:

```
public PLMREG
rseg   at     1CH
PLMREG EQU    $
dsl    2
end
```

5.4 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. A `register` variable can be any data type and is read or written using 8-bit addressing instead of 16-bit addressing.

5.4.1 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 3 for the description of each control.

5.4.2 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 iC-96 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 iC-96 and PL/M-96 compilers overlay registers within each module. The RL96 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 iC-96 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. See the `overlay` control in Chapter 3 for more information on overlaying registers. 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.

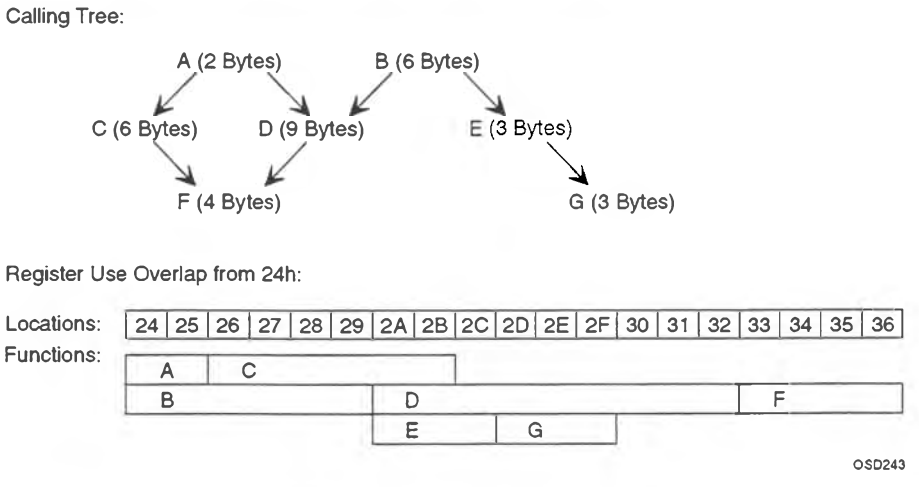

Figure 5-2 Calculating Register Memory Requirements

You can calculate the number of bytes of register memory needed by a module as in the following example, illustrated in Figure 5-2:

- 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 `B` 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 5-2 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 5-2 occupies the same number of bytes (19) but in different locations. The registers used by any given module are not necessarily contiguous.

5.4.3 Support for Vertical Windows

The 80C196KC and the 80C196KR processors have 256 bytes of additional registers from 100H through 1FFH. 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 *16-Bit Embedded Controller Handbook*, listed in Chapter 1, for more information on register windowing.

The 80C196KC and 80C196KR processors provide 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 into 32-, 64-, or 128-byte windows onto the top 32-, 64-, 128-byte portion of the register file. Use the Window Select Register (WSR) to switch between windows.

The iC-96 compiler uses the additional registers for the block-scope register variables allocated in overlay segments. Block-scope variables are variables declared within non-reentrant functions. Figure 5-3 shows the register allocation scheme that the linker uses to locate register and overlay segments on the 80C196KC processor.

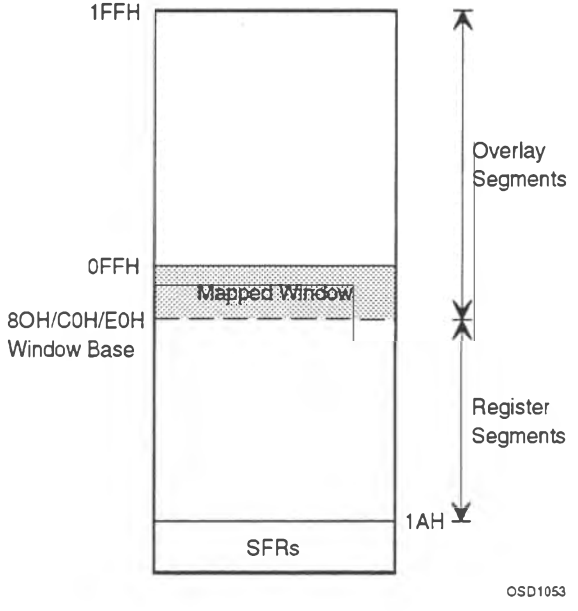

Figure 5-3 80C196KC Register Allocation Scheme

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 changes to the WSR. Then, the linker locates the overlay segments after all register segments are

located. If gaps are 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.

To read or write to the local register variables, the iC-96 compiler generates the WSR management code in the prolog and epilog of all public functions, except for functions that do not contain local register 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 `windows` 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]+
inc   SP
```

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.

When linking modules together, specify the range of the registers available to the application with the RL96 `registers` control and the desired window size through the RL96 `windowsize` control. See the *MCS®-96 Utilities User's Guide*, listed in Chapter 1, 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 RL96 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 *MCS®-96 Utililities User's Guide*, listed in Chapter 1, 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 ASM96 modules together with iC-96 modules and you want the overlay segment from your ASM96 module to use the vertical windowing done in C, declare `?wsr` as an external byte variable in your ASM96 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 ASM96 module. This example assumes that the ASM96 module is called by `main()`.

Your iC-96 module contains the following line:

```
void func(void);

main()
{
     func();
}
```

Your ASM96 module must contain the following lines:

```
example module
$include(80c196.inc)

rseg
     extrn ?wsr:byte

oseg
     var1: dsw 1

cseg
     public func
func:
     push wsr
     andb wsr, #80h
     orb wsr,#?wsr
     .
     .
     .
     ld var1, #10
     .
     .
     .
     ldb wsr,[sp]+
     inc sp
     ret
end
```

If you have a specific use for vertical windows and do not want the iC-96 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.

Contents

6

Assembly Code Instructions

Assembly Code Instructions

6

This chapter describes ways to include assembly language instructions inside your iC-96 program without requiring a separately written and translated assembly language routine.

6.1 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_list;
       .
       .
       .
    }
```

Where:

`asm`	is the keyword indicating an assembly instruction follows.
`pseudo_asm_inst`	is the 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.

