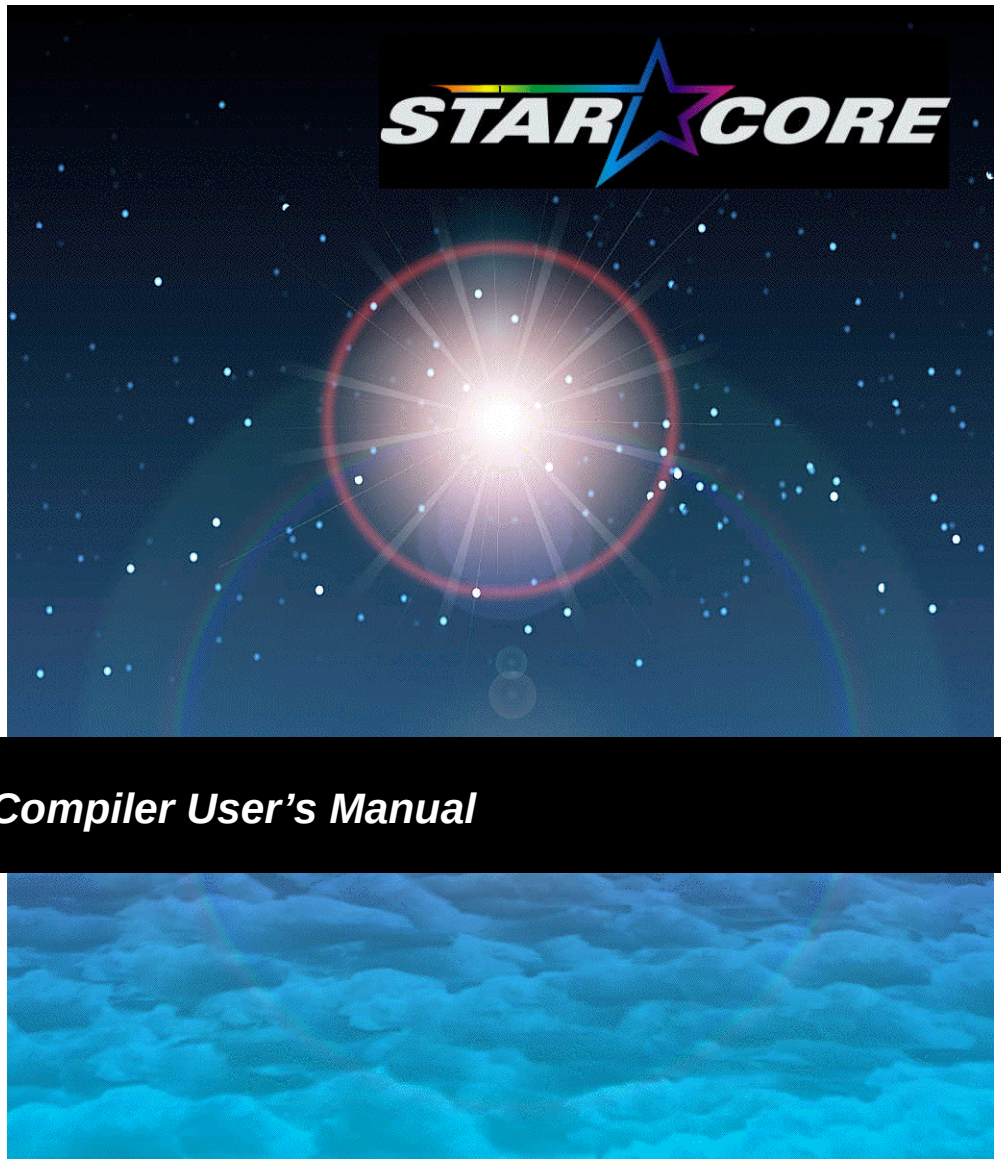
MNSC100CC/D
Rev. 2.0, 11/2001



SC100 C Compiler User's Manual

MOTOROLA digital dna

agere systems

MNSC100CC/D
Rev. 2.0, 11/2001

SC100 C Compiler

User's Manual

MOTOROLA digital dna™

agere systems

This document contains information on a new product. Specifications and information herein are subject to change without notice.

Table of Contents

About This Book

Audience xvii
Organization xvii
Suggested Reading xviii
References xviii
Conventions xviii
Acronyms and Abbreviations xviii
Revision History xviii

Chapter 1
Introduction

1.1 Overview of the SC100 C Compiler 1-1
1.2 The Cross-File Optimization Approach 1-2
1.3 Compiling Applications 1-3
1.3.1 The Compiler Shell Program 1-3
1.3.2 Stages in the C Compilation Process 1-4

Chapter 2
Getting Started

2.1 A Quick Start 2-1
2.1.1 Creating and Executing a Program 2-1

Chapter 3
Using the SC100 C Compiler

3.1 The Shell Program 3-1
3.1.1 The C Compilation Process 3-2
3.1.2 Cross-File Optimization 3-4
3.1.3 File Types and Extensions 3-7
3.1.4 Environment Variables 3-9
3.2 Invoking the Shell 3-9
3.2.1 Command Line Syntax 3-9
3.2.1.1 Command Line Syntax Rules 3-9
3.2.1.2 Command Files 3-10
3.3 Shell Control Options 3-10
3.3.1 Controlling the Behavior of the Shell 3-14
3.3.1.1 Controlling where the shell stops processing 3-14
3.3.1.2 Specifying a shell command file 3-15
3.3.1.3 Displaying the shell Help page 3-15
3.3.2 Specifying Preprocessing Options 3-16

3.3.2.1 Changing preprocessed output 3-16
3.3.2.2 Defining and undefining preprocessor macros 3-17
3.3.2.3 Adding directories to the #include file path 3-17
3.3.3 Overriding Input File Extensions 3-18
3.3.4 Output Filename and Location Options 3-19
3.3.5 Specifying C Language Options 3-20
3.3.5.1 Defining the language version 3-20
3.3.5.2 Adding debugging information to files 3-20
3.3.5.3 Changing the default char sign setting 3-20
3.3.5.4 Indicating fractional data-types in saturation 3-21
3.3.6 Passing Options Through to Specific Tools 3-21
3.3.7 Setting the Options for Listings and Messages 3-22
3.3.7.1 Generating listing files 3-22
3.3.7.2 Controlling the type of information displayed 3-23
3.3.7.3 Suppressing warnings 3-23
3.3.7.4 Reporting all remarks and warnings 3-23
3.3.8 Specifying the Hardware Model and Configuration 3-24
3.3.8.1 Defining the architecture 3-24
3.3.8.2 Configuration and Startup files 3-24
3.3.9 Specifying modes 3-25
3.3.9.1 Specifying big memory mode 3-25
3.3.9.2 Specifying tiny memory mode 3-25
3.3.9.3 Copying initialized variables from ROM 3-25
3.3.9.4 Specifying big-endian mode 3-25
3.4 Language Features 3-26
3.4.1 C Language Dialects 3-26
3.4.1.1 Standard Extensions 3-26
3.4.1.2 K&R/PCC mode 3-31
3.4.2 Types and Sizes 3-36
3.4.2.1 Characters 3-37
3.4.2.2 Integers 3-38
3.4.2.3 Floating point 3-39
3.4.2.4 Fractional representation 3-40
3.4.2.5 Pointers 3-40
3.4.2.6 Bit-fields 3-40
3.4.3 Fractional and Integer Arithmetic 3-42
3.4.4 Intrinsic Functions 3-45
3.4.4.1 Data types for intrinsic functions 3-45
3.4.4.2 Intrinsic function categories 3-46
3.4.4.3 Intrinsic functions examples 3-51
3.4.5 Pragmas 3-52
3.4.5.1 Syntax 3-52
3.4.5.2 Placement 3-52
3.4.5.3 Pragmas which apply to functions 3-54
3.4.5.4 Pragmas which apply to statements 3-56
3.4.5.5 Pragmas which apply to variables 3-59
3.4.6 Predefined Macros 3-61

Chapter 4
Interfacing C and Assembly Code

4.1 Inlining a Single Assembly Instruction 4-1
4.2 Inlining a Sequence of Assembly Instructions 4-2
4.2.1 Guidelines for Inlining Assembly Code Sequences 4-2
4.2.2 Defining an Inlined Sequence of Assembly Instructions 4-3
4.3 Calling an Assembly Function in a Separate File 4-7
4.3.1 Writing the Assembly Code 4-7
4.3.2 Calling the Assembly Function 4-8
4.3.3 Integrating the C and Assembly Files 4-8
4.4 Including Offset Labels in the Output File 4-9

Chapter 5
Optimization Techniques and Hints

5.1 Optimizer Overview 5-2
5.1.1 Basic Blocks 5-2
5.1.2 Linear and Parallelized Code 5-3
5.1.3 Optimization Levels and Options 5-4
5.2 Using the Optimizer 5-6
5.2.1 Invoking the Optimizer 5-6
5.2.2 Optimizing for Space 5-7
5.2.3 Using Cross-File Optimization 5-7
5.3 Optimization Types and Functions 5-8
5.3.1 Dependencies and Parallelization 5-8
5.3.2 Target-Independent Optimizations 5-9
5.3.2.1 Target-Independent Strength reduction (loop transformations) 5-10
5.3.2.2 Function inlining 5-16
5.3.2.3 Common subexpression elimination 5-17
5.3.2.4 Loop invariant code 5-17
5.3.2.5 Constant folding and propagation 5-18
5.3.2.6 Jump-to-jump elimination 5-19
5.3.2.7 Dead code elimination 5-19
5.3.2.8 Dead storage/assignment elimination 5-19
5.3.3 Target-Specific Optimizations 5-20
5.3.3.1 Instruction scheduling 5-22
5.3.3.2 Target-specific software pipelining 5-23
5.3.3.3 Conditional execution and predication 5-26
5.3.3.4 Speculative execution 5-27
5.3.3.5 Post-increment detection 5-28
5.3.3.6 Target-specific peephole optimization 5-29
5.3.3.7 Extract peephole optimization 5-30
5.3.3.8 Target-Specific Strength Reduction 5-32
5.3.3.9 Prefix grouping 5-33
5.3.4 Space Optimizations 5-33
5.3.4.1 Code Sinking Optimization 5-34
5.3.5 Cross-File Optimizations 5-35

5.3.5.1 Rules for using Cross-file Optimization 5-35
5.4 Guidelines for Using the Optimizer 5-35
5.4.1 Partial Summation Techniques 5-37
5.4.2 Multisample Techniques 5-40
5.4.2.1 Multisample implementation issues 5-42
5.4.2.2 Implementation example 5-44
5.4.3 General Hints 5-49
5.4.3.1 Software pipelining 5-49
5.4.3.2 Passing and returning large structs 5-49
5.4.3.3 Arithmetic operations 5-49
5.4.3.4 Local variables 5-49
5.4.3.5 Resource limitations 5-50
5.5 Optimizer Assumptions 5-50

Chapter 6
Runtime Environment

6.1 Startup Code 6-1
6.1.1 Bare Board Startup Code 6-2
6.1.2 C Environment Startup Code 6-3
6.1.2.1 C environment initialization code 6-3
6.1.2.2 Initialization of variables 6-3
6.1.2.3 C environment finalization code 6-3
6.1.2.4 Low level I/O services 6-3
6.1.3 Configuring Your Startup Code 6-4
6.2 Memory Models 6-5
6.2.1 Small and Tiny Memory Models 6-5
6.2.2 Big Memory Model 6-5
6.2.3 Linker Command Files 6-6
6.3 Memory Layout and Configuration 6-7
6.3.1 Stack and Heap Configuration 6-8
6.3.1.1 Runtime stack 6-9
6.3.1.2 Dynamic memory allocation (heap) 6-9
6.3.2 Static Data Allocation 6-10
6.3.3 Configuring the Memory Map 6-10
6.3.3.1 Memory map configuration example 6-10
6.3.4 Machine Configuration File 6-11
6.3.4.1 Defining the memory configuration 6-11
6.3.5 Application Configuration File 6-13
6.3.5.1 File structure and syntax 6-14
6.3.5.2 Schedule section 6-14
6.3.5.3 Binding section 6-16
6.3.5.4 Overlay section 6-17
6.4 Calling Conventions 6-19
6.4.1 Stack Pointer 6-19
6.4.2 Stack-Based Calling Convention 6-19
6.4.3 Optimized Calling Sequences 6-21
6.4.4 Stack Frame Layout 6-22

6.4.5 Interrupt Handlers 6-23
6.4.6 Frame Pointer and Argument Pointer 6-23
6.4.7 Hardware Loops 6-24
6.4.8 Operating Modes 6-24
6.5 Saturation 6-25
6.5.1 Saturation switches 6-25
6.5.2 Saturation states 6-25

Chapter 7
Runtime Libraries

7.1 Providing Runtime Libraries 7-2
7.1.1 Using Libraries with debug 7-2
7.1.2 Building the Libraries 7-2
7.2 Character Typing and Conversion (ctype.h) 7-2
7.2.1 Testing Functions 7-3
7.2.2 Conversion Functions 7-3
7.3 Floating Point Characteristics (float.h) 7-4
7.3.1 Floating Point Library Interface (fltmath.h) 7-5
7.3.1.1 Round_Mode 7-5
7.3.1.2 FLUSH_TO_ZERO 7-6
7.3.1.3 IEEE_Exceptions 7-6
7.3.1.4 EnableFPExceptions 7-7
7.4 Integer Characteristics (limits.h) 7-8
7.5 Locales (locale.h) 7-8
7.6 Floating Point Math (math.h) 7-9
7.6.1 Trigonometric Functions 7-9
7.6.2 Hyperbolic Functions 7-9
7.6.3 Exponential and Logarithmic Functions 7-10
7.6.4 Power Functions 7-10
7.6.5 Other Functions 7-10
7.7 Nonlocal Jumps (setjmp.h) 7-11
7.8 Signal Handling (signal.h) 7-11
7.9 Variable Arguments (stdarg.h) 7-11
7.10 Standard Definitions (stddef.h) 7-12
7.11 I/O Library (stdio.h) 7-12
7.11.1 Input Functions 7-12
7.11.2 Stream Functions 7-13
7.11.3 Output Functions 7-14
7.11.4 Miscellaneous I/O Functions 7-14
7.12 General Utilities (stdlib.h) 7-15
7.12.1 Memory Allocation Functions 7-15
7.12.2 Integer Arithmetic Functions 7-15
7.12.3 String Conversion Functions 7-16
7.12.4 Searching and Sorting Functions 7-16
7.12.5 Pseudo Random Number Generation Functions 7-16
7.12.6 Environment Functions 7-17
7.12.7 Multibyte Character Functions 7-17

7.13 String Functions (string.h) 7-17
7.13.1 Copying Functions 7-18
7.13.2 Concatenation Functions. 7-18
7.13.3 Comparison Functions 7-18
7.13.4 Search Functions. 7-19
7.13.5 Other Functions. 7-19
7.14 Time Functions (time.h) 7-20
7.14.1 Time Constant. 7-20
7.14.2 Process Time. 7-20
7.15 Built-in Intrinsic Functions (prototype.h). 7-21

Appendix A
Migrating from Other Environments

A.1 Code Migration Overview A-1
A.2 Migrating Code Developed for DSP56600. . . . . A-2
A.3 Migrating Code Developed for TI6xx A-6

List of Tables

3-1 File Types and Extensions 3-7
3-2 Shell Options Summary 3-11
3-3 Data Types and Sizes 3-36
3-4 Interpretation of 16-bit Data Values 3-43
3-5 Interpretation of 40-bit Data Values 3-43
3-6 Intrinsic Functions 3-47
3-7 Pragmas 3-53
3-8 Predefined Macros 3-61
5-1 Optimization Levels 5-4
5-2 Optimization Options Summary 5-5
5-3 Summary of Target-Independent Optimizations 5-9
5-4 Summary of Target-Specific Optimizations 5-20
6-1 Status Register Default Settings 6-2
6-2 Memory Models 6-5
6-3 Small Memory Model Default Values 6-8
6-4 Big Memory Model Default Values 6-8
6-5 Tiny Memory Model Default Values 6-8
6-6 Register Usage in the Stack-based Calling Convention 6-20
7-1 Supported ISO Libraries 7-1
7-2 Supported Non-ISO Libraries 7-1
7-3 Testing Functions 7-3
7-4 Conversion Functions 7-3
7-5 Contents of File float.h 7-4
7-7 Locale Functions 7-8
7-6 Contents of File limits.h 7-8
7-8 Trigonometric Functions 7-9
7-9 Hyperbolic Functions 7-9
7-10 Exponential and Logarithmic Functions 7-10
7-11 Power Functions 7-10
7-12 Other Functions 7-10
7-13 Nonlocal Jumps 7-11
7-14 Signal Handling (signal.h) 7-11
7-15 Variable Arguments (stdarg.h) 7-11
7-16 Standard Definitions (stddef.h) 7-12

7-17 Input Functions 7-12
7-18 Stream Functions 7-13
7-19 Output Functions. 7-14
7-20 Miscellaneous I/O Functions 7-14
7-21 Memory Allocation Functions 7-15
7-22 Integer Arithmetic Functions 7-15
7-23 String Conversion Functions 7-16
7-24 Searching and Sorting Functions 7-16
7-25 Pseudo Random Number Generation Functions. 7-16
7-26 Environment Functions. 7-17
7-27 Multibyte Character Functions 7-17
7-28 Copying Functions 7-18
7-29 Concatenation Functions. 7-18
7-30 Comparison Functions 7-18
7-31 Search Functions. 7-19
7-32 Other Functions. 7-19
7-33 Time Functions 7-20
7-34 Time Constant. 7-20
7-35 Built-in Intrinsic Functions. 7-21
A-1 DSP56600 Integer Data Type Differences A-2
A-2 DSP56600 Fractional Data Type Differences A-2
A-3 DSP56600 Pointer Size Differences. A-3
A-4 DSP56600 Fractional Arithmetic Differences A-3
A-5 DSP56600 Intrinsic Function Differences A-4
A-6 DSP56600 Storage Specifiers. A-5
A-7 DSP56600 Miscellaneous Differences A-5

List of Figures

1-1 The SC100 C Compilation Process 1-5
3-1 The C Compilation Process 3-3
3-2 Traditional Optimization. 3-5
3-3 Cross-File Optimization 3-6
3-4 File Extensions in the Shell Cycle 3-8
3-5 Characters—Memory Layout 3-37
3-6 Characters—Dn Register Layout 3-37
3-7 Characters—Rn Register Layout 3-37
3-8 Integers—Memory Layout 3-38
3-9 Integers—Alignment 3-38
3-10 Integers—Dn Register Layout 3-39
3-11 Integers—Rn Register Layout 3-39
3-12 Fractional Integers—Dn Register Layout. 3-40
3-13 Extended Precision Fractional—Dn Register Layout. 3-45
5-1 Linear and Parallelized Code 5-3
5-2 Square Loop 5-14
5-3 Triangular Loop 5-15
5-4 Sequence of Target-Specific Transformation Optimizations 5-21
5-5 Single Sample and Multisample Kernels 5-40
5-6 Single ALU Operand and Memory Bandwidth 5-41
5-7 Quad ALU Operand and Memory Bandwidth 5-41
5-8 Options for Increasing Operand Bandwidth 5-42
5-9 Number of Samples and ALUs for Implementing DSP Algorithms 5-42
5-10 Quad Coefficient Loading from Memory. 5-43
5-11 Misalignment when Loading Quad Operands 5-43
5-12 Quad ALU, Quad Sample FIR Filter Data Flow 5-44
5-13 FIR Filter Equations for Four Samples. 5-45
5-14 Generic Kernel For Quad ALU FIR 5-45
6-1 SC100 Default Memory Layout 6-7
6-2 Stack Frame Layout 6-22

List of Examples

2-1 Sample source file: hello.c 2-1
3-1 Invoking the shell 3-10
3-2 Defining a shell command file 3-15
3-3 Contents of a shell command file 3-15
3-4 Shell Help page (extract) 3-15
3-5 Overriding file extensions. 3-18
3-6 Specifying output files 3-19
3-7 Passing multiple options to the same tool. 3-21
3-8 Defining the architecture 3-24
3-9 Defining a predicate name 3-26
3-10 Deleting a predicate 3-27
3-11 Declaring an asm function 3-28
3-12 Returning the alignment requirement 3-29
3-13 Out of range warning 3-29
3-14 External entities in other scopes 3-30
3-15 Pointers to incomplete arrays 3-30
3-16 Prototyped parameter list 3-32
3-17 Omitting the declarator list 3-33
3-18 Bit-field alignment to long word (1). 3-41
3-19 Bit-field alignment to character 3-41
3-20 Bit-field alignment to long word (2). 3-41
3-21 Bit-field offset. 3-41
3-22 Fractional arithmetic examples. 3-42
3-23 Integer arithmetic examples 3-42
3-24 Integer arithmetic computation. 3-44
3-25 Fractional arithmetic computation 3-44
3-26 Intrinsic functions 3-51
3-27 Intrinsic functions using extended precision 3-51
3-28 #pragma noinline 3-54
3-29 #pragma save_ctxt 3-54
3-30 #pragma external. 3-55
3-31 #pragma interrupt 3-56

3-32 #pragma profile with constant value 3-56
3-33 #pragma profile with frequency ratio 3-57
3-34 #pragma loop count 3-58
3-35 #pragma align 3-60
4-1 Inlining a single assembly instruction 4-2
4-2 Inlining syntax 4-4
4-3 Simple inlined assembly function 4-4
4-4 Inlined assembly function with labels and hardware loops 4-5
4-5 Referencing global variables in an inlined assembly function 4-6
4-6 Assembly function in a separate file 4-7
4-7 C code calling assembly function 4-8
4-8 Integrating C and assembly files 4-8
4-9 Data structure shared between C and assembly 4-9
4-10 Specifying the output of offset information 4-9
4-11 Data structure offsets in the assembly output file 4-9
4-12 Using symbolic offsets in assembly code 4-10
5-1 Invoking the optimizer with default settings 5-6
5-2 Invoking the optimizer for target-independent optimizations only 5-6
5-3 Invoking the optimizer with cross-file optimization 5-6
5-4 Simple instruction dependency 5-8
5-5 Algorithm instruction dependency 5-8
5-6 Loop transformation - simple loop 5-10
5-7 Loop transformation - dynamic loop 5-11
5-8 Loop transformation - multi-step loop 5-12
5-9 Loop transformation - composed variables 5-13
5-10 Loop transformation - square loop 5-14
5-11 Loop transformation - triangular loop 5-15
5-12 Function inlining 5-17
5-13 Common subexpression elimination 5-17
5-14 Loop invariant code motion 5-17
5-15 Constant folding and propagation 5-18
5-16 Jump-to-jump elimination 5-19
5-17 Dead code elimination 5-19
5-18 Dead storage/assignment elimination 5-19
5-19 Instruction scheduling 5-22
5-20 Filling delay slots 5-22
5-21 Avoiding pipeline restrictions 5-23

5-22 Software pipelining - complex FIR 5-24
5-23 Software pipelining - vector multiplication by a constant 5-25
5-24 Conditional execution and predication 5-26
5-25 Speculative execution 5-27
5-26 Post-increment detection. 5-28
5-27 Target-specific peephole optimization 5-29
5-28 Combined pipelining and peephole optimizations 5-29
5-29 When the AND constant does not fit in lower or upper 16 bits 5-30
5-30 ASR followed by an AND (no ASLL necessary). 5-31
5-31 Using an EXTRACT instead of an EXTRACTU. 5-31
5-32 Invoking the optimizer for space optimization. 5-33
5-33 Code sinking optimization 5-34
5-34 Simple and complex array accesses 5-35
5-35 MAC usage limited by dependency in loop 5-37
5-36 Partial summation for dual MAC usage 5-38
5-37 Alignment restrictions in algorithms 5-39
5-38 Single instruction quad ALU generic filter kernel 5-46
5-39 FIR_A4S4 quad ALU, quad sample C simulation code. 5-46
5-40 Avoiding software pipelining in source code 5-49
6-1 Creating a new startup file 6-4
6-2 Assembling the modified startup file 6-4
6-3 Using the modified startup file 6-4
6-4 Big, small, and tiny memory models 6-6
6-5 Small and tiny memory mode instruction. 6-6
6-6 Allocating large arrays from the heap. 6-10
6-7 Modified memory map configuration. 6-10
6-8 Modified memory configuration in the linker command file 6-11
6-9 Defining a data memory space 6-12
6-10 Defining a program memory space. 6-13
6-11 Defining multiple memory spaces 6-13
6-12 Defining additional entry points for an application 6-16
6-13 Placing a variable at an absolute location. 6-17
6-14 Defining global variable overlays. 6-18
6-15 Function call and allocation of parameters. 6-20
7-1 Changing the round mode. 7-5
7-2 Disabling flushing to zero. 7-6
7-3 Using the exception status word 7-6

7-4 Setting a signal for exceptions 7-7
7-5 Timing an application 7-20
A-1 Migrating code from other environments A-2

About This Book

This manual describes the C compiler for the StarCore SC100 generation of digital signal processor (DSP) cores (SC110 and SC140), and provides detailed guidelines for its use.

Audience

This manual is intended for systems software developers, applications developers, system hardware developers, and microprocessor designers.

Organization

This manual is organized into seven chapters and one appendix, as follows:

- Chapter 1, “Introduction,” provides an overview of the SC100 C compiler and outlines the SC100 C compilation process.
- Chapter 2, “Getting Started,” provides the essential information and instructions that enable you to start using the SC100 C compiler.
- Chapter 3, “Using the SC100 C Compiler,” explains how to use the compiler, and describes the options and features that it supports.
- Chapter 4, “Interfacing C and Assembly Code,” describes the support provided by the compiler for interfacing between C source code and assembly code, and provides instructions for using this interface.
- Chapter 5, “Optimization Techniques and Hints,” explains how the SC100 optimizer operates, and describes the optimization levels and individual optimizations that can be applied.
- Chapter 6, “Runtime Environment,” describes the startup code used by the compiler, the layout and configuration of memory, and the calling conventions that the compiler supports.
- Chapter 7, “Runtime Libraries,” describes the C libraries and I/O libraries supported by the SC100 C compiler.
- Appendix A, “Migrating from Other Environments,” provides guidelines for migrating C code from other environments to the SC100 C compiler.

Suggested Reading

C, A Reference Manual, Samuel P. Harbison and Guy L. Steele Jr, Tartan Inc. (Prentice Hall: 1995).

Compilers Principles, Techniques, and Tools, Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman (Addison-Wesley Publishing Company: 1986).

References

- *SC100 Application Binary Interface Reference Manual* (MNSC100ABI/D)
- *SC100 Assembly Language Tools User's Manual* (MNSC100ALT/D)
- *SC110 DSP Core Reference Manual* (MNSC110CORE/D)
- *SC140 DSP Core Reference Manual* (MNSC140CORE/D)
- *SC100 Simulator Reference Manual* (MNSC110SIM/D)

Conventions

This manual uses the following notational conventions:

- Courier monospaced type indicate commands, command parameters, code examples, expressions, datatypes, and directives.
- Italic type indicates replaceable command parameters.
- All source code examples are in C.

Acronyms and Abbreviations

The following list defines the acronyms and abbreviations used in this manual.

AGU	Address Generation Unit
ALU	Arithmetic Logic Unit
CFE	C Front End
DSP	Digital Signal Processor
FIR	Finite Impulse Response
IR files	Immediate Representation files
LLT	Low Level Transformations

Revision History

Rev 1.8 of the *SC100 C Compiler User's Manual* has been updated with SC110 information.

Chapter 1
Introduction

The StarCore Technology Center has focuses on ensuring a wide selection of best-in-class development tools for StarCore-based System on Chip (SoC) products. The result is an unusually high level of support for a new architecture that includes multiple compilers, development environments, and real-time operating system software products.

Specifically, StarCore is developing baseline tools such as a C compiler, assembler, linker, and simulator. These common SC100 baseline tools will be featured in visually integrated development environments (IDEs) that Lucent Technologies and Motorola provide in support of their respective SC100 chip products. The IDEs include real-time source-level debugging and profiling tools.

1.1 Overview of the SC100 C Compiler

A key feature of the SC100 C compiler is its ability to generate code that is exceptionally compact, approaching the code density of the best RISC microprocessors while demonstrating high performance that is comparable to assembly code running on other DSPs. To achieve such a high performance, the compiler optimizes code for maximum parallelism in order to take full advantage of the core's multiple execution units.

In addition to its extensive optimization capabilities, the compiler offers a host of other features that make it ideal for DSP software development, including:

- Conformance to the ANSI C standard
- Intrinsic function support for ITU/ETSI primitives: saturating, non-saturating, and double-precision arithmetic
- Runtime libraries and environments
- Easy integration of assembly code into C code

1.2 The Cross-File Optimization Approach

The SC100 optimizer converts preprocessed source files into assembly output code, applying a range of code transformations that can significantly improve the efficiency of the executable program. The goal of the optimizer is to improve its performance in terms of execution time and/or code size by producing output code that is functionally equivalent to the original source code.

The method that traditional compilers use is to optimize each source file individually before compiling the optimized code and submitting all the compiled files to the linker. Because all the necessary information is not available when files are optimized individually, the compiler must make various assumptions, and is unable to produce the most efficient result.

To ensure optimal performance, the optimizer can take advantage of visibility of as much of the application as possible. The SC100 global binder links all modules into a single module on which all optimizations are performed. As a result of this approach, the performance of the optimizer is substantially improved, and the generated code is typically more efficient than if produced without cross-file optimization.

Chapter 5, “Optimization Techniques and Hints,” describes the optimization modes and functions in detail.

1.3 Compiling Applications

The SC100 compilation process consists of a series of steps, starting from the submission of source files and options to the C Front End (CFE), through the creation of Intermediate Representation (IR) files, the optimization of these files, and the output of optimized assembly code for linking into the final executable program.

You can perform all these processes in one single step, using the compiler shell program.

1.3.1 The Compiler Shell Program

The Compiler's shell provides a one-step command-line interface, in which you specify the files to be processed for each compilation. At each stage, a different tool accepts the input files according to their file extensions, processes them, and outputs the transformed code for processing by the next development tool.

By default, the compiler automatically progresses the input files through all the processing phases. The shell command line lets you select the exact development tools and processing stages that you require, and enables you to define any specific processing options, settings and default overrides that you need.

The options that you specify in the command line control the operation of the shell and of the tools used in the application development process. These options either affect the behavior of the shell itself or the compiler dispatches the options to the different programs, which the shell invokes.

The shell accepts a wide range of option types, including those which perform specific actions, such as:

- generating a list of included files,
- dictating how to treat a source file, and
- controlling specific aspects of the C language features.

When you invoke the shell, the application development process automatically implements through all its various stages to the final production of the executable program.

Chapter 3, "Using the SC100 C Compiler," provides a full description of the shell program and options.

1.3.2 Stages in the C Compilation Process

Following is an outline of the steps involved in compiling C source files into an executable program. These stages are illustrated in the flow diagram shown in Figure 1-1 on page 1-5:

1. The shell is invoked with the list of the C source files and assembly files to be processed, and the various options to be applied.
2. The C Front End (CFE) identifies each C source file by its file extension, preprocesses the source files, converts the files into Intermediate Representation (IR) files, and passes these to the optimizer.
3. The high-level phase of the optimizer translates each intermediate representation file into an assembly ASCII file, and performs a number of target-independent optimizations. The optimizer extracts library files that were created in IR form, and included at this stage of processing. The optimization process also includes any relevant information contained in the application and machine configuration files.
4. The low-level phase of the optimizer carries out target-specific optimizations, and transforms the linear assembly code output by the previous phase into parallel assembly code.
5. At the end of the optimization, the compiler outputs the optimized assembly files to the assembler, joined together with any specified external assembly files.
6. The compiler outputs the assembly files to the linker. The linker combines the assembly object files, extracts any required object modules from the library, and produces the executable application.

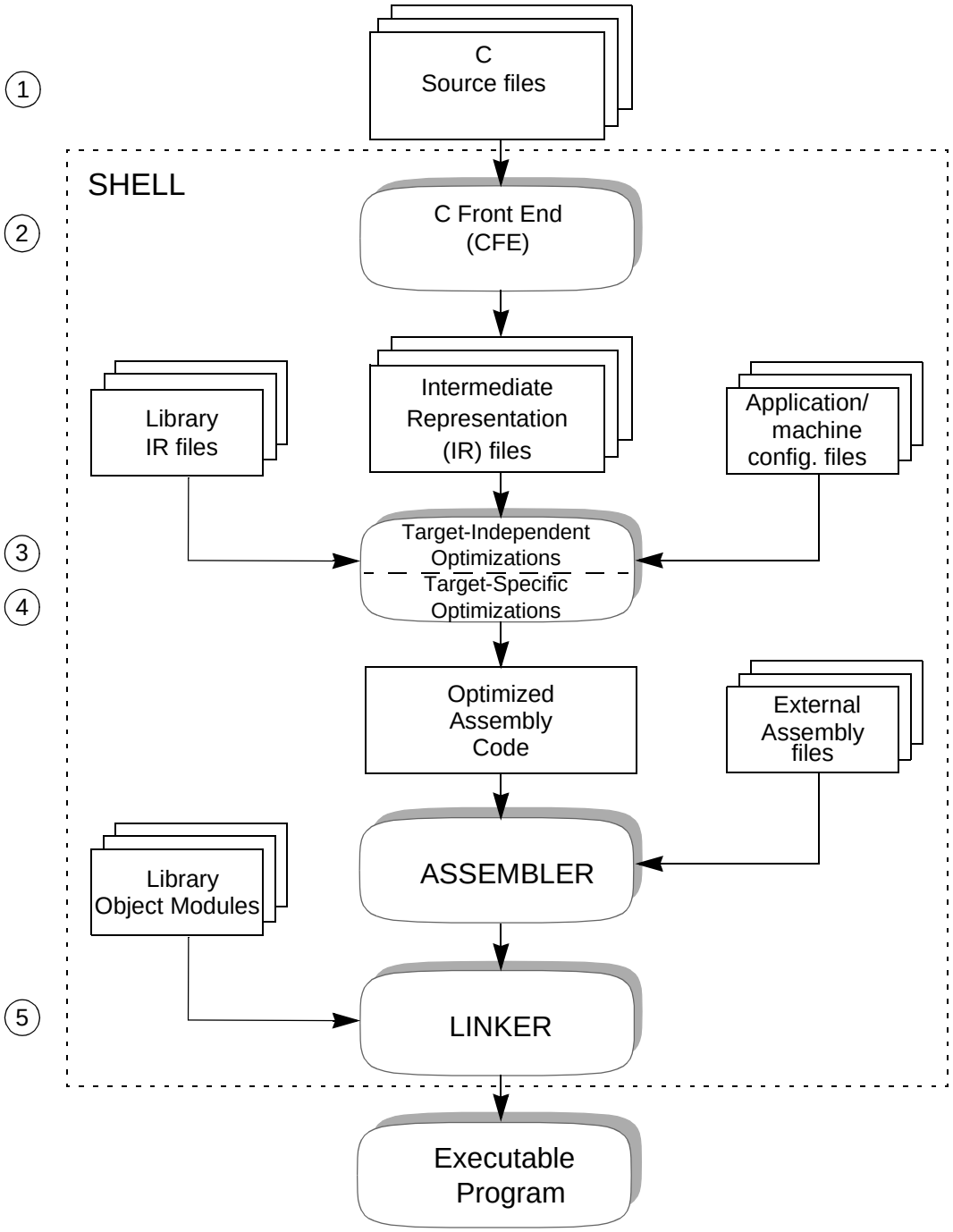


Figure 1-1. The SC100 C Compilation Process

Chapter 2
Getting Started

This example walks you through building and running a simple program using the SC100 C compiler.

2.1 A Quick Start

Creating and executing a program includes the following three phases:

1. Writing the C source code, using the utility of your choice. In this example we will use a sample C source code file provided with your installation.
2. Compiling and linking the file, using the compiler shell.
3. Running the executable application you created.

2.1.1 Creating and Executing a Program

To create and execute a program:

1. Locate the file `hello.c` in the `$SCTOOLS_HOME/src/appnotes` directory, where `$SCTOOLS_HOME` is your installation directory. Copy the `hello.c` file into your working directory.

   Example 2-1 shows the C source code contained in the `hello.c` file:

   Example 2-1. Sample source file: hello.c

   ```
   #include <stdio.h>

   void main()
   {
    printf("Hello there!\n");
   }
   ```

2. Type the following command, which instructs the shell program to compile and link the program:

   ```
   scc -o hello.eld hello.c
   ```

3. Run the executable program, by typing:

   ```
   sc100-sim -quiet -exec hello.eld
   ```

   RESULT: The message `Hello there!` is displayed.

Congratulations! You successfully compiled, linked, and executed a program using the SC100 C compiler.

Chapter 3, “Using the SC100 C Compiler,” describes the shell program and the various file types in detail, and explains how to use the many options and language features that the compiler supports.

Chapter 3
Using the SC100 C Compiler

This chapter contains sections that explain how to use the SC100 C compiler, and describes the options and features that the compiler supports. The sections are:

- Section 3.1, “The Shell Program,” provides an overview of the compiler shell program, outlines the application development process and optimization modes, and lists the file types and environment variables that the shell recognizes.
- Section 3.2, “Invoking the Shell,” explains how to initiate execution of the shell.
- Section 3.3, “Shell Control Options,” describes the options for controlling the operation of the shell and the development tools.
- Section 3.4, “Language Features,” describes the supported extensions and modes, data types, intrinsic functions, pragmas, and predefined macros.

3.1 The Shell Program

The shell program controls the processing of C source files and other files into an executable application, through the preprocessing, compilation, optimization, assembly and linking stages.

The shell provides a one-step command line interface, where you specify the files that you want processed for each compilation. At each stage a different tool accepts the input files according to their file extensions, processes them, and outputs the transformed code for processing by the next development tool.

By default, the input files automatically progress through all the processing phases. The command line enables you to select the exact development tools and processing stages that you require. You can also define any specific processing options, settings and default overrides that you need.

Section 3.3, “Shell Control Options,” describes in detail the options that you can include in the command line to change the behavior of the shell and the specific processing tools at the relevant stages of the development process.

3.1.1 The C Compilation Process

The following is an outline of the steps involved in compiling C source files into an executable program, as illustrated in the flow diagram shown in Figure 3-1 on page 3-3.

1. The shell is invoked with the list of the C source files and assembly files to be processed, and the various options to be applied.
2. The C Front End (CFE) identifies each C source file by its file extension, preprocesses the source files, converts the files into Intermediate Representation (IR) files, and passes these to the optimizer.
3. The high-level phase of the optimizer translates each intermediate representation file into an assembly ASCII file, and performs a number of target-independent optimizations. The optimization process also includes any relevant information contained in the application and machine configuration files.
4. The low-level phase carries out target-specific optimizations, and transforms the linear assembly code output by the previous phase into parallel assembly code.
5. At the end of the optimization, the optimized assembly files are output to the assembler, assembled together with any specified external assembly files, and from there output to the linker. The linker combines the assembly object files, together with any specified external assembly files, extracts any required object modules from the library, and produces the executable application.

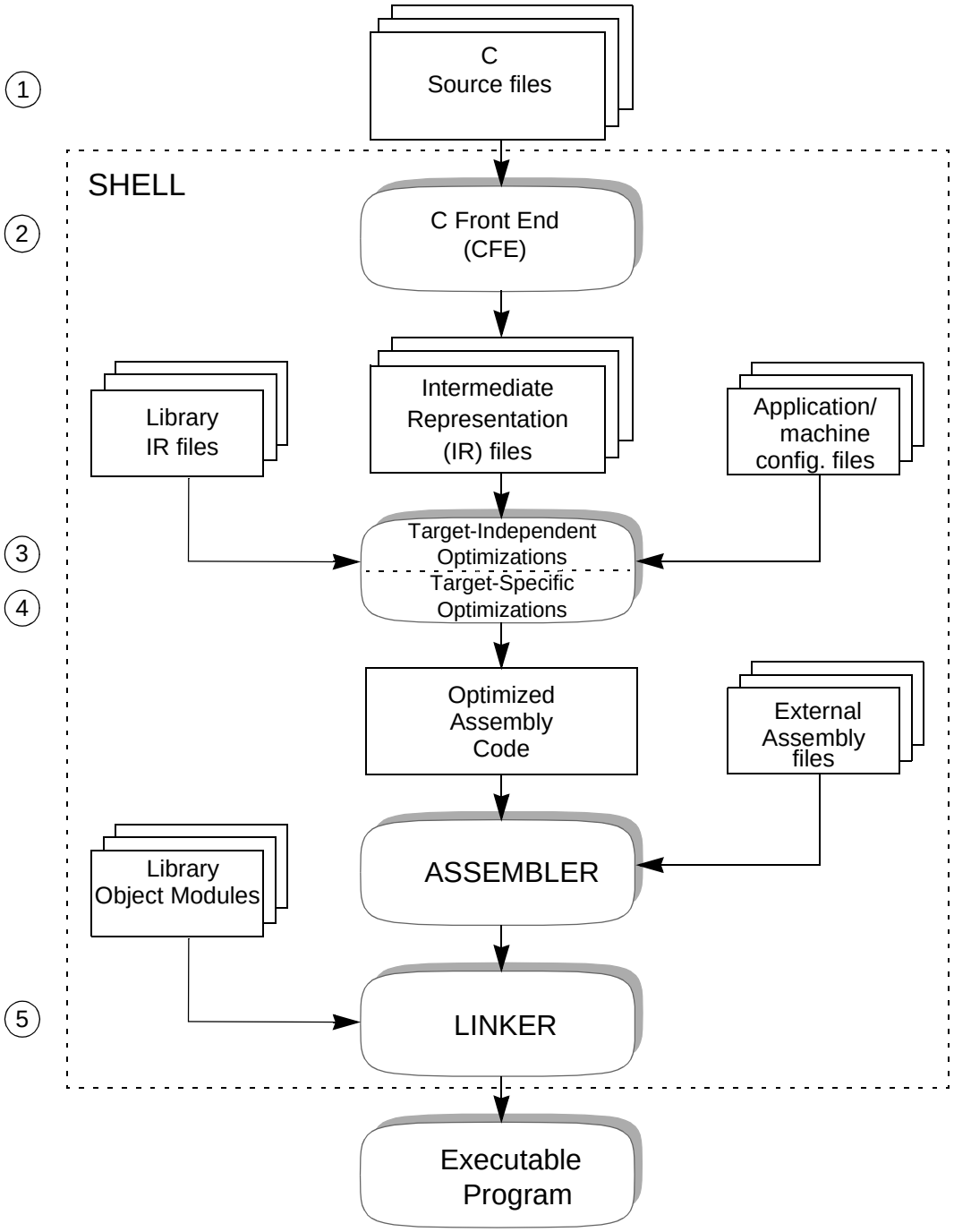


Figure 3-1. The C Compilation Process

3.1.2 Cross-File Optimization

The SC100 optimizer converts preprocessed source files into assembly output code, applying a range of code transformations that can significantly improve the efficiency of the executable program. The goal of the optimizer is to improve its performance in terms of execution time and/or code size by producing output code that is functionally equivalent to the original source code.

The method that traditional compilers use is to optimize each source file individually, before compiling the optimized code, and then submit all the compiled files to the linker. Because not all the necessary information is available when files are optimized individually, the compiler must make various assumptions, and is unable to produce the most efficient result.

To ensure optimal performance, the optimizer takes advantage of visibility of as much of the application as possible. The SC100 global binder links all modules into a single module on which all optimizations can be performed. As a result of this cross-file approach, the performance of the optimizer substantially improves, and the generated code is typically more efficient than if produced without cross-file optimization.

Section 5.3.5, “Cross-File Optimizations,” on page 5-35, provides detailed information and rules for utilizing cross-file optimization.

Figure 3-2 on page 3-5, and Figure 3-3 on page 3-6, illustrate the different processing routes for traditional and cross-file optimization, respectively.

Traditional optimization provides faster compilation, but produces less optimized code. This can be useful during the early stages of development, when you may need to compile different parts of the application separately.

Cross-file optimization produces more efficient code, but the optimization process itself is slower than traditional optimization.

By default, the shell compiles source files without cross-file optimization, for development purposes. You can choose to specify cross-file optimization when you invoke the shell.

Chapter 5, “Optimization Techniques and Hints,” describes the optimization modes and functions in detail.

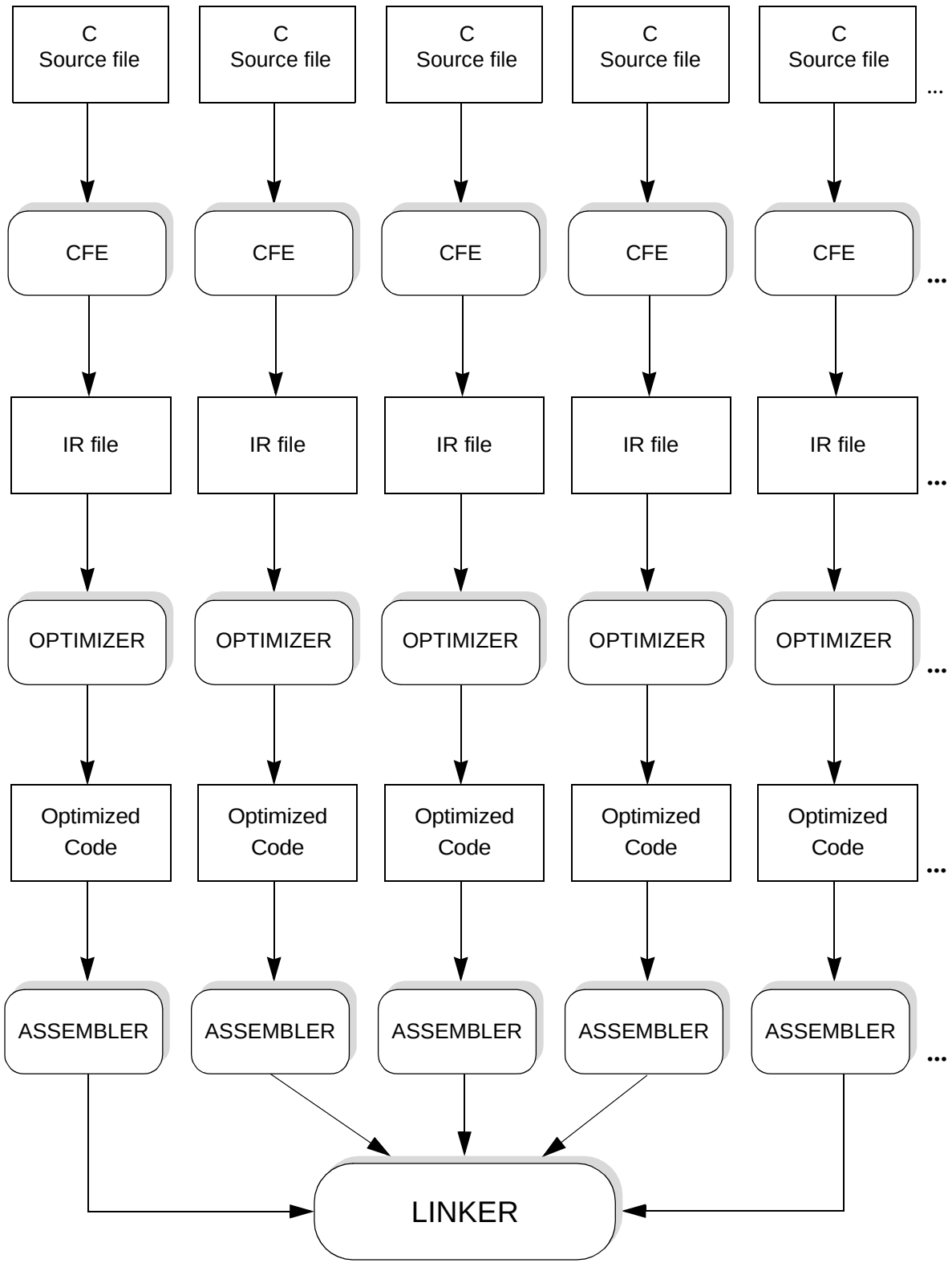


Figure 3-2. Traditional Optimization

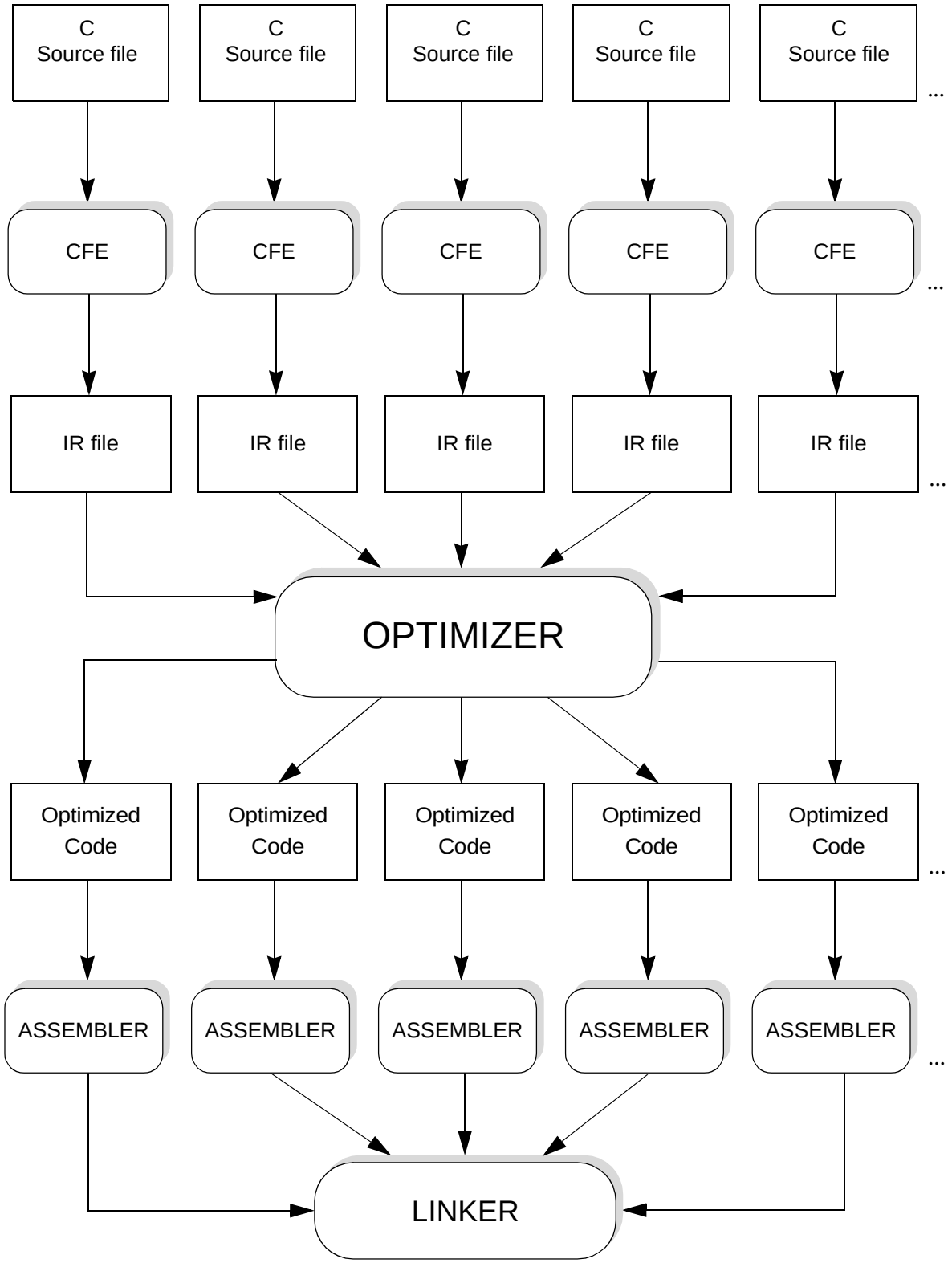


Figure 3-3. Cross-File Optimization

3.1.3 File Types and Extensions

The shell program assumes that all items included in the command line that are not recognizable as options or option arguments are input file names. The extension for each file identifies the file type, and determines at which stage the shell will start processing the file. If none of the tools recognize the file extension, the shell treats the file as an input file to the linker.

The following table lists the file extensions and their corresponding file types, and shows which tool processes each file type.

Table 3-1. File Types and Extensions

Extension	File	Tool
.c	C source file	C Preprocessor
.h	C header file	
.i	Preprocessed C source	Front End
.obj	IR language file	Optimizer
.lib	IR library	Optimizer
.asm, .sl	Assembly file	Assembler
.eln	Relocatable ELF object file	Linker
.elb	ELF library file	Linker
.cmd	Linker command file	Linker

Note: It is possible to cause the shell to process a file as if it were a different file type, as described in Section 3.3.3, “Overriding Input File Extensions.”

The end result of the compilation process is an executable object file, with a file extension of `.eld`. Figure 3-4 on page 3-8 illustrates the assignment of file extensions at each stage of the shell processing cycle.

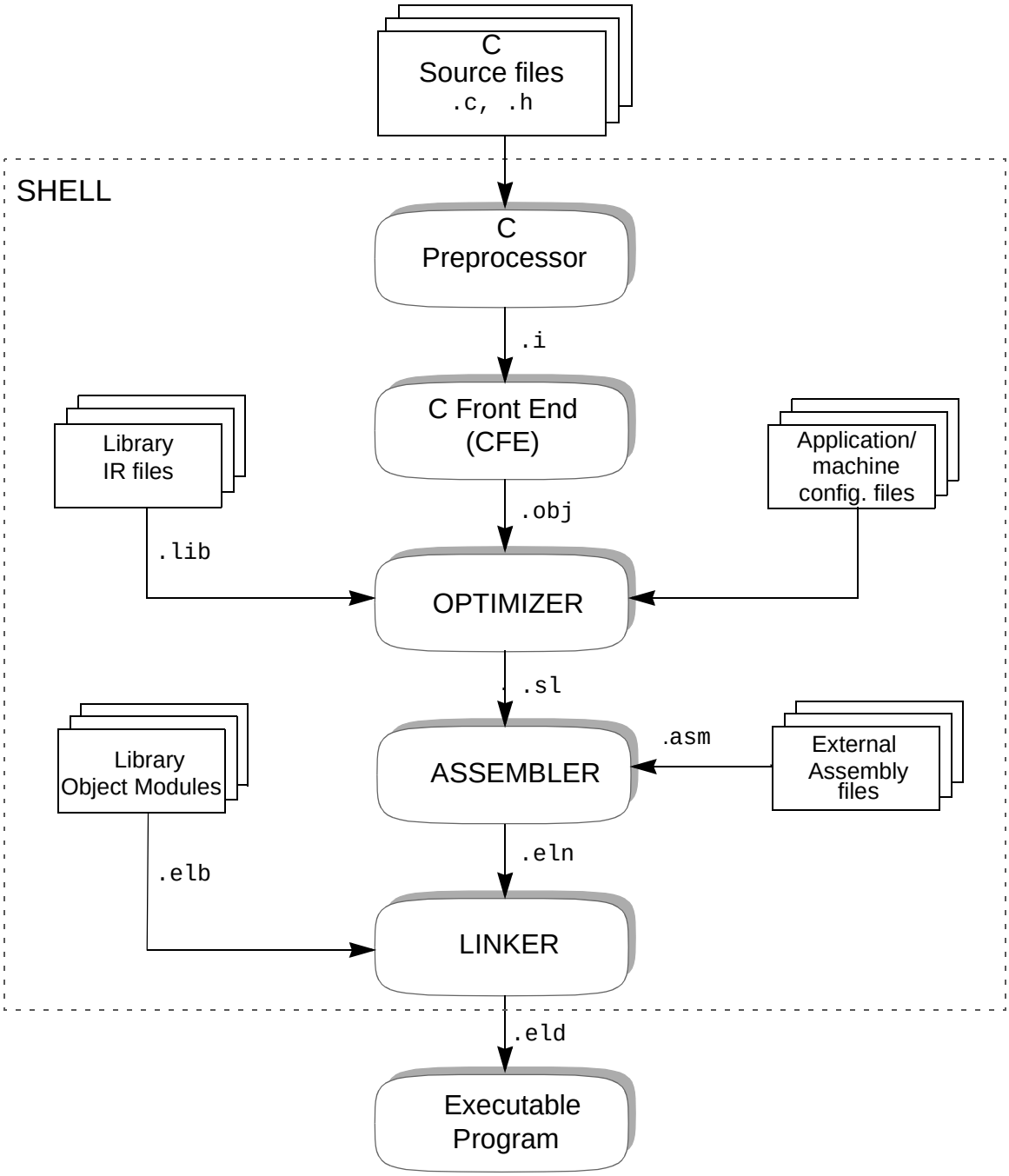


Figure 3-4. File Extensions in the Shell Cycle

3.1.4 Environment Variables

Each time the shell executes, it refers to certain environment variables that determine specific aspects of its behavior. These environment variables are defined during installation, and include the following:

`$SCTOOLS_HOME` — Defines the root directory where the executables, libraries, and tools are stored. This is set to the default location at installation. The compiler searches this directory for all the configuration and executable files that it requires.

3.2 Invoking the Shell

Invoke the shell using a single command line, entered at a UNIX® or MS-DOS® prompt. This command line consists of the shell invocation command, one or more file names, and optionally, one or more shell options.

3.2.1 Command Line Syntax

The syntax of the shell command line is as follows:

`scc` *`[option...] file...`*

The three components of the command line are:

`scc` — Formerly `ccsc100`. Invokes the compiler shell.

`option` — One or more options that control the way in which the shell and the various development tools operate. Section 3.3, “Shell Control Options,” describes the available options and their effect on the shell and the specified files. It is not mandatory to specify options in the command line.

`file` — The names (including extensions) of one or more files that you want the shell to process. These can be source, object, library, and/or command files.

3.2.1.1 Command Line Syntax Rules

The following syntax rules apply:

- The command line must consist of only one line.
- You can include individual options and files in the command line in any order, but you must separate them from each other using at least one space.
- Do not combine options. Specify the options individually.
- Options which specify an argument, such as a file name or directory name, must be followed immediately by their argument(s), separated by at least one space.
- All file names, options, and arguments are case sensitive. File names may be any combination of alphanumeric characters.
- You can not start a file name with a numeric character. Numbers can appear within the filename, but not at the beginning of the file name.
- The underscore (_) is the only special character that the compiler accepts.

The shell command line shown in Example 3-1 specifies three C source files and the option `-c`, which instructs the shell to compile and assemble these files.

Example 3-1. Invoking the shell

```
scc -c one.c two.c three.c
```

3.2.1.2 Command Files

You can include one or more shell command files on the command line. Command files are files that you can create containing any number of options and arguments, which the shell uses as if the files are part of the command line. Section 3.3.1.2, "Specifying a shell command file," describes the use of shell command files in greater detail.

3.3 Shell Control Options

The options specified in the command line and command files control the operation of the shell, and control the operation of the tools used in the application development process.

The following categories of options are provided:

- Options that control the behavior of the shell
- Preprocessing options
- Options that override the file extension for input files
- Output filename and location options
- C language options
- Optimization pragma and code options
- Options that control the output of listing files and messages
- Pass-through options
- Hardware model and configuration options

Table 3-2 on page 3-11 provides a summary of the available options. This table is followed by detailed descriptions of each of the options, with the exception of the options relating to optimization, which are described in detail in Chapter 5, "Optimization Techniques and Hints."

Table 3-2. Shell Options Summary

Options that control the behavior of the shell			
Option	**Effect**	**Section**	**Page**
`-E [file]`	Stops after preprocessing source files. Removes comments.	3.3.1.1	3-14
`-cfe`	Stops after Front End. Does not invoke the optimizer. Enables the creation of libraries of object files for use with cross-file optimization.	3.3.1.1	3-14
`-S`	Stops after compilation. Does not invoke the assembler.	3.3.1.1	3-14
`-c`	Compiles and assembles only. Does not invoke the linker.	3.3.1.1	3-14
`-F file`	Reads options from the specified file, and appends to command line.	3.3.1.2	3-15
`-h` or none	Displays the shell Help page, listing all available options.	3.3.1.3	3-15
Preprocessing Options			
Option	**Effect**	**Section**	**Page**
`-C`	Preserves comments in the preprocessing output.	3.3.2.1	3-16
`-M file`	Generates a MAKE file showing dependencies.	3.3.2.1	3-16
`-MH file`	Generates a list of `#include` files.	3.3.2.1	3-16
`-D mac [=def]`	Defines preprocessor macro.	3.3.2.2	3-17
`-U macro`	Undefines preprocessor macro.	3.3.2.2	3-17
`-I dir`	Adds directories to the `#include` file search path.	3.3.2.3	3-17
	Syntax note:The options `-D`, `-U`, and `-I` do not require a space before the argument.		
Options that override the file extension for input files			
Option	**Effect**	**Section**	**Page**
`-xc file [file2 ...]`	Treats specified file(s) as C source file(s) (`.c`).	3.3.3	3-18
`-xobj file [file2 ...]`	Treats specified file(s) as IR language file(s) (`.obj`).	3.3.3	3-18
`-xasm file [file2 ...]`	Treats specified file(s) as assembler source file(s) (`.asm` or `.sl`).	3.3.3	3-18
Output filename and location options			
Option	**Effect**	**Section**	**Page**
`-o file`	Assigns a filename (and extension) to the output file.	3.3.4	3-19
`-r dir`	Redirects all output to the specified directory.	3.3.4	3-19

Table 3-2. Shell Options Summary (Continued)

C Language Options

Option	Effect	Section	Page
`-ansi`	Strict ANSI mode. Assumes all C source files contain ANSI/ISO versions of the language, with no extensions. The default mode is the ANSI/ISO version with extensions.	3.3.5.1	3-20
`-kr`	K&R/pcc mode. Assumes all C source files contain K&R/pcc versions of the language. The default mode is the ANSI/ISO version with extensions.	3.3.5.1	3-20
`-g`	Adds debug information to generated files.	3.3.5.2	3-20
`-sc` (Default)	Makes char type variables signed.	3.3.5.3	3-20
`-usc`	Makes char type variables unsigned. The default setting is signed.	3.3.5.3	3-20
`-fractional`	Tells the compiler what to do about saturation. Indicates that your code contains intrinsics.	3.3.5.4	3-21

Optimization Pragma and Code Options

Option	Effect	Section	Page
`-O0`	Disables all optimizations. Outputs unoptimized assembly code.		
`-O1`	Performs all target-independent optimizations, and outputs optimized linear assembly code. Omits all target-specific optimizations.	5.3.2	5-9
`-O2` (Default)	Performs all optimizations, producing the highest performance code possible without cross-file optimization. Outputs optimized non-linear assembly code.	5.3.3	5-20
`-Os`	Performs space optimization for the indicated level of optimization. Outputs optimized assembly code which is small. This option can be specified together with any of the optimization options except -O0.	5.3.4	5-33
`-Og`	Performs cross-file optimization, which applies the indicated level of optimization across all input files at once. The default is non-cross file optimization. This option can be specified together with any of the optimization options except -O0.	5.3.5	5-35
`-no_overflow`	Tells the compiler that the application does not rely on the ANSI/ISO C defined overflow behavior of operations on unsigned integral data-types.		

Pass-through Options

Option	Effect	Section	Page
`-Xasm` *`option`*	Passes *option* to the assembler.	3.3.6	3-21
`-Xlnk` *`option`*	Passes *option* to the linker.	3.3.6	3-21

Options that control the output of listing files and messages

Option	Effect	Section	Page
`-de`	Retains a generated error file for each source file.	3.3.7.1	3-22

Table 3-2. Shell Options Summary (Continued)

`-dm [file]`	Generates a link map file.	3.3.7.1	3-22
`-do`	Adds to the assembly output file the offsets for C data structure field definitions.	3.3.7.1	3-22
`-dL`	Generates a C list file for each source file.	3.3.7.1	3-22
`-dL1`	Generates a C list file for each source file, including a list of `#include` files.	3.3.7.1	3-22
`-dL2`	Generates a C list file for each source file, including expansions.	3.3.7.1	3-22
`-dL3`	Generates a C list file for each source file, including both `#include` files and expansions.	3.3.7.1	3-22
`-dx [file]`	Generates a cross-reference information file.	3.3.7.1	3-22
`-dc [0-4]`	Generates a file showing calls in graphical tree form, in postscript. The number 0 to 4 specifies the paper size, A0 through A4.	3.3.7.1	3-22
`-q` or `-w` (Default)	Quiet mode. Displays errors only.	3.3.7.2	3-23
`-v`	Verbose mode. Displays full information.	3.3.7.2	3-23
`-n`	Displays command lines without executing.	3.3.7.2	3-23
`-Wj`	Suppresses warnings on local automatic variables that are used before their values are set.	3.3.7.3	3-23
`-Wg`	Suppresses cross-file optimization warnings.	3.3.7.3	3-23
`-Wall`	Reports all warnings and remarks.	3.3.7.4	3-23

Hardware Model and Configuration Options

Option	Effect	Section	Page
`-arch target`	Specifies the target architecture. `sc140` is the default architecture.	3.3.8.1	3-24
`-mc file`	Specifies the file to be used as the machine configuration file, if different from the default file defined at installation.	3.3.8.2	3-24
`-ma file`	Specifies the file to be used as the application configuration file, if different from the default file defined at installation.	3.3.8.2	3-24
`-crt file`	Specifies the file to be used as the startup file, if different from the default file defined at installation.	3.3.8.2	3-24
`-mb`	Compiles in big-memory mode. Small-memory mode is the default.	3.3.9	3-25
`-mt`	Compiles in tiny-memory mode.Small-memory mode is the default.	3.3.9.2	3-25
`-mrom`	Copies all initialized variables from ROM at startup.	3.3.9.3	3-25
`-be`	Generates output for a big-endian target configuration. The default is a little-endian configuration.	3.3.9.4	3-25
`-mem file`	Specifies the linker command file to be used, if different from the default file defined at installation.	3.3.8.2	3-24

3.3.1 Controlling the Behavior of the Shell

The options described in this section enable you to control the overall actions of the shell. You can tell the shell program at what stage to stop processing, define files containing command line options, and display the invocation commands.

3.3.1.1 Controlling where the shell stops processing

By default, the shell completes the entire processing cycle, from the input of source files through all of the intermediate stages to the output of the final executable. If you want to stop the processing at a specific stage, you can use one of the options `-E`, `-cfe`, `-S`, or `-c`. In this way, you can process and check individual files or groups of files through different stages, until the files are finally ready to be compiled and linked together.

Following is the process for controlling where the shell stops processing.

1. Select one of the following options:

Option	Description
`-E [`*`file`*`]`	The shell stops after preprocessing the C source files. Include an `.i` extension if you want the file input to the compiler at a later time. If you do not specify a file, the compiler sends the output to the standard output stream, `stdout`. Comments are not preserved in the preprocessing output, unless you specify the option `-C`. See Section 3.3.2.1, "Changing preprocessed output," for details of `-C` and other options that add specific features to the preprocessing output.
`-cfe`	The shell stops after it processes the input source files through the Front End. You can use this option to check that the files are valid source files that meet the essential requirements for processing by the shell, for example, they contain no syntax errors. This is primarily useful when preparing files for cross-file optimization. Output files are IR files, assigned the extension `.obj`. The `-cfe` option enables you to create libraries of object files to use later when compiling in cross-file optimization mode.
`-S`	The shell stops after it compiles the source files to assembly files, and does not invoke the assembler. Output files are assigned the extension `.sl`.
`-c`	The shell stops after compiling C and assembly source files to object code, and does not invoke the linker. The object code output files are assigned the extension `.eln`.

2. Following processing with the `-E`, `-cfe`, `-S`, or `-c` options, the output files are written to the current directory. If you use the `-r` option the output files are written to the specified directory.
3. The compiler assigns the output files the same names as the input files, with the extension for the selected option.
4. The compiler overwrites any existing files in the directory with the same name and extension.

The starting point for the processing of each input file is determined by its file extension. Refer to Table 3-1 on page 3-7 for an explanation of file extensions. See Section 3.3.3, "Overriding Input File Extensions," for a description of the options you can use to override these extensions.

3.3.1.2 Specifying a shell command file

You can create command files containing options and arguments, which the shell program will treat as if they were included on the command line.

Defining options and arguments within command files can save you input time when you invoke the shell program. This also helps you overcome any imposed limitation on the length of the command line. Each time you invoke the shell, you can select the command file with the set of options that suit your specific requirements.

To specify a shell command file, specify the option `-F` followed by a filename. A command file can itself contain the option `-F` specifying another shell command file.

Example 3-2 illustrates the use of the `-F` option to specify the command file `proj.opt`.

Example 3-2. Defining a shell command file

```
scc -F proj.opt
```

Within the command file, each separate option (with or without an argument), file, or list of files must reside on a new line. You can specify as many lines as you wish, in any order. You can include comments in the file using the # character. The shell ignores all characters between # and the end of the line.

The command file shown in Example 3-3 contains four lines that instruct the shell to invoke the linker with three application object files and one library file, generate a link map file, and output the executable program to a file named `appl.eld`.

Example 3-3. Contents of a shell command file

```
-o appl.eld                          # output file name
-dm appl.map                         # generate map file
file1.eln file2.eln file3.eln        # object files
-l mylib.elb                         # shared library
```

Note: If you do not specify a map file, the shell generates a file with the same file name as the specified `.eld` file, and the extension `.map`.

3.3.1.3 Displaying the shell Help page

The shell Help page lists all of the available shell options and arguments. Select the option `-h` to display this list.

Example 3-4 shows a section of the shell Help page:

Example 3-4. Shell Help page (extract)

```
-c     Compile and assemble only. Don’t invoke the linker
-cfe   Stop after Front-End. (Used for cross-file optimization)
-S     Generate assembly output file. Don’t invoke assembler
-E     Preprocess only
-C     Preprocess only and keep comments
```

3.3.2 Specifying Preprocessing Options

The options described in this section enable you to control the preprocessing stage of the shell program, before the input files proceed through the Front End.

Using these preprocessing options, you can:

- change the output that the preprocessor produces,
- define one or more preprocessor macros, and
- define the directories you want searched for `#include` files.

3.3.2.1 Changing preprocessed output

You can specify any of the following options to change the format and content of the preprocessed output. You can specify these options in addition to the `-E` option, or instead of the `-E` option.

`-C`
: Keeps all comments (preprocessor directives) in the preprocessing output. If you specify the `-E` option only, the preprocessed text is written to the output file with line control information only, and with all comments removed.

`-M` *`[file]`*
: Instead of the normal preprocessing output, the compiler generates an output file in `MAKE` format, containing a list that shows the dependencies between the input source files.

  If you do not specify a file, the compiler sends the output to the standard output stream, `stdout`.

`-MH` *`[file]`*
: Instead of the normal preprocessing output, the compiler generates an output file containing a list of all the `#include` files used in the source. This list includes all levels of `#include` files, together with any nested files.

  If you do not specify a file, the compiler sends the output to the standard output stream, `stdout`.

3.3.2.2 Defining and undefining preprocessor macros

You can define one or more preprocessor macros, and you can remove the definition of a macro. Section 3.4.6, "Predefined Macros," provides details of all the predefined macros supplied with the SC100 C compiler.

You can specify the following macro options more than once in the command line, to define and undefine different preprocessor macros:

`-D` *`macro [=value]`*
: Defines the named macro as a preprocessor macro, with the specified value. If *`value`* is omitted, the value 1 (one) is assumed. Once a preprocessor macro is defined with this option, it is passed by the shell to the preprocessor for all subsequent compilations until it is undefined with the `-U` option.

  The space between the `-D` option and the named macro is optional.

`-U` *`macro`*
: Undefines the named macro by removing its previous definition. The macro will not be passed to the preprocessor unless it is redefined with the `-D` option. Any `-U` options in the command line are processed only *after* all `-D` options have been processed.

  It is not necessary to enter a space between the `-U` option and the named macro.

3.3.2.3 Adding directories to the #include file path

The option `-I` *`dir`* adds the specified directory or directories to the path that the shell uses to search for `#include` files. The string *`dir`* can be a list of directories.

To specify directory or directories for the `#include` file search path:

1. Specify the option `-I`.
2. Follow the `-I` option with a directory name or a list of directories.
3. The space between the `-I` option and the *`dir`* string is optional.
4. On UNIX hosts, separate the individual directories in the list with colons (:).
5. On PC hosts, separate the individual directories with semicolons (;).

You can use this option more than once in a command line, and the directories or lists will be searched in the order in which the options are supplied.

3.3.3 Overriding Input File Extensions

You can change how the shell program treats a specific input file, by overriding the assumptions made by the shell based on the file's extension.

You can select any of the following options, as many times as required. After the selected option you can specify one or more filenames, separated by spaces.

`-xc` *`file [file2 ...]`* This option identifies the specified files as C language source files, as if they had the extension `.c`. The shell processes these files in exactly the same way as any other C source files specified in the command line, subject to any other processing options selected.

`-xobj` *`file [file2 ...]`* This option identifies the files as IR language files, as if they were output by the Front End with the extension `.obj`. The compiler inputs the files for processing.

`-xasm` *`file [file2 ...]`* This option instructs the shell program to identify the specified files as assembler source files, as if they had the extension `.asm` or `.sl`. The files are assembled at the appropriate processing stage, and the object code is made available to the linker.

For a list of the default extensions for each file type, see Table 3-1 on page 3-7.