6.2 Pseudo-assembly Instruction Interpretation

The pseudo-assembly instruction statement follows the same format as any regular ASM96 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 *ASM96 Assembler User's Guide for DOS Systems*, listed in Chapter 1, 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` and `dcw` code definition directives are supported. See Section 6.3 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` and `dcw`, such as `dcl` and `dcr`.
  - 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 6.4 for a complete list of supported assembly instructions and the *ASM96 Assembler User's Guide for DOS Systems*, listed in Chapter 1, 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 80C196KB or 80C196KC instruction set.
- You cannot use numeric expressions. Expressions must consist of simple numbers.
- For `dcb` and `dcw` 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. */
```

6.3 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 separate from any in-line assembly instructions. See example 2 in Section 6.6 for an application of this process.

6.4 Assembly Instructions

The following 8096 assembly instructions are supported by the iC-96 compiler:

Arithmetic, Logical, and Memory Transfer Instructions:

```
add             ld              push
addb            ldb             st
addc            ldbse           stb
addcb           ldbze           sub
and             mul             subb
andb            mulb            subc
cmp             mulu            subcb
cmpb            mulub           xch
div             or              xchb
divb            orb             xor
divu            pop             xorb
divub
```

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
```

NOTE

The iC-96 compiler supports a pseudo-instruction, `call` *`register`*, which implements the indirect call via a `push` and `br[indirect]` sequence.

Zero-operand Instructions:

```
clrc    epts    pushf
clrvt   nop     ret
di      popa    rst
dpts    popf    setc
ei      pusha
```

Extended Instructions:

```
bmov    cmpl    tijmp
bmovi   idlpd
```

6.5 Unsupported Instructions

The following 8096 instructions are not supported:

Non-generic Branch Instructions:

```
djnz            jh              jnvt
djnzw           jle             jst
jbc             jlt             jv
jbs             jnc             jvt
jc              jne             lcall
je              jnh             ljmp
jge             jnst            scall
jgt             jnv             sjmp
```

Module-level Directives:

```
end             module          public
extrn
```

Location Counter Control Directives:

```
cseg            org             rseg
dseg            oseg
```

Symbol Definition Directives:

```
equ             set
```

Code Definition Directives:

```
dcl             dcr
```

Storage Reservation Directives:

```
dsb             dsr
dsl             dsw
```

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 *ASM96 Assembler User's Guide for DOS Systems* listed in Chapter 1.

6.6 Examples

1. The following example shows how you can add in-line assembly code to iC-96 source text:

```
#include <stdio.h>

register int ri;

main()
{
    int i;

    init_putchar();
    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;
}
```

Contents

7

Libraries

7 Libraries

The iC-96 libraries provide some of the ANSI standard C library functionality and some additional functionality specific to MCS®-96 architecture. This chapter describes iC-96 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 Chapter 1.

7.1 Library Files

You can link your application with any of the iC-96 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 iC-96 compiler includes two run-time library files. They are:

`c96.lib`	Defines all standard functions and Intel extensions.
`fpal96.lib`	Defines all floating-point operations.

The compiler also includes two object files. They are:

`printf.obj`	Supports floating-point format printf, sprintf.
`scanf.obj`	Supports floating-point format scanf, sscanf.

7.1.2 Linking Library Files

The RL96 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. To do so, specify the libraries and object files for linking in the following order:

1. startup code
2. program modules
3. user-defined libraries, if any
4. the floating-point versions of `printf` and `scanf`, which are `printf.obj` and `scanf.obj`, if you are performing floating-point formatted input and output
5. the floating-point library, `fpal96.lib`, if you are using floating-point functions
6. the C library, `c96.lib`

7.2 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 iC-96 source text to declare the library function prototypes and macros. The function declarations in the iC-96 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 7-1 lists the names and functionality of the iC-96 library header files and the manuals which describe each header file.

Table 7-1 Header Files

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
8096.h	8096 processor special facilities	this chapter
80c196.h	80C196 processor special facilities	this chapter
kr.h	80C196KR processor special facilities as they are described in the *8XC196KR User's Manual.*	this chapter
auto_kr.h	80C196KR processor special facilities as they are described in the *8XC196KR User's Guide.*	this chapter

8096.h
80c196.h
kr.h
auto_kr.h

Processor-specific facilities
non-ANSI

Discussion

The `8096.h`, `80c196.h`, `kr.h`, and `auto_kr.h` header files define variables to access the Special Function Registers (SFRs) and declare functions to manipulate the processor hardware. The `8096.h` header file defines the SFR symbols for the 8096-90 and 8096-BH processors. The `80c196.h` header file builds from the `8096.h` header file and defines the additional SFR symbols of the 80C196KB and 80C196KC processors. The `kr.h` and `auto_kr.h` header files define all of the SFR symbols plus the additional SFRs of the 80C196KR processor. The `kr.h` header file defines the SFR symbols of the 80C196KR based on the register names found in the *8XC196KR User's Manual*, listed in Chapter 1. The `auto_kr.h` header file defines the SFR symbols of the 80C196KR based on the register names found in the *8XC196KR User's Guide*, listed in Chapter 1.