These options can appear any number of times in the command line. Each option relates to one specified file or a list of files. The files that these options identify are processed normally in all other respects, and in the same relative order as other listed files.

In the following example, the input files `file1.ext` and `file2.bar`, specified after the option `-xc`, will be compiled as if they were C source files:

Example 3-5. Overriding file extensions

```
scc -c -xc file1.ext file2.bar
```

3.3.4 Output Filename and Location Options

Output filename and location options allow you to specify the name and/or directory for the output files that the shell program produces. By default, the compiler assigns each output file the same name as the input file and is stored in the current directory.

The stage at which the shell stops processing determines the default file type and extension for the output files. For example, when you select the `-cfe` option, the output files that the Front End produces have the extension `.obj`. If you wish, you can specify a different extension when you specify the file name. This alters the way the shell treats this file. For more information about overriding file extensions, refer to Section 3.3.3, "Overriding Input File Extensions."

You can select either or both of the following options.

`-o` *`file`*	The output file is assigned the specified filename, and optionally the specified extension. Any existing file with the same name in the current directory, or in the specified directory, if the `-r` option is selected, is overwritten. You can specify this option more than once in the command line, for different files.
`-r` *`dir`*	All output files are redirected to the specified directory. This option can be specified only once in the command line.

In Example 3-6, the input file `file1.foo` will be treated as an input file to the linker (the default).

Example 3-6. Specifying output files

```
scc -o file.eld file1.foo
```

3.3.5 Specifying C Language Options

You can use the C language options described in this section to:

- inform the shell of the language version used in the source files,
- add debugging information to generated files, and
- to define whether variables of type `char` default to signed or unsigned.

3.3.5.1 Defining the language version

The default C language mode is the normal ANSI/ISO version with extensions, with all source files using the standard `.c` extension. You do not need to specify any language option if you use this mode. However, if you use a different language version, you must select either the `-ansi` or the `-kr` option.

Language Version	Option to Select	Assumptions
Strict ANSI/ISO version of C	`-ansi` option	Front End assumes that all input source files are to be in the strict ANSI/ISO version of C with no extensions. The compiler flags any extensions that it finds with warnings.
K&R (Portable C Compiler, or PCC) dialect of C	`-kr` option	Shell program assumes that all source files are in this version of C.

See Section 3.4, "Language Features," for details of the C language features supported in the default, strict ANSI and K&R modes.

You cannot compile source files in different C language versions simultaneously. If you need to compile source files in different versions, you must use a separate shell command line for each version.

3.3.5.2 Adding debugging information to files

The option `-g` tells the shell program to include debugging information in the output files produced by all C compilations. The produced object files are somewhat larger as they will contain source-level debugging information.

The `-O0` option disables optimization. When you debug, we recommend combining the `-g` option and the `-O0` option, so that optimization is disabled while you debug. You can not use any other optimization level with the `-g` option. If you specify an optimization level other than `-O0` in combination with `-g`, the compiler issues the following warning message: "`Illegal combination of options: -g cannot be used with any code optimization`." The SCC shell exits after displaying this message.

3.3.5.3 Changing the default char sign setting

Signed is the default setting for all `char` type variables.

- To make all `char` type variables default to unsigned use the `-usc` option.
- To change the setting back to make all `char` type variables default to signed, specify the `-sc` option.

3.3.5.4 Indicating fractional data-types in saturation

The `-fractional` option tells the compiler that the application you are compiling operates on fractional data-types with the expected ITU saturation behavior. This is not the compilation default. You must specify this switch to use the ITU fractional intrinsics. `-fractional` implies that your code contains intrinsics.

You must intricately know your code when using the `-fractional` option. To obtain correct results, type `-fractional` if your code contains intrinsics. If your code contains intrinsics and you do not type `-fractional`, you will receive erroneous results.

3.3.6 Passing Options Through to Specific Tools

The options described in this section enable you to instruct the shell program to pass options to specific tools, such as the assembler or linker, as shown in Example 3-7.

Example 3-7. Passing multiple options to the same tool

```
scc -Xasm -occ
```

You can instruct the compiler to pass multiple options to the same tool in the same option statement, along with the arguments for each option. You must list multiple options and their arguments, where relevant, within quotation marks.

When invoking a tool several times, the compiler passes the pass-through options on each invocation. It then continues to pass any other options that the shell program passes directly to the tool from the command line.

Specify either of the following options:

`-Xasm` *`option`*	Passes the specified option(s) and arguments to the assembler.
`-Xlnk` *`option`*	Passes the specified option(s) and arguments to the linker.

Note: Use the `-mem` option to pass a command file other than the default to the linker. If you use the `-Xlnk` option to do this, both the command file you are specifying and the default command file are passed to the linker, resulting in errors.

3.3.7 Setting the Options for Listings and Messages

The options in this section enable you to control the retention, display, and printing of diagnostic and informational messages, and the generation of various listing and map files.

3.3.7.1 Generating listing files

By default the shell program does not retain the diagnostic and cross-reference information produced at different processing stages. You can select to retain one or more different types of information in listing files.

Use any combination of the following options to generate listing files that contain the types of information you require. You can specify each individual option only once in a shell command line.

`-de` The Front End creates a file containing all error messages generated during the compilation. The `-de` option retains this error file. If you do not specify this option, the errors display during processing, but are not kept. The compiler creates an error file for each source file, with the same name as the source file and the extension `.err`.

`-dm` *`[file]`* Generates a link map file listing all the specific variables, applications and addresses that the linker uses. If you do not specify a file name, the compiler creates a file with the same name as the executable, and the extension `.map`.

`-do` Includes the details of C data structures in the output assembly file, showing the offsets for all field definitions in each data structure. Refer to Chapter 4, “Interfacing C and Assembly Code,” for a detailed description of the C/assembler interface.

`-dL` Generates a C list file for each source file, listing the entire contents of the source file. The compiler creates each list file with the same name as its corresponding source file, and the extension `.lis`.

`-dL1` Generates a C list file for each source file, listing the entire contents of the source file, with the addition of a list of `#include` files that the source uses. The compiler creates each list file with the same name as its corresponding source file, and the extension `.lis`.

`-dL2` Generates a C list file for each source file, listing the entire contents of the source file, with the addition of expansions, such as macro expansions, line splices, and trigraphs. The compiler creates each list file with the same name as its corresponding source file, and the extension `.lis`.

`-dL3` Generates a C list file for each source file, listing the entire contents of the source file, with the addition of a list of `#include` files, and expansions, such as macro expansions, line splices, and trigraphs. The compiler creates each list file with the same name as its corresponding source file, and the extension `.lis`.

`-dx` *`[file]`* Generates a cross-reference information file, providing details of cross-references in the source file. If you do not specify a file name, the compiler creates a file with the same name as the source file, and the extension `.xrf`.

`-dc [0-4]` Generates a file showing calls in graphical tree form, which you can print using a postscript printer. Specify the size of the paper to use for the printout: 0 for paper size A0, 1 for A1, and so on.

3.3.7.2 Controlling the type of information displayed

You can control the level and type of messages and information that the shell program displays using the following options:

`-q` or `-w` Quiet mode (the default). The shell program displays the minimum amount of information (errors only), omitting normal notices and banners. This option is useful when running the shell in batch mode or with the `MAKE` utility, when the display of normal progress information is not required.

`-v` Verbose mode. The shell program displays/prints all the commands and command line options used, as it proceeds through the different processing stages and invokes the individual tools. The exact information output depends on the processing stages that the shell performs.

`-n` Displays the specified shell processing actions without executing them. You can use this option before you invoke the shell and to check the actions the shell will take, based on the list of files and arguments specified in the command line.

3.3.7.3 Suppressing warnings

By default the shell reports all errors and warnings. You can suppress specific types of warnings using the `-Wj` and `-Wg` options, which reduce the number of messages that the shell program generates. This is useful if, for example, you are testing incomplete sections of the program and you know in advance that certain warnings will be produced.

You can select either or both of the following options:

`-Wj` This option suppresses warnings on local automatic variables that are used before their values are set. If you are testing partial code, which you know does not assign values to all the local automatic variables, you can use this option to suppress all the "false" warnings that would otherwise be issued.

`-Wg` By default, the compiler produces a warning for each module identified as missing during the cross-file optimization process. Use this option if you wish to suppress these warnings, for example when testing an incomplete application, or one that uses external modules.

3.3.7.4 Reporting all remarks and warnings

By default, the shell reports all errors and warnings, but does not report remarks unless you specifically instruct it to do so. Select the option `-Wall` to ensure that all remarks are reported, as well as all warnings and errors.

3.3.8 Specifying the Hardware Model and Configuration

The options in this category let you override various default hardware and configuration settings.

3.3.8.1 Defining the architecture

The default architecture is SC140, which utilizes four MAC units. Unless instructed otherwise, the compiler assumes, during the optimization phase, that four execution units are in use, and parallelizes the code accordingly.

If you are compiling for a hardware configuration other than SC140, it is essential that you specify the correct architecture. To change the assumed architecture, specify the `-arch` *`target`* option, as illustrated in Example 3-8.

Valid values for *`target`* are `sc110` and `sc140` (default).

Example 3-8. Defining the architecture

```
scc -arch sc110 file1.c
```

3.3.8.2 Configuration and Startup files

The default machine and application configuration files that the compiler uses, and the startup file that the linker uses, are defined during the installation process. The following table provides brief descriptions of the systems configuration and startup files. These files, and their use in the run-time environment, are described in greater detail in Chapter 6, “Runtime Environment.”

File	Description
Machine configuration file	includes information about logical and physical memory maps. This information enables the global optimizer to dispatch variables to different memory areas in internal ROM or RAM.
Application configuration file	contains information about how the application software and the hardware interact. The file includes sections about binding interrupt handlers, overlays, and application objects to specific addresses.
Startup file	the linker uses the startup files when it links the assembly code files with the standard libraries, and defines such items as the interrupt vector and set up code executed upon system initialization.

3.3.8.2.1 Defining specific configuration and startup files

You may wish to select other files to be used for configuration setup and initialization instead of the default files, for example, to specify certain devices that need initializing at startup.

To specify different files for use at initialization, select one or all of the following options. For each option, specify the file name, and if the file is not in the current directory, specify the path.

`-mc` *`file`* The compiler reads the specified file instead of the default machine configuration file.

`-ma` *`file`* The compiler reads the specified file instead of the default application configuration file.

`-crt` *`file`*	The linker links into the application of the specified file instead of the default startup file.
`-mem` *`file`*	The linker uses the specified command file instead of the default linker command file (`crtscsmm.cmd` or `crtscbmm.cmd`).

For more detailed information, refer to Chapter 6, “Runtime Environment.”

3.3.9 Specifying modes

The SC100 architecture instruction set supports 15-bit, 16-bit, and 32-bit addresses. If the application is small enough to allow all static data to fit into the lower 64K of the address space, then the compiler generates more efficient code.

Small memory mode is the default, and assumes that all addresses are 16-bit. The compiler uses small memory mode unless you specify big memory mode or tiny memory mode.

3.3.9.1 Specifying big memory mode

If your application does not fit into 64K bytes, meaning that the use of 32-bit absolute addresses is required, you must instruct the shell to use the big memory model, by specifying the `-mb` option.

3.3.9.2 Specifying tiny memory mode

If your application can fit in less than 32K bytes, use the tiny memory mode, which is the most efficient for code size and performance. Instruct the shell to use the tiny memory model, by specifying the `-mt` option. The .data section is loaded into the lowest address in memory and references to the .data section use code sequences with smaller encodings.

3.3.9.3 Copying initialized variables from ROM

During development you would normally use a loader to set the values for global variables, and to load these initialized variables into RAM at startup, together with the executable application.

When you finish development, if your final application does not use a loader, you must ensure that when the completed application executes, the initialized variables are copied from ROM into RAM. To do this, when you compile the final application version, specify the `-mrom` option.

Refer to Chapter 6, “Runtime Environment,” for more detailed information about the initialization of variables in the runtime environment.

3.3.9.4 Specifying big-endian mode

By default, the compiler generates code based on the assumption that the architecture operates in little-endian mode, meaning the least significant bits in the lower address. If you want to run the application in an environment that operates in big-endian mode, meaning the most significant bits in the lower address, specify the option `-be`.

3.4 Language Features

This section describes the different language modes that the SC100 C compiler accepts. It also provides detailed information about the data types and sizes supported, fractional arithmetic representation, intrinsic functions, pragmas, and predefined macros.

3.4.1 C Language Dialects

The compiler accepts three variations of the C language.:

- Normal ANSI/ISO version with extensions — This is the default mode. See Section 3.4.1.1, "Standard Extensions." for more details.
- Strict ANSI/ISO mode: — Specified with the shell option `-ansi`. Any ISO C extensions are flagged with warnings.
- K&R/PCC mode: — Specified with the shell option `-kr`. The compiler accepts the older K&R dialect of C, and provides almost complete compatibility with the widely used UNIX PCC (`pcc`) dialect. See Section 3.4.1.2, "K&R/PCC mode," for details.

You can not compile source files of different C language types together; however, once compiled you can link them together into a single application.

3.4.1.1 Standard Extensions

This section lists the extensions that standard C programs normally accept. When compiling in strict ANSI/ISO mode, the compiler issues warnings when it detects these extensions.

3.4.1.1.1 Preprocessor extensions

The compiler accepts the following preprocessor extensions:

- Comment text can appear at the end of preprocessing directives.
- The compiler scans numbers according to the syntax for numbers. Thus, `0x123e+1` is scanned as three tokens instead of one invalid token.
- The compiler allows the `#assert` preprocessing extensions of AT&T System V release 4. These enable the definition and testing of predicate names. Such names are in a name space distinct from all other names, including macro names. You can define a predicate name using a preprocessing directive in one of two forms, as shown in Example 3-9:

Example 3-9. Defining a predicate name

```
#assert name
#assert name(token-sequence)
```

In the first form, the predicate is not given a value. In the second form, it is given the value *token-sequence*. Such a predicate can be tested in a `#if` expression, as follows: `#name(token-sequence)`. This expression has the value 1 if a `#assert` of that *name* with that *token-sequence* has appeared, otherwise it has the value 0. You can assign a predicate more than one value at a given time.

- A predicate may be deleted by a preprocessing directive in one of two forms, as shown in Example 3-10:

  Example 3-10. Deleting a predicate

  ```
  #unassert name
  #unassert name(token-sequence)
  ```

  The first form removes all definitions of the indicated predicate name. The second form removes only the indicated definition, leaving any remaining definitions unchanged.

A number of predefined preprocessor macros are provided, as described in Section 3.4.6, “Predefined Macros.”

The pragmas described in Section 3.4.5, “Pragmas,” are available in all modes.

3.4.1.1.2 Syntax

The compiler accepts:

- An empty translation unit (input file), containing no declarations.
- An extra comma at the end of an `enum` list. Similarly, you can omit the final semicolon preceding the closing `}` of a struct or union specifier. The compiler issues a remark in both cases, except in `pcc` mode.
- A label definition followed immediately by a right brace. (Normally, a label definition must be followed by a statement.) The compiler issues a warning.
- An empty declaration (a semicolon with nothing before it). The compiler issues a remark.
- An initializer expression that is a single value and used to initialize an entire static array, `struct`, or `union` not enclosed in braces, except in strict ANSI C mode.
- By default, the compiler accepts a `struct` with no named fields, but that has at least one unnamed field. The compiler issues a diagnostic warning or error in strict ANSI C mode.

3.4.1.1.3 Declarations

The compiler accepts the following declaration extensions:

- Static functions declared in function and block scopes. The compiler moves their declarations to the file scope.
- The compiler allows benign redeclarations of `typedef` names, meaning that you can re declare a `typedef` name in the same scope as the same type. The compiler issues a warning.
- The compiler always accepts `asm` statements and declarations, with one exception, which is when compiling in strict ANSI C mode. The reason for this is that there is a conflict with the ANSI C standard. For example, the Front End interprets `asm("xyz");` as an `asm` statement by default, while ANSI C interprets this as a call of an implicitly-defined function `asm`.
- The compiler accepts functions declared as `asm` functions, and recognizes`__asm` as a synonym for `asm`. An `asm` function body is represented by an uninterpreted null-terminated string containing the text that appears in the source.

- An `asm` function must be declared with no storage class, with a prototyped parameter list, and with no omitted parameters, as shown in Example 3-11:

  Example 3-11. Declaring an asm function

```
asm void f(int,int) {
  ...
}
```

- As an `asm` function must be output with a prototyped parameter list, these functions are valid for ANSI C modes only.

3.4.1.1.4 Types Extensions

The compiler accepts:

- Bit-fields with base types that are `enums` or integer types, as well as the types `int` and `unsigned int`. The use of any signed integer type is equivalent to using type `int`, and the use of any unsigned integer type is equivalent to using type `unsigned int`.
- The last member of a `struct` containing an incomplete array type. It may not be the only member of the `struct` (otherwise, the `struct` would have zero size).
- A file-scope array with an incomplete `struct`, `union`, or `enum` type as its element type. The type must be completed before the array is subscripted (if it is subscripted), and by the end of the compilation if the array is not `extern`.
- Incomplete `enum` tags. You can define and resolve the tag name later by specifying the brace-enclosed list.
- Object pointer types and function parameter arrays that decay to pointers may use `restrict` as a type qualifier. Its presence is recorded in the compiler so that optimizations are performed that would otherwise be prevented because of possible aliasing.
- The type `long float` as a synonym for `double`.
- Assignment of pointer types in cases where the destination type has added type qualifiers that are not at the top level (for example, `int **` to `const int **`).

3.4.1.1.5 Expressions and statements

The compiler accepts the following extensions for expressions and statements:

- The compiler allows assignment and pointer differences between pointers to types that are interchangeable, but not identical, for example, `unsigned char *` and `char *`. This includes pointers to same-sized integral types (e.g., typically, `int *` and `long *`). The compiler issues a warning, except in `pcc` mode. Without a warning, the compiler may assign a string constant to a pointer to any kind of character, without a warning.
- In operations on pointers, a pointer to `void` is always implicitly converted to another type if necessary, and a null pointer constant is always implicitly converted to a null pointer of the right type if necessary. In ANSI C, some operators allow such conversions, while others do not, generally where such a conversion would not be logical.
- In an initializer, a pointer constant value may be cast to an integral type if the integral type is big enough to contain it.

- In an integral constant expression, an integer constant may be cast to a pointer type and then back to an integral type.
- In character and string escapes, if the character following the `\` has no special meaning, the value of the escape is the character itself. Thus "`\s`" `==` "`s`". A warning is issued.
- Adjacent wide and non-wide string literals are not concatenated.
- In duplicate size and sign specifiers (e.g., `short short` or `unsigned unsigned`) the redundancy is ignored, and a warning is issued.
- `__ALIGNOF__` is similar to `sizeof`, but returns the alignment requirement value for a type, or `1` if there is no alignment requirement. It may be followed by a type or expression in parentheses, as shown in Example 3-12:

Example 3-12. Returning the alignment requirement

```
__ALIGNOF__(type)
__ALIGNOF__(expression)
```

The expression in the second form is not evaluated.

- Identifiers may not contain dollar signs.
- `__INTADDR__(expression)` scans the enclosed expression as a constant expression, and converts it to an integer constant (it is used in the `offsetof` macro).
- The values of enumeration constants may be given by expressions that evaluate to unsigned quantities which fit in the `unsigned int` range but not in the `int` range. A warning is issued when such a result is possible, as shown in Example 3-13:

Example 3-13. Out of range warning

```
/* When ints are 32 bits: */
enum a {w = -2147483648};  /* No warning */
enum b {x = 0x80000000};   /* No warning */
enum c {y = 0x80000001};   /* No warning */
enum d {z = 2147483649};   /* Warning */
```

- The address of a variable with `register` storage class may be taken, and a warning is issued.
- The expression `&...` is accepted in the body of a function in which an ellipsis appears in the parameter list.
- An ellipsis may appear by itself in the parameter list of a function declaration, for example, `f(...)`. A diagnostic is issued in strict ANSI mode.

- External entities declared in other scopes are visible, as shown in Example 3-14. A warning is issued.

Example 3-14. External entities in other scopes

```
void f1(void) { extern void f(); }
void f2() { f(); /* Using out of scope declaration */ }
```

- Pointers to incomplete arrays may be used in pointer addition, subtraction, and subscripting, as shown below in Example 3-15. A warning is issued if the value added or subtracted is anything other than a constant zero. Since the type pointed to by the pointer has zero size, the value added to or subtracted from the pointer is multiplied by zero and therefore has no effect on the result. Comparisons and pointer differences of such pairs of pointer types are also allowed. A warning is issued.

Example 3-15. Pointers to incomplete arrays

```
int (*p)[];
...
q = p[0];
```

- Pointers to different function types may be assigned or compared for equality (==) or inequality (!=) without an explicit type cast, and a warning is issued.
- A pointer to `void` may be implicitly converted to or from a pointer to a function type.
- Intrinsic functions are recognized as extensions only in the default C language mode (ANSI C with extensions). In all other modes they are treated as function calls.

3.4.1.2 K&R/PCC mode

When you specify `pcc` mode, the SC100 C compiler accepts the traditional C language that the *The C Programming Language*, first edition, by Kernighan and Ritchie (K&R), Prentice-Hall, 1978 defined. This mode provides almost complete compatibility with the Reiser CPP and Johnson PCC (`pcc`), both widely used as part of UNIX systems. Since there is no documentation of the exact behavior of those programs, complete compatibility cannot be guaranteed.

In general, when compiling in `pcc` mode, the compiler attempts to interpret a source program that is valid to `pcc` in the same way that `pcc` would. However, ANSI features that do not conflict with this behavior are not disabled.

In some cases where `pcc` allows a highly questionable construct, the compiler accepts it but gives a warning, where `pcc` would be silent. For example: `0x`, a degenerate hexadecimal number, is accepted as zero, but a warning is issued.

3.4.1.2.1 K&R/PCC mode preprocessor differences

The following are the preprocessor differences relative to the default standard mode:

- When preprocessing output is generated, the line-identifying directives have the `pcc` form instead of the ANSI form.
- `__STDC__` is left undefined.
- In preprocessing output, the compiler deletes entire comments instead of replacing them with one space. Extra spaces are not generated in textual preprocessing output to prevent pasting of adjacent confusable tokens. As a result, the characters `a/**/b` are `ab` in preprocessor output.
- The first directory searched for include files is the directory containing the file that contains the `#include` instead of the directory containing the primary source file.
- The compiler does not recognize Trigraphs.
- Macro expansion is implemented differently. Arguments to macros are not macro- expanded before being inserted into the expansion of the macro. Any macro invocations in the argument text are expanded when the macro expansion is rescanned. With this method, macro recursion is possible and is checked for.
- Token pasting inside macro expansions is implemented differently. End-of-token markers are not maintained, so tokens that abut after macro substitution may be parsed as a single token.
- The compiler recognizes macro parameter names inside character and string constants and gives them substitutes.
- The compiler flags macro invocations having too many arguments with a warning rather than an error. The compiler ignores the extra arguments.
- The compiler flags macro invocations having too few arguments with a warning rather than an error. A null string is used as the value of the missing parameters.
- The compiler ignores extra occurrences of `#else`, after the first has appeared in an `#if` block; and instead issues a warning.

3.4.1.2.2 K&R/PCC mode syntax differences

The following are the syntax differences relative to the default standard mode:

- The keywords `signed`, `const`, and `volatile` are disabled, so that they can be user identifiers. The other non-K&R keywords (`enum` and `void`) are judged to have existed already in code and are not disabled.
- The = preceding an initializer may be omitted. A warning is issued. This was an anachronism even in K&R.
- `0x` is accepted as a hexadecimal 0, with a warning.
- `1E+` is accepted as a floating point constant with an exponent of `0`, with a warning.
- The compound assignment operators may be written as two tokens (for example, += may be written + =).
- The compound assignment operators may be written in their old-fashioned reversed forms (for example, -= may be written =-). A warning is issued.
- The digits 8 and 9 are allowed in octal constants. (For example, the constant 099 has the value 9*8+9, or 81.)
- The escape `\a` (alert) is not recognized in character and string constants.

3.4.1.2.3 K&R/PCC mode differences for declarations

The following are the declaration differences relative to the default ANSI mode:

- Declarations of the form `typedef` *`some-type`* `void;` are ignored.
- The names of functions and of external variables are always entered at the file scope.
- A function declared `static`, which is used and never defined, is treated as if its storage class were `extern` (instead of causing an error for being undefined).
- A file-scope array that has an unspecified storage class and remains incomplete at the end of the compilation will be treated as if its storage class is `extern.` In ANSI mode, the number of elements is changed to 1, and the storage class remains unspecified.
- When a function parameter list begins with a `typedef` identifier, the parameter list is considered prototyped only if the `typedef` identifier is followed by something other than a comma or right parenthesis, as shown below in Example 3-16. Function parameters are allowed to have the same names as `typedef` identifiers. In the normal ANSI mode, any parameter list that begins with a `typedef` identifier is considered prototyped, and Example 3-16 would produce an error.

Example 3-16. Prototyped parameter list

```
typedef int t;
int f(t) {}         /* Old-style list */
int g(t x) {}       /* Prototyped list, parameter x of type t */
```

- The empty declaration `struct` *`x;`* will not hide an outer-scope declaration of the same tag. It is taken to refer to the outer declaration.

- In a declaration of a member of a `struct` or `union`, the declarator list may be omitted entirely, to specify an unnamed field which requires padding, as shown in Example 3-17. Such a field may not be a bit-field.

  Example 3-17. Omitting the declarator list

  ```
  struct s {char a; int; char b[2];} v; /* sizeof(v) is 3 */
  ```

- No warning is generated for a storage specifier appearing in other than the first position in a list of specifiers (as in `int static`).
- Free-standing tag declarations are allowed in the parameter declaration list for a function with old-style parameters.
- Declaration specifiers are allowed to be completely omitted in declarations. (ANSI C allows this only for function declarations.) Thus `i;` declares `i` as an `int` variable. A warning is issued.
- An identifier in a function is allowed to have the same name as a parameter of the function. A warning is issued.

3.4.1.2.4 K&R/PCC mode type differences

The following are the type differences relative to the default standard mode:

- Integral types with the same representation (size, signedness, and alignment) will be considered identical and may be used interchangeably. For example, this means that `int` and `long` will be interchangeable if they have the same size.
- `All enums` are given type `int`. In ANSI mode, smaller integral types will be used if possible.
- A “plain” `char` is considered to be the same as either `signed char` or `unsigned char`, depending on the command-line options. In ANSI C, “plain” `char` is a third type distinct from both `signed char` and `unsigned char`.
- All `float` functions are promoted to `double` functions, and any `float` function parameters are promoted to `double` function parameters.
- All `float` operations are executed as `double`.
- The types of large integer constants are determined according to the K&R rules. They will not be `unsigned` in some cases where ANSI C would define them that way.

3.4.1.2.5 K&R/PCC mode differences: expressions and statements

The following are the differences for expressions and statements relative to the default standard mode:

- Assignment is allowed between pointers and integers, and between incompatible pointer types, without an explicit cast. A warning is issued.
- A field selection of the form `p->field` is allowed even if `p` does not point to a `struct` or `union` that contains `field`. In this context, `p` must be a pointer or an integer. Similarly, `x.field` is allowed even if `x` is not a `struct` or `union` that contains `field`. In this case, `x` must be an `lvalue`. In both cases, if `field` is declared as a `field` in more than one `struct` or `union`, it must have the same offset in all instances.
- Overflows detected while folding signed integer operations on constants will cause warnings rather than errors.
- A warning will be issued for an `&` operator applied to an array. The type of such an operation is “address of array element” rather than “address of array”.
- For the shift operators `<<` and `>>`, the usual arithmetic conversions are done on the operands as they would be for other binary operators. The right operand is then converted to `int`, and the result type is the type of the left operand. In ANSI C, the integral promotions are done on the two operands separately, and the result type is the type of the left operand. The effect of this difference is that, in `pcc` mode, a `long` shift count will force the shift to be done as `long`.
- String literals will not be shared. Identical string literals will cause multiple copies of the string to be allocated.
- The expression `sizeof` may be applied to bit-fields. The size is that of the underlying type (for example `unsigned int`).
- Any `lvalues` cast to a type of the same size remain `lvalues`, except when they involve a floating point conversion.
- A warning rather than an error is issued for integer constants that are larger than can be accommodated in an `unsigned long`. The value is truncated to an acceptable number of low-order bits.
- Expressions in a `switch` statement are cast to `int`. This differs from the ANSI C definition in that a `long` expression may be truncated.
- The promotion rules for integers are different: `unsigned char` and `unsigned short` are promoted to `unsigned int`.

3.4.1.2.6 K&R/PCC differences: remaining incompatibilities

The additional known cases where the compiler is not compatible with `pcc` are as follows:

- Token pasting is not implemented outside of macro expansions (meaning, in the primary source line) when two tokens are separated only by a comment. That is, `a/**/b` is not considered to be `ab`. The `pcc` compiler's behavior in such a case can be obtained by preprocessing to a text file and then compiling that file.

  The textual output from preprocessing is also equivalent but not identical. The blank lines and white space will not be exactly the same as those produced in `pcc`.
- The `pcc` compiler considers the result of a `?:` operator to be an `lvalue` if the first operand is constant and the second and third operands are compatible `lvalues`. The compiler never treats the result of the `?:` operator as an `lvalue`.
- The `pcc` compiler misparses the third operand of a `?:` operator in a way that some programs exploit, as follows:

  `i ? j : k += l` is parsed by `pcc` as `i ? j : (k += l)`

  This is not correct, since the precedence of the `+=` operator is lower than the precedence of the `?:` operator. The compiler will generate an error in such a case.
- The `lint` utility recognizes the keywords for its special comments anywhere in a comment, regardless of whether they are preceded by other text in the comment. The compiler only recognizes the keywords when they are the first identifier following an optional initial series of blanks and/or horizontal tabs. In addition, `lint` recognizes only a single digit of the `VARARGS` count. The compiler accumulates as many digits as appear in the count.

3.4.2 Types and Sizes

The data types that the compiler supports are summarized in Table 3-3 below. The table shows the size for each data type in memory and in the two register types, the 40-bit data register (Dn), and the 32-bit address register (Rn). Table 3-3 also shows the required alignment and the value range for each data type, together with a reference to the section in this chapter which provides greater detail about the data type.

Table 3-3. Data Types and Sizes

	Size (in bits)				Range		Details	
Type	**Memory**	**Dn**	**Rn**	**Align**	**Minimum**	**Maximum**	**Section**	**Page**
char	8	40	32	8	-128	127	3.4.2.1	3-37
unsigned char	8	40	32	8	0	255	3.4.2.1	3-37
short	16	40	32	16	-32,768	32,767	3.4.2.2	3-38
unsigned short	16	40	32	16	0	65,535	3.4.2.2	3-38
int	32	40	32	32	-2,147,483,648	2,147,483,647	3.4.2.2	3-38
unsigned int	32	40	32	32	0	4,294,967,295	3.4.2.2	3-38
long	32	40	32	32	-2,147,483,648	2,147,483,647	3.4.2.2	3-38
unsigned long	32	40	32	32	0	4,294,967,295	3.4.2.2	3-38
float, double, and long double	32	40	32	32	-1.17E-38	1.17E+38	3.4.2.3	3-39
fractional short[1]	16	40	-	16	-1	0.99969842	3.4.2.4	3-40
fractional long / int[2]	32	40	-	32	-1	0.9999999953	3.4.2.4	3-40
pointer	32	40	32	32	0	0xFFFFFFFF	3.4.2.5	3-40

1. Fractional short is not a language type. It can be used with intrinsic functions only, and maps to the predefined type short.
2. Fractional long/int is not a language type. It can be used with intrinsic functions only, and maps to the predefined type long/int.

3.4.2.1 Characters

A character, whether signed or unsigned, is stored in memory in one byte (8 bits), and is always aligned on an 8-bit boundary. Arrays of characters occupy one byte per character. Figure 3-5 shows the memory layout for characters.

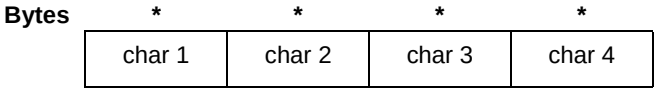


Figure 3-5. Characters—Memory Layout

When loaded into registers, signed characters are signed extended, while unsigned characters are zero extended. Figure 3-6 illustrates the layout for signed and unsigned characters in the Dn (40-bit) data register. “S” indicates the signed extension of the value.

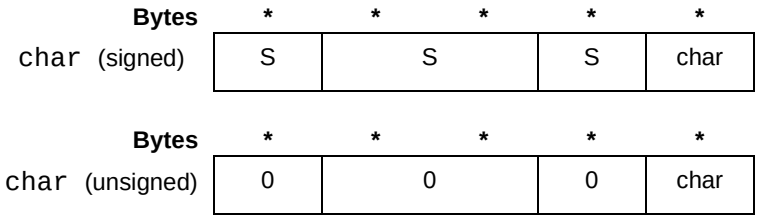


Figure 3-6. Characters—Dn Register Layout

Figure 3-7 shows the layout for signed and unsigned characters in the Rn (32-bit) address register.

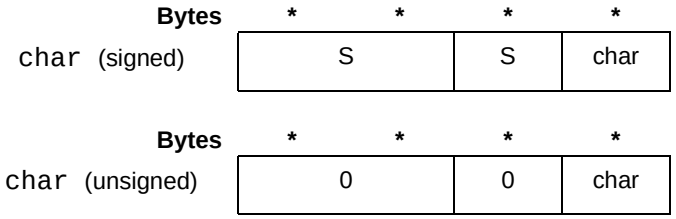


Figure 3-7. Characters—Rn Register Layout

3.4.2.2 Integers

Integer arithmetic is performed using data sizes appropriate to the arithmetic operation. Short integers use at least 16-bit wide operations (single-precision integer arithmetic), and long integers use at least 32-bit (double-precision integer arithmetic).

Short and long integers are stored in memory using little-endian representation (the least significant bits in the lower address), unless the option `-be` is specified.

Integer arithmetic overflow wraps around and does not result in any additional side effects.

Figure 3-8 shows the memory layout for short and long integers.

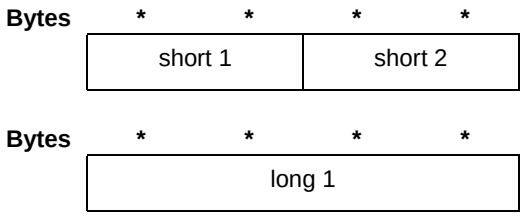


Figure 3-8. Integers—Memory Layout

Short integers must be aligned on 2-byte (16-bit) boundaries, while long integers must be aligned on a 4-byte (32-bit) boundary. Figure 3-9 illustrates the alignment of short and long integers, in conjunction with characters.

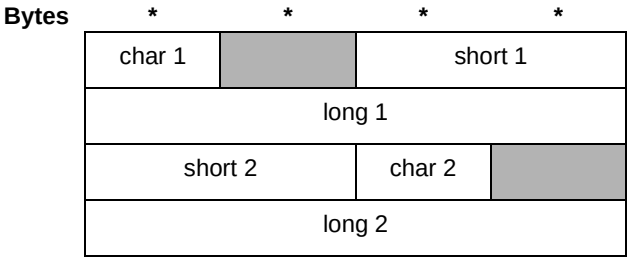


Figure 3-9. Integers—Alignment

As with characters, when loaded into registers, signed integers are signed extended, while unsigned integers are zero extended.

Figure 3-10 illustrates the layout for signed and unsigned short and long integers in the Dn (40-bit) data register. “S” indicates the signed extension of the value.

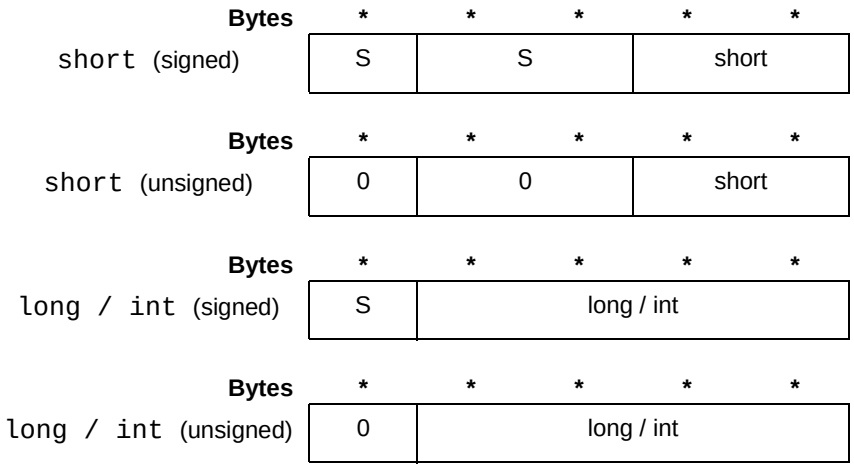


Figure 3-10. Integers—Dn Register Layout

Figure 3-11 shows the layout for signed and unsigned short and long integers in the Rn (32-bit) address register.

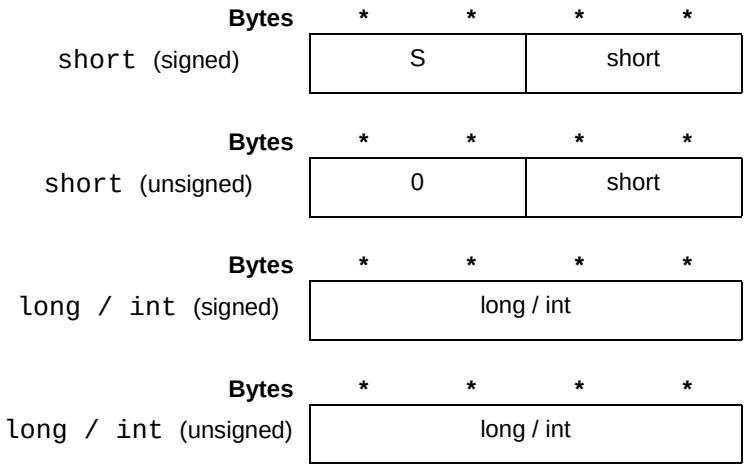


Figure 3-11. Integers—Rn Register Layout

3.4.2.3 Floating point

Floating point, double, and long double type integers are mapped to a single precision IEEE-754 type, using 32 bits (4 bytes). The compiler generates calls for library functions to evaluate floating point expressions. The representation of these integers in memory and in the registers is exactly the same as for long integers.

3.4.2.4 Fractional representation

Since C does not provide built-in support for fractional types, the syntactic representation of fractional types and operations is implemented by intrinsic functions using integer data types. See Section 3.4.4, “Intrinsic Functions,” for details of the intrinsic functions supported.

Fixed -point arithmetic is performed using 16-bit, 32-bit, 40-bit, and 64-bit operations. Fractional integers are stored in memory using little-endian representation, meaning the least significant bits in the lower address, unless the option `-be` is specified.

Fractional type overflows may saturate and do not result in any additional side effect. Rounding and saturation modes are determined as part of the startup code, or with optional intrinsic function calls.

Operations on double and extended precision type objects are limited to assignments and fractional arithmetic using intrinsic functions only. See Section 3.4.3, “Fractional and Integer Arithmetic,” for further information. Integer operations on extended precision types are not supported.

Fractional types are mapped to their corresponding predefined types. A fractional `short` maps to the predefined type `short`, a fractional `long` maps to the predefined type `long`, and a fractional `int` maps to the predefined type `int`.

Figure 3-12 illustrates the layout for fractional short and long integers in the Dn (40-bit) data register, which is the only register used for fractional integer types. “S” indicates the signed extension of the value.

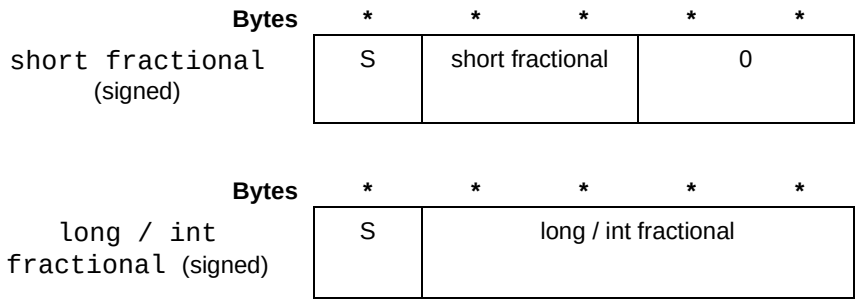


Figure 3-12. Fractional Integers—Dn Register Layout

When loading data from memory into data registers, the compiler aligns the data in the registers according to the context in which the data is used.

3.4.2.5 Pointers

Pointers contain addresses of data objects or functions. Pointers are represented in memory using 32 bits (4 bytes). In the small memory model, although pointers are represented in memory using 32 bits, only 16 bits are meaningful. The representation of pointers in memory and in the registers is exactly the same as for unsigned long integers, as shown in Section 3.4.2.2, “Integers.”

3.4.2.6 Bit-fields

Members of structures are always allocated on byte boundaries, and are aligned according to their fundamental base type. However, bit-fields in structures can be allocated at any bit and of any length not exceeding the size of a long word (32 bits). Signed and unsigned bit-fields are permitted and are sign extended when fetched. A bit-field of type `int` is considered `signed`.

Bit-fields are always allocated from the low-order end of a word (right to left or little-endian), even if the option `-be` is specified. Bit-field sizes are not allowed to cross a long word boundary.

In the following example, the structure `more` has 4-byte alignment and will have a size of 4 bytes. This is because the bit-fields in the structure are governed by the fundamental type `long` which requires a 4-byte alignment.

Example 3-18. Bit-field alignment to long word (1)

```
struct more {
       long first : 3;
       unsigned int second : 8;
};
```

The structure `less` shown in Example 3-19 requires only a one byte alignment because this is the requirement of the fundamental type `char` used in this structure.

Example 3-19. Bit-field alignment to character

```
struct less {
       unsigned char third : 3;
       unsigned char fourth : 8;
};
```

The alignments are driven by the underlying type, not the width of the fields. These alignments are to be considered along with any other structure members.

In Example 3-20 below, the structure `careful` requires a 4-byte alignment; its bit-fields require only a one byte alignment, but the field `fluffy` requires a 4-byte alignment because its fundamental type is `long`.

Example 3-20. Bit-field alignment to long word (2)

```
struct careful {
       unsigned char third : 3;
       unsigned char fourth : 8;
       long fluffy;
};
```

Fields within structures and unions begin on the next possible suitably aligned boundary for their data type. For fields that are not bit-fields, this is a suitable byte alignment. Bit-fields begin at the next available bit offset, with the following exception: the first bit-field after a member that is not a bit-field will be allocated on the next available byte boundary.

In the following example, the offset of the field `c` is one byte. The structure itself has 4-byte alignment and is four bytes in size because of the alignment restrictions introduced by using the `long` underlying data type for the bit-field.

Example 3-21. Bit-field offset

```
struct s {
       int bf: 5;
       char c;
};
```

3.4.3 Fractional and Integer Arithmetic

The ability to perform both integer and fractional arithmetic is one of the strengths of the SC100 C compiler.

Fractional arithmetic is typically required for computation-intensive algorithms such as digital filters, speech coders, vector and array processing, digital control, or other signal processing tasks. In this mode, the data is interpreted as fractional values, and the computations are performed interpreting the data as fractional. Fractional arithmetic examples are shown in Example 3-22.

Often, saturation is used when performing calculations in this mode to prevent the severe distortion that occurs in an output signal generated from a result where a computation overflows without saturation. Saturation can be selectively enabled or disabled so that intermediate calculations can be performed without limiting, and limiting is only done on final results.

Example 3-22. Fractional arithmetic examples

```
0.5 * 0.25  -> 0.125

0.625 + 0.25 -> 0.875

0.125 / 0.5 -> 0.25

0.5 >> 1 -> 0.25
```

It is important to note that the notation used in Example 3-22 is for illustration purposes only, since C does not support the specification of fractional constants using floating-point notation. The compiler implements fractional arithmetic using intrinsic functions based on integer data types. For more information, see Section 3.4.2.4, “Fractional representation,” and Section 3.4.4, “Intrinsic Functions.”

Integer arithmetic is invaluable for controller code, array indexing and address computations, peripheral setup and handling, bit manipulation, and other general purpose tasks, as shown in Example 3-23.

Example 3-23. Integer arithmetic examples

```
4 * 3      -> 12

1201 + 79  -> 1280

63 / 9     -> 7

100 << 1   -> 200
```

Data in a memory location or register can be interpreted as fractional or integer, depending on the needs of a user's program. Table 3-4 shows how a 16-bit value can be interpreted as either a fractional or integer value, depending on the location of the binary point.

Table 3-4. Interpretation of 16-bit Data Values

Binary Representation[1]	Hexadecimal Representation	Integer Value (decimal)	Fractional Value (decimal)
0.100 0000 0000 0000	0x4000	16384	0.5
0.010 0000 0000 0000	0x2000	8192	0.25
0.001 0000 0000 0000	0x1000	4096	0.125
0.111 0000 0000 0000	0x7000	28672	0.875
0.000 0000 0000 0000	0x0000	0	0.0
1.100 0000 0000 0000	0xC000	-16384	-0.5
1.110 0000 0000 0000	0xE000	-8192	-0.25
1.111 0000 0000 0000	0xF000	-4096	-0.125
1.001 0000 0000 0000	0x9000	-28672	-0.875

1. Note: This corresponds to the location of the binary point when interpreting the data as fractional. If the data is interpreted as integer, the binary point is located immediately to the right of the LSB.

The following equation shows the relationship between a 16-bit integer and a fractional value:

Fractional Value = Integer Value / (2^{15})

There is a similar equation relating 40-bit integers and fractional values:

Fractional Value = Integer Value / (2^{31})

Table 3-5 shows how a 40-bit value can be interpreted as either an integer or fractional value, depending on the location of the binary point.

Table 3-5. Interpretation of 40-bit Data Values

Hexadecimal Representation	40-bit Integer in Entire Accumulator (decimal)	16-bit Integer in MSP (decimal)	Fractional Value (decimal)
0x0 4000 0000	1073741824	16384	0.5
0x0 2000 0000	536870912	8192	0.25
0x0 0000 0000	0	0	0.0
0xF C000 0000	-1073741824	-16384	-0.5
0xF E000 0000	-536870912	-8192	-0.25

The following code fragment illustrates the use of integer arithmetic:

Example 3-24. Integer arithmetic computation

```
a = a + b*c;
```

Example 3-25 provides an example of the use of an intrinsic function to implement fractional arithmetic.

Example 3-25. Fractional arithmetic computation

```
a = L_mac(a,b,c);
```

Section 3.4.4, “Intrinsic Functions,” describes the use of intrinsic functions in greater detail.

3.4.4 Intrinsic Functions

The compiler supports a large number of intrinsic (built-in) functions that map directly to SC100 assembly instructions. As C does not support fractional types and operations, these intrinsic functions enable implementation of fractional operations using integer data types.

The syntax of the compiler group of intrinsic functions is structured for full compatibility with the ETSI and ITU reference implementations of bit-exact standards.

3.4.4.1 Data types for intrinsic functions

The following four data types are defined for specific use with intrinsic functions:

- Fractional short, a 16-bit fractional value mapped to a short, as described in Section 3.4.2.4, “Fractional representation.”
- Fractional long, a 32-bit fractional value mapped to a long, as described in Section 3.4.2.4, “Fractional representation.”
- Extended precision fractional, a 40-bit value which can only be used in intrinsic functions. See Section 3.4.4.1.1, “Extended precision fractional,” for details.
- Double precision fractional, a 64-bit value which can only be used in intrinsic functions. See Section 3.4.4.1.2, “Double precision fractional,” for details.

Extended and double precision fractional types enable algorithms to be defined which require precision larger than 32 bits. These data types can be used only with intrinsic functions and with assignments. Variables defined as extended and double precision fractionals cannot be used for standard arithmetical or other operations.

3.4.4.1.1 Extended precision fractional

The extended precision fractional (`Word40`) is a 40-bit data type which occupies the entire Dn (40-bit) register, as shown in Figure 3-13:

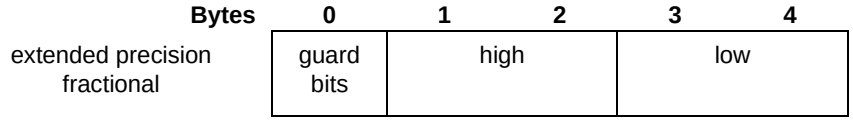


Figure 3-13. Extended Precision Fractional—Dn Register Layout

This data type is mapped in the compiler as a structure containing two elements:

- A 32-bit integer placed to the right of the binary point.
- An 8-bit integer placed to the left of the binary point. These “guard bits” can be used to ensure a more accurate result when an overflow occurs.

When stored in memory, an extended precision fractional variable occupies 64 bits. The least significant 32 bits are stored in the first 32-bit word, and the 8 most significant guard bits are stored in the second 32-bit word in an undefined position.

See Table 3-6 on page 3-47 for a list of intrinsic functions for fractional arithmetic using guard bits.

See Example 3-27 on page 3-51 for an illustration of the use of intrinsic functions with extended precision fractional variables.

3.4.4.1.2 Double precision fractional

The double precision fractional data type (`Word64`) consists of 64 bits, all of which are assumed to be to the right of the binary point. This data type is mapped in the compiler as a structure containing two 32-bit elements.

See Table 3-6 on page 3-47 for a list of intrinsic functions for double precision data types.

3.4.4.1.3 Fractional constants

Fractional constants require integer notation, since floating point notation is not supported. For example, to express the value `0.5` as a fractional constant, the integer representation in hexadecimal must be used in the source code, in this case `0x4000`. For further examples of fractional values and their corresponding hexadecimal representations, see Table 3-4 on page 3-43.

3.4.4.1.4 Initializing variables with fractional values

Variables can be initialized as fractional values, using the following macros:

- `WORD16` initializes a value as a fractional short.
- `WORD32` initializes a value as a fractional long.

For example, `short x = WORD16(0.5)` initializes `x` as a fractional short with the value `0x4000.`

3.4.4.2 Intrinsic function categories

The following categories of intrinsic functions are provided:

- Fractional arithmetic
- Long fractional arithmetic
- Double precision fractional arithmetic
- Extended precision fractional arithmetic, with guard bits
- Architecture primitives
- Architecture primitives that generate identical assembly instructions
- Bit reverse addressing

Table 3-6 on page 3-47 lists all the supported intrinsic functions by category, with a brief description of each function. See Section 7.15, “Built-in Intrinsic Functions (prototype.h),” on page 7-21, for more detailed information about each of the fractional arithmetic and long fractional arithmetic intrinsic functions.

Section 3.4.4.3, “Intrinsic functions examples,” which follows Table 3-6, contains example code segments illustrating the use of a number of intrinsic functions.

Table 3-6. Intrinsic Functions

a) Fractional arithmetic

Intrinsic Function	Declaration	Description
add	short add(short,short)	Short add
sub	short sub(short,short)	Short subtract
mult	short mult(short,short)	Short multiply
div_s	short div_s(short,short)	Short divide
mult_r	short mult_r(short,short)	Multiply and round
L_mac	long L_mac(long,short,short)	Multiply accumulate
mac_r	short mac_r(long,short,short)	Multiply accumulate and round
L_msu	long L_msu(long,short,short)	Multiply subtract
msu_r	short msu_r(long,short,short)	Multiply subtract and round
abs_s	short abs_s(short)	Short absolute value
negate	short negate(short)	Short negate
round	short round(long)	Round
shl	short shl(short,short)	Short shift left
shr	short shr(short,short)	Short shift right
shr_r	short shr_r(short,short)	Short shift right and round
norm_s	short norm_s(short)	Normalize any fractional value
max	short max(short,short)	Maximum value of any two short fractional values
min	short min(short,short)	Minimum value of any two short fractional values
saturate	short saturate(short)	Short saturation

Table 3-6. Intrinsic Functions (Continued)

b) Long fractional arithmetic

Intrinsic Function	Declaration	Description
L_add	long L_add(long,long)	Long add
L_sub	long L_sub(long,long)	Long subtract
L_mult	long L_mult(short,short)	Long multiply
extract_h	short extract_h(long)	Extract 16 MSB of long word
extract_l	short extract_l(long)	Extract 16 LSB of long word
L_deposit_h	long L_deposit_h(short)	Deposit short in MSB
L_deposit_l	long L_deposit_l(short)	Deposit short in LSB
L_abs	long L_abs(long)	Long absolute value
L_negate	long L_negate(long)	Long negate
norm_l	short norm_l(long)	Normalize any long fractional value
L_max	long L_max(long,long)	Maximum value of any two long fractional values
L_min	long L_min(long,long)	Minimum value of any two long fractional values
L_shl	long L_shl(long,short)	Long shift left
L_shr	long L_shr(long,short)	Long shift right
L_shr_r	long L_shr_r(long,short)	Long shift right and round
L_sat	long L_sat(long)	Long saturation

c) Double precision fractional arithmetic

Intrinsic Function	Declaration	Description
D_mult	Word64 D_mult(long,long)	Double precision multiply
D_mac	Word64 D_mac(Word64,long,long)	Double precision multiply accumulate
D_msu	Word64 D_msu(Word64,long,long)	Double precision multiply subtract
D_add	Word64 D_add(Word64,Word64)	Double precision add
D_sub	Word64 D_sub(Word64,Word64)	Double precision subtract
D_cmpeq	short D_cmpeq(Word64,Word64)	Double precision compare equal
D_cmpgt	short D_cmpgt(Word64,Word64)	Double precision compare greater than
D_sat	Word64 D_sat(Word64)	Double precision saturation
D_round	long D_round(Word64)	Double precision round
D_set	Word64 D_set(long,unsigned long)	Concatenate two longs into one double precision value
D_extract_l	unsigned long D_extract_l(Word64)	Extract 32 LSB of double precision value
D_extract_h	long D_extract_h(Word64)	Extract 32 MSB of double precision value

Table 3-6. Intrinsic Functions (Continued)

d) Extended precision fractional arithmetic (with guard bits)

Intrinsic Function	Declaration	Description
X_mult	Word40 X_mult(short,short)	Short multiply to long long word
X_mac	Word40 X_mac(Word40,short,short)	Short multiply accumulate to long long word
X_msu	Word40 X_msu(Word40,short,short)	Short multiply subtract to long long word
X_set	Word40 X_set(char,unsigned long)	Concatenate char and unsigned long into one long long word
X_add	Word40 X_add(Word40,Word40)	Long add including guard bits
X_sub	Word40 X_sub(Word40,Word40)	Long subtract including guard bits
X_shl	Word40 X_shl(Word40,short)	Long shift left with guard bits
X_shr	Word40 X_shr(Word40,short)	Long shift right with guard bits
X_extract_h	short X_extract_h(Word40)	Extract 16 MSB of long long word
X_extract_l	short X_extract_l(Word40)	Extract 16 LSB of long long word
X_round	short X_round(Word40)	Round long long value
X_norm	short X_norm(Word40)	Normalize any long long fractional value
X_rol	Word40 X_rol(Word40)	Rotate left a long long word
X_ror	Word40 X_ror(Word40)	Rotate right a long long word
X_abs	Word40 X_abs(Word40)	Long absolute value with guard bits
X_sat	long X_sat(Word40)	Long saturation including guard bits
X_or	Word40 X_or(Word40,Word40)	Logical OR two long values with guard bits
X_trunc	long X_trunc(Word40)	Truncate guard bits
X_extend	Word40 X_extend(long)	Sign extend long value to include guard bits
X_cmpeq	short X_cmpeq(Word40,Word40)	Fractional compare equal with guard bits
X_cmpgt	short X_cmpgt(Word40,Word40)	Fractional compare greater than with guard bits

Table 3-6. Intrinsic Functions (Continued)

e) Architecture primitives

Intrinsic Function	Declaration	Description
L_rol	`long L_rol(long)`	Rotate left a long
L_ror	`long L_ror(long)`	Rotate right a long
mpyuu	`long mpyuu(long,long)`	Long multiply 16 LSB of two long words, treating both words as unsigned values
mpyus	`long mpyus(long,long)`	Long multiply 16 LSB of the first long word, treated as an unsigned value, by 16 MSB of the second long word, treated as signed
mpysu	`long mpysu(long,long)`	Long multiply 16 MSB of the first long word, treated as a signed value, by 16 LSB of the second long word, treated as unsigned
setnosat	`setnosat()`	Set saturation mode off
setsat32	`setsat32()`	Set saturation mode on
set2crm	`set2crm()`	Set rounding mode to two's-complement rounding mode
setcnvrm	`setcnvrm()`	Set rounding mode to convergent rounding mode

f) Architecture primitives that generate identical assembly instructions

Intrinsic Function	Declaration	Description
debug	`void debug()`	Enter `Debug` mode
debugev	`void debugev()`	Generate `Debug` event
mark	`void mark()`	If trace buffer enabled, write program counter to trace buffer
stop	`void stop()`	Enter `Stop` low power mode
trap	`void trap()`	Execute `Trap` exception
wait	`void wait()`	Enter `Wait` low power mode
ei	`void ei()`	Enable interrupts
di	`void di()`	Disable interrupts
illegal	`void illegal()`	Execute `illegal` exception

g) Bit reverse addressing

Intrinsic Function	Declaration	Description
InitBitReverse	`InitBitReverse`	Allocate a bit reverse iterator
BitReverseUpdate	`BitReverseUpdate`	Increment the iterator with bit reverse
EndBitReverse	`EndBitReverse`	Free bit reverse iterator

3.4.4.3 Intrinsic functions examples

The following example illustrates the use of a number of intrinsic functions.

Example 3-26. Intrinsic functions

```
#include <prototype.h>
void  Iir(short Input[], short Coef[], short FiltOut[])
{
      long L_Sum;
      short int Stage, Smp;
      FiltOut[0] = Input[0];

      for (Smp = 1; Smp < S_LEN; Smp++)
      {
            L_Sum = L_msu(LPC_ROUND, FiltOut[Smp - 1], Coef[0]);
            for (Stage = 1; ((0 < (Smp - Stage)) && Stage < NP); Stage++)
                  L_Sum = L_msu(L_Sum, FiltOut[Smp - Stage - 1], Coef[Stage]);
            L_Sum = L_shl(L_Sum, ASHIFT);
            L_Sum = L_msu(L_Sum, Input[Smp], 0x8000);
            FiltOut[Smp] = extract_h(L_Sum);
  }
}
```

Example 3-27 illustrates the use of extended precision variables and intrinsic functions using guard bits:

Example 3-27. Intrinsic functions using extended precision

```
#define M1 10
#define M2 10
#include <prototype.h>
docorr()
{
      int L_sample[10];
      int coeff[10]
      int sample[10]
      int j, i;
      int shift_val;
      short corr_0;
      Word40 E_acc, E_sum;
      E_acc = X_extend(0); E_sum = X_extend(0);
      for (i = 0; i < M1; i++)
      {
            for (j = 0; j < M2; j++)
                  E_acc = X_mac (E_acc, sample[j], coeff[j] );
            L_sample[i] = X_sat(E_acc);
            E_acc = X_abs(E_acc);
            E_sum = X_add(E_sum, E_acc);
      }
       shift_val = X_norm(E_sum);
       corr_0 = 0;
       for (i = 0; i < M1; i++)
      {
            sample[i] = round (L_shr (L_sample[i], shift_val));
            corr_0 = sub (corr_0, sample[i]);
      }
      corr = corr_0;
}
```

3.4.5 Pragmas

Pragmas allow you greater control over your application, enabling you to give the compiler specific additional information about how to process certain statements. The pragmas that you specify in your code provide the compiler with context-specific hints which can save the compiler unnecessary operations, and help to further enhance the optimization process.

You can include as many pragmas as necessary in your source code. The sections that follow describe the syntax and placement rules for pragmas.

3.4.5.1 Syntax

The pragmas supported by the compiler have the following general syntax:

```
#pragma pragma-name [argument(s)]
```

One or more of the arguments may be optional. Arguments are comma-delimited.

Each pragma must fit into one line.

3.4.5.2 Placement

Each pragma is applicable only in a certain context, and must be placed accordingly. Four categories of pragmas can be defined according to the placement rules, as follows:

- Pragmas which apply to functions can appear only in the scope of the function, after the opening "{".
- Pragmas which apply to statements must be placed immediately before the relevant statement, or immediately before any comment lines which precede the statement.
- Pragmas which apply to variables must follow the object definition, or any comment lines which follow that definition. Objects referred to by pragmas must be explicitly defined.

The pragmas supported by the compiler are listed in Table 3-7 on page 3-53. The sections that follow the table provide a brief summary and example of the syntax and use of each pragma. The detailed functioning of each pragma is described in Chapter 5, "Optimization Techniques and Hints."

Table 3-7. Pragmas

h) Function pragmas	**Description**	**Section**	**Page**
`#pragma inline`	Forces function inlining.	3.4.5.3.1	3-54
`#pragma noinline`	Disables function inlining.	3.4.5.3.1	3-54
`#pragma save_ctxt`	Forces save and restore of all registers that are used in this procedure.	3.4.5.3.2	3-54
`#pragma external` *`func`* `[name =` *`string`*`, convention =` *`number`*`, nosideeffects]`	Defines a function as external to the C application, or as a function that can be called from outside the application.	3.4.5.3.3	3-55
`#pragma interrupt` *`func`*	Defines the specified function as an interrupt handler.	3.4.5.3.4	3-56
i) Pragmas which apply to statements	**Description**	**Section**	**Page**
`#pragma profile` *`value`*	Sets profiling information for a statement.	3.4.5.4.1	3-56
`#pragma loop_count` *`(lower_bound, upper_bound, {2/4}, remainder)`*	Specifies the minimum and maximum limits for a loop, the loop count divider (2 or 4), and the use of the remainder.	3.4.5.4.2	3-57
j) Pragmas which apply to variables	**Description**	**Section**	**Page**
`#pragma align` *`var_name`* `{4/8}`	Forces stricter alignment on an object. Needed for paired moves.	3.4.5.5.1	3-59
`#pragma align *`*`var_name`* `{4/8}`	Indicates that the address of the variable referenced by a pointer is aligned as specified.	3.4.5.5.1	3-59

3.4.5.3 Pragmas which apply to functions

The pragmas in this category provide additional information about specific functions, and are defined in the scope of the function to which they apply, directly after the "{" which marks the start of the scope.

3.4.5.3.1 Forcing or disabling function inlining

Inlining enables the compiler to improve optimization by replacing a function call by the entire function. For very small functions, for example, where the overhead of the function call is greater than the size of the function itself, this can be very efficient. For more information about function inlining, refer to Chapter 5, "Optimization Techniques and Hints."

You can use `#pragma inline` to force the compiler to inline a specific function, or `#pragma noinline` to prevent the compiler from inlining a certain function. In the code segment shown in Example 3-28, any calls to the function which follows `#pragma noinline` will not be inlined.

Example 3-28. #pragma noinline

```
static int proc_30(int a)
{
#pragma noinline
    int tab_30[1000];

    tab_30[0] = 4*a;
    return(tab_30[0]);
}
```

3.4.5.3.2 Saving the entire context of the system

During normal processing, the compiler saves the contents of registers that have been changed, and any other essential data. You can force the compiler to save the entire context of the machine, including all registers that are used in this procedure, so that it can be restored if necessary to its previous state, at the exact point at which the specific function started to execute.

Using `#pragma save_ctxt` to save the entire system status can incur a large overhead, and should only be used where absolutely necessary.

The following example illustrates the use of `#pragma save_ctxt` to force the compiler to save the complete machine context upon entry to the specified function.

Example 3-29. #pragma save_ctxt

```
void EntryPoint()
{
#pragma save_ctxt
...
}
```

3.4.5.3.3 Defining a function as external

When the compiler encounters an unresolved function call, it assumes by default that this is a call to an external function that exists outside the application. The pragma `#pragma external` enables you to:

- Confirm this assumption, by informing the compiler that the call is to an external function defined outside the application
- Define the function as an internal function that can be called from outside the application

The effect of the pragma depends on its placement, as described below:

- If `#pragma external` is specified in the global scope, the compiler does not expect to find the body of the function within the current application. The compiler uses standard calling conventions to call the function, and does not issue warnings for unresolved references. Specifying `#pragma external` in the global scope is valid only with cross-file optimization.
- If `#pragma external` is specified within the function scope, followed by the body of the defined function, the compiler recognizes this as an internal function that can be called from outside the application.

The following optional parameters can be specified with `#pragma external`:

- Specify `name = string` to provide a specific function name, to override the default linkage name allocated to the function.
- Define `convention = number` to select the calling convention to be used instead of the default standard convention. See Chapter 6, “Runtime Environment,” for further information about calling conventions.
- Specify `nosideeffects` if the function does not change any variable values in the application, and can be moved or duplicated in other parts of the application without making any changes.

When `nosideeffects` is specified, the compiler does not need to make worst case assumptions about any possible impact that the function may have within the application.

In the first part of Example 3-30, `printf` is defined as an external function that does not exist within the application, and that has no effect on any variables in the application. In the second part of the example, the function `ICanBeCalled` is defined inside the application and may be called by external function calls. This function therefore has to obey the standard calling conventions.

Example 3-30. #pragma external

```
extern void printf();
#pragma external printf [nosideeffects]

void main()
{
  printf("Hello there\n");
}

void ICanBeCalled(int X, int Y)
{
#pragma external ICanBeCalled [name ="xyz"]
...
}
```

3.4.5.3.4 Defining a function as an interrupt handler

A function that operates as an interrupt handler differs from other functions in three basic respects:

- It must save and restore all resources that it uses, as it can be called at any time an interrupt occurs, and cannot assume any conventions.
- It runs in “exception” mode, which forces the compiler to generate instructions that are slightly different from the instructions issued in normal mode.
- It cannot be passed parameters nor return a value.

You can use `#pragma interrupt` to define a function as an interrupt handler, as shown in the following example.

Example 3-31. #pragma interrupt

```
void IntHandler();
#pragma interrupt IntHandler
extern long Counter;
void IntHandler()
{
Counter++;
}
```

3.4.5.4 Pragmas which apply to statements

Pragmas which apply to statements are placed immediately before the relevant statement.

3.4.5.4.1 Specifying a profile value

By default, the profiler provided with the compiler enables it to make the necessary assumptions about the number of times to execute a given statement. You can specify `#pragma profile`, followed by a value and immediately preceding a statement, to specify to the compiler the exact number of times that the statement executes.

In Example 3-32, the value following `#pragma profile` notifies the compiler that the loop executes only 10 times. If `#pragma profile` is not specified, the compiler assumes that, since this is a loop with dynamic bounds, the loop executes 25 times (the default). It is important to note that this assumption affects the optimization of the program, and not its correctness.

Example 3-32. #pragma profile with constant value

```
#include <prototype.h>
int energy (short block[], int N)
{
    int i;

    long int L_tmp = 0;

    for (i = 0; i < N; i++)
#pragma profile 10
      L_tmp = L_mac (L_tmp, block[i], block[i]);

    return round (L_tmp);
}
```

With `if-then-else` constructs, `#pragma profile` can be used to inform the compiler which branch executes more frequently, and the frequency ratio between the two branches, meaning the number of times one branch executes in relation to the other.

In Example 3-33, the two `#pragma profile` statements have the values 5 and 50. These values notify the compiler that the `else` branch section executes 10 times more frequently than the first (implied `then`) section. When used in this way, the exact `#pragma profile` values are not significant, since they indicate the frequency ratio, and not the absolute values. In this example, the values 1 and 10 would convey the same information.

Example 3-33. #pragma profile with frequency ratio

```
#include <prototype.h>
int energy (short block[], int N)
{
       int i;
       long int L_tmp = 0;

       if ( N>50)
#pragma profile 5
              for (i = 0; i < 50; i++)
                     L_tmp = L_mac (L_tmp, block[i], block[i]);
       else
#pragma profile 50
              for (i = 0; i < N; i++)
                     L_tmp = L_mac (L_tmp, block[i], block[i]);

    return round (L_tmp);
}
```

3.4.5.4.2 Defining a loop count

The compiler tries to evaluate the number of times a loop iterates using the static information available. In cases where this static information is not supplied to the compiler, if you know the upper and lower limits of a loop, you can use `#pragma loop_count` to provide these values. Supplying such information, which cannot always be discerned automatically by the compiler, enables generation of more efficient code.

Similarly, specifying a divider for the loop count enables the optimizer to unroll loops in the most efficient way. The loop count can be divided by either 2 or 4, corresponding to the number of execution units. You can also instruct the compiler whether to use the remainder, if there is one following division of the loop count, to execute the loop an additional number of times.

The syntax of `#pragma loop_count` is:

```
#pragma loop_count (lower_bound, upper_bound, [{2/4}, [remainder]])
```

Define a value for lower_bound for the minimum number of times the loop will iterate, and a value for `upper_bound` for the maximum number of times.

The divider parameter is optional. Only the values 2 or 4 may be specified as the divider.

To specify that a remainder should be used for the loop count, specify a value for `remainder`. The `remainder` argument is only valid if a value has been specified for the divider.

The pragma `#pragma loop_count` must be placed inside the loop to which it relates, and outside any nested loops which the loop contains.

In Example 3-34, the loop will always iterate at least 4 times and at most 512 times. The iteration count will always be divisible by 4. As no remainder is specified, any remainder from the division will be disregarded.

Example 3-34. #pragma loop count

```
void correlation2 (short vec1[], short vec2[], int N, short *result)
{
    long int L_tmp = 0;
    int i;

for (i = 0; i < N; i++)
             #pragma loop_count (4,512,4)
             L_tmp = L_mac (L_tmp, vec1[i], vec2[i]);

    *result = round (L_tmp);
}
```

3.4.5.5 Pragmas which apply to variables

These pragmas are placed immediately after the definition of the object(s) to which they refer. Objects referred to by pragmas must first be explicitly defined.

3.4.5.5.1 Alignment of variables

Objects are usually aligned according to their size, as described in Section 3.4.2, “Types and Sizes.” The default alignment for arrays is determined by their base type.

An array may need to be aligned to a specified value before it can be passed to an external function. The pragma `#pragma align` can be used to force the alignment of arrays passed to an external function, to meet the specific alignment requirements of the function.

To force the alignment of an array before passing it to an external function, specify `#pragma align`, followed by the defined array object, and either the value 4 for 4-byte (32-bit double word) alignment or 8 for 8-byte (64-bit quad word) alignment.

Certain instructions, such as `move.2w` and `move.4w`, which move words in pairs, may require alignment to be applied that is stricter than the alignment defined for the data types involved.

In certain cases, the compiler cannot assess the alignment for dynamic objects and has to assume that the objects have the alignment requirements for their base type. As a result, the compiler cannot use the multiword move instructions for these objects. By specifying the exact alignment for one or more objects, you can enable the compiler to use these multiword moves and generate more efficient code.

You can use the pragma `#pragma align` to provide the compiler with specific alignment information about pointers to arrays, in order to enable the compiler to use multiword move instructions.

To inform the compiler that the address of an array is aligned as required for multiword moves, specify `#pragma align`, followed by the pointer to the array object, and either the value 4 for 4-byte alignment or 8 for 8-byte alignment. When using `#pragma align` in this way, you should ensure that the object is in fact aligned as required, since this form of the pragma does not force the alignment.

In the first part of Example 3-35, array `a` is forced to 8-byte alignment before being passed to the external function `Energy`. The second part of the example informs the compiler that both input vectors are aligned to 32 bits. The instruction `move.2f` may be used here.

Example 3-35. #pragma align

```
#include <prototype.h>
short a[10];
#pragma align a 8

extern int Energy( short a[] );
int foo()
{
       return Energy(a);
}
short Cor(short vec1[], short vec2[], int N)
{
#pragma align *vec1 4
#pragma align *vec2 4

       long int L_tmp = 0;
       long int L_tmp2 = 0;
       int i;

       for (i = 0; i < N; i += 2)
              L_tmp = L_mac(L_tmp, vec1[i], vec2[i]);
              L_tmp2 = L_mac(L_tmp2, vec1[i+1], vec2[i+1]);

       return round(L_tmp + L_tmp2);
}
```

3.4.6 Predefined Macros

The compiler shell maintains a number of predefined macros, including standard C macros, and additional macros which are specific to the SC100 C compiler and the SC100 architecture. Table 3-8 lists these predefined macros.

Table 3-8. Predefined Macros

Macro Name	Description
`__LINE__`	The line number of the current source line.
`__FILE__`	The name of the current source file.
`__DATE__`	The compilation date, as a character string in the form Mmm dd yyyy e.g. Jan 23 1999.
`__TIME__`	The compilation time, as a character string in the form hh:mm:ss.t
`__STDC__`	Decimal constant 1, indicating ANSI conformance.
`__STDC_VERSION__`	Defined in ANSI C mode as 199409L.
`__SIGNED_CHARS__`	Defined when char is signed by default
`__VERSION__`	The version number of the compiler, as a character string in the form nn.nn.

`_ENTERPRISE_C_`	Defined for use with the Enterprise compiler. If your source file may be compiled with other compilers apart from the Enterprise, this macro should be included in a conditional statement to ensure that the appropriate commands are activated, for example:
	`#ifdef _ENTERPRISE_C_`
	`(Enterprise-specific commands)`
	`#else`
	`....`
	`#endif`

`BIG_ENDIAN`	The most significant bits in the lower address.
`LITTLE_ENDIAN`	The least significant bits in the lower address.