The functions declared in the `8096.h` header file are:

`enable`	enables interrupts.
`disable`	disables interrupts.

The functions declared in the `80C196.h`, `kr.h`, and `auto_kr.h` header files, in addition to `enable` and `disable`, are:

`enable_pts`	enables PTS interrupts.
`disable_pts`	disables PTS interrupts.
`power_down`	puts the processor into powerdown mode.
`idle`	puts the processor into idle mode.

Discussion

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 Chapter 1.

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 7-1.

```
iC-96  Compiler   CYTPE_X                  10/28/91  11:29:19  Page  1
DOS 5.0 (046-N) iC-96 Compiler Vx.y, Compilation of module CTYPE_X
Object module placed in CTYPE_X.obj
Compiler invoked by: c:\ic96\bin\IC96.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 7-1 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 7-2:

```
iC-96  Compiler   CTYPE_XU                  10/28/91 12:41:49  Page  1

DOS 5.0 (046-N) iC-96 Compiler Vx.y, Compilation of module CTYPE_XU
Object module placed in CTYPE_XU.obj
Compiler invoked by: d:\ic96\bin\IC96.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 7-2 Example Using the Function Prototype

string.h

Character array manipulation
ANSI

Discussion

The ANSI contents of `string.h` are described in *C: A Reference Manual*, listed in Chapter 1. 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

7.3 Functions

This section provides descriptions of the iC-96 library functions that differ from or are not covered in *C: A Reference Manual*.

Each entry in this section is organized as follows:

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.

atof

Converts a string to floating-point

Prototype Declaration

```
double atof (const char *str_ptr);
```

Where:

str_ptr points to the string to be converted.

Header File

```
stdlib.h
```

Description

Use this function to convert a string of ASCII characters to a floating-point value. *C: A Reference Manual*, listed in Chapter 1, describes the `atof` function.

Before using the `atof` function, you must call the `fpinit` function to initialize floating-point capability. You must also specify the `fpal96.lib` library when you link your program to provide floating-point support.

Returns

The `atof` function returns the converted floating-point value.

Cross-reference

`fpinit`

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
```

Discussion

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

Disables the processor's interrupts

Prototype Declaration

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

Header File

```
8096.h, 80c196.h, kr.h, auto_kr.h
```

Discussion

Use this function to disable the processor's interrupts.

Returns

The `disable` function does not return a value.

Prototype Declaration

```
void disable_pts (void);
```

Header File

```
80c196.h, kr.h, auto_kr.h
```

Discussion

Use this function to disable the peripheral transaction server's (PTS) interrupts. This function is valid only for the 80C196KC processor.

Returns

The `disable_pts` function does not return a value.

enable

Enable the processor's interrupts

Prototype Declaration

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

Header File

```
8096.h, 80c196.h, kr.h, auto_kr.h
```

Discussion

Use this function to enable the processor's interrupts.

Returns

The `enable` function does not return a value.

Prototype Declaration

```
void enable_pts (void);
```

Header File

```
80c196.h, kr.h, auto_kr.h
```

Discussion

Use this function to enable the peripheral transaction server's (PTS) interrupts. This function is valid only for the 80C196KC processor.

Returns

The `enable_pts` function does not return a value.

fabs

Converts floating-point numbers to positive

Prototype Declaration

```
double fabs (double);
```

Header File

```
math.h
```

Discussion

Use this function to convert the sign of a floating-point number to positive. You can use this function as an alternative to using the `fpabs` floating-point function.

Returns

The `fabs` function returns a positive floating-point number.

Prototype Declaration

```
void fpinit (void);
```

Header File

```
fpal96.h
```

Discussion

Use this function to perform the following necessary initializations for the functions in the `fpal96.lib` library:

- 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 *8096 Floating-point Library Supplement*, listed in Chapter 1, for more information on floating-point numbers and initialization.

Returns

The `fpinit` function does not return a value.

idle

Enters a power-saving mode

Prototype Declaration

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

Header File

```
80c196.h, kr.h, auto_kr.h
```

Discussion

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 *16-Bit Embedded Controller Handbook*, listed in Chapter 1, 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.

Prototype Declaration

```
void init_putchar (void);
```

Header File

```
stdio.h
```

Discussion

Use this function to set the TI bit in the static variable that holds status of the serial port. Setting the TI bit initializes the mechanism used by the `putchar` function. You must call the `init_putchar` function once after reset or exit from the powerdown mode to ensure correct operation of subsequent calls to `putchar`. The `putchar` 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_putchar` before calling `putchar`, the `putchar` function can wait indefinitely for the TI bit to be set.

Returns

The `init_putchar` function does not return a value.

power_down

Enters a power-saving mode

Prototype Declaration

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

Header File