`_SC100_`	Defined for use with all compilers based on the SC100 architecture. If your source file may be compiled with other compilers apart from those based on the SC100 architecture, this macro should be included in a conditional statement to ensure that the appropriate commands are activated, as shown in the following example:
	`#ifdef _SC100_`
	`(SC100-specific commands)`
	`#else`
	`....`
	`#endif`

`_SC110_` `_SC140_`	The architecture variant, which specifies the number of MAC units to be used by the compiler: `_SC110_` indicates 1 MAC unit. `_SC140_` indicates 4 MAC units. Only one of these macros is valid for each invocation of the compiler. The macro that is selected, and the value of the architecture variant, are determined by the value set for the `-arch` option when the compiler is invoked. If no value is specified for `-arch`, the default is SC140 (`_SC140_`). See Section 3.3.8.1, “Defining the architecture,” for further information about the `-arch` option.

Chapter 4
Interfacing C and Assembly Code

The SC100 C compiler supports interfacing between C source code and assembly code, enabling access to functionality not provided by C. This chapter describes the features of this interface and provides instructions, guidelines, and examples.

The following sections are contained in this chapter:

- Section 4.1, “Inlining a Single Assembly Instruction,” explains how to use an individual assembly instructions in your C source code.
- Section 4.2, “Inlining a Sequence of Assembly Instructions,” describes how to embed an assembly function consisting of a sequence of assembly instructions into your C code.
- Section 4.3, “Calling an Assembly Function in a Separate File,” explains how an assembly function that is contained in a separate file can be used in conjunction with your C source files.
- Section 4.4, “Including Offset Labels in the Output File,” describes the use of symbolic offsets for C data structures in the assembly output file.

4.1 Inlining a Single Assembly Instruction

A single assembly instruction can be inlined in a sequence of C statements and compiled by the compiler. To ensure successful compilation of an inlined assembly instruction, note the following guidelines:

- The compiler passes an inlined instruction to the assembly output file in text form, and therefore has no knowledge of the contents or side effects of the instruction. It is important that you ensure that there is no risk of the instruction affecting the C and/or assembly environment and producing unpredictable results. For example, do not use an inlined assembly instruction to change the contents of registers, as the compiler has no knowledge of such changes. Similarly, do not include any jumps or labels, which access the C code and may affect the correctness of the tracking algorithms.
- The compiler ignores inlined assembly code instructions.
- Since the compiler treats the assembly instruction as a string of text, it cannot perform any error checking on the instruction. Check the syntax and text of the instruction carefully prior to compilation. Errors in assembly code are identified only at the assembly stage of the compilation process.
- A single inlined assembly instruction cannot reference a C object. The only way to reference a C object in assembly code is by inlining a sequence of assembly instructions, as described in Section 4.2, “Inlining a Sequence of Assembly Instructions.”

To inline a single assembly instruction, use the `asm` statement. The syntax is as for a standard function call, with one argument enclosed in double quotation marks, as shown below in Example 4-1.

Example 4-1. Inlining a single assembly instruction

```
asm("wait");
```

4.2 Inlining a Sequence of Assembly Instructions

It is possible to use assembly code that references C objects, by defining a separate function that consists of a sequence of assembly instructions, and inlining this in your C code. Such a function is implemented entirely in assembly and may not include C statements, but can accept parameters referenced by the assembly code.

4.2.1 Guidelines for Inlining Assembly Code Sequences

The following guidelines are similar to those for the inlining of individual assembly instructions, described in Section 4.1, “Inlining a Single Assembly Instruction,” and apply also to the use of inlined sequences of assembly code:

- The compiler passes a sequence of inlined instructions to the assembly output file as a string of text, and therefore has no knowledge of the contents or side effects of the instructions. It is important that you ensure that the assembly function does not affect the C and/or assembly environment and does not produce unpredictable results. For example, do not use inlined assembly instructions to change the contents of registers, and do not alter the sequence of C code instructions by specifying jumps, as the compiler has no knowledge of such changes.
- The optimizer cannot use functions based on inlined sequences of assembly code; thus, they are ignored during optimization. Avoid using assembly-based functions if a C alternative is available, in order to ensure maximum optimization of the code.
- The compiler performs no error checking on the sequence of assembly instructions. Assembly code errors are identified only at the assembly stage of the compilation process.
- By definition, inline assembly functions are static and declare no external linkage; therefore, you must call inline assembly functions from within the same module that they are declared.

The following guidelines apply specifically to the use of inlined sequences of assembly code for `asm` functions:

- When passing parameters to an inlined sequence of assembly instructions, registers are not automatically allocated. For each parameter, you must specify the register in which the parameter enters or exits the function. There is no need to save and restore the registers before and after the function.
- The compiler cannot deduce whether an inlined function is likely to affect the application, for example, if it modifies global variables. It is important that you provide the compiler with such information if there is a possibility that the function may have any side effects.
- A function that is initially defined as stand-alone may in certain circumstances be included in another sequence of instructions. Therefore, inlined functions should not use statements such as RTS. If the function is used in a sequence of instructions, the compiler automatically adds the necessary return statements.

- The compiler does not automatically allocate local variables for assembly functions to use. If a function requires the use of local variables, you must allocate these variables specifically on the stack or define them as static variables.
- Assembly functions defined as a sequence of instructions can access global variables in the C source code, since these are static by definition.

4.2.2 Defining an Inlined Sequence of Assembly Instructions

When defining a sequence of inlined assembly instructions:

- define the header for the function before the body of the instructions, and
- specify the registers that each parameter will use.

You can define a list of read parameters, a list of write parameters, and/or a list of modified registers, as appropriate.

The syntax for inlining a sequence of assembly instructions is as follows:

```
asm <func prototype>
{
asm_header
       optional arg binding
       optional return value
       optional read list
       optional write list
       optional modified reg list
asm_body
       <asm code>
asm_end
}

optional arg binding
       .arg
              <ident> in <reg>;
              <ident> in <reg>;
              ...
optional return value
       return in <reg>

optional read list:
       .read <ident>,<ident>,...;

optional write list:
       .write <ident>,<ident>,...;

optional modified reg list:
       .reg <reg>, <reg>, ...;
```

The following syntax conventions apply:

- Identifiers must have the prefix _ (underscore).
- Registers must have the prefix $ (dollar sign).
- Labels must have the suffix . (period).

Example 4-2 shows the syntax for an inlined assembly function that takes two arguments as input parameters and returns one value. The first argument is passed in the register `d0`, and the second parameter is passed in the register `r1`. The result is returned in `d0`.

Example 4-2. Inlining syntax

```
asm int t6( int param1, int *param2)
{
asm_header
.arg
      _param1 in $d0;
      _param2 in $r1;
return in $d0;
.reg $d0,$d1,$r1;
asm_body
      move.l (r1),d1
      add    d0,d1,d0
asm_end
}
```

In Example 4-3, the function `t6` accepts two parameters, an integer `p1` passed in register `d14,` and a pointer `p2` passed in `r7`. The result of the function is returned in `d14`.

Example 4-3. Simple inlined assembly function

```
#include <stdio.h>
int A[10] = {1,2,3,4,5,6,7,8,9,0};
asm int t6(int p1, int *p2)
{
asm_header
.arg
      _p1 in $d14;
      _p2 in $r7;
return in $d14;
.reg $d14,$d1,$r7;
asm_body
      move.l (r7),d1
      add    d14,d1,d14
asm_end
}
int main()
{
      int k = 8;
      int s;

      s = t6(k,&A[3]);

      printf("S= %d\n",s);

return s;
}
```

Example 4-4 shows the use of labels and hardware loops within inlined assembly functions. You should use hardware loops within assembly functions only if you know that the loop nesting is legal. In this example, the function is called from outside a loop, and the use of hardware loops is therefore allowed.

Example 4-4. Inlined assembly function with labels and hardware loops

```
#include <stdio.h>

char sample[10] = {9,6,7,1,0,5,1,8,2,6};

int status;

asm char t7(int p)
{
 asm_header
.arg
       _p in $d7;
       return in $d8;
       .reg $d7,$d8,$r1;
asm_body
       clr d8
       move.l #_sample,r1
       doen3  d7
       dosetup3 _L10

       loopstart3
_L10:  move.b (r1),d1
       add d8,d1,d8
       inc d1
       move.b d1,(r1)+
       loopend3

asm_end
}

int main()
{
       int m=8;
       int s,i;
       for(i=0;i < 10;i++)
       {
              sample[i] *= 2;
              printf("%d ",sample[i]);
       }
       printf("\n");
       s = (int)t7(m);
       printf("S= %d\n",s);

       for(i=0;i < 10;i++)
              printf("%d ",sample[i]);
       printf("\n");
       return 1;
}
```

Example 4-5 shows how global variables are referenced within an inlined assembly function. Global variables are accessed using their linkage name, which is by default the variable name prefixed by the character _ (underscore). The variables `vector1` and `vector2` are therefore accessed within the function as `_vector1` and `_vector2` respectively.

Example 4-5. Referencing global variables in an inlined assembly function

```
#include <stdio.h>

short vector1[] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
int vector2[] = {11,12,13,14,15,16,17,18,19,1,2,3,4,5,6};
short result_1=0;
int result_2=0;

asm void test(int n, short *r1,int *r2)
{
 asm_header
.arg
   _n  in $r1;
   _r1 in $r3;
   _r2 in $r7;
.reg $d0,$r1,$r6,$r11,$r3,$r7;

asm_body
  move.l        #_vector1,r6
  move.l        #_vector2,r11
  addl1a        r1,r6
  addl2a        r1,r11
  move.w        (r6),d0
  asrr          #<2,d0
  move.w        d0,(r3) move.l   (r11),d1
  asl           d1,d2
  move.l        d2,(r7)
asm_end
}

int main(void)
{
       test(12,&result_1,&result_2);
       printf("Status = %d %d\n",(int)result_1, result_2);
       return (int)result_2;
}
```

4.3 Calling an Assembly Function in a Separate File

The compiler supports calls to assembly functions that are contained in separate files, and enables you to integrate these files with your C application.

To include a call to an assembly function in your program, follow the steps described below:

1. Write the assembly function in a separate file from your C source files. Use the standard calling conventions, as described in Chapter 6, “Runtime Environment.”
2. Assemble the file, if required. This step is optional.
3. In your C source file, define the assembly function as an external function.
4. Specify both the C source file and the assembly file as input files in the shell command line to integrate the files during compilation.

The following examples show how a segment of C code calls a function that performs an FFT algorithm implemented in assembly.

4.3.1 Writing the Assembly Code

Example 4-6 shows the assembly code for the FFT algorithm, in the file `fft.sl`.

Example 4-6. Assembly function in a separate file

```
;
; extern void fft(short *, short*);
;
; Parameters:  pointer to input buffer in r0
; pointer to output buffer in r1
;
_fft:
   push d6 push d7 ;Save and restore d6, d7, r6, r7, according to push r6 push r7
                   ;calling conventions.

   implementation of FFT algorithm >

      pop r6  pop r7
      pop d6  pop d7
      rts
```

4.3.2 Calling the Assembly Function

The C code that calls the FFT function is shown in Example 4-7. This source code is saved in the file `test_fft.c`.

Example 4-7. C code calling assembly function

```
#include <stdio.h>
extern void fft(short *, short*);
#pragma external fft

short in_block[512];
short out_block[512];
int in_block_length, out_block_length;

void main()
{
       int i;
       FILE *fp;
       int status;
       in_block_length=512;
       out_block_length=512;
       fp=fopen("in.dat","rb");

       if( fp== 0 )
       {
              printf("Can't open parameter file: input_file.dat\n");
              exit(-1);
       }

       printf("Processing function fft \n");

       while ((status=fread(in_block, sizeof(short), in_block_length, fp)) ==
       in_block_length)
       {
              fft(in_block,out_block);
       }
}
```

4.3.3 Integrating the C and Assembly Files

Example 4-8 shows how the two input files are specified in the shell command line:

Example 4-8. Integrating C and assembly files

```
scc -o test_fft.eld test_fft.c fft.sl
```

4.4 Including Offset Labels in the Output File

In some cases when assembly functions are called, data structures need to be shared between the C source code and the assembly code. In the following example, the layout of the structure `complex` needs to be used by the assembly code.

Example 4-9. Data structure shared between C and assembly

```
struct complex
{
    short r;
    short i;
};

struct complex CVEC1, CVEC2;
volatile struct complex res;

void main()
{
    cmpy (&CVEC1, &CVEC2, &res);
}
```

The `-do` option in the shell command line instructs the compiler to include the details of C data structures in the output assembly file. You can specify this as an additional option in the command line, as shown in Example 4-10:

Example 4-10. Specifying the output of offset information

```
scc -o test.eld test.c cmpy.sl -do
```

When the `-do` option is specified, the output file shows the offsets for all field definitions in each data structure defined in the C source code. The symbolic label is composed of:
`<module name>_<structure name>_<field name>`, as shown in the following example:

Example 4-11. Data structure offsets in the assembly output file

```
test_complex_r    equ         0
test_complex_i    equ         2
```

The symbolic labels in the output file can be used in the assembly code, making the code more readable, as illustrated in Example 4-12. Using these symbolic labels also makes maintenance of the assembly code easier when changes are made to the C code.

Example 4-12. Using symbolic offsets in assembly code

```
;=======================================================
; Function cmpy
;
; Parameter x        passed in r0
; Parameter y        passed in r1
; Parameter result   passed in (sp-12)
;=======================================================

        section.txt
        global  _cmpy
        align 2

_cmpy
          [
          move.2f  (r0),d0:d1
          move.2f  (r1),d2:d3
          ]

          [
          mpy d0,d2,d5
          mpy d0,d3,d7
          [
          ]
          macr -d1,d3,d5
          macr d1,d2,d7
          move.l (sp-12),r2
          ]
          rtsd
          moves.f d5,(r2+test_complex_r)   moves.f  d7,(r2+test_complex_i)

          endsec
```

Chapter 5
Optimization Techniques and Hints

This chapter explains how the SC100 optimizer operates, and describes the optimization levels and individual optimizations which can be applied. The following sections are included in this chapter:

- Section 5.1, “Optimizer Overview,” provides a general description of the optimizer, illustrates how the optimizer transforms code, and outlines the available optimization levels, modes and options.
- Section 5.2, “Using the Optimizer,” explains how to invoke the optimizer, and how to achieve the required results for your application.
- Section 5.3, “Optimization Types and Functions,” describes the individual optimizations in detail.
- Section 5.4, “Guidelines for Using the Optimizer,” provides advice on ways to write source code that can make the best use of the optimizer’s capabilities.
- Section 5.5, “Optimizer Assumptions,” describes the assumptions made by the optimizer in different circumstances.

5.1 Optimizer Overview

The SC100 optimizer converts preprocessed source files into assembly output code, applying a range of code transformations which can significantly improve the efficiency of the executable program. The goal of the optimizer is to produce output code which is functionally equivalent to the original source code, while improving its performance in terms of execution time and/or code size.

5.1.1 Basic Blocks

The majority of the code transformations operate on basic blocks of code. A basic block of code is a linear sequence of instructions for which there is only one entry point and one exit point. There are no branches in a basic block. In general, bigger basic blocks enable better optimization, since the scope for further optimization is increased.

5.1.2 Linear and Parallelized Code

The optimizer can produce code that takes full advantage of the multiple execution units provided by the SC100 architecture.

Executable programs process instructions in the form of execution sets, with one execution set per cycle. The optimizer can increase the number of instructions in an execution set, enabling two or more execution units to process instructions in parallel, in the same cycle. In this way, linear code is transformed into parallelized code:

- **Linear code** uses only one execution unit, regardless of the number of units available. Each execution set consists of one instruction only.
- **Parallelized code** execution sets can comprise multiple instructions which execute in parallel using the available number of execution units. Parallelized code executes faster and more efficiently than linear code.

Figure 5-1 illustrates the transformation of linear code, comprising a series of single instruction execution sets, into parallelized code, which consists of execution sets containing one or more instructions each:

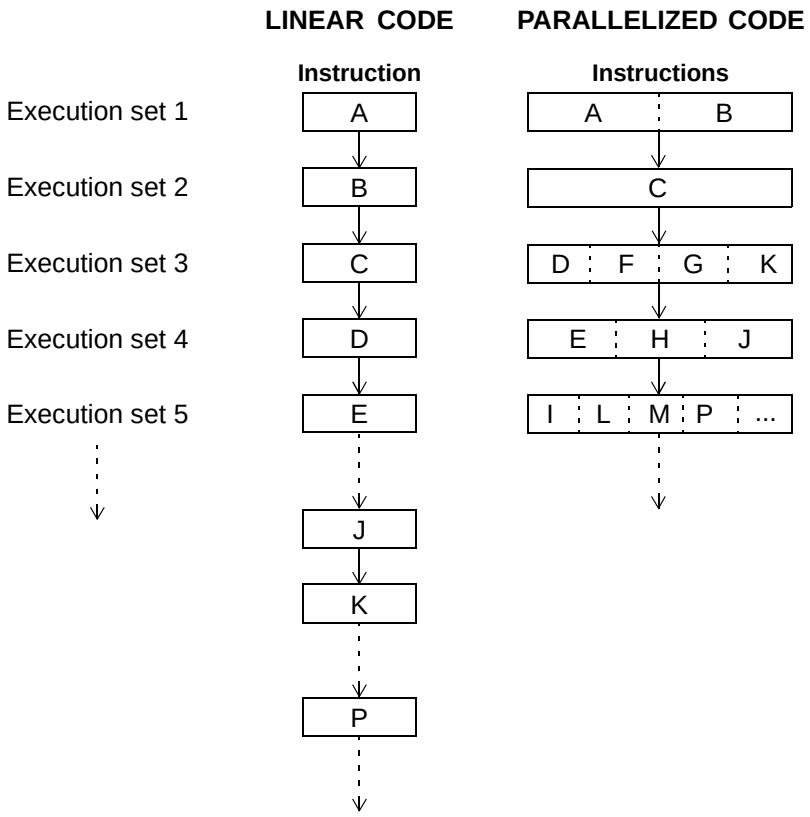


Figure 5-1. Linear and Parallelized Code

Dependencies between instructions can restrict the level of parallelization that the optimizer can achieve. For more information, see Section 5.3.1, “Dependencies and Parallelization,” and Section 5.4, “Guidelines for Using the Optimizer.”

5.1.3 Optimization Levels and Options

Three basic optimization levels are provided, all of which maintain a balance between code density and speed:

Table 5-1. Optimization Levels

Optimization Levels	Description
Level 0	compiles the fastest and produces the slowest output as linear code. Level 0 produces unoptimized code.
Level 1	takes longer to compile, applies target-independent optimizations, and produces optimized linear code.
Level 2	(the default) compiles more slowly than Level 1, applies all target-independent optimizations, as well as all target-specific optimizations, and can produce faster, parallelized code

Select only one of the above optimization levels for each compilation.

Two supplemental optimizations are available that you can combine with Level 1 or Level 2 optimization:

- Space optimization enables you to apply the indicated level of optimization, while weighting the optimization process in favor of program size. Programs or modules optimized for space require a smaller amount of memory but may sacrifice program speed.
- Cross-file optimization is a complex process, which requires significantly more compilation time than non-cross file optimization. With cross-file optimization, the optimizer applies the required level of optimization across all the files in the application at the same time, and as a result produces the most efficient program code.

  Cross-file optimization is generally applied at the end of the development cycle, after all source files are compiled and optimized individually or in groups. By default, the optimizer operates without cross-file optimization.

Table 5-2 below summarizes the optimization options. This table also includes a cross-reference to the sections containing detailed descriptions of the individual optimizations applied by each of the options.

Table 5-2. Optimization Options Summary

Option	Description	Benefits	Section	Page
`-O0` (Level 0)	• Disables all optimizations. • Outputs non-optimized, linear assembly code.	• Compiles fastest. • Generates assembly code which correlates clearly with the C source code, and can assist debugging.		
`-O1` (Level 1)	• Performs all target-independent (non-parallelized) optimizations, such as function inlining. • Omits all target-specific optimization steps. • Outputs optimized, linear code.	• Compiles faster than option `-O2` (the default). • Produces faster programs than option `-O0.`	5.3.2	5-9
`-O2` (Level 2) (Default)	• Performs all optimizations. • Outputs optimized, non-linear assembly code.	• Takes advantage of parallel execution units, producing the highest performance code possible without cross-file optimization.	5.3.3	5-20
`-Os` Space Optimization	• Performs the indicated level of optimization, with emphasis on reducing code size. • Can be specified together with any of the other optimization options except `-O0`.	• Produces optimized assembly code which is small.	5.3.4	5-33
`-Og` Cross-file Optimization	• Performs cross-file optimization. • Can be specified together with any of the other optimization options except `-O0`. • Produces the most efficient results when specified together with the `-O2` (default) option. • Compiles significantly slower than the other options.	• Takes advantage of the visibility of all input files to implement the specified optimization level across the entire application. • Produces the fastest runtime code.	5.3.5	5-35
`-no_overflow`	• Tells the compiler that the application does not rely on the ANSI/ISO C defined overflow behavior of operations on unsigned integral data-types.	• Generates more efficient code sequences. • Results in smaller code size. • Produces faster runtime code.	6.5.1	6-25

5.2 Using the Optimizer

By default, the compiler optimizes all source code files using Level 2 optimization without cross-file optimization. You can choose to optimize your source code at the level that you require at each stage of program development, and you can optimize individual sections of the program according to their purpose in the application. For example, you may wish to prepare your application as follows:

- **During initial development stages**: Use the default Level 2 optimization to compile your source code files, individually or in groups. If required, optimize certain sections of the application for maximum speed, and optimize other sections for size, to reduce the memory space they occupy.
- **During final development stages**: Select Level 2 and cross-file optimization, in order to apply all optimizations across the entire application. The compilation is slower, but produces the most effective optimization results.

You select the optimization level and mode to be applied by specifying one or more options in the shell command line, as described below in Section 5.2.1, “Invoking the Optimizer.”

5.2.1 Invoking the Optimizer

The optimizer can be invoked by including the required option(s) in the shell command line or command file, as illustrated in the examples that follow. For more detailed information about the use and syntax of the shell command line, refer to Section 3.2, “Invoking the Shell,” on page 3-9.

The command line shown in Example 5-1 invokes the optimizer with one input source file, and the default optimization settings. The optimizer applies Level 2 optimizations without cross-file optimization.

Example 5-1. Invoking the optimizer with default settings

```
scc -o file.eld file.c
```

Example 5-2 shows how to invoke the optimizer with the Level 1 option, to apply target-independent optimizations only. The optimizer operates without cross-file optimization.

Example 5-2. Invoking the optimizer for target-independent optimizations only

```
scc -O1 -o file.eld file.c
```

The command line shown in Example 5-3 invokes the optimizer in cross-file optimization mode. The optimizer processes all the specified source files together, applying the default Level 2 optimizations to all the modules in the application.

Example 5-3. Invoking the optimizer with cross-file optimization

```
scc -Og -o file.eld file1.c file2.c file3.c
```

5.2.2 Optimizing for Space

Your application, or specific parts of it, may require code that occupies the least possible space in memory. You can optimize the file(s) for space at the expense of program speed.

To activate space optimization, specify the `-Os` option in the shell command line. See Section 5.3.4, "Space Optimizations," for details of the optimization functions for this option.

The `-Os` option generates the smallest code size for the given optimization level. If no optimization level is specified with `-Os`, the `-O2` optimization level is selected by default. All optimizations associated with the given optimization level are applied, except those noted in Section 5.3.4, "Space Optimizations," with the emphasis on functions which reduce code size.

Depending on your application, the best code density might be achieved using `-Og` and `-Os` together along with the appropriate memory model switch for your application. See Section 6.2, "Memory Models," on page 6-5 for details about the memory models.

5.2.3 Using Cross-File Optimization

Once you have optimized your individual source files and groups of files, you can invoke the optimizer in cross-file mode to ensure maximum optimization across the entire application, in order to produce the most efficient code.

With cross-file optimization, all the code in the application is processed by the compiler at the same time. The optimizer has no need to make worst case assumptions since all the necessary information is available. This enables the optimizer to achieve an extremely powerful level of optimization.

Compiling with cross-file optimization entails high consumption of required resources, and has a slow compilation time. In addition, because of the interdependency that cross-file optimization creates between all segments of the application, the entire application needs to be recompiled if any one source code file is changed. For these reasons, cross-file optimization is generally used at the final stage of development.

To receive optimal results using cross-file optimization, follow these rules:

1. You must compile the entire application together.
2. You can only link the Standard C library that is shipped with the Compiler.
3. Assembly functions can only call other assembly functions and library functions

For a graphic representation of how the compiler operates with and without cross-file optimization, see Section 3.2, "Invoking the Shell," on page 3-9.

To activate cross-file optimization, specify the `-Og` option in the shell command line, as shown in Example 5-3 on page 5-6. While you can specify this option together with any of the other optimization level options, cross-file optimization is generally recommended with optimization Level 2. The `-O2` option is the default and may be omitted.

5.3 Optimization Types and Functions

The optimizer implements two main types of optimization:

- Target-independent optimizations improve the output code without taking into account the properties of the target machine. These optimizations are described in detail in Section 5.3.2, "Target-Independent Optimizations."
- Target-specific optimizations achieve code improvements by exploiting the architecture features of the target machine. Section 5.3.3, "Target-Specific Optimizations," provides a description of these optimizations.

Both sets of optimizations can be applied to individual files and groups of files, with or without cross-file optimization. Refer to Section 5.2.3, "Using Cross-File Optimization," and Section 5.3.5, "Cross-File Optimizations," for more information.

Changes in the code as a result of one optimization may enable another optimization to be applied, producing an accumulative effect.

5.3.1 Dependencies and Parallelization

Dependency between instructions directly limits how successfully the optimizer can apply the various optimizations. An instruction is considered to be dependent on another if a change in their order of execution influences the result of the operation.

The optimizer can group instructions into parallelized execution sets only if these instructions do not contain dependencies. For a description of parallelized execution sets, refer to Section 5.1.2, "Linear and Parallelized Code." Parallelization of different parts of the program, or of iterations of the same loop, can significantly increase the speed of the executable application.

Example 5-4 illustrates a simple dependency between two instructions. The value of `d0` is entirely different when the order of these instructions is reversed. These instructions cannot be executed in parallel.

Example 5-4. Simple instruction dependency

```
move.w  #5,d0                   ; Sets register d0 to 5
add     d0,d1,d2                ; Adds the values in d0 and d1 into register d2
```

An example of dependency arising from an algorithm is shown in Example 5-5. The value of the variable `sum` must be calculated before it can be used in the `L_mac` instruction.

Example 5-5. Algorithm instruction dependency

```
sum = mpy(a,b);
result = L_mac(sum,c,d);
```

The optimizer can operate most effectively with code which contains as few dependencies as possible. Section 5.4, "Guidelines for Using the Optimizer," provides more detailed advice for writing code that avoids dependencies and makes the best use of the optimizations.

The sections that follow describe the operation of individual optimizations in detail, and are intended for advanced users of the SC100 C compiler. If you do not require this level of detail, you may wish to skip these sections, and turn directly to Section 5.4, "Guidelines for Using the Optimizer."

5.3.2 Target-Independent Optimizations

In the high-level optimization phase, a number of general, target-independent optimizations are implemented. All target-independent optimizations are applied when either optimization Level 1 (option `-O1`) or the default optimization Level 2 (option `-O2`) is selected.

These target-independent optimizations are summarized in Table 5-3, and examples of each are given in the sections that follow.

For a detailed discussion of the principles behind target-independent optimizations, refer to *Compilers Principles, Techniques, and Tools,* by Aho, Sethi, and Ullman.

Table 5-3. Summary of Target-Independent Optimizations

Optimization	Description	Section	Page
Target-Independent Strength reduction (loop transformations)	Transforms array access patterns and induction variables in loops, and replaces them with pointer accesses	5.3.2.1	5-10
Function inlining	Substitutes a function call with the code of the function	5.3.2.2	5-16
Common subexpression elimination	Replaces an expression with its value if it occurs more than once	5.3.2.3	5-17
Loop invariant code	Moves code outside a loop if its value is unchanged by the loop	5.3.2.4	5-17
Constant folding and propagation	Calculates the value of an expression at compilation time if it contains known static constants	5.3.2.5	5-18
Jump-to-jump elimination	Combines jump instructions	5.3.2.6	5-19
Dead code elimination	Removes code that is never executed	5.3.2.7	5-19
Dead storage/assignment elimination	Removes redundant variables and value assignments	5.3.2.8	5-19

The output from the target-independent optimizations is in the form of linear assembly code.

5.3.2.1 Target-Independent Strength reduction (loop transformations)

The purpose of strength reduction is to increase the effectiveness of the code by transforming operations which are “expensive” in terms of resources, into less expensive, linear operations. For example, addition and subtraction are linear functions which require less operation cycles than multiplication and division.

When an address calculation that contains multiplication is replaced by one containing addition, the amount of resources required by the code is significantly reduced, since addition can be implemented using the complex addressing mode of the Address Generation Unit (AGU). When the multiplication appears within a loop, the benefit of the replacement is further increased.

The strength reduction optimization identifies and transforms induction variables, meaning variables whose successive values form an arithmetic progression, usually within a loop. An example of an induction variable is a subscript which points to the addresses of array elements, and increases with each iteration of the loop. The computation of such a variable can be moved to a position outside the loop to avoid repeated operations, and/or transformed for use with linear operations.

Simple and complex loops and array access patterns are transformed where possible into simpler, linear forms, as described in the sections that follow.

5.3.2.1.1 Simple loops

Example 5-6 shows the generated pseudocode and output assembly code for a simple loop which initializes an array. The loop structure is static, meaning that its induction variables, the loop counter `i` and the array offset `t1`, both increase by increments of known constant values.

Example 5-6. Loop transformation - simple loop

C source code

```
int table[100];
step = 1;

for(i=0; i<100; i+=step)
       table[i] = 0;
```

Pseudocode before optimization

```
       i = 0;
L1     t1 = i * 4;
       table[t1] = 0;
       i++;
       if(i<100) goto L1
```

Pseudocode after optimization

```
       i = 0;
       t1 = i * 4;
L1     table[t1] = 0;
       t1 = t1 + 4;
       i++;
       if(i<100) goto L1
```

Assembly code output

```
  move.l #_table,r0   clr d2
  loopstart3
       move.l  d2,(r0)+
  loopend3
```

Before optimization, the calculation of the value of `t1` is within the loop, and is incremented by multiplication. After optimization, the initial value of `t1` is set outside the loop, and its value is incremented inside the loop by addition. The resulting values are identical for both forms, but in the optimized version the resource overhead is considerably lower.

The same principles also apply to more complex loop structures and array access patterns, as described in the sections that follow:

- Dynamic loops, in which increments are based on a variable whose value is not known at compilation time
- Multi-step loops, in which the loop iterator increments more than once in each iteration of the loop
- Composed variable loops, in which one or more variables or iterators are linked to each other in a linear relationship
- Square loops, which access elements in a two-dimensional array as in a matrix, on a row-by-row basis
- Triangular loops, which are similar to square loops, but which access each row in the matrix from an incremented starting position in each subsequent row

5.3.2.1.2 Dynamic loops

In a dynamic loop, one or more increments are based on variables whose values are not known at compilation time.

Example 5-7 shows the generated code for a dynamic loop in which the value of the loop increment and its upper limit are not known at the time of compilation. The optimization removes the initial multiplication instruction from the body of the loop, and inside the loop the multiplication increment instruction is replaced by an addition instruction.

Example 5-7. Loop transformation - dynamic loop

C source code

```
step = step_table[1];
for(i=0; i<MAX; i+=step)
      table[i] = 0;
```

Pseudocode before optimization

```
      step = step_table[1];
      i = 0;
L1    t1 = i * 2;
      table[t1] = 0;
      i = i + step;
      if(i<MAX) goto L1
```

Pseudocode after optimization

```
      i = 0;
      step = step_table[1];
      t1 = i * 2;
      t2 = step * 2;
L1    table[t1] = 0;
      t1 = t1 + t2;
      i = i + step;
      if(i<MAX) goto L1
```

Assembly code output

```
L2
      clr    d3          add         d1,d0,d1
      move.l d3,(r1)     cmpge.w     #100,d1
      adda   r2,r1
      jf     L2
```

5.3.2.1.3 Multi-step loops

Loops in which the loop iterator increments more than once in each iteration of the loop are defined as multi-step loops.

In the multi-step loop shown in Example 5-8, the loop iterator `i` increments twice within the loop. In this case, `i` is transformed into an induction variable which increments in linear progression in three stages.

Example 5-8. Loop transformation - multi-step loop

C source code

```
int table[10];
for(i=0; i<10; i++)
{
      table[i] = i;
      i++;
      table[i] = 0;
}
```

Pseudocode before optimization

```
      i = 0;
L1    t1 = i * 2;
      table[t1] = i;
      i = i + 1;
      t2 = i * 2;
      table[t2] = i;
      i= i + 1;
      if(i<10) goto L1
```

Pseudocode after optimization

```
i = 0;
t1 = i * 2;
t2 = i * 2 + 2;
t3 = i;
Repeat 10 times:
      table[t1] = t3;
      table[t2] = 0;
      t1 = t1 + 4;
      t2 = t2 + 4;
      t3 = t3 + 2;
```

Assembly code output

```
      loopstart3
L93

      move.l      d0,(r0)+n3          add    #<2,d0

      move.l      d2,(r1)+n3
      loopend3
```

5.3.2.1.4 Composed variable loops

A composed variable loop incorporates one or more variables or iterators which have a linear relationship between them. The loop transformation optimizes such loops by moving the multiplication instruction to a position outside the loop, and by substituting one of the variables with a constant.

This optimization can be applied only when the variables are linked by linear arithmetic functions, meaning those calculations involving addition or subtraction of the variables, or multiplication of a variable by a constant. Functions which include non-linear operations, such as multiplication of two induction variables, cannot be optimized in this way.

Example 5-9 illustrates the generated code for a composed variables loop. In this example the increment is the result of a linear calculation using the two induction variables `i` and `j`.

Example 5-9. Loop transformation - composed variables

C source code

```
int table[100];
for(i=0, j=0; i<10; i++)
{
   table[10 * i + j] = i;
   j++;
}
```

Pseudocode before optimization

```
   i = 0;
   j = 0;
   t1 = i * 10;
L1 t2 = t1 + j;
   t3 = t2 * 2; /* address */
   table[t3] = i;
   i = i + 1;
   t1 = t1 + 10;
   j = j + 1;
   if(j < 10) goto L1
```

Pseudocode after optimization

```
i = 0; j = 0;
t1 = 1 * 10;
t2 = t1 + j;
t3 = t2 * 2;
Repeat 10 times:
   table[t3] = i;
   i = i + 1;
   t3 = t3 + 22;
```

Assembly code output

```
   loopstart3
L93
   move.l    d0,(r0)+n3   inc    d0
   loopend3
```

5.3.2.1.5 Square loops

A square loop is a two-dimensional array access pattern which is similar to a matrix in which cells are accessed horizontally in rows, starting at the first cell in each row.

The code that is initially generated for a square loop uses a doubly-nested loop with two induction variables. These variables are incremented by multiplication, as the loop progresses through the array elements in each row, and at the start of each new row, as shown in Figure 5-2.

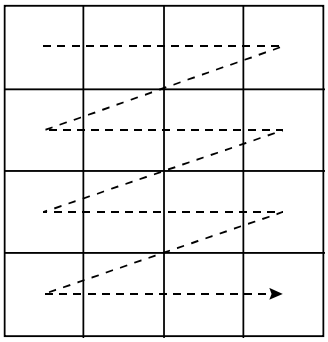


Figure 5-2. Square Loop

The loop transformation changes such a two-dimensional array into one row containing all the elements in one straight string. The multiplication instructions are replaced by additions, as the progression can now be performed on a linear basis. An example of the transformation of a square loop is shown below in Example 5-10.