```
80C196.h, kr.h, auto_kr.h
```

Discussion

Use this function to place the 80C196 processors in powerdown mode. The `power_down` function is available only on the 80C196 processor. See the *16-Bit Embedded Controller Handbook*, listed in Chapter 1, 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.

Prototype Declaration

```
int printf (const char *format_ptr,...);
int sprintf (char *buf_ptr, const 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
```

Discussion

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 Chapter 1. The `printf` and `sprintf` functions do not support the `g` and `G` floating-point conversion operations.

Before using `printf` for the first time, you must call the `init_putchar` function once after a reset or exit from the powerdown mode to ensure correct operation of subsequent calls to `putchar`. The `printf` function calls the `putchar` function. The `init_putchar` 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 `putchar` function. The `putchar` 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_putchar` before calling `putchar`, the `putchar` function can wait indefinitely for the TI bit to be set.

The sp_stat and sbuf variables are defined in the 8096.h and 80C196.h header files. If you redefine putchar 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 the printf.obj module and the fpal96.lib library 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.

Prototype Declaration

```
int scanf (const char *format_ptr,...);
int sscanf (char *buf_ptr, const 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
```

Discussion

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 Chapter 1. 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 `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 the `scanf.obj` module and the `fpal96.lib` library when you link your program.

Returns

The `scanf` and `sscanf` functions return the number of successfully read input values.

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
```

Discussion

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 Chapter 1, 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.

Contents

8

Messages and Error Recovery

Messages and Error Recovery

8

The iC-96 compiler can issue the following types of messages:

- Sign-on and sign-off messages (discussed in Section 8.1)
- Fatal errors (discussed in Section 8.2)
- Errors (discussed in Section 8.3)
- Warnings (discussed in Section 8.4)
- Remarks (discussed in Section 8.5)

All messages, except fatal error messages, are reported in the print file. Fatal error messages appear on the scree; 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.

8.1 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:

```
system-id iC-96 Compiler Vx.y
Copyright years Intel Corporation
```

Where:

system-id	identifies the host operating system.
Vx.y	identifies the version of the compiler.
years	identifies the copyright years.

On normal completion, the compiler displays a message similar to the following:

```
iC-96 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:

```
iC-96 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.

8.2 Fatal Error Messages

Fatal error messages have the following syntax:

```
iC-96 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 Intel 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.

`function call nesting limit exceeded`

The nesting of function calls within an expression exceeded 32.

`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 not one of `90`, `bh`, `196`, `kb`, `kc`, or `kr`.

`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 `cases` 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.

8.3 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`. 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. The iC-96 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 */
};
```

`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`.

`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.

`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 6.3 for a list of supported assembler instructions. Also see the *ASM96 Assembler User's Guide for DOS Systems* 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 *ASM96 Assembler User's Guide for DOS Systems* 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 *ASM96 Assembler User's Guide for DOS Systems* 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 *ASM96 Assembler User's Guide for DOS Systems*, listed in Chapter 1, 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.

`nesting too deep`

One of the nesting limits described in Chapter 8 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.

8.4 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.

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`.

`filename too long; truncated`

The filename length exceeded the limit of the operating system.

`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.

`illegal character: hex_value (hex)`

The character with the ASCII value *hex_value* is not part of the iC-96 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 not one of `90`, `bh`, `196`, `kb`, `kc`, or `kr`.

`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.

8.5 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.

`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.

`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.

Contents

9

Language Implementation

9 Language Implementations

This chapter describes compatibility issues regarding data types and calling conventions when linking modules written in other languages for the MCS®-96 processor with iC-96 modules. It also describes iC-96 conformance to ANSI C and explains how iC-96 implements some characteristics of the C language.

9.1 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, ASM96, or an older version of Intel C for the MCS-96 processor, variables referenced by external symbols must be represented in memory in a format compatible with iC-96 data type representations, as described in this chapter.

9.1.1 Data Types

The iC-96 compiler supports all ANSI data types except wide characters. *C: A Reference Manual*, listed in the Chapter 1, describes the ANSI data types. Floating-point data types in iC-96 are always 32 bits.

Table 9-1 shows the scalar data types for the MCS-96 processor, the amount of memory occupied by the data type, the arithmetic format, and the range of accepted values.

Table 9-1 MCS®-96 Processor Scalar Data Types

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
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
pointer	2	address	64 kilobytes
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

A character constant can contain up to two characters and is stored in integer (2-byte) 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. That is, the compiler zero-extends the unassigned (high-order) byte of a single-character value in a character constant.

9.1.2 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.

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 9-1 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.

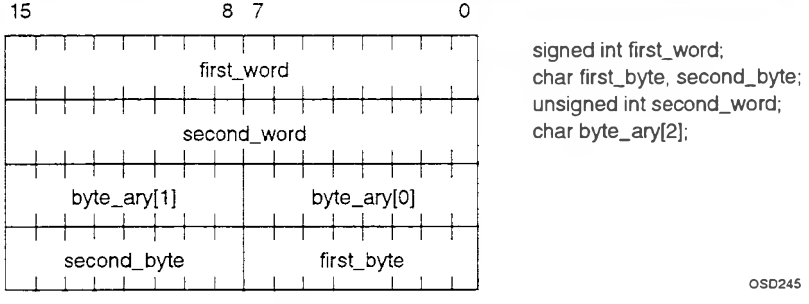

Figure 9-1 Contiguity of Variables

9.1.3 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 9-2 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.

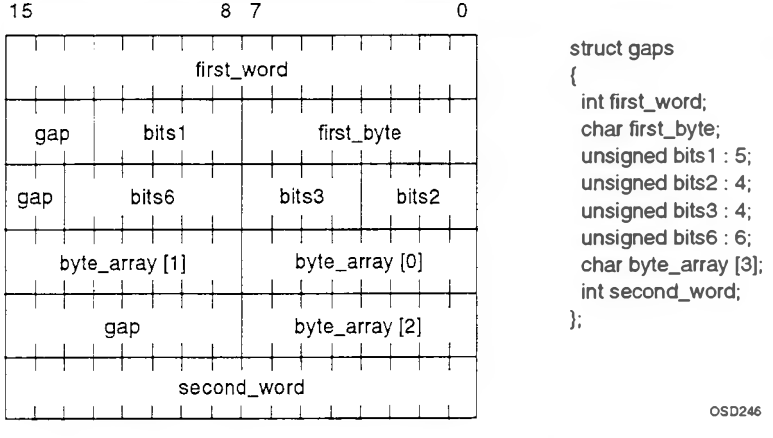

Figure 9-2 Alignment of Structure Members

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 9-3 shows a structure allocation using bit fields for padding.

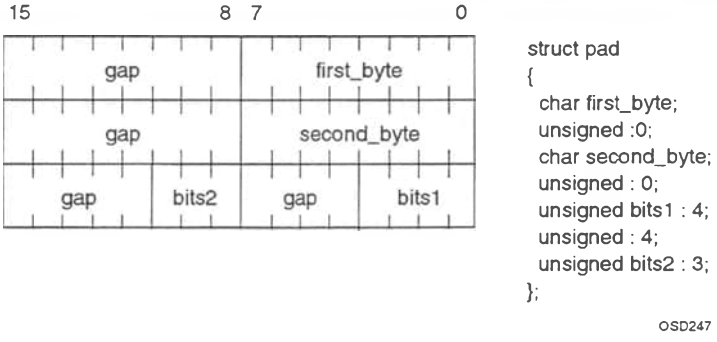

Figure 9-3 Alignment of Structure Members With Padding

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 register than what you have specified in the `registers` control, if any.

9.2 Calling Conventions

This section describes the four sections of object code (shown in Figure 9-4) 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.

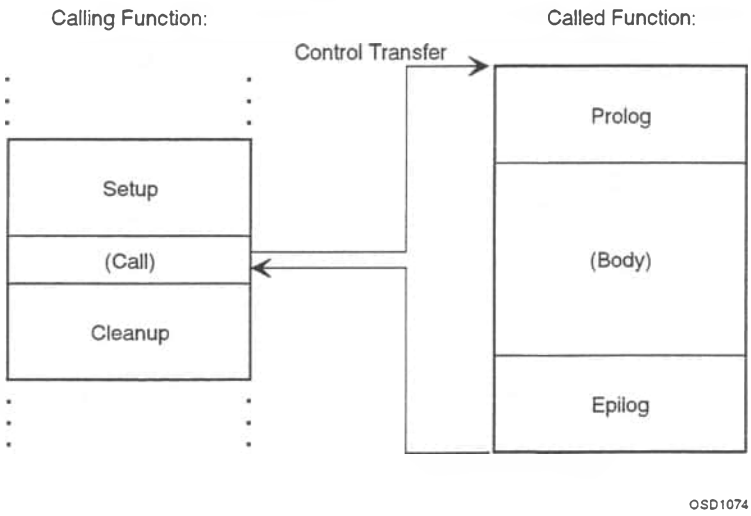

Figure 9-4 The Four Sections of Code For a Function Call

The iC-96 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 iC-96 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.

9.2.1 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.

9.2.2 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, `PLMREG`. For aggregate return values, `PLMREG` contains a pointer to a temporary aggregate variable.

9.2.3 Local Variables

The prolog allocates space on the stack for local variables. This space is commonly called a frame. The `?FRAMEØ1` variable is a relocatable word register that points to the beginning of the frame and is commonly called the frame pointer. A module named `FRAMØ1`, in the `c96.lib` library, defines `?FRAMEØ1` 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 ?FRAMEØ1. To find the address of the ?FRAMEØ1 variable, examine the map file produced by RL96.

If your iC-96 function calls or is called by an ASM96 or PL/M-96 module, you must know the possible differences in stack usage for local variables. For example, since the iC-96 compiler does not support nested function definitions, it uses only one frame pointer. However, the PL/M-96 compiler supports up to 18 nested procedures and can use a maximum of 17 frame pointers, with names other than ?FRAMEØ1.