Example 5-10. Loop transformation - square loop

C source code

```
int table[70][70];
int i, j;
for(i=0; i<35; i++)
   for(j=0; j<70; j++)
      c+=table[i][j];
```

Pseudocode before optimization

```
   i = 0;
L1 j = 0;
L2    tmp1 = i * 140;
      tmp2 = j * 2;
      tmp3 = tmp1 + tmp2;
      tmp4 = table[tmp3];
      c = c + tmp4;
      j++;
      if(j < 70) goto L2
   i++;
   if(i<35) goto L1
```

Pseudocode after optimization

```
tmp1 = 0;
Repeat 35 times
   tmp2 = table + tmp1;
    /* row base address */
   Repeat 70 times
      tmp3 = *tmp2;
       /* pointer to cell */
      c = c + tmp3;
      tmp2 = tmp2 + 2;
   tmp1 = tmp1 + 140;
    /* next matrix row */
```

Assembly code output

```
   loopstart2
L98
   doensh3   d0
   loopstart3
L97
   move.l    d1,(r0)+
   loopend3
L94
   loopend2
```

5.3.2.1.6 Triangular loops

A triangular loop array access pattern is similar to the square loop described above, except that the pointer moves to an incremented starting position in each row. The starting position pointer increments by linear progression, as shown in Figure 5-3:

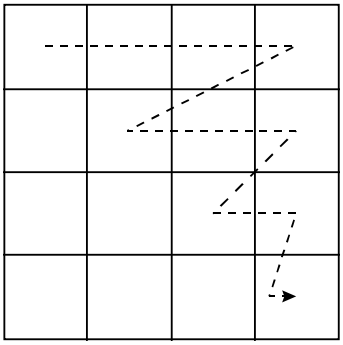


Figure 5-3. Triangular Loop

A triangular loop is transformed into a mainly linear based loop, incorporating the offset increment as an addition operation. Example 5-11 illustrates the transformation of a triangular loop.

Example 5-11. Loop transformation - triangular loop

C source code

```
int table[70][70];
int i, j;
for(i=0; i<70; i++)
   for(j=i+3; j<70; j++)
   table[i][j] = 0;
```

Pseudocode before optimization

```
   i = 0;
L1 j = i
   if(j>=70) goto L3
L2    tmp1 = i * 140;
      tmp2 = j * 2;
      tmp3 = tmp1 + tmp2;
      table[tmp3] = 0;
      j++;
      if(j < 70) goto L2
L3 i++;
   if(i<70) goto L1
```

Pseudocode after optimization

```
i = 0;
tmp1 = 0;
Repeat 70 times
   j = 1 + 3;
   if(j>=70) goto L3
   tmp2 = j * 2;
   tmp3 = tmp1 + tmp2; /* offset */
   tmp4 = table + tmp3;
      /* base + offset */
   Repeat (70-j)
      *tmp4 = 0;
      *tmp4 = tmp4 + 2;
L3 i++;
   tmp1 = tmp1 + 140;
      /* next matrix row */
```

Example 5-11. Loop transformation - triangular loop

Assembly code output

```
   loopstart2
L98
   cmpgt     d0,d2
   jf        L3
   tfr       d0,d4  sub    d0,d2,d3
   doensh3   d3
   asll      #<2,d4
   add       d4,d1,d5
   move.l    d5,r1
   adda      #>_table,r1,r0
   loopstart3
L97
   move.l    d6,(r0)+
   loopend3
L3
   add       #280,d1,d1
   inc       d0
   loopend2
```

5.3.2.2 Function inlining

Inlining replaces a call to a function with a copy of the code for the function. In cases where the procedure call and return may be more time-consuming than the function itself, function inlining can significantly increase the speed of the program.

Inlining can decrease code size by removing overhead and enabling optimization opportunities.The function inlining optimization is particularly effective with cross-file optimization, as the inlining can be applied across all available files; thus functions that were inlined every where can be deleted.

The inlining heuristics were tuned to deal with code size and performance. At the command line, type `-Os` to inline for code size; otherwise, the compiler performs performance inlining. The following table illustrates the conditions for performance inlining and for code size inlining.

	-Os on	**-Os off**
-Og on	code size inlining occurs	performance inlining occurs
-Og off	limited code size inlining occurs (code size inlining only occurs for static functions with `-Os` on	performance inlining occurs

The following example shows how the operation executed by the function `Check` is incorporated into the code itself, removing the call to the function.

Example 5-12. Function inlining

Before optimization

```
int Check(int x);
{
   return (x>10);
}
void main()
{
   if (Check(y))
      a = 5;
}
```

After optimization

```
void main()
{
   if (y>10)
      a = 5;
}
```

You can force or suppress function inlining at specific points in the code, using the pragmas `#pragma inline` and `#pragma noinline`. Refer to Section 3.4.5, “Pragmas,” on page 3-52, for further details.

5.3.2.3 Common subexpression elimination

Where an expression appears in more than one place in the code and has the same computed value in each instance, this optimization replaces the expression itself with its result. Values loaded from memory can be included in this process, as well as values based on arithmetic computations. In the example shown below, the variable `x` replaces the repeated subexpression `e + f`.

Example 5-13. Common subexpression elimination

Before optimization

```
d = e + f + g;
y = e + f + z;
```

After optimization

```
x = e + f;
d = x + g;
y = x + z;
```

5.3.2.4 Loop invariant code

The term “invariant code” refers to an instruction which appears inside a loop, but whose value is not directly affected by the execution of the loop. This optimization moves such an instruction to a position outside the loop, with the result that the instruction is not repeated each time the loop executes. In Example 5-14, the variable `z` is set to the computed value of `2 * b + 1` before the loop executes, and this calculation is removed from the iteration.

Example 5-14. Loop invariant code motion

Before optimization

```
b = c;
for(i=0; i<3; i++)
   d[i] = 2 * b + 1;
```

After optimization

```
b = c;
z= 2 * b + 1;
for(i=0; i<3;i++)
   d[i] = z;
```

5.3.2.5 Constant folding and propagation

This optimization identifies expressions which contain `int` values known to be constants and calculates their value at compilation time. The value of the expression then replaces the expression itself, as shown in Example 5-15 below.

Example 5-15. Constant folding and propagation

Before optimization	After optimization
`X = 2;`	`X = 2;`
`Y = X + 10;`	`Y = 12;`
`Z = 2 * Y;`	`Z = 24;`

5.3.2.6 Jump-to-jump elimination

This optimization combines two jump operations into one, in cases where the code executes a jump to an address, and at that address immediately jumps to a different address.

In Example 5-16, the two jump instructions `goto J1;` and `goto J2;` are replaced by a direct jump to `J2`.

Example 5-16. Jump-to-jump elimination

Before optimization

```
if(x)
...
else
   goto J1;

J1:
   goto J2;
```

After optimization

```
if(x)
...
else
   goto J2;
```

5.3.2.7 Dead code elimination

This optimization removes segments of “dead” code, meaning code that cannot possibly be executed. The code may be dead from the start, or it may become dead as a result of other optimizations. For example, the code may specify a condition which can never be true. In the example shown below, the variable `c` is type `char`, which can never have a value greater than 255, and therefore the `if` condition will never be met.

Example 5-17. Dead code elimination

Before optimization

```
char c;
if c > 300
   a = 1;
else
   a = 2;
```

After optimization

```
a = 2;
```

5.3.2.8 Dead storage/assignment elimination

Dead storage or assignment occurs when a variable is assigned a value, either directly or as a result of an expression, and is not used again anywhere in the code, or receives another value before being used. This optimization removes any unnecessary instructions and unused memory locations which may result from such cases. This redundancy may arise as a result of other optimizations.

In Example 5-18, before optimization the variable `a` is assigned the value 5, and is not used before it is reassigned the value 7. The dead storage/assignment elimination optimization removes the redundant instruction `a = 5`. If the variable `a` was not used at all after being assigned a value, it would be removed completely.

Example 5-18. Dead storage/assignment elimination

Before optimization

```
a = 5;
..
a = 7;
```

After optimization

```
a = 7;
```

5.3.3 Target-Specific Optimizations

The Low-Level Transformations (LLT) phase is a separate modular stage of the optimization process which implements a number of target-specific optimizations. This phase transforms the linear code generated by the target-independent optimization phase into parallel assembly code, which can take advantage of the parallel execution units of the SC100 architecture.

The degree of parallelization that the optimizer is able to achieve is limited by the number and type of dependencies within the source code. See Section 5.3.1, “Dependencies and Parallelization,” for a summary of these issues.

Section 5.4, “Guidelines for Using the Optimizer,” provides detailed advice about preparing your source code with a view to reducing dependencies and realizing the maximum potential for optimization.

All target-specific optimizations are applied when the Level 2 optimization (option -O2) is selected. Target-specific optimizations are not activated at all when either option -O0 or option -O1 is selected.

The major target-specific optimizations are summarized in Table 5-4, and examples of each are given in the sections that follow.

Table 5-4. Summary of Target-Specific Optimizations

Optimization	Description	Section	Page
Instruction scheduling	Executes multiple instructions in the same cycle, fills delay slots associated with a branch operation, and avoids pipeline restrictions	5.3.3.1	5-22
Target-specific software pipelining	Rearranges instructions in a loop to minimize dependencies	5.3.3.2	5-23
Conditional execution and predication	Transforms a branch into a sequence of conditional actions	5.3.3.3	5-26
Speculative execution	Moves instructions from conditional to unconditional paths	5.3.3.4	5-27
Post-increment detection	Combines the functions of incrementing (or decrementing) a pointer and accessing the computed address into one instruction	5.3.3.5	5-28
Target-specific peephole optimization	Merges a sequence of instructions into a single instruction	5.3.3.6	5-29
Extract peephole optimization	Replaces multiple instructions and cycles with one word or one cycle (combines AND instructions)	5.3.3.7	5-30
Multiply strength reduction	Replaces integer multiplies by constants with combinations of ASRs and ASLLs	5.3.3.8.1	5-32

The optimizer applies the target-specific optimizations in a predefined sequence, and invokes some of the optimizations more than once, as illustrated in Figure 5-4. Each optimization is directly affected by the result of the preceding optimization.

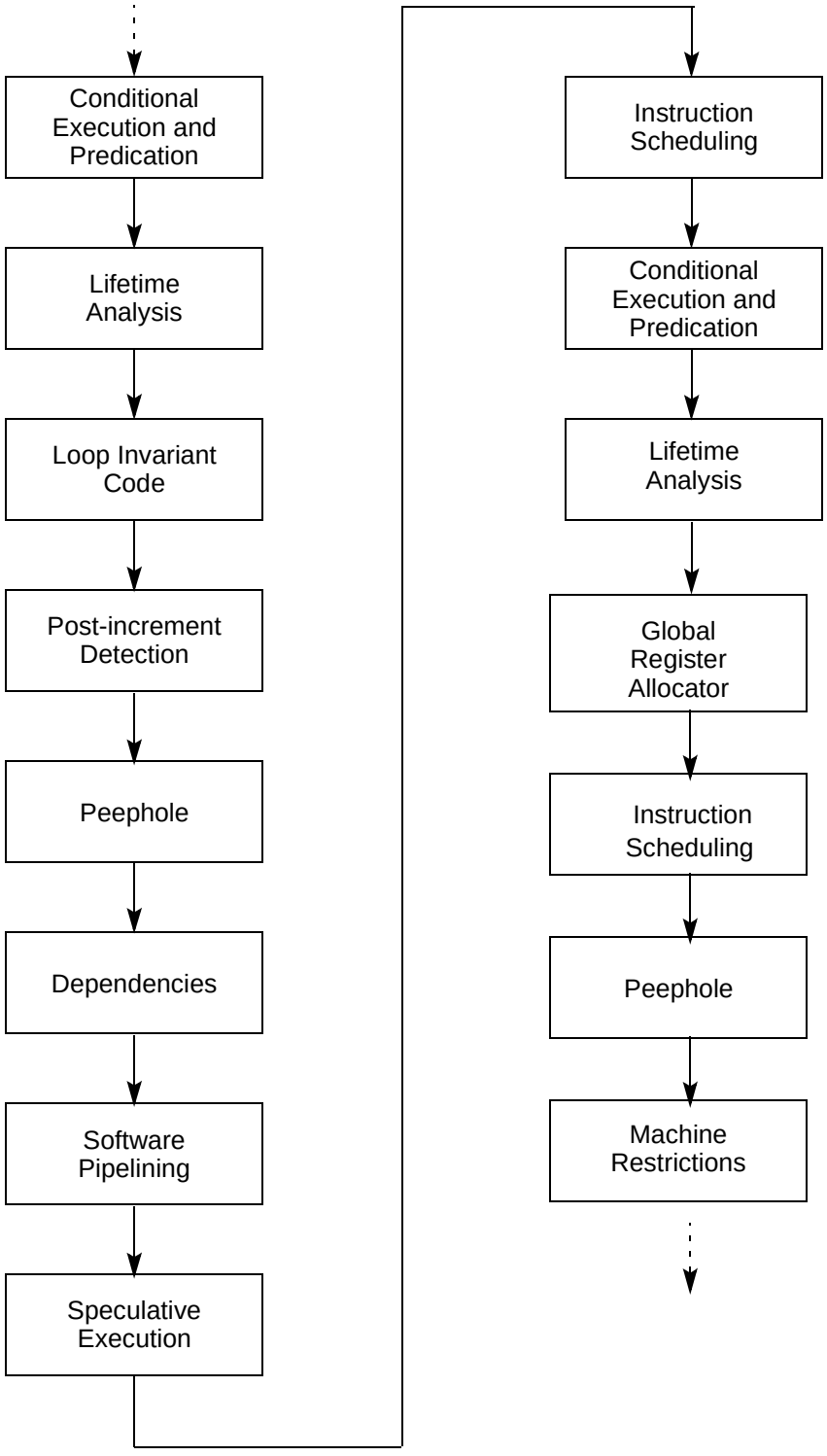


Figure 5-4. Sequence of Target-Specific Transformation Optimizations

5.3.3.1 Instruction scheduling

The main purpose of this optimization is to execute as many instructions as possible from the same instruction stream in the same cycle. The amount of dependency between the instructions limits the extent to which this can be achieved.

The instruction scheduling optimization organizes instructions into execution sets wherever it is possible to do so, making best use of the Data Arithmetic Units and Address Generation Units provided by the SC100 architecture.

Example 5-19 illustrates the use of instruction scheduling:

Example 5-19. Instruction scheduling

Before optimization

```
move.l d0,(r0)
inc    d0
tfra   r3,r0
adda   #12,r3
move   (r1)+,d1
```

After optimization

```
move.l d0,(r0)   inc    d0
tfra   r3,r0     move   (r1)+,d1
adda   #12,r3
```

Instruction scheduling serves two further purposes:

- Filling delay slots when branch instructions are executed, as described below
- Rescheduling operations that are not dependent on pipeline restricted instructions, as described in Section 5.3.3.1.2, “Avoiding pipeline restrictions,” on page 5-23

5.3.3.1.1 Filling delay slots

A branch instruction requires three cycles to execute if the branch is taken. When a branch executes, the prefetch queue is lost, and the cycles used for the other instructions are wasted, since they cannot execute until the branch instruction has completed. The wasted cycles are termed *delay slots*.

The instruction scheduling optimization checks whether other operations can be executed at the same time as the branch instruction. This is not possible if there are limiting factors, for example:

- The branch instruction is directly affected by the instructions which precede it.
- There are specific dependencies between the branch and the other instructions.

If there are no limiting factors, the scheduler rearranges the code, in order to use the delay slots efficiently. In the following example, the code has been reorganized to enable three instructions to execute during the time that the branch requires to complete its operation.

Example 5-20. Filling delay slots

Before optimization

```
move.l d0,(r0)
inc    d0
tfr    d5,d2
rts
```

After optimization

```
rtsd
move.l    d0,(r0)      inc    d0     tfr  d5,d2

move.1    d0,(r0)      inc    d0     tfr d5, d2
```

5.3.3.1.2 Avoiding pipeline restrictions

Certain instructions, for example, a move to an Rn register, are subject to pipeline restrictions. The effect of these instructions may not be implemented until two or more cycles after the instruction executes. In such cases, an operation which is dependent on the result of such an instruction, and which follows it immediately, must wait until the result is available.

The instruction scheduling optimization rearranges the sequence of such instructions where possible, using the cycle(s) which would otherwise be wasted to implement one or more operations that are not dependent on the restricted instruction.

In Example 5-21, the `clr` instruction has been rescheduled, since it can execute before the effect of the `move.l` instruction is implemented, whereas the `move.w` instruction must wait for the results of the `move.l` operation.

Example 5-21. Avoiding pipeline restrictions

Before optimization	After optimization
`move.l d0,(r0)`	`move.l d0,(r0)`
`nop`	`clr d0`
`move.w (r0),dl`	`move.w (r0),dl`
`clr d0`	

5.3.3.2 Target-specific software pipelining

Software pipelining provides a further level of loop optimization, in addition to the target-independent optimizations which operate on loops.

The software pipelining optimization attempts to rearrange the sequence of instructions inside a loop, in order to minimize dependencies between such instructions, and thus increase the level of parallelization.

For example, a segment of code may consist of three instructions, A, B and C, within a loop which iterates 4 times. In some cases, the code may be reorganized into a different sequence without affecting its result, for example:

1. Instruction A
2. Instructions B, C, A, in a loop which iterates 3 times
3. Instruction B
4. Instruction C

The revised arrangement of the instructions results in fewer dependencies than in the original code.

This optimization is applied only to innermost loops of small or moderate size, which contain no branches or function calls within the loop. It is most effective when applied to loops that execute a large number of times.

Each iteration of a software pipelined loop may contain instructions from a different iteration of the original loop.

Software pipelining increases code size in almost all circumstances. When optimization for size is specified, software pipelining is suppressed entirely.

The following example shows how the software pipelining optimization reduces the number of iterations and rearranges instructions both within and outside the loop, thus enabling the maximum number of instructions that are not dependent on each other to execute in parallel.

Example 5-22. Software pipelining - complex FIR

C source code

```
for (i = 0; i < N; i++)
{

L_tmpr = L_mac (L_tmpr, sample[i].r, coeff[N - i - 1].r);
L_tmpr = L_msu (L_tmpr, sample[i].i, coeff[N - i - 1].i);

L_tmpi = L_mac (L_tmpi, sample[i].i, coeff[N - i - 1].r);
L_tmpi = L_mac (L_tmpi, sample[i].r, coeff[N - i - 1].i);

}
```

Before optimization

```
loop n times:
   move.w (r0)+,d4
   move.w (r1)-,d3
   mac    d3,d4,d5
   move.w (r0)+,d1
   move.w (r1)-,d2
   mac    -d1,d2,d5
   mac    d3,d1,d6
   mac    d2,d4,d6
```

After optimization

```
/* Prolog */
move.w (r0)+,d4     move.w    (r1)-,d3
mac    d3,d4,d5     move.w    (r0)+,d1     move.w(r1)-,d2

loop n-1 times:
 /*start loop*/
   [
      mac          d3,d1,d6            mac       -d1,d2,d5
      move.w       (r0)+,d4            move.w    (r1)-,d3
   ]
   [
      mac          d3,d4,d5            mac       d2,d4,d6
      move.w       (r0)+,d1            move.w    (r1)-,d2
   ]
/*endloop*/
/* Epilog */
mac    d3,d1,d6     mac    -d1,d2,d5
mac    d2,d4,d6
```

In the following example, the loop iterates only 8 times, instead of the 10 in the original code, since two iterations have been unrolled. The loop executes in a single cycle. During this cycle the loop:

- Loads a value from iteration `i+2`
- Multiplies the value from iteration `i+1`
- Stores the result value from iteration `i`

Example 5-23. Software pipelining - vector multiplication by a constant

C source code

```
for (i=0; i<10; i++)
    b[i] = mult(a[i], 0x4000);
```

Assembly code after optimization

```
   doensh3  #<8 ; Pipelining loop twice
   move.l   #_a,r1
    move.f   #16384,d1
    move.f   (r1)+,d0  move.l #_b,r0
    mpy      d0,d1,d2  move.f (r1)+,d0
    loopstart3

L93
   [
    moves.f  d2,(r0)+
    mpy      d0,d1,d2
    move.f   (r1)+,d0
   ]
    loopend3

L92
    moves.f  d2,(r0)+  mpy d0,d1,d2
    moves.f  d2,(r0)+
```

5.3.3.3 Conditional execution and predication

The conditional execution and predication optimization simplifies small conditional structures and transforms the branch into one sequence.

An example of this transformation is shown in Example 5-24, in which two branches are removed.

Example 5-24. Conditional execution and predication

C source code

```
If(a < 0){
  lower_bound = 0;
  i = 0;
}else
  lower_bound = a;
```

Generated code before optimization

```
      move.w    a,d0
      tstgt     d0
      bf        L_False
      clr       d2
      clr       d3
      bra       L_AfterIf
L_False
      tfr       d0,d2
L_AfterIf
      move.w    d2,lower_bound
```

Generated code after optimization

```
move.w    a,d0
tstgt     d0
ift       clr  d2      clr d3
iff       tfr d0,d2
move.w    d2,lower_bound
```

An additional advantage of this optimization is that it increases the size of the basic blocks in the optimized code segment, making further optimization more effective.

It is important to note, however, that the conditional execution optimization adds one word for each branch that it replaces (`ift` and `iff` in the above example). As a result, the impact on the size of the program can be considerable. Generally, this optimization is only activated for small structures where the number of instructions added is less or equal to the number of instructions saved.

5.3.3.4 Speculative execution

The speculative execution optimization moves instructions from conditional to unconditional paths, in order to fill execution slots that would not otherwise be used.

If an empty execution slot is available when a condition statement is encountered, the instructions are rearranged so that the conditional instructions execute unconditionally in previous cycles to the condition. If the condition is true and the `ift` instruction has been executed, or if the condition is false and the `iff` instruction has been executed, a cycle has been gained. If the condition result does not match the moved instruction, the appropriate instruction is executed as normal, with no loss of cycles.

Example 5-24 shows an example of this transformation. In this example, the first `iff` instruction is moved so that it executes in the same cycle as the `cmpgt` instruction. If the result of the conditional operation is true, the `ift` instruction is executed in the next cycle. If the result is false, the instruction that was previously the second `iff` is executed, with the result that only one cycle is used instead of two.

Example 5-25. Speculative execution

C source code

```
If(var > 5)
  x[3] = a;
else
  y = b;
```

Generated code before optimization

```
cmpgt #5,d1
nop
iff    move.l x+6,r0
iff    move.l d3,_y
ift    move.l d2,(r0)
```

Generated code after optimization

```
move.l x+6,r0    cmpgt #5,d1
nop
iff    move.l d3,_y
ift    move.l d2,(r0)
```

This optimization can be implemented successfully for one or more instructions if:

- Sufficient slots are available.
- There are no dependencies between the instruction in the conditional path and other instructions.
- The conditional instruction does not have any specific side effects.

5.3.3.5 Post-increment detection

This optimization exploits the features of the SC100 architecture, and increases code efficiency in terms of both size and speed. It identifies the instructions which use arithmetic functions to modify pointers, and which access the computed addresses, and replaces them with special post-increment or post-decrement address mode instructions which combine both functions.

The increment (or decrement) factor is not limited to the values 2 or 4, since any one of the four index registers (n0 through n3) may be used, as illustrated in Example 5-26.

Example 5-26. Post-increment detection

Generated code before optimization

```
L150
   move.l  #_L_R,r4
   move.l  #_CGUpdates,r5
   doen3   #<8
   dosetup3        L183
   loopstart3
L183
   move.l  (r4),d0
   move.l  (r5),d1
   mac     d0,d1,d2
   adda    #<4,r4
   adda    #<12,r5
   loopend3
L152
```

Generated code after optimization

```
L150
   doensh3   #<7      ;Pipelining loop once
   move.w    #3,n3
   move.l    #_L_R,r4
   move.l    #_CGUpdates,r5
   move.l    (r4)+,d0
   move.l    (r5)+n3,d1
   loopstart3

L183
   [
    mac      d0,d1,d2
    move.l   (r5)+n3,d1
    move.l   (r4)+,d0
   ]
    loopend3

L152
    mac      d0,d1,d2
```

5.3.3.6 Target-specific peephole optimization

The target-specific peephole optimization identifies sequences of instructions that can be merged into a single instruction, and implements this transformation, as shown in Example 5-27.

Example 5-27. Target-specific peephole optimization

Generated code before optimization

```
        deca        r0
        move.w      #33,d0
        tstgea.l    r0
```

Generated code after optimization

```
        decgea      r0
        move.w      #33,d0
```

Example 5-28 illustrates a combination of pipelining and peephole optimizations. After pipelining, the final `mac` instruction, which has been moved outside the loop, is merged with the `rnd` instruction to form a `macr` instruction.

Example 5-28. Combined pipelining and peephole optimizations

Generated code before optimization

```
        doen        #9
        dosetup0    L1

        loopstart0
L1
        move.w      (r0)+,d3
        move.w      (r1)+,d2
        mac         d2,d3,d7

        loopend0

        rnd         d7
```

Generated code after optimization

```
        doen   #8           ;Pipelining loop once
        dosetup0    L1
        move.w      (r0)+,d3
        move.w      (r1)+,d2
        loopstart0

L1
        mac         d2,d3,d7
        move.w      (r1)+,d2
        move.w      (r0)+,d3
        loopend0

        macr        d2,d3,d7
```

5.3.3.7 Extract peephole optimization

Extract peephole optimization replaces ASR and AND operations with EXTRACT and ASLL operations. The compiler only performs this optimization when the AND operation has one constant operand and that constant operand contains only one string of contiguous ones.

The EXTRACT instruction performs an implicit ASR that places the resulting bits of interest into the LSB positions. Therefore, the compiler may add an ASLL after the EXTRACT/EXTRACTU instruction so that the bits end up in the correct position, as illustrated in example 5-30. The EXTRACT/EXTRACTU instruction can also encompass a ASR that may be present around the original AND instruction, as illustrated in example 5-31.

The optimization needs to use the correct extract instruction, EXTRACT or EXTRACTU. Normally, the EXTRACTU instruction is used. The EXTRACT instruction is used when the original ASR instruction may cause the MSB positions to be set to one (assuming the sign bit is set to 1).

The extract peephole optimization is intelligent and aggressive, cognizant of the code size and cycle implications of the transformation; therefore, the optimizer only applies this optimization when it determines that a benefit is possible. Examples 5-29, 5-30, and 5-31 illustrate when extract peephole optimization is beneficial.

Example 5-29. When the AND constant does not fit in lower or upper 16 bits

Generated code before optimization

```
;;total words = 4 and total cycles = 2

move.l     #$18000,d2       ; 3-word instruction
and        d0,d2            ; 1-word instruction
```

Generated code after optimization

```
;;total words = 3 and total cycles = 2

extractu   #<2,#<15,d0,d2   ; 2-word instruction
asll       #<15,d2          ; 1-word instruction
```

Example 5-30. ASR followed by an AND (no ASLL necessary)

Generated code before optimization

```
;;total words = 3 and total cycles = 2

asrr        #<13,d1             ; 1-word instruction
and         #$e000,d1, d1       ; 2-word instruction
```

Generated code after optimization

```
;;total words = 2 and total cycles = 1
extractu   #<3,#<13,d1,d1   ; 2-word instruction
```

Example 5-31. Using an EXTRACT instead of an EXTRACTU

Generated code before optimization

```
;;total words = 3 and total cycles = 2

and         #$f0000000,d1, d1    ; 2-word instruction
asrr        #<28,d1              ; 1-word instruction
```

Generated code after optimization

```
;;total words = 2 and total cycles = 1

extract  #<4,#<28,d1,d1    ; 2-word instruction
```

5.3.3.8 Target-Specific Strength Reduction

Strength reduction increases the effectiveness of the code by transforming operations that are “expensive” in terms of resources, into less expensive, linear operations. For example, addition and subtraction are linear functions that require less operation cycles than multiplication and division.

When an address calculation that contains multiplication is replaced by one containing addition, the amount of resources required by the code is significantly reduced, since addition can be implemented using the complex addressing mode of the Address Generation Unit (AGU).

5.3.3.8.1 Multiply Strength Reduction

Multiply strength reduction is an enhanced strength reduction for multiplication operations. It occurs when one of the operands in an instruction is a constant. During multiply strength reduction, the compiler determines which combinations of shifts, by power of two, and adds/subtracts can create the equivalent to the constant multiplication. This is an important optimization because it can take seven assembly instructions to implement 32-bit multiplication of integers in SC100 architecture.

5.3.3.9 Prefix grouping

Instruction grouping is applied by the optimizer wherever possible, in order to make best use of the available multiple execution units. In addition to “natural” grouping of instructions, which increases efficiency and does not increase code size, the optimizer can implement prefix grouping. Prefix grouping is a mechanism whereby an additional word is introduced into the code in order to force more than one instruction to execute in the same cycle.

Prefix grouping improves performance in terms of speed, but increases the size of the code. The optimizer activates prefix grouping on the entire code.

5.3.4 Space Optimizations

When you select the `-Os` option, the optimizer aims to produce code that occupies as little memory space as possible for the given optimization level. In certain cases, the reduced memory space may be at the expense of program speed.

The compiler executes all optimizations associated with the specified optimization level, except for those that increase the code size, as noted below:

- For target-independent optimizations, `-Os` specifies the use of inlining heuristics.
- For target-specific optimizations, `-Os` does the following:
  - — Disables software pipelining.
  - — Omits conditional execution.
  - — Uses only serial grouping when encoding assembly instructions, since code size is increased when prefixes are added, as described in Section 5.3.3.9, “Prefix grouping.”

You can use the `-Os` option in combination with any other optimization option except `-O0`. If no optimization level is specified with `-Os`, Level 2 optimization (`-O2` option) is selected by default.

The command line shown in Example 5-32 invokes the optimizer with the default Level 2 optimizations. All target-independent and target-specific optimizations, except those noted above, are applied across all modules in the application.

Example 5-32. Invoking the optimizer for space optimization

```
scc -Os -Og -o file1.eld file1.c file2.c
```

5.3.4.1 Code Sinking Optimization

Code sinking is a space optimization that downsizes replicated code by sinking duplicated operations. It creates basic blocks when necessary, enabling the compiler to apply the optimization more often.

Code sinking optimization is only executed when optimizing for code size using the `-Os` option.

Two important aspects of code sinking optimization are:

- Code sinking optimization makes educated guesses about the code size implications of the transformation. It only performs the optimization when it believes that a code-size reduction is likely.
- Code sinking optimization alters the control flow of the program in order to enable code sinking. This alteration allows code from a subset of a join's preceding blocks to be sunk. This is only performed when there is likely to be a positive result.

Example 5-34 illustrates code before and after code sinking optimization occurs.

Example 5-33. Code sinking optimization

Generated code before optimization

```
if( index > 0x37 )
{
   array[ index ]= array[ index - 0x37 ];
   array[ index - 0x37 ]=tmp;
}
else
{
   array[ index ]= array[ index + 0x37 ];
   array[ index + 0x37 ]=tmp;
}
```

Generated code after optimization

```
if( index > 0x37 )
{
   temp_index = index - 0x37;
}
else
{
   temp_index = index + 0x37;
}
array[ index ]= array[ temp_index ];
array[ temp_index ]=tmp;
```

5.3.5 Cross-File Optimizations

Cross-file optimization produces the most effective form of optimization, since optimizations are applied across all the files in the application. You can specify the `-Og` option in the command line together with any of the optimization options except the `-O0` option. The `-Og` option is most effective when used with the default level `-O2`.

In addition to implementing the selected level of optimization across all the files, cross-file optimization also applies two specific optimizations:

- Function inlining across multiple files, which applies the optimization described in Section 5.3.2.2, "Function inlining," to the whole program. As with function inlining for individual files, this may increase the size of the code, but can considerably increase execution speed. If you specify `-Os`, the code size may actually decrease.
- Optimization of access to global and static variables.

5.3.5.1 Rules for using Cross-file Optimization

To receive optimal results using cross-file optimization, follow these rules:

1. You must compile the entire application together.
2. You can only link the Standard C library that is shipped with the Compiler.
3. Assembly functions can only call other assembly functions and library functions.

5.4 Guidelines for Using the Optimizer

The optimizer produces the best possible results when the source code is written in a simple and straightforward manner. Complex structures and algorithms should be avoided wherever possible, since these can reduce the effectiveness of many of the optimizations.

During the various optimization phases, the compiler attempts to convert all the structures in the code into a form that is independent of the style of an individual user, and that can be processed efficiently by the individual optimizations. By following the basic rules of clarity and simplicity when writing your code, you help the optimizer to retrieve the specific information it needs, and to apply the maximum amount of optimization.

For example, when accessing arrays you should use simple access instructions wherever possible, and avoid using complex access instructions which use pointers, as shown in Example 5-34:

Example 5-34. Simple and complex array accesses

a) Simple array access (recommended)

```
a[i]
```

b) Complex array access (not recommended)

```
p = &a[0]
*p++;
```

Section 5.4.3, "General Hints," provides further general guidelines for writing simple code structures to assist optimization.

You can further enhance the results of the optimization by applying two specific techniques that help the optimizer take full advantage of the multiple execution units of the SC100 architecture:

- Partial summation, which reduces dependencies in a loop, enabling multiple iterations of a loop in parallel
- Multisample processing, a programming technique which processes multiple samples simultaneously

These techniques are described in the sections that follow.

5.4.1 Partial Summation Techniques

One of the optimizer's major functions is to produce parallelized code that fully utilizes the available number of multiply-accumulate (MAC) units. The number of MAC units that can be used in an execution set, meaning the number of instructions executed in the same cycle, is usually limited by the degree of dependency within the code.

The partial summation programming technique helps you reduce the dependencies in the loops in your source code, in such a way that the iterations can execute in parallel. By structuring your source code using partial summation techniques wherever possible, you enable the optimizer to further reduce dependencies and increase parallelization.

In Example 5-35, the inner loop can use only a single MAC per cycle, because of the inner dependency within the algorithm. The same output code is generated when compiling for a single, dual, or quad MAC StarCore system.

Example 5-35. MAC usage limited by dependency in loop

Source code

```
 void  Iir(short Input[], short Coef[], short FiltOut[])
{
    long L_Sum = 0;  short int Stage, Smp;  int LoopCount;

    FiltOut[0] = Input[0];
    for (Smp = 1; Smp < S_LEN; Smp++)
    {
        L_Sum = LPC_ROUND;   LoopCount = (Smp<NP ? Smp : NP );

        for (Stage = 0; Stage < LoopCount; Stage++)
            L_Sum = L_msu(L_Sum, FiltOut[Smp - Stage -1],Coef[Stage]);

        L_Sum = L_shl(L_Sum, ASHIFT);
        L_Sum = L_msu(L_Sum, Input[Smp], 0x8000);
        FiltOut[Smp] = extract_h(L_Sum);
    }
}
```

Generated code

```
    doensh1 d0
    move.f r2)+,d0      move.f (r0)-,d1
    loopstart1
PL001
    mac    -d0,d1,d2    move.f (r0)-,d1     move.f (r2)+,d0
    loopend1
PL000
    mac    -d0,d1,d2
```

Example 5-36 illustrates how you can use partial summation to split the inner loop in the above example to enable two parallel iterations. The loop iterates half the number of times. The sum is accumulated using two variables, which are combined outside the loop.

Example 5-36. Partial summation for dual MAC usage

Source code

```
for (Stage = 0; Stage < (LoopCount>>1); Stage++)
   {
      L_Sum = L_msu(L_Sum, FiltOut[Smp - 2*Stage -1], Coef[2*Stage]);
      L_Sum1 = L_msu(L_Sum1, FiltOut[Smp - 2*Stage -2], Coef[2*Stage+1]);
   }

L_Sum = L_shl(L_Sum+L_Sum1, ASHIFT);
L_Sum = L_msu(L_Sum, Input[Smp], 0x8000);
```

Generated code

```
doen  sh1 d0
   move.2f   (r2)+,d0d1   move.2f   (r0)-,d6d7
   loopstart0
PL001
   [
   mac       -d0,d6,d2
   mac       -d1,d7,d5
   move.2f   (r0)-,d6:d7
   move.2f   (r2)+,d0:d1
   ]
   loopend0
PL000
   mac       -d0,d6,d2    mac    -d1,d7,d5
```

The same technique can be used for compiling with a quad MAC system, by splitting the loop into four iterations, using four variables and one quarter the number of iterations.

It is important to note that partial summation is not suitable for algorithms with bit-exact requirements. This technique changes the order of the calculation, and may affect the value of the result in cases where statements must be executed in the exact order specified.

In certain algorithms the effectiveness of the partial summation technique may be limited because of alignment restrictions. For example, the `move.2f` instruction, which is required for partial summation, must be used on a long word boundary.

In Example 5-36, this restriction is satisfied, and the partial summation technique can be used successfully. Example 5-37 shows an algorithm for which partial summation cannot be used. This is because the second iteration produces an odd value for the variable `i`, with the result that the `move.2f` instruction violates the alignment requirement.

Example 5-37. Alignment restrictions in algorithms

```
for (i = 0; i < DataBlockSize; i++)
{
   Delay[(DataBlockSize-i)] = DataIn[i];
   sum1 = 0;  sum2 = 0;
   for (j = 0; j < FirSize/2 ; j++)
   {
       sum = L_mac(sum,Coef[2*j],Delay[2*j-i]);
       sum = L_mac(sum,Coef[2*j+1],Delay[2*j-i+1]);
   }
   Result = round(sum);
}
```

The multisample techniques described in the following section can help you write source code which enables the optimizer to take further advantage of multiple execution units. You can apply multisample techniques even if you cannot use partial summation for certain algorithms because of alignment restrictions or bit-exact requirements.