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 ?FRAMEØ1       ;Saves the frame pointer of the calling
                    ;function, to allow for reentrancy.
ld   ?FRAMEØ1,SP    ;Loads the stack pointer of the called
                    ;function into the frame pointer.
```

Figure 9-5 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:

```
[?FRAMEØ1]     ; The old frame pointer saved on the stack
2[?FRAMEØ1]    ; The local variable 'a'
4[?FRAMEØ1]    ; The local variable 'b'
6[?FRAMEØ1]    ; The local variable 'c'
```

The compiled program uses the frame pointer as follows:

```
sub   SP,#6H        ;These first three instructions set up the
push  ?FRAMEØ1      ;stack frame.  ?FRAMEØ1 is set up as
ld    ?FRAMEØ1,SP   ;a pointer to local variables a, b, and c.

ld    TmpØ,4[?FRAMEØ1]    ;Load the contents of the variable
                          ;'b' into a temporary register.

add   TmpØ,6[?FRAMEØ1]    ;Add the variable 'c' to the variable
                          ;'b' which was stored in TMPØ,
                          ;then store the sum in TMPØ.

st    TmpØ,2[?FRAMEØ1]    ;Store the result of the
                          ;addition into the variable 'a'.

pop   ?FRAMEØ1            ;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 the RL96, lists the address of `?FRAMEØ1`. For example, linking the `exfrm.obj` object file with the following RL96 invocation produces the `exfrm.m96` map file:

```
rl96 cstart.obj,exfrm.obj,c96.lib to exfrm.out print(exfrm.m96)
```

Figure 9-6 contains a section of the `exfrm.m96` map file. The address of `?FRAMEØ1`, `1AH` in this example, appears in the `VALUE` column, beside the `?FRAMEØ1` entry in the `NAME` column. The local variables reside on the stack immediately following the top of the frame.

```
iC-96  Compiler   EXFRM                    10/27/91 17:56:19  Page   1

DOS 5.Ø (Ø46-N) iC-96 Compiler Vx.y, Compilation of module EXFRM
Object module placed in EXFRM.obj
Compiler invoked by: d:\ic96\bin\IC96.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  ?FRAMEØ1
          0006  A01800               E         ld    ?FRAMEØ1,SP
                                               ;   Statement     4
          0009  A300041C             E         ld    TmpØ,b[?FRAMEØ1]
          000D  6700061C             E         add   TmpØ,c[?FRAMEØ1]
          0011  C300021C             E         st    TmpØ,a[?FRAMEØ1]
                                               ;   Statement     5
          0015  CCØØ                 E         pop   ?FRAMEØ1
          0017  65060018                       add   SP,#6H
          001B  FØ                             ret
```

Figure 9-5 Print File

```
ATTRIBUTES              VALUE     NAME
----------              -----     ----

                                  PUBLICS:
CODE     ENTRY          2083H       MAIN
REG      WORD           001AH       ?FRAME01
REG      NULL           001CH       PLMREG
NULL     NULL           002EH       MEMORY
NULL     NULL           1FD2H       ?MEMORY_SIZE
```

Figure 9-6 Map File Illustrating Frame Pointer

9.2.4 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 `?OVRBASE`. 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 RL96 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:

```
fool()
{
   register char a, b, c;
   a = b = c;
}
```

The compiler produces the following code:

```
fool:
      push  ?OVRBASE
      push  ?OVRBASE+2H
      ldb   b,c
      ldb   a,b
      pop   ?OVRBASE+2H
      pop   ?OVRBASE
      ret
```

9.2.5 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.

9.3 Implementation-dependent iC-96 Features

This section provides information about how iC-96 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.

9.3.1 Characters

The iC-96 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.

9.3.2 Identifiers

The iC-96 compiler supports 40-character significance in external and internal names. The compiler forces external names to uppercase. Case is significant in internal names.

9.3.3 Extended Semantics and Syntax

The iC-96 compiler supports the `alien`, `reentrant`, and `nonreentrant` keywords, and allows file-level `register` variables when the `extend` control is in effect. These iC-96 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 iC-96 compiler allows an extended syntax for type qualifiers that does not conflict with ANSI C.

In iC-96, 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 iC-96 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;              /* iC-96 extended syntax */
```

9.3.4 Initialization

The iC-96 compiler supports initialization of only constant objects at file scope and initialization of only constant and automatic objects at block scope.

Examples of file-scope (global) initializations are:

```
const int i = 1;

long l;
long *const lp = &l;

const struct {  int i, j; } s1 = { 1, 2};
```

Examples of invalid file-scope initializations are:

```
int si = 1;                     /* non-const object */

struct { const int i;
         int j;
       } s2 = {3};

static register int sri = 1;

const register int cri = 1;     /* register variables cannot */
                                /* be initialized at file    */
                                /* scope */
```

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) */
```

9.3.5 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 pointer to an `int` or `short` preserves the bit representation. If cast to a signed integer, the result can be negative.

- Casting a pointer to a `long` or `char` is not supported.

The result of casting an integer data type to a pointer is as follows:

- Casting an `int` or a `short` to a pointer preserves the bit representation.
- Casting a `char` to a pointer zero- or sign-extends the result depending on whether `signedchar` or `nosignedchar` is in effect.
- Casting a `long` to a pointer discards the high-order 16 bits.

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.

9.3.6 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.

9.3.7 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`.

9.4 Compiler Limits

The values listed in Table 9-2 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.

Table 9-2 Compiler Limits

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 c96inc 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	256
nesting level of functions specified in function argument lists	20
number of functions defined in a module	255
number of external references in a module	256
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

Installation

iC-96 is supplied on two low-density, 5.25-inch diskettes or on one 3.5-inch diskette. Before installation, make a backup copy of both product diskettes using the DOS `diskcopy` command.

Hardware

All of Intel's development tools for DOS require an IBM PC XT, PC AT (recommended), or a true compatible. This software requires at least a minimum of 192 kilobytes of RAM, and DOS Version 3.0 or later.