5.4.2 Multisample Techniques

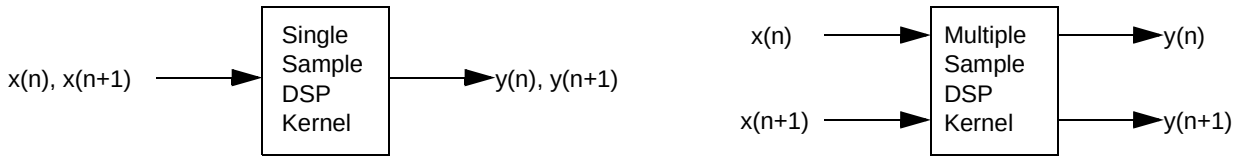
To obtain high performance, a pipelining technique called “multisample” programming is used to process multiple samples simultaneously. The multisample programming techniques enable you to obtain high performance by taking full advantage of the SC100 multiple-ALU architecture.

This following terminology is used throughout this section:

- **Generic Kernel**: The minimum required operations of the algorithm. The generic kernel is the theoretical minimum size of the kernel without considering implementation constraints.
- **Basic Kernel**: The inner loop of a DSP algorithm. This may contain several replications of the generic kernel or additional code for pipelining. The basic kernel is actually implemented on the DSP and is subject to implementation constraints.
- **Operand:** A value used as an input to an ALU.
- **Delays**: Values stored as a delay line for referencing past values.
- **Iteration**: The complete execution of a basic kernel.
- **Loop pass**: The execution of the instructions within the basic kernel. Many loop passes may be needed to complete a single iteration of the kernel.

To process several samples simultaneously, operands (both coefficients and variables) are reused within the kernel. Although a coefficient or operand is loaded once from memory, multiple ALUs may use the value, or the value may be used in a later step of the kernel.

Figure 5-5 illustrates the structure of a single sample and multisample algorithm.



A. Single Sample Algorithm **B. Multiple Sample Algorithm**

Figure 5-5. Single Sample and Multisample Kernels

In a single sample algorithm (Figure 5-5 A), samples are processed by the algorithm serially. The kernel processes a single input sample and generates a single output sample. For an algorithm such as an FIR, samples are input to the FIR kernel one at a time. The FIR kernel generates a single output for each input sample. Blocks of samples are processed using loops and executing the FIR kernel several times.

In contrast, the multisample algorithm (Figure 5-5 B) takes multiple samples at the input in parallel and generates multiple samples at the output simultaneously. The multisample algorithm operates on data in small blocks. Operands and coefficients are held in registers, and applied to both samples simultaneously, resulting in fewer memory accesses.

Multisample algorithms are ideal for block processing algorithms where data is buffered and processed in groups (such as speech coders). Figure 5-5 B shows two samples being processed simultaneously. However, the number of simultaneous samples depends on the processor architecture and type of algorithm.

Most DSP algorithms have a multiply-accumulate (MAC) at their core. On a load/store machine, the register file is the source/destination of operands to/from memory. For the ALU, the register file is the source/destination of operands. On a single sample, single ALU algorithm, the memory bandwidth is typically equal to the operand bandwidth, as shown in Figure 5-6.

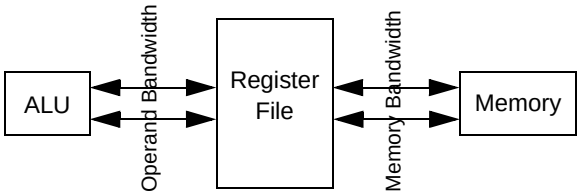


Figure 5-6. Single ALU Operand and Memory Bandwidth

When increasing the number of ALUs to four, the bandwidth increases as shown in Figure 5-7.

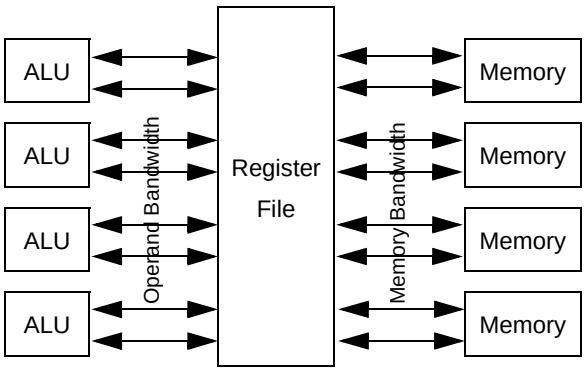


Figure 5-7. Quad ALU Operand and Memory Bandwidth

Quadrupling the number of ALUs quadruples the operand bandwidth. If there is one address generator per operand, this results in eight address generators. This is undesirable because it requires an 8 port memory and a significant amount of address generation hardware.

The SC140 DSP core solves this problem by providing up to a quad operand load/store over a single bus. With two quad operand loads, eight operands can be loaded using two address generators.

Although quad operand loading provides the proper memory bandwidth, some algorithms have special memory alignment requirements. These alignment requirements make it difficult to use multiple operand load/stores.

Multisample algorithms provide a solution for implementing algorithms with memory alignment requirements. By reusing previously loaded values, the number of operands loaded from memory is reduced, which relaxes the alignment constraints.

Both techniques for increasing operand bandwidth, by using wider data buses or by reusing operands, are shown in Figure 5-8.

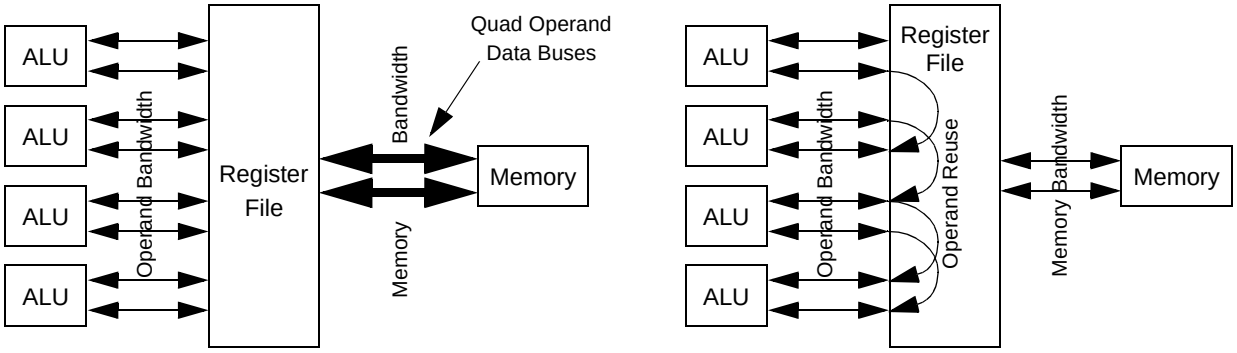


Figure 5-8. Options for Increasing Operand Bandwidth

To introduce the multisample technique, four example DSP kernels are written in multisample form. The DSP kernels presented are direct form FIR filter, direct form IIR filter, correlation and biquad filter.

5.4.2.1 Multisample implementation issues

When implementing a DSP algorithm such as an FIR filter, trade-offs are made between the number of samples processed and the number of ALUs as shown in Figure 5-9.

Number of ALUs

Number of Samples	1	2	4
1	1 sample, 1 ALU	1 sample, 2 ALUs	1 sample, 4 ALUs
2	2 samples, 1 ALU	2 samples, 2 ALUs	2 samples, 4 ALUs
4	4 samples, 1 ALU	4 samples, 2 ALUs	4 samples, 4 ALUs

Figure 5-9. Number of Samples and ALUs for Implementing DSP Algorithms

As the kernel computes more samples simultaneously, the number of memory loads decreases because data and coefficient values are being reused. However, to obtain this reuse, more intermediate results are required, which typically requires more registers in the processor architecture.

If the operand memory requires wait states, this technique improves the speed of the algorithm. If the operand memory is full speed, then the algorithm does not execute any faster, but may reduce power consumption because the number of memory accesses has been reduced.

By using more ALUs, it is theoretically possible to compute an algorithm faster. Moving across the row theoretically applies 1, 2 or 4 ALUs to the algorithm. To apply multiple ALUs, some degree of parallelism is required in the algorithm to partition the computations.

Although computing a single sample with multiple ALUs is theoretically possible, limitations in the DSP hardware may not allow this style of algorithm to be implemented. In particular, most processors typically require operands to be aligned in memory and multiple operand load/stores to be aligned.

For example, a double operand load requires an even address and a quad operand load requires a double even address. These types of restrictions are typical to reduce the complexity of the address generation hardware (particularly for modulo addressing).

Restricting the boundaries of the load makes implementing some algorithms very difficult or impossible. This is easiest to explain by way of example. Consider a series of (aligned) quad operand loads from memory, as shown in Figure 5-10.

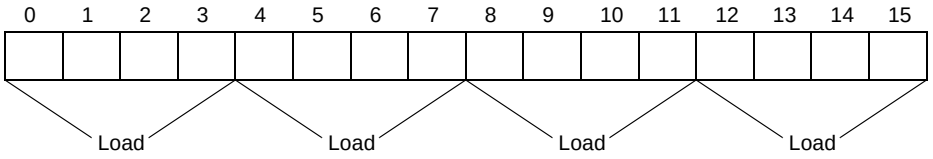


Figure 5-10. Quad Coefficient Loading from Memory

The loads in Figure 5-10 do not have a problem with alignment because loads occur from double even addresses.

Alignment problems typically occur with algorithms implementing delay lines in memory. These algorithms delete the oldest delay and replace it with the newest sample. This is typically done by using modulo addressing and “backing up” the pointer after the sample is processed. This leads to an addressing alignment problem as shown in Figure 5-11.

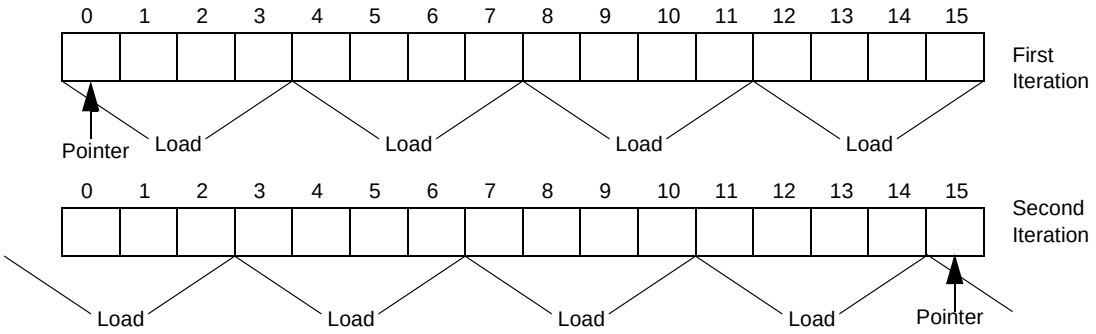


Figure 5-11. Misalignment when Loading Quad Operands

On the first iteration of the kernel, quad data values are loaded starting from a double even address. This does not create an alignment problem. However, at the end of the first iteration, the pointer is backed up by one to delete the oldest sample. On the next iteration, the pointer is not at a double even address and the quad data load is not aligned.

A solution to the alignment problem is to reduce the number of operands moved on each data bus. This relaxes the alignment issue. However, in order to maintain the same operand bandwidth, each loaded operand must be used multiple times. This is a situation where multisample processing is useful.

As the number of samples per iteration increases, more operands are reused and the number of moves per sample is reduced. With fewer moves per sample, the number of memory loads is decreased allowing fewer operands per bus. Fewer operands per bus allows the data to be loaded with fewer restrictions on alignment.

5.4.2.2 Implementation example

The FIR_A4S4 Quad ALU, quad sample, is the highest performance implementation on a quad ALU SC100 DSP.

To further increase the performance of the FIR filter, four ALUs may be used. To avoid misalignment, four samples are processed simultaneously. The quad ALU, quad sample FIR data flow is shown in Figure 5-12.

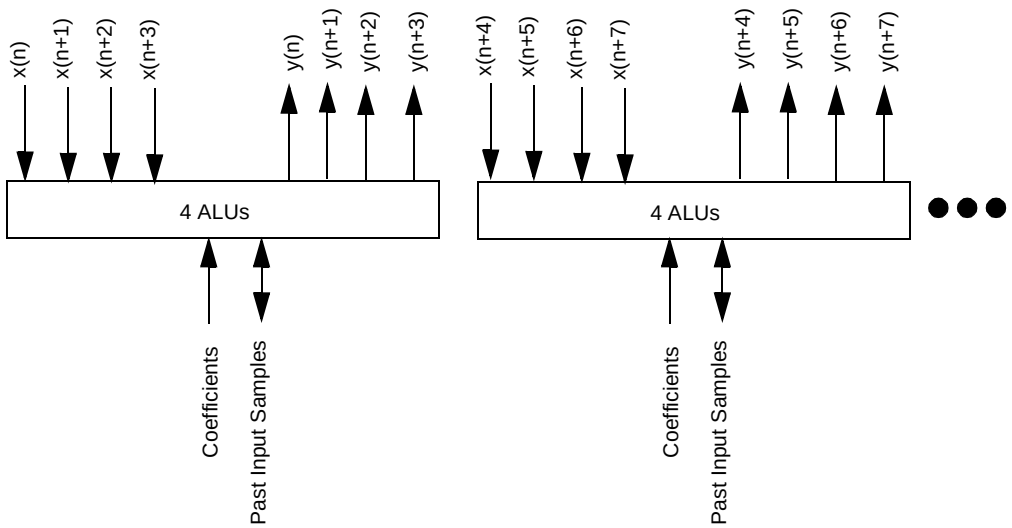


Figure 5-12. Quad ALU, Quad Sample FIR Filter Data Flow

Input samples are grouped together four at a time. Coefficients and delays are loaded and applied to all four input values to compute four output values. By using four ALUs, the execution time of the filter is only one quarter the execution time of a single ALU filter.

To develop the FIR filter equations for processing four samples simultaneously, the equations for the current sample `y(n)` and the next three output samples `y(n+1)`, `y(n+2)` and `y(n+3)` are as shown in Figure 5-13.

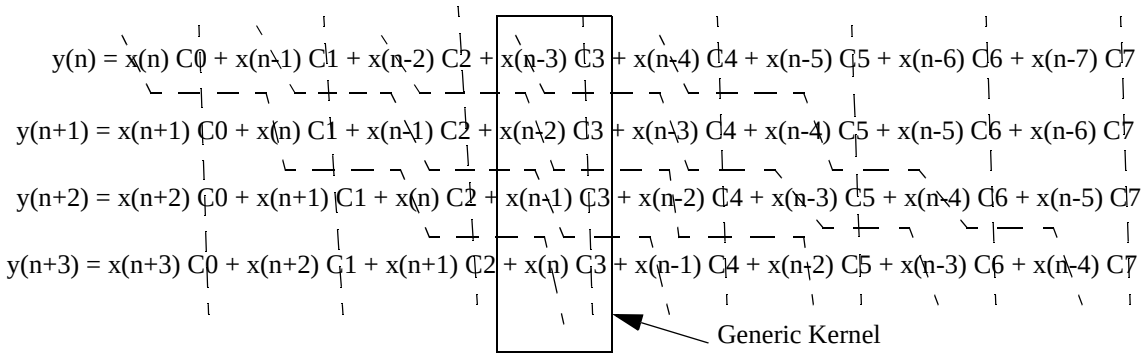


Figure 5-13. FIR Filter Equations for Four Samples

The generic kernel has the following characteristics:

- Four parallel MACs.
- One coefficient is loaded and used by all four MACs in the same generic kernel.
- One delay value is loaded, used by the generic kernel and saved for the next three generic kernels.
- Three delays are reused from the previous generic kernel.

To develop the structure of the quad ALU kernel, the filter operations are written in parallel and the loads are moved ahead of where they are first used. This creates the generic kernel shown in Figure 5-14.

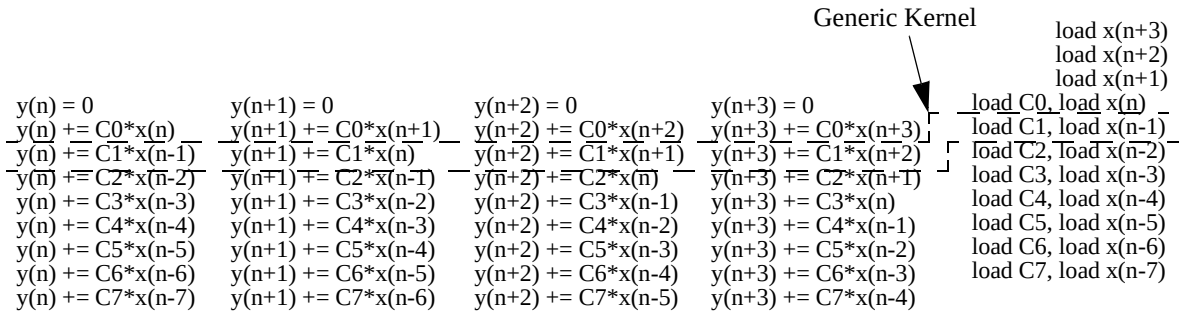


Figure 5-14. Generic Kernel For Quad ALU FIR

The generic kernel requires four MACs and two parallel loads. Example 5-38 illustrates how the kernel in Figure 5-14 is implemented in a single instruction.

Example 5-38. Single instruction quad ALU generic filter kernel

```
y(n) += C * D1      y(n+1) += C * D2      y(n+2) += C * D3      y(n+3) += C * D4
Load C, Copy D3 to D4, Copy D2 to D3, Copy D1 to D2, Load D1
```

To provide delay reuse, the delays are copied by using temporary variables D1, D2, D3 and D4 as a delay line. This imposes a requirement on the kernel to perform two MACs and five move operations (two loads and three copies) in a single instruction.

Example 5-39 contains an example of C simulation code which implements the generic kernel shown in Figure 5-14 on page 5-45.

Example 5-39. FIR_A4S4 quad ALU, quad sample C simulation code

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

#define DataBlockSize 40  // size of data block to process
#define FirSize      8    // number of coefficients in FIR

Word16 DataIn[DataBlockSize] = {
    328, 9830, 8192, -6553, -3277, 3277, 3277, -6553, -9830, 4915,
    8192, -6553, 328, 9830, 4915, -6553, -3277, 3277, 3277, -9830,
    4915, -3277, -9830, 8192, -6553, 328, 9830, -6553, 3277, 3277,
    3277, 328, 9830, 4915, -3277, -9830, 8192, -6553, -6553, 3277
};

Word16 Coef[FirSize] = {
    3277, 6553, -9830, -6553, -4915, 3277, 8192, -6553
};
Word16 Delay[FirSize+3];

#define IncMod(a) (a=((a+1)%(FirSize+3)))
#define DecMod(a) (a=((a+FirSize+2)%(FirSize+3)))

int main()
{
   int DelayPtr;
   Word32 sum1,sum2,sum3,sum4;
   Word16 D1,D2,D3,D4;
   int i,j;

   DelayPtr = 0;// init delay ptr

   for (i = 0; i < DataBlockSize; i += 4) {// do 4 samples at a time

      Delay[DelayPtr] = DataIn[i];   DecMod(DelayPtr);
      Delay[DelayPtr] = DataIn[i+1]; DecMod(DelayPtr);
      Delay[DelayPtr] = DataIn[i+2]; DecMod(DelayPtr);
```

```
Delay[DelayPtr] = DataIn[i+3];

sum1 = 0;          // init sum to zero
sum2 = 0;          // init sum to zero
sum3 = 0;          // init sum to zero
sum4 = 0;          // init sum to zero
```

```
        D4 = Delay[DelayPtr];    IncMod(DelayPtr);
                  D3 = Delay[DelayPtr];    IncMod(DelayPtr);
                  D2 = Delay[DelayPtr];    IncMod(DelayPtr);

        for (j = 0; j < FirSize / 4 ; j++) {// evaluate FIR
            D1 = Delay[DelayPtr];    // get delay
            IncMod(DelayPtr);

                      sum1 = L_mac ( sum1, Coef[4*j], D1 );
                      sum2 = L_mac ( sum2, Coef[4*j], D2 );
                      sum3 = L_mac ( sum3, Coef[4*j], D3 );
                      sum4 = L_mac ( sum4, Coef[4*j], D4 );

            D4 = Delay[DelayPtr];    // get delay
            IncMod(DelayPtr);

                      sum1 = L_mac ( sum1, Coef[4*j+1], D4 );
                      sum2 = L_mac ( sum2, Coef[4*j+1], D1 );
                      sum3 = L_mac ( sum3, Coef[4*j+1], D2 );
                      sum4 = L_mac ( sum4, Coef[4*j+1], D3 );

            D3 = Delay[DelayPtr];    // get next delay
            IncMod(DelayPtr);

                      sum1 = L_mac ( sum1, Coef[4*j+2], D3 );
                      sum2 = L_mac ( sum2, Coef[4*j+2], D4 );
                      sum3 = L_mac ( sum3, Coef[4*j+2], D1 );
                      sum4 = L_mac ( sum4, Coef[4*j+2], D2 );

            D2 = Delay[DelayPtr];    // get next delay
            IncMod(DelayPtr);

                      sum1 = L_mac ( sum1, Coef[4*j+3], D2 );
                      sum2 = L_mac ( sum2, Coef[4*j+3], D3 );
                      sum3 = L_mac ( sum3, Coef[4*j+3], D4 );
                      sum4 = L_mac ( sum4, Coef[4*j+3], D1 );
        }
        DecMod(DelayPtr);

        printf("Index: %d, output: %d\n",i,round(sum1));
        printf("Index: %d, output: %d\n",i+1,round(sum2));
        printf("Index: %d, output: %d\n",i+2,round(sum3));
        printf("Index: %d, output: %d\n",i+3,round(sum4));
        }
}
```

5.4.3 General Hints

In addition to the specific techniques described in the previous sections, there are a number of general guidelines that you should follow when writing source code, in order to assist the optimizer to produce the most efficient results. These guidelines are described in the sections that follow.

5.4.3.1 Software pipelining

The optimizer implements sophisticated levels of software pipelining, saving you the need to introduce software pipelining into your source code. It is important that you do not include any manual form of software pipelining into your source code, as this can conflict with the algorithms used by the optimizer, resulting ultimately in less efficient optimization.

Example 5-40 shows two forms of source code for the same loop. The first version contains no pipelining, and is the recommended source code form. This will generate more efficient and smaller code than the second version, which pipelines the first iteration at the C level outside the loop. The type of manual pipelining shown in the second version should be avoided.

Example 5-40. Avoiding software pipelining in source code

1. No pipelining (recommended)

```
L_R = 0;
for (J = 0; J < S_LEN; J++)
   L_R = L_mac(L_R,WBasisVecs[J + (I * S_LEN)], WInput[J]);
```

2. Manual pipelining (not recommended)

```
L_R = L_mult(WBasisVecs[I * S_LEN], WInput[0]);
for (J = 1; J < S_LEN; J++)
   L_R = L_mac(L_R,WBasisVecs[J + (I * S_LEN)], WInput[ J]);
```

5.4.3.2 Passing and returning large structs

Instead of passing and returning large `structs` using their value, use pointers to large `structs` wherever possible.

5.4.3.3 Arithmetic operations

Whenever you can, use constants instead of variables for shift, division, or remainder operations.

5.4.3.4 Local variables

Any local variable that you specify should be initialized before it is used.

5.4.3.5 Resource limitations

The SC100 architecture provides a total of 16 Dn registers and 16 Rn registers. If the number of active variables is greater than the number that the registers can accommodate, the compiler maps the extra variables to memory, resulting in less efficient code.

For best results, you should take account of these physical limitations when writing your source code. For example, when preparing a set of instructions to execute in one cycle, remember that there is a restriction on the number of operands that can be used in a single cycle.

5.5 Optimizer Assumptions

The optimizer uses the information passed to it by the compiler, in order to ensure that the optimizations applied during the various optimization stages do not affect the original accuracy of the program.

At the time that the compiler accumulates this information, it assumes that only two types of variables can be accessed while inside a function, either indirectly through a pointer or by another function call:

- Global variables, meaning all variables within the file scope or application scope
- Local variables, whose addresses are retrieved implicitly by the automatic conversion of array references to pointers, or explicitly by the `&` operator

If your programs conform to the standard ANSI/ISO version of C, this assumption does not affect your code. If the code that you are compiling is not standard, and it violates this assumption, the optimization process may adversely affect the behavior of the program.

To avoid unexpected results, and to ensure that your program executes correctly once optimized, follow the coding guidelines listed below:

- Don’t make assumptions based on memory layout when using pointers. For example, if `x` points to the first member of a structure, `x+1` may not necessarily point to the second member of the same structure. Similarly, if `y` is defined as a pointer to the first declared variable in a list, do not assume that `y+1` points to the second variable in the list.
- When referencing an array, keep the references inside the array bounds.
- Ensure that all the required arguments are passed to functions.
- When subscribing one array, don’t access another array indirectly. For example, if in the construct `x[y-x]`, `x` and `y` are the same type of array, the construct is equivalent to `*(x+(y-x))`, which is equivalent to `*y`. Thus the construct actually references the array `y`.
- When pointing to objects, don’t reference outside the bounds of these objects. The optimizer assumes that all references of the form `*(p+i)` apply within the bounds of the variable(s) to which `p` points.
- When the need arises for variables that are accessed by external processes, be sure to declare the variables as `volatile`. Use this keyword judiciously, as it may have adverse effects on optimization.

Chapter 6
Runtime Environment

This chapter describes the startup code used by the SC100 C compiler, the layout and configuration of memory, and the calling conventions which the compiler supports.

It contains the following sections:

- Section 6.1, “Startup Code,” provides details of the runtime code used for system initialization and finalization.
- Section 6.2, “Memory Models,” describes the two memory models supported by the compiler.
- Section 6.3, “Memory Layout and Configuration,” describes the way that the compiler maps memory, and explains how you can change this configuration to suit your requirements.
- Section 6.4, “Calling Conventions,” describes the stack-based and other calling conventions that the compiler supports.
- Section 6.5, “Saturation,” provides details about saturation switches and saturation states.

6.1 Startup Code

The compiler runtime startup code consists of the following components:

- Initialization code, which is executed when the program is initiated and before its main function is called
- Finalization code, which controls the closedown of the application after the program’s main function terminates
- Entry points for low level I/O services
- The interrupt vector table
- Support for debugging tools

The entire startup code resides in assembly code files, named `crt.asm and crtnosat.asm`, which are located in the directory `$SCTOOLS_HOME/src/rtlib/`. `crt.asm` turns saturation on and is the default. `crtnosat.asm` sets the saturation mode bit to off. The object module for the files is located in the directory `$SCTOOLS_HOME/lib`.

The compiler startup code contains two phases:

- **Bare board startup code**, which is used for programs which execute without the support of any runtime executive or operating system. This phase resets the interrupt vector and initializes all necessary hardware registers.
- **C environment startup code**, which is a mandatory phase for all configurations. This phase initializes the runtime structure of the application for the C environment, and includes the finalization code used following termination of the program.

6.1.1 Bare Board Startup Code

The bare board startup phase assumes that no operating system or runtime executive is running. It performs the various actions which are normally carried out automatically by the operating system or runtime executive, as follows:

1. The reset interrupt vector is set to point to the system entry point `__crt0_start`, as if the system has just been reset. The interrupt vector table holds the addresses of all interrupt handlers. The first entry in this table is the system entry point. All other entries in the interrupt vector table point by default to the `abort` function. Further information about interrupt handlers is provided in Section 6.4.5, "Interrupt Handlers." See Chapter 7, "Runtime Libraries," for more information about `abort` and other runtime functions.
2. The hardware registers are initialized as follows:
   - The four modulo (M) registers (`m0-m3`) are initialized to linear addressing.
   - The status register is set to an initial value taken from the linker command file used at link time. This file includes a label `SR_setting`, which defines the initial value to be assigned to the status register following system reset. Table 6-1 shows the default status register settings.

Table 6-1. Status Register Default Settings

Setting Type	Value
Mode:	Exception mode
Interrupt level:	7
Saturation:	On
Rounding mode:	NEAREST_EVEN

3. If the system includes a timer, the timer is activated.
4. The bare board startup phase terminates by jumping to the C environment startup code entry point, `___start`.

6.1.2 C Environment Startup Code

The C environment startup phase is applicable to all programs. The entry point for this phase is `___start`. This phase includes initialization code used prior to program start, and finalization code used after the application terminates.

6.1.2.1 C environment initialization code

The following initialization actions are executed before the application starts:

1. The memory map is set up and initialized. The stack pointer (SP) value is loaded into memory by the stack start address, located at `StackStart`. This label is defined in the linker command file and used by the linker at link time. For further information about the memory map, see Section 6.2, “Memory Models.”
2. If the `-mrom` option has been specified in the shell command line, initialized variables are copied from ROM into RAM. This option is required for applications which do not use a loader.
3. The `argv` and `argc` arguments are set up.
4. Interrupts are enabled. Until this point, interrupts have been disabled.
5. The application main procedure entry point is called using the function `main`.

6.1.2.2 Initialization of variables

If your system uses a loader, this will by default initialize all variables. In systems that do not include a loader, it is important that you specify the `-mrom` option when you compile the final version of your application, to ensure that the initialized variables are copied from ROM into RAM at startup.

Note: Before a C program executes, certain global variables may assume the assignment of an initial value of zero. The compiler does not preinitialize variables automatically. You must ensure that your code includes explicit initialization of any variable that must have an initial value of zero.

6.1.2.3 C environment finalization code

On return from the application main function, the runtime function `exit` is called. This terminates any I/O services which have not yet terminated, and stops the processor by issuing the `stop` instruction.

Note: Certain embedded real time applications never terminate. Such termination activities do not usually pertain to embedded applications, but may be of use during early development and debugging stages.

6.1.2.4 Low level I/O services

The C environment startup code includes the input and output of low level, buffered I/O services. The code uses calls to `__send` and `__receive` in order to interface with debugging tools and/or runtime systems.

6.1.3 Configuring Your Startup Code

If the default runtime setup does not match your configuration, you need to modify your startup code accordingly.

To create your own runtime configuration code, follow the steps described below:

1. Make your own copy of the default startup file, `crtsc100.asm`, with a name of your choice, as shown in the following example:

   Example 6-1. Creating a new startup file

   ```
   cp install-dir/src/rtlib/crtsc100.asm mysc100.asm
   ```

2. Make the required changes to the new file.
3. Assemble the modified file, as shown in Example 6-2.

   Example 6-2. Assembling the modified startup file

   ```
   asmsc100 -b -l mysc100.asm
   ```

   The generated object file has the same file name as the source file, and the extension `.eln`. In this example, the object file generated is `mysc100.eln`.
4. Use the modified file by specifying the `-crt` option in the shell command line, as shown in Example 6-3, to ensure that the modified startup file is used at link time.

   Example 6-3. Using the modified startup file

   ```
   scc -crt mysc100.eln my-object-files.eln
   ```

6.2 Memory Models

The StarCore architecture supports big, small, and tiny memory models. These memory models save code size and enhance performance. The following table provides information about each memory model:

Table 6-2. Memory Models

Memory Models	Option	Bit	Description
Big memory model	-mb	unsigned 32-bit addresses	Does not restrict the amount of space allocated to addresses. Uses a longer instruction that includes 32-bit instructions.
Small memory model	default	unsigned 16-bit addresses	The default model. Assumes that all addresses are 16-bit immediate.
Tiny memory model	-mt	signed 16-bit addresses	Assumes that all addresses are within the range of a signed 16-bit immediate, effectively an unsigned 15-bit range.

The three compilation models allow the compiler to generate references to global and static data without global knowledge as to the variables final allocation address in memory. For each model the compiler assumes that references to global and static data fit within the corresponding size implied by the model. The expectation is that the linker will generate errors whenever a variable is resolved to not fit within the range defined by the memory model.

6.2.1 Small and Tiny Memory Models

If the application is small enough to allow all static data to fit into the lower 64K of the address space, then more efficient code can be generated. This small memory model is the default and assumes that all addresses are 16-bit immediate. The tiny memory model assumes that all addresses are within the range of a signed 16-bit immediate (effectively an unsigned 15-bit range).

6.2.2 Big Memory Model

The big memory model does not restrict the amount of space allocated to addresses. When the compiler uses the big memory model to access a data object, whether static or global, it must use a longer instruction that includes a 32-bit address. This operation requires an additional word, and as a result it produces code that is larger, and in some cases slower than a similar operation using the small or tiny memory models.

Example 6-4 illustrates the code sequence to generate the address of a global symbol in memory and the sequence to reference the memory contents of a global symbol for each memory model.

Example 6-4. Big, small, and tiny memory models

Big memory model:

```
move.l address,d0     (3 16-bit words)
moveu.l #address, d0  (3 16-bit words)
```

Small memory model:

```
move.l <address,d0    (2 16-bit words)
moveu.l #address, d0  (3 16-bit words)
```

Tiny memory model:

```
move.l <address,d0    (2 16-bit words)
move.w #address, d0   (2 16-bit words)
```

You can use certain instructions only in small memory mode. If < is omitted in conjunction with these instructions, an error results. Example 6-5 shows the instruction `bmset.w`, which sets bit #zero in the specified address, and is valid only in small memory models.

Example 6-5. Small and tiny memory mode instruction