Installation on DOS Systems

The installation program `install.exe` installs iC-96, RL96, LIB96, and OH on your DOS host system. The program sets up directories and places files in them. At certain points during the installation, you can:

- name a directory structure or choose the default
- choose to install the compiler, the utilities, or both
- change your `autoexec.bat` and `config.sys` files

To begin the installation, place the 5.25-inch diskette labelled `iC-96 COMPILER DISK 1 OF 2` or the 3.5-inch diskette labelled `iC-96 COMPILER DISK 1 OF 1` in the appropriate drive and run the `install` program. In the following example, the diskette is in the `a:` drive:

```
C:>  a:\install
```

NOTE

The installation program used to install the iC-96 compiler and MCS®-96 utilities, *INSTALL*, is licensed software provided by Knowledge Dynamics Corporation, Highway Contract 4 Box 185-H, Canyon Lake, Texas 78133-3508 (USA). *INSTALL* is provided to you for the exclusive purpose of installing the iC-96 and MCS-96 software.

Environment Variables

DOS and the iC-96 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` is recognized by DOS 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 iC-96 compiler, you need not retype the full pathname each time you invoke the compiler. Your DOS literature explains how to define the `path` environment variable.

`c96inc` 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 `c96inc` in turn and uses each resulting pathname as the name of the include file.

The compiler uses the pathnames formed from the list in `c96inc` in addition to any you specify with the `searchinclude` control. For example, the following definition of `c96inc` locates the files `c:\ic96\include\stdio.h` and `c:\working\kal4.h` when the `include (stdio.h,kal4.h)` control is specified:

```
C:> set C96INC=c:\ic96\include;c:\working
```

`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:

```
C:> 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 on in a preprocessor directive.

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. The `:work:` environment variable specifies the directory where the compiler is to put these temporary files. If `:work:` is not defined or is empty, the compiler uses your current working directory for the temporary files. For example, setting `:work:` as follows causes the compiler to use the `c:\temps\cwrk` directory for temporary work files:

```
C:> set :work:=c:\temps\cwrk
```

Glossary

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
ASM96	MCS®-96 assembly language
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
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
dead-code optimization	compile process that eliminates code that can never be executed
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

file-level variable	a variable defined outside of any control block
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
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
header file	source text file containing variable declarations, function prototypes, in-line assembly language functions, and macro definitions
iC-96	MCS-96 C language
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 an iC-96 program, that is assembled as ASM96 source text rather than compiled as iC-96 source text
instruction set	the set of machine codes recognized by the MCS-96 processor
integral types	Types that include all forms of integers, characters, and enumerations.
length-prefixed string	character string beginning with a value that indicates how many characters long it is
LIB96	MCS-96 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
MCS-96	8096 microcontroller system: 8096-90, 8096-BH, 80C196KB, 80C196KC, 80C196KR
multiply-aliased	having more than one name
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
OH	MCS-96 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
overlaying registers	allocating the same registers to more than one function
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 instruction sequences to reduce overall code size
PL/M-96	the MCS-96 PL/M compiler
portable	not dependant 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
RAM	random access memory
reentrant	a function that calls itself or gets called again in a call loop
register file	MCS-96 on-chip memory used for high-speed data access and for hardware control; also called register memory
registers	bytes in the register file
RL96	MCS-96 relocation and linking utilities
ROM	read-only memory
run-time	during execution
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
target	system on which the application program executes
UDI string	length-prefixed string

vector table	table containing addresses pointing to code
VPL	variable-parameter list calling convention
word	16 bits; 2 bytes
zero-extend	in promoting a data type, filling the bits of the unused high-order part of the longer data type with zeroes

Index

intel

product release notes

iC-96 COMPILER VERSION 2.3 ON DOS SYSTEMS

These Product Release Notes are divided into the following sections:

- **PRODUCT CHECKLIST**
- **PRODUCT ENHANCEMENTS**
- **iC-96 PRODUCT NOTES**
- **KNOWN ANOMALIES**

PRODUCT CHECKLIST

Item	Description
1	Binder containing:

- *iC-96 Compiler User's Guide for DOS Systems*
- *MCS®-96 Utilities User's Guide*
- *The 8096 Floating-point Library User's Guide*
- Product Release Notes-iC-96 Compiler
- Intel Software License and Registration Certificate
- One 3-1/2" diskette
- Two 5-1/4" diskettes

PRODUCT ENHANCEMENTS

8XC196KR PROCESSOR SUPPORT

Support for the 8XC196KR processor has been improved. You no longer need to link in a separate object file. The SFR definitions are now contained within `c96.lib`. Also, there are two new header files, `kr.h` and `auto_kr.h`. These header files contain external declarations of the 8XC196KR SFRs, which correspond to the *8XC196KR User's Manual* definitions, and the *8XC196KR User's Guide and Data Sheet*, respectively. See the *iC-96 Compiler User's Guide for DOS Systems* for complete details.

NEW BMOV CONTROL

The new `bmov` control provides support for the use of the `bmov` instruction on the 8XC196KB processor. Thus, you must also specify the `model(kb)` control. The compiler uses this instruction for copying large structures or unions. The iC-96 compiler, by default, generates the BMOVI instruction, which is available in later models of the 8XC196 processor, to do the same process. The 8XC196KB processor does not have the BMOVI instruction. The compiler does not generally use the BMOV instruction because it is not interruptable and can cause significant delays in responding to interrupts. To instruct the compiler to use the BMOV instruction, specify the `bmov` control with the `model(kb)` control on the command line or in a `#pragma` directive.

NEW /HELP OPTION

Invoke the iC-96 compiler with the `/help` option to get on-line help regarding use of the various controls for iC-96.

This facility displays the file `ic96.hlp` one screenful at a time. You can customize the `ic96.hlp` file to suit your needs.

iC-96 PRODUCT NOTES

USING LONG REGISTERS IN ASM INSTRUCTIONS

iC-96 currently does not contain support for aligning long registers on long-word boundaries. In-line assembly instructions `mul`, `div`, `shll`, `shral`, and `shrl` require a long-word aligned operand. If the register is not long-word aligned, the instruction generates unpredictable results.

You can use `#pragma locate` to position the long operands on long-word boundaries. See the *iC-96 Compiler User's Guide for DOS Systems* for more details on how to use the pragma `locate` control.

EXTERNAL CALLBACK WITH OVERLAID REGISTERS

If module A contains nonreentrant functions A1 and A2, which the compiler determines can never be active simultaneously, the compiler can allocate the same overlayable registers to both functions. However, if A1 calls an external function X1, and X1 then calls A2, A2 uses those shared registers without preserving them, thus destroying A1's data. Neither the compiler nor the linker can detect this situation.

Avoid this problem by moving the external function X1 into module A, or by putting A1 and A2 in separate modules. Using the first method, the compiler can determine that A1 and A2 cannot share registers. Using the second method, the registers are not shared, since overlayable register segments are not shared between modules.

KNOWN ANOMALIES

NOLIST CONTROL LEAVES LISTING TURNED OFF

When you use `#pragma list` and `#pragma nolist` to turn source listing on and off at various places in the source text, make sure that you use `#pragma list` last. Otherwise, no source listing is produced.

REGISTER OVERLAYING CAN MAKE MISTAKES

In some cases, the allocation of registers in the overlay register segment erroneously assigns two variables to the same register. A complicated set of conditions causes the compiler to make this mistake. One such condition is when at least one function contains an odd number of bytes of overlaid registers.

A work-around for this problem is to make sure that every function uses an even number of overlaid registers. Declare an extra register char variable, if needed.

CAST ERRONEOUSLY DELETED FOR CERTAIN OPERATIONS

In an expression such as `(long)(i - j) * 32 / 128`, the cast to long is not done because a multiply operation is about to take place, which produces a long operand anyway. The compiler does not anticipate the multiply changing to a shift because 32 is a power of two.

Casting one of the deeper operands, either `i` or `j` in this case, as in `((long)i - j) * 32 / 128`, forces the cast to take place earlier, resulting in long operations. Explicitly using a shift works too, as in `((long)(i - j) << 5) / 128`, or `(long)(i - j) << 5 >> 7`.

ADDRESS of ARRAY NOT HANDLED CORRECTLY

The compiler should take no action on the & (address of) operator applied to an array name, but the compiler does not handle this construct properly. The name of an array is already an address, so the & (address of) operator is unnecessary. The only useful place for & array is in the expression given to sizeof, where this construct changes the result of sizeof to the size of the whole array, rather than the size of one element.

Except for its use in sizeof do not apply the & operator to an array name.

CONSTANT INITIALIZATION NOT PERFORMED FOR CERTAIN CONST POINTERS

In the following code sample, the local variable is not initialized.

```
main()
{
    char array[5];              /* Local variable */
    char *const cptr = array;
    ...
}
```

One work-around is to use non-const pointers. Another is to declare const pointers at file scope, along with the type to which they point.

READER RESPONSE CARD

We'd Like Your Opinion

Please use this form to help us evaluate the effectiveness of this manual and improve the quality of future versions.

To order publications, contact the Intel Literature Distribution Center (see page ii of this manual).

Fill in the blanks below with a rating of 1 through 10, where 1 is poor and 10 is excellent:

___ Readability
___ Technical depth
___ Technical accuracy
___ Usefulness of material for your needs
___ Comprehensibility of material
___ OVERALL QUALITY OF THIS MANUAL

If you have any comments, please include them here:

_____

_____

_____

What suggestions would you have for improving this manual:

_____

_____

If you would like us to call you for more specific suggestions about this book, please additionally fill in your phone number below.

Name _____

Phone Number (_____)_____

Address _____

_____

_____

Thanks for taking the time to fill out this form.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 79 HILLSBORO OR

POSTAGE WILL BE PAID BY ADDRESSEE

DTO TECHNICAL PUBLICATIONS JF1-59
INTEL CORPORATION
5200 NE ELAM YOUNG PARKWAY
HILLSBORO OR 97124-9978

Please fold here and close the card with tape. Do not staple.

WE'D LIKE YOUR COMMENTS....

This document is one of a series describing Intel products. Your comments on the other side of this form will help us produce better manuals. Each reply will be reviewed. All comments and suggestions become the property of Intel Corporation.

If you are in the United States and are sending only this card, postage is prepaid.

If you are sending additional material or if you are outside the United States, please insert this card and any enclosures in an envelope. Send the envelope to the above address, adding "United States of America" if you are outside the United States.

Thanks for your comments.