```
bmset.w #0001,<address
```

Note: For maximum efficiency, it is recommended that you place data in the smallest possible locations of the memory map (lower 32K or lower 64K), in order to enable the compiler to use small or tiny memory modes.

6.2.3 Linker Command Files

The SC100 Linker refers to a linker command file at link time, for various runtime values, addresses and labels. Three linker command files are provided, one for each memory mode.

These files are:

- `crtscsmm.cmd`, used in small memory mode,
- `crtsctmm.cmd`, used in tiny memory mode, and
- `crtscbmm.cmd`, used when big memory mode is selected.

All three files are located in the *`install-dir`*`/etc` directory.

6.3 Memory Layout and Configuration

The SC100 default memory layout is a single linear block which is divided into data and code areas. C programs generate code and data in sections. The compiler places each of these sections in its own continuous space in memory.

The default layout of the SC100 memory is illustrated in Figure 6-1

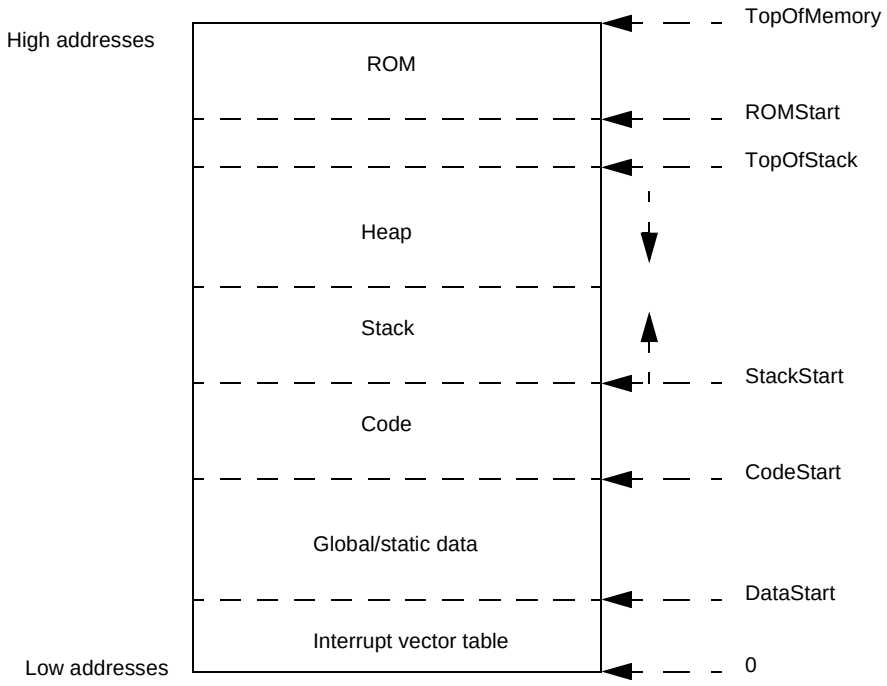


Figure 6-1. SC100 Default Memory Layout

All three memory models use the same default layout, but with different default values that define the distribution of the memory areas, as shown in Table 6-3, Table 6-4, and Table 6-5 on page 6-8. You can change these default values, and configure the memory map to meet your specific requirements, as described in Section 6.3.3, “Configuring the Memory Map.”

The layout and functionality of the stack and heap are common to all the memory models, and are described in the sections that follow.

The default memory map values for the small memory model are listed in Table 6-3. These values are held in the file `crtscsmm.cmd`.

Table 6-3. Small Memory Model Default Values

From	Default value	To	Default value	Contents
0		0x1ff		Interrupt vector table
DataStart	0x0200	DataStart+DataSize-1	0x101ff	Global and static variables
CodeStart	0x100000	StackStart-1	0x27fff	Program code
StackStart	0x200000	TopOfStack	0x2fff00	Stack and heap
ROMStart	0x300000	TopOfMemory	0x3fffff	ROM

Table 6-4 lists the default memory map values for the big memory model. These values are held in the file `crtscbmm.cmd`.

Table 6-4. Big Memory Model Default Values

From	Default value	To	Default value	Contents
0		0x1ff		Interrupt vector table
DataStart	0x0200	DataStart+DataSize-1	0xfffff	Global and static variables
CodeStart	0x100000	StackStart-1	0x3ffff	Program code
StackStart	0x200000	TopOfStack	0x2fff00	Stack and heap
ROMStart	0x300000	TopOfMemory	0x3fffff	ROM

Table 6-5 lists the default memory map values for the tiny memory model. These values are held in the file `crtsctmm.cmd`.

Table 6-5. Tiny Memory Model Default Values

From	Default value	To	Default value	Contents
0	0	0x1ff	0x1ff	Interrupt vector table
DataStart	0x200	DataStart+DataSize-1	0x81ff	Global and static variables
CodeStart	0x100000	StackStart-1	0x27fff	Program code
StackStart	0x200000	TopOfStack	0x2fff00	Stack and heap
ROMStart	0x300000	TopOfMemory	0x3fffff	ROM

6.3.1 Stack and Heap Configuration

The heap and stack are allocated from the same area of memory and must be contiguous. The compiler always treats the stack and heap as a continuous area of memory. The other sections of memory can be distributed, and there are no restrictions relating to their location.

6.3.1.1 Runtime stack

The compiler allocates an area of memory to the runtime stack, which is used for the following purposes:

- Allocation of local variables
- Passing arguments to functions
- Saving function return addresses
- Saving temporary results

The stack is allocated in the area above the space used for code, and grows in an upward direction toward the top of memory. The compiler uses the SP register to manage this stack.

The SC100 architecture includes two stack pointers:

- NSP, used when the processor is running in Normal mode
- ESP, used when the processor is running in Exception mode

As shown in Table 6-1 on page 6-2, the default mode at initialization is Exception mode.

The compiler makes no assumptions about which stack pointer to use, and uses the pointer for the current processor mode to point to the address at the top of the stack.

When the system is initialized, the stack pointer for the current mode is set by default to the address of the location directly after the code area, as defined in `StackStart` in the linker command file. The actual address of the stack is determined at link time.

The stack pointer for the current processor mode is automatically incremented by the C environment at the entry to a function. This ensures that sufficient space is reserved for the execution of the function. At the function exit, the stack pointer is decremented, and the stack is restored to its previous size prior to function entry. If your application includes assembly language routines and C code, you must ensure at the end of each assembly routine that the current stack pointer is restored to its pre-routine entry state.

Note: If you change the default memory configuration, remember to allow sufficient space for the stack to grow. If a stack overflow occurs at runtime, this will cause your program to fail. The compiler does not check for stack overflow during compilation or at runtime.

6.3.1.2 Dynamic memory allocation (heap)

The runtime libraries supported by the compiler include a number of functions that enable you to allocate memory dynamically for variables. See Chapter 7 for details of the runtime libraries supported. Since C does not support the dynamic allocation of memory, the compiler assigns an area of memory as a heap for this purpose.

The compiler allocates memory from a global pool for the stack and the heap together. The lower address of the area assigned to the stack and heap is defined in `StackStart`, in the linker command file. The heap starts at the top of memory, and is allocated in a downward direction toward the stack.

Objects that are dynamically allocated are addressed only with pointers, and not directly. The amount of space that can be allocated to the heap is limited by the amount of available memory in your system.

To make more efficient use of the space allocated to data, you can use the heap to allocate large arrays, instead of defining them as static or global.

For example, a definition such as `struct large array1[80];` can be defined using a pointer and the `malloc` function, as illustrated in Example 6-6.

Example 6-6. Allocating large arrays from the heap

```
struct large *array1;
array1 = (struct large *)malloc(80*sizeof(struct large));
```

6.3.2 Static Data Allocation

When you compile your application without cross-file optimization, the allocations for each file are assigned to different sections of data memory. At link time these are dispatched to different addresses.

When compiling with cross-file optimization, the compiler uses the same data section for all allocations. If you want to override this and to instruct the compiler to use non-contiguous data blocks, you can edit the machine configuration file to define the exact memory map of the system that you want to use. For more details, see Section 6.3.4, "Machine Configuration File."

6.3.3 Configuring the Memory Map

The default values in the SC100 memory map are easily configurable, by modifying the linker command file. When making such changes, it is important that you ensure that the code size and data size values that you specify do not overlap.

The stack and the heap must be always be located together in one contiguous area of memory. The compiler makes no assumptions about the layout of the other sections of memory, which can be split and distributed over non-contiguous parts of memory, as required.

Section 6.3.3.1, "Memory map configuration example," provides an example of a requirement for a modified memory map configuration, and describes the changes to be defined in the linker command file for this sample configuration.

Note: If you choose not to modify the default command file, but rather save the changes in a new command file instead, use the `-mem` option to pass the new command file to the linker. If you use the `-Xlnk` option to do this, both the new command file and the default command file will be passed to the linker, resulting in errors.

6.3.3.1 Memory map configuration example

This example assumes that you have a system with non-contiguous memory, and would like to configure the memory as follows:

- All code placed in external memory (addresses `0x10000000` through `0x10100000`)
- All data placed in internal memory
- Some local memory reserved for the most frequently used functions and overlays (addresses `0x10000` through `0x20000`)
- All data placed in the lower 64K addresses, in order to be able to use the small memory model compilation mode

The memory map that meets these requirements is shown in Example 6-7:

Example 6-7. Modified memory map configuration

From	To	Contents
0	`0x1ff`	Interrupt vectors

Example 6-7. Modified memory map configuration

0x200	`0xfffd`	Global and static variables
0x10000	`0x1ffff`	Local code
0x20000	`0x7fff0`	Stack and heap
0x80000	`0xfffff`	ROM
0x10000000	`0x100fffff`	External code

Example 6-8 shows the definitions in the `crtscsmm.cmd` file that specify this memory map configuration:

Example 6-8. Modified memory configuration in the linker command file

```
.provide _DataSize,   0x10000          ; Sets the data size.
.provide _CodeStart,  0x10000000       ; Sets the loader code start address.
.provide _StackStart, 0x20000          ; Sets the stack start address;
                                       ;     the stack grows upwards.
.provide _TopOfStack, 0x7fff0          ; The heap start address;
                                       ;     the heap grows downwards.
.provide _ROMStart,   0x80000          ; Sets the ROM start address.
```

6.3.4 Machine Configuration File

The machine configuration file contains the following:

- Information about data types and alignment requirements, used by the compiler for reference. This data must **not** be changed.
- Memory structure information, used by the compiler to allocate variables in the data sections of memory. This information can be modified if required.

By default, the compiler uses the file `proc.config`, located in the *`install-dir`*`/etc` directory. A different machine configuration file can be specified using the `-mc` option in the shell command line.

The SC100 memory structure consists of physical and logical memory maps, as follows:

- Physical memory is divided into several memory spaces. Each memory space is a physical entity consisting of a data bus and an address bus. A physical memory space is defined in terms of its size in words and the width of its address bus, and comprises blocks of words with contiguous addresses, described as physical memory areas.
- Logical memory areas are defined as blocks of memory words with contiguous addresses. These words are used by the compiler as if they were in physical memory areas. The addresses of the logical areas are mapped as offsets to physical memory addresses at link time.

This dual memory map structure provides a high degree of flexibility during the loading of application code.

6.3.4.1 Defining the memory configuration

Each memory space is defined individually in the machine configuration file, by specifying a space identifier and a description, comprising:

- Memory space type: program or data.
- Word size, in bytes.
- Area list, defining one or more logical areas.

- The addresses in the logical areas, as positive integers, used as offsets to physical memory areas.
- Physical area type: single-port RAM (`ramsp`), dual-port RAM (`ramdp`) or ROM (`rom`).
- Attached spaces (optional). This is used for dual-port RAM only, when `ramdp` is the defined area type, to specify the two memory spaces. It is important that the code ensures address consistency between the corresponding spaces.

The syntax for defining a memory space is as follows:

```
space definition:
   define space <space identifier>:
      space_type;
      word_size;
      area_list;
   end define
;
space_type:
   program | data
;
word_size:
   word : byte_number
;
area_list:
   area | area_list area
;
area:
   address_value .. address_value : area_type opt_attached_spaces ;
;
area_type:
   ramsp | ramdp | rom
;
opt_attached_spaces:
   [ space_number , space_number ]
;
```

In Example 6-9, a one-word data space is defined, providing one logical area that can be used for the allocation of variables.

Example 6-9. Defining a data memory space

```
define space data_0 :
   data;
   word : 2;
   0x0000 .. 0xfffff : ramsp;
end define
```

Example 6-10 shows the definition of a 2-word program space in ROM.

Example 6-10. Defining a program memory space

```
define space pgm :
   program;
   word : 4;
   0x0000 .. 0x3fff : rom;
end define
```

At link time, these areas are mapped to the relevant physical memory space, and the actual addresses are calculated as offsets to the physical space starting address.

A data space can be divided into multiple logical areas, as shown in Example 6-11. When the compiler executes with cross-file optimization, it divides memory into these logical areas, and allocates variables accordingly.

Example 6-11. Defining multiple memory spaces

```
define space data_1 :
   data;
   word : 2;
   0x0000 .. 0x3fff : ramsp;
   0x0800 .. 0xffff : ramdp [data_0,data_1];
   0x10000 .. 0x13fff : ramsp;
   0x40000 .. 0x47fff : ramsp;
end define
```

Note: If you define new memory spaces in the machine configuration file, it is important that you also add these space definitions in the linker command file, to enable the linker to locate them at link time.

6.3.5 Application Configuration File

The application configuration file contains information about the interaction between the application software and the hardware. This file indicates to the compiler how to compile specific software units in order to ensure efficient sharing of hardware resources, in particular memory space. This information can be modified, to suit the requirements of your application.

The default application configuration file is named `minimal.appli`, and is located in the *`install-dir`*`/etc` directory. A different application configuration file can be specified, using the `-ma` option.

This file contains the following functional section types:

- Schedule section, which defines the entry points for the software units in the application, and their overlay capabilities for local variables. See Section 6.3.5.2, “Schedule section,” for details.
- Binding section, which specifies the links between software interrupt routines and hardware interrupt vectors, and between software-defined variables and fixed memory addresses. See Section 6.3.5.3, “Binding section,” for details.
- Overlay section, which specifies the overlay capabilities of global variables. See Section 6.3.5.4, “Overlay section,” for details.

6.3.5.1 File structure and syntax

More than one section of each type can be included in the file. The order in which the sections are defined in the file is unimportant. Each of the section types is optional and can be omitted.

The syntax of the application configuration file is as follows:

```
translation_unit:
   header_section
      configuration section_list
   end configuration
;
header_section:
   opt_version
;
opt_version:
   version string_content
;
section_list:
   section | section_list section
;
section:
   schedule_section | binding_section | overlay_section
;
```

6.3.5.2 Schedule section

The schedule section defines the entry point structure of an application, by specifying a "call tree". The call tree root is a C function name that defines the starting entry point for an application. Each node in the call tree is the name of an entry point of a unit that can be called during the execution of the application.

Each call tree node is defined as a call tree item, and is given a `ct` number that is unique for the application. A call tree item can be one of three types:

- Background task, identifying the main entry point, defined as `main`
- Interrupt handler, identifying an interrupt routine entry point, defined as `it_entry`, with a number that is used by the binding section to link to the associated hardware interrupt vector
- Task entry point, defined as `task_entry`, for example, an operating system task

The schedule section can optionally include an overlay specification, which informs the compiler which groups of local variables can use the same memory location during execution of the application. The compiler is able to overlay groups of local variables automatically, but only when it is clear that the two sets of variables do not share the same lifetime, and are therefore not active simultaneously. By specifying overlays in this file, you provide the necessary information in advance to help the compiler make more efficient use of memory space.

The overlay specification in the schedule section relates to local variables only. Overlays for global variables are specified in the overlay section, as described in Section 6.3.5.4, “Overlay section.”

The syntax of the schedule section is as follows:

```
schedule_list:
   schedule_elmt | schedule_list schedule_elmt
;
schedule_elmt:
   call_tree_list ; opt_overlay_spec
;
call_tree_list:
   call_tree_item | call_tree_list call_tree_item
;
call_tree_item:
   ct [int_constant] : main = ident ;
   ct [int_constant] : it_entry int_constant = ident ;
   ct [int_constant] : task_entry = ident ;
;
opt_overlay_spec:
   overlay = entry_overlay_list ;
;
entry_overlay_list:
   [group_list]
;
group_list:
   group | group_list, group
;
group:
   [entry_number_list]
;
entry_number_list:
   entry_number | entry_number_list, entry_number
;
entry_number:
   ct[int_constant] | int_constant
;
```

Example 6-12 defines two entry points, in addition to `main`. The function `task1()` is defined as a task entry point and the function `int_entry()` is defined as an interrupt handler.

Note that defining a function as an interrupt handler in the application configuration file is equivalent to using `#pragma interrupt` in the source file. For more details, see Section 6.4.5, "Interrupt Handlers."

Example 6-12. Defining additional entry points for an application

```
configuration

schedule
   ct[0] : main = _main;
   ct[1] : task_entry = _task1;
   ct[2] : it_entry 0 = _int_entry;
end schedule

binding
    place ___stackX on space 0 at 1;
end binding

end configuration
```

6.3.5.3 Binding section

The binding section performs the following functions:

- Assignment of fixed memory addresses to variables. A full memory address is specified with a memory binding directive, using the following syntax:

  `memory_binding_directive:`
  `place` *`full_ident`* `on` *`space_identifier`* `at` *`number`*

- Specification of the links between fixed interrupt entries and hardware interrupt vector addresses. An interrupt binding directive is used to specify an interrupt entry number, in the range 0-15, and the corresponding hardware vector number, in the range 1-16, using the following syntax:

  `it_binding_directive:`
  `place it_vector` *`interrupt_number`* `on` *`space_identifier`* `at` *`vector_number`*

The syntax of the binding section is as follows:

```
binding_directive:
   memory_binding_directive | it_binding_directive
;
binding_directive_list:
   binding_directive binding_directive_list ; binding_directive
;
binding section:
   binding
      binding_directive_list
   end binding
;
```

In Example 6-13, the location of global variable `mem` is fixed at absolute address `0x2000`:

Example 6-13. Placing a variable at an absolute location

```
configuration

schedule
   ct[0] : main = _main;
   ct[1] : it_entry 0 = _int_entry;
end schedule

binding
   place ___stackX on space 0 at 1;
   place _mem on space 0 at 0x2000;
end binding

end configuration
```

6.3.5.4 Overlay section

The overlay section specifies how the compiler should overlay global variables in order to further reduce the amount of memory required for data. As with local variables, in many cases the compiler can automatically detect that two data objects do not share the same lifetime and as a result, the memory allocated to these objects can be shared. This feature is needed for cases where the compiler cannot identify statically that the object lifetimes of global variables do not conflict.

Defining the overlay specification for global variables includes the following:

- Grouping the global variables into sets that can share the same memory space. In the overlay section syntax, the full identity is specified for each global variable, or list of variables, and defined as `symbol_list`.
- Defining each set of global variables as a `symbol_group`, associated with a `symbol_list` and an identifying group number.
- Specifying compatibility clauses that define which symbol groups can be overlaid, using the keyword `discern`.
- Specifying a list of compatibility clauses to indicate which symbol groups in the application can share the same memory space.

The syntax of the overlay section is as follows:

```
overlay section:
   overlay
      opt_overlay_spec
      compatibility_list
   end overlay
;
symbol_list:
   full_ident | symbol_list, full_ident
;
symbol_group:
   SG [number] = [symbol_list] ;
;
```

```
symbol_group_list:
   symbol_group | symbol_group_list symbol_group
;
sg_ref:
   SG [number]
;
sg_list:
   sg_ref | sg_list, sg_ref
;
compatibility_clause:
   discern_sg_ref : sg_list ;
;
compatibility_list:
   compatibility_clause | compatibility_list compatibility_clause
;
```

Example 6-14 shows an overlay section that specifies that the application will never access the two global arrays, `arr1` and `arr2`, at the same time, and they can therefore share the same physical memory location.

Example 6-14. Defining global variable overlays

```
configuration
schedule
   ct[0] : main = _main;
end schedule

binding
   place ___stackX on space 0 at 1;
end binding

overlay
   sg[0] = [_arr1];
   sg[1] = [_arr2];
   discern sg[0] : sg[1];
end overlay

end configuration
```

6.4 Calling Conventions

The compiler supports a stack-based calling convention. Additional calling conventions are also supported. Calling conventions can be mixed within the same application.

Specific calling conventions can be enforced using pragmas. For further information about the use of pragmas for this purpose, refer to Section 3.4.5, “Pragmas,” on page 3-52.

When compiling in separate compilation mode, non-static functions use the stack-based calling convention.

6.4.1 Stack Pointer

The SP register serves as the stack pointer, which points to the first available location. The stack direction is toward higher addresses, meaning that a push is implemented as `(sp)+`. The stack pointer must always be 8-byte aligned.

6.4.2 Stack-Based Calling Convention

The following calling conventions are supported:

- The first (left-most) function parameter is passed in `d0` if it is a numeric scalar or in `r0` if it is an address parameter, regardless of its size. The second function parameter is passed in `d1` if it is a numeric scalar, or in `r1` if it is an address parameter, regardless of its size. The remaining parameters are pushed on the stack. Long parameters are pushed on the stack using little endian mode, with the least significant bits in the lower addresses.
- Structures and union objects that can fit in a register are treated as numeric parameters, and are therefore candidates to be passed in a register.
- Numeric return values are returned in `d0`. Numeric address return values are returned in `r0`. Functions returning large structures, meaning structures that do not fit in a single register, receive and return the returned structure address in `r2`. The space for the returned object is allocated by the caller.
- Functions with a variable number of parameters allocate all parameters on the stack.
- Parameters are aligned in memory according to the base parameter type, with the exception of characters and unsigned characters that have a 32-bit alignment.

The following registers are saved by the caller: `d0-d5`, `r0-r5`, `n0-n3`.

The following registers are saved by the callee, if actually used: `d6-d7`, `r6-r7`.

The compiler assumes that the current settings of the following operating control bits are correct:

- Saturation mode
- Round mode
- Scale bits

The application is responsible for setting these mode bits correctly.

Example 6-15 shows two function calls and the parameters that are allocated for each call.

Example 6-15. Function call and allocation of parameters

Function call:

```
foo(int a1, struct fourbytes a2, struct eightbytes a3, int *a4)
```

Parameters:

```
a1 - in d0
a2 - in d1
a3 - in stack
a4 - in stack
```

Function call:

```
bar(long *b1, int b2, int b3[])
```

Parameters:

```
b1 - in r0
b2 - in d1
b3 - in stack.
```

The stack-based calling convention must be used when calling functions that are required to maintain a calling stack.

The compiler is able to use optimized calling sequences for functions that are not exposed to external calls.

Local and formal variables are allocated on the stack and in registers.

Table 6-6 summarizes register usage in the stack-based calling convention.

Table 6-6. Register Usage in the Stack-based Calling Convention

Register	Used as	Caller Saved	Callee Saved
d0	First numeric parameter Return numeric value	+	
d1	Second numeric parameter	+	
d2-d5		+	
d6-d7			+
d8-d15		+	
r0	First address parameter Return address value	+	
r1	Second address parameter	+	
r2	Big object return address	+	
r3-r5		+	
r6	Optional argument pointer		+
r7	Optional frame pointer		+
n0-n3, m0-m3		+	

6.4.3 Optimized Calling Sequences

A stack-less convention may be used when calling functions that are not reentrant, if this technique generates more efficient code than other conventions.

This convention will be used only if the function is not visible to external code.

When using this calling convention, local variables may be allocated statically, meaning not on a stack. Functions with mutually exclusive lifetimes may share space for their local variables.

Actual parameters are placed by the calling function at the locations allocated for the formal parameters in the called function. The compiler may use registers and memory locations as required when allocating locations for the formal parameters.

Under this calling convention, all registers are classified as caller-saved.

Return values from functions are placed in the space allocated for the function return value in the calling function. The compiler may use a register or a memory location as the space for the function return value.

6.4.4 Stack Frame Layout

The stack pointer points to the top (high address) of the stack frame. Space at higher addresses than the stack pointer is considered invalid and may actually be unaddressable. The stack pointer value must always be a multiple of eight.

Figure 6-2 shows typical stack frames for a function, indicating the relative position of local variables, parameters and return addresses.

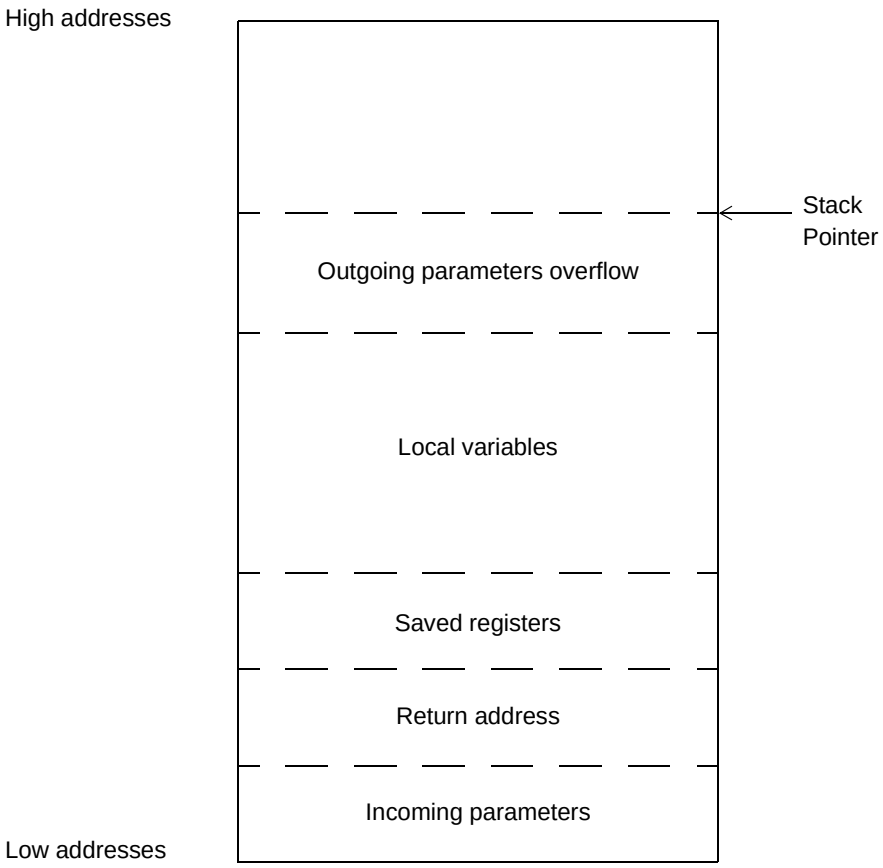


Figure 6-2. Stack Frame Layout

The caller must reserve stack space for return variables that do not fit in registers. This return buffer area is typically located with the local variables. This space is typically allocated only for functions that make calls that return structures. Beyond these requirements, a function is free to manage its stack frame as necessary.

The outgoing parameter overflow block is located at the top (higher addresses) of the frame. Any incoming argument spill generated for `varargs` and `stdargs` processing must be at the bottom (low addresses) of the frame.

The caller puts argument variables that do not fit in registers into the outgoing parameter overflow area. If all arguments fit in registers, this area is not required. A caller has the option to allocate argument overflow space sufficient for the worst case call, use portions of this space as necessary, and/or leave the stack pointer unchanged between calls.

Local variables that do not fit into the local registers are allocated space in the local variables area of the stack. If there are no such variables, this area is not required.

6.4.5 Interrupt Handlers

Functions which require no parameters and return no result can be designated as interrupt handler functions. The process of creating an interrupt handler function includes:

- Defining the function as an interrupt handler
- Linking the function to the appropriate interrupt vector entry

An interrupt handler can be be defined in one of two ways:

- Using `#pragma interrupt` in the source code. For more detail about this pragma, refer to Section 3.4.5.3.4, “Defining a function as an interrupt handler,” on page 3-56.
- Defining an interrupt entry point in the application, by editing the schedule section of the application configuration file, as described in Section 6.3.5, “Application Configuration File.”

To create the link between the function and the interrupt vector entry, you can use any one of the following options:

- In the code that calls the function, place a call to the handler function in the interrupt vector entry.
- Use the `signal.h` library function to insert a call to the interrupt handler function into the required interrupt vector entry. For syntax details, see Section 7.8, “Signal Handling (signal.h),” on page 7-11.
- If the function is very small, you can embed it in the interrupt vector entry, by modifying the startup code file, `crt.asm`. The size of each interrupt vector entry is 64 bytes. With this option, there is no need for an explicit call from the vector to the function.

Interrupt handler functions always follow the stack-based calling convention. When an interrupt function is called, the interrupt handler saves all registers and all other resources that are modified by the function. Upon returning from the function all registers and hardware loop state saved at entry are restored to their original state.

Local variables are saved on the stack. Interrupt handlers that are known to be non-interruptible may also allocate data statically.

Return from interrupt is implemented using an RTE instruction.

6.4.6 Frame Pointer and Argument Pointer

The compiler does not use a frame pointer or an argument pointer.

If, however, the use of a frame pointer or an argument pointer is required by external code, `r7` may be allocated as a frame pointer and `r6` as an argument pointer. When these registers are allocated as frame pointer and/or argument pointer they should be saved and restored as part of the function prolog/epilog code.

6.4.7 Hardware Loops

All hardware loop resources are available for use by the compiler. It is assumed that no nesting occurs when entering a function. As a result, a function may use all 4 nesting levels for its own use. An additional side effect of this assumption is that loops that include a function call as part of the loop code cannot be implemented using hardware loops, unless the compiler can infer the nesting level of the called function from static variables known at compilation time.

Loops are nested beginning with loop counter 3 at the innermost nesting level.

6.4.8 Operating Modes

The compiler makes the following assumptions regarding runtime operating modes and the machine state:

- All modulo (M) registers (`m0-m3`) are assumed to contain the value 0 (linear addressing). If the use of an M register is required, the using function must restore the M register to the value 0 before returning or before calling another function.
- No specific settings are assumed for the operating mode settings in the EMR register. The compiler assumes that the default settings in the startup code, including saturation modes, rounding mode and scale bits, are set by the user. You can control and change these operating modes during execution of the application. Refer to the SC100 architecture documentation for further details.

6.5 Saturation

By default, saturation is turned off; however, there is a switch option that allows you to turn saturation on. It is important that you intricately know your program in order to get the desired results on overflow. Choose the compilation model based upon your program.

Use a combination of the `-fractional` and `-no_overflow` options to tell the compiler the overflow requirements of your particular program. `-fractional` indicates that your code contains intrinsics that rely on saturation when overflowing. The compiler gives an error message to standard error whenever it finds an intrinsic in the compilation module and the `-fractional` switch is not specified. The `-no_overflow` option is an optimization that allows the compiler to relax the definition of unsigned integers overflowed by saturating rather than modular wrap-around behavior.

6.5.1 Saturation switches

The `-fractional` switch combined with the `-no_overflow` switch determines the model the compiler uses to generate code for the SC140. Combined, the two switches give the compiler three approaches to generate code:

- **default**: saturation mode-bit off, generating non-intrinsic code that uses the saturation bit. When you do not use `-fractional` and/or `-no_overflow`, the compiler automatically uses this default setting. This is the most efficient and correct for generic ANSI/ISO C codes that do not use fractional operations.
- **`-fractional`**: saturation mode-bit on, generating non-intrinsic code that does not rely on saturation bit. This generates intrinsic code that has the proper saturation overflow semantics and non-intrinsic code that correctly conforms to ANSI/ICO C overflow rules. This incurs a performance degradation over the intrinsic code today, but will be correct.
- **`-fractional` and `-no_overflow`**: saturation mode-bit on, generating non-intrinsic code that uses the saturation bit. This code does not honor the ANSI/ICO C defined behavior of overflow on unsigned values as they will saturate on overflow. This may result in an incorrect program. The `-fractional` and `-no_overflow` combination is most efficient on C code that uses the fractional intrinsics. Use this combination for compatibility with prior StarCore compiler releases.

6.5.2 Saturation states

The following table illustrates the various saturation states based on the combinations of the compiler switches. For example, if you have `-fractional` turned off, and `-no_overflow` turned on, then the saturation bit is off, and the compiler's performance is fast.

Option Combinations		Results	
`-fractional`	**`-no_overflow`**	**sat_bit sr{2}**	**Compiler performance**
off	off	off	fast
off	on	off	fast
on	off	on	slow
on	on	on	fast

Chapter 7
Runtime Libraries

This chapter describes the C libraries and I/O libraries that the SC100 C compiler supports. Each table in this chapter is organized in alphabetical order, according to the file, function, or constant name in the first column in the table.

Table 7-2 summarizes the ISO standard C libraries that the compiler supports.

Table 7-1. Supported ISO Libraries

Header file	Description	Section	Page
`ctype.h`	Character Typing and Conversion	7.2	7-2
`float.h`	Floating Point Characteristics	7.3	7-4
`limits.h`	Integer Characteristics	7.4	7-8
`locale.h`	Locales	7.5	7-8
`math.h`	Floating Point Math	7.6	7-9
`setjmp.h`	Nonlocal Jumps	7.7	7-11
`signal.h`	Signal Handling	7.8	7-11
`stdarg.h`	Variable Arguments	7.9	7-11
`stddef.h`	Standard Definitions	7.10	7-12
`stdio.h`	I/O Library	7.11	7-12
`stdlib.h`	General Utilities	7.12	7-15
`string.h`	String Functions	7.13	7-17
`time.h`	Time Functions	7.14	7-20

The non-ISO C library supported by the compiler is shown in Table 7-2. This library contains the built-in intrinsic functions supplied with the compiler.

Table 7-2. Supported Non-ISO Libraries

Header file	Description	Section	Page
`prototype.h`	Built-in Intrinsic Functions	7.15	7-21

7.1 Providing Runtime Libraries

Starting with the StarCore Tools release 2.2.0, the compiler includes the sources and the makefile necessary for rebuilding the runtime libraries. The default libraries are located in the `lib` directory, which resides in the SW Tools directory (ex: `$ SC100_HOME/lib`). This `lib` directory has the libraries compiled with the small (`-Os`) optimization. Another set is provided in the `lib_debug` directory that were built with the debug (`-g`) option. When using the compiler, please note that the runtime libraries that are used (linked during compilation of a target source program) are the ones in the `lib` directory.

7.1.1 Using Libraries with debug

To use libraries with debug:

1. Rename the `lib` directory to `lib_small`.
2. Rename the `lib_debug` directory to `lib`.

7.1.2 Building the Libraries

The following example shows the steps for rebuilding the libraries under solaris with -O2 optimization.

1. Source the appropriate env.sh, for example:
   ```
   $ source /sw_tools-2.x.x/env.sh
   ```
2. Change the directory to the `rtlib` directory, where `Make` file resides.
   ```
   $ cd /sw_tools-2.x.x/src/rtlib/
   ```
3. Edit the CFLAGS variable in `Make`file to match your desired compiler options.
   ```
   CFLAGS = -O2
   ```
4. Run `Make`.
   ```
   $ make install
   ```

Note: The libraries are copied into the `lib` directory by the `Make` install command, thereby overwriting any existing libraries in the `lib` directory. You can only build the libraries in the Solaris environment.

7.2 Character Typing and Conversion (ctype.h)

The `ctype.h` library contains the following function types:

- Testing functions
- Conversion functions

7.2.1 Testing Functions

Table 7-3 lists the testing functions that the compiler supports.

Table 7-3. Testing Functions

Function	Purpose
`int isalnum(int)`	Tests for `isalpha` or `isdigit`
`int isalpha(int)`	Tests for `isupper` or `islower`
`int iscntrl(int)`	Tests for any control character
`int isdigit(int)`	Tests for decimal digit character
`int isgraph(int)`	Tests for any printing character except space
`int islower(int)`	Tests for lowercase alphabetic character
`int isprint(int)`	Tests for any printing character including space
`int ispunct(int)`	Tests for any printing character not space and not `isalnum`
`int isspace(int)`	Tests for white-space characters
`int isupper(int)`	Tests for uppercase alphabetic character
`int isxdigit(int)`	Tests for hexadecimal digit character

7.2.2 Conversion Functions

Table 7-4 lists the conversion functions that the compiler supports.

Table 7-4. Conversion Functions

Function	Purpose
`int tolower(int)`	Converts uppercase alphabetic character to the equivalent lower case character
`int toupper(int)`	Converts lowercase alphabetic character to the equivalent uppercase character

7.3 Floating Point Characteristics (float.h)

The compiler represents floating point numbers using IEEE format (ANSI/IEEE Std 754-1985). Only single precision floating point format is supported.

The contents of `float.h` are listed in Table 7-5.

Table 7-5. Contents of File float.h

Constant	Value	Purpose
FLT_DIG	6	Number of decimal digits of precision
DBL_DIG	15	
LDBL_DIG	15	
FLT_EPSILON	1.19209290E-07F	Minimum positive number χ such that 1.0 + χ does not equal 1.0
DBL_EPSILON	2.2204460492503131E-16	
LDBL_EPSILON	2.2204460492503131E-16L	
FLT_MANT_DIG	24	Number of base-2 digits in the mantissa
DBL_MANT_DIG	53	
LDBL_MANT_DIG	53	
FLT_MAX_10_EXP	38	Maximum positive integers *n* such that 10^n is representable
DBL_MAX_10_EXP	308	
LDBL_MAX_10_EXP	308	
FLT_MAX_EXP	128	Maximum positive integer *n* such that 2^{n-1} is representable
DBL_MAX_EXP	1024	
LDBL_MAX_EXP	1024	
FLT_MAX	3.40282347E+38F	Maximum positive floating point number
DBL_MAX	1.7976931348623157E+308	
LDBL_MAX	1.7976931348623157E+308L	
FLT_MIN_10_EXP	(-37)	Minimum negative integer *n* such that 10^n is representable
DBL_MIN_10_EXP	(-307)	
LDBL_MIN_10_EXP	(-307)	
FLT_MIN_EXP	(-125)	Minimum negative integer *n* such that 2^{n-1} is representable
DBL_MIN_EXP	(-1021)	
LDBL_MIN_EXP	(-1021)	
FLT_MIN	1.175494351E-38F	Minimum positive number
DBL_MIN	2.2250738585072014E-308	
LDBL_MIN	2.2250738585072014E-308L	
FLT_RADIX	2	Floating point exponent is expressed n radix 2.
FLT_ROUNDS	-1	Floating point rounding is to nearest even number.

7.3.1 Floating Point Library Interface (fltmath.h)

This header file defines the software floating point library interface. Most of these functions are called by the code generator of the compiler for floating point expression evaluation. They may also be called directly by user code.

The floating point library supports the full IEEE-754 single-precision floating point standard.

Three configuration parameters and one status word can be used. Each of these is described in the following sections.

- `Round_Mode`
- `FLUSH_TO_ZERO`
- `IEEE_Exceptions`
- `EnableFPExceptions`

7.3.1.1 Round_Mode

Four rounding modes are supported:

- `ROUND_TO NEAREST_EVEN`. The representable value nearest to the infinitely precise intermediate value is the result. If the two nearest representable values are equally near (tie), then the one with the least significant bit equal to zero (even) is the result.
- `ROUND_TOWARDS_ZERO`. The result is the value closest to, and no greater in magnitude than, the infinitely precise intermediate result.
- `ROUND_TOWARDS_MINUS_INF`. The result is the value closest to and no greater than the infinitely precise intermediate result (possibly minus infinity).
- `ROUND_TOWARDS_PLUS_INF`. The result is the value closest to and no less than the infinitely precise intermediate result (possibly plus infinity).

By default, the rounding mode is set to `ROUND_TO_NEAREST_EVEN`.

Following is an example of changing the round mode to `ROUND_TOWARDS_MINUS_INF`:

Example 7-1. Changing the round mode

```
#include <fltmath.h>
. . .
Round_Mode = ROUND_TOWARDS_MINUS_INF.
```

7.3.1.2 FLUSH_TO_ZERO

This is a boolean configuration item that sets the behavior of un-normalized numbers. When set to true (default) all un-normalized values are flushed to zero. This leads to better performance, but a smaller dynamic range.

For example, to disable the `FLUSH_TO_ZERO` option, you would specify the following:

Example 7-2. Disabling flushing to zero

```
#include <fltmath.h>
. . .
FLUSH_TO_ZERO = 0;
```

7.3.1.3 IEEE_Exceptions

This is a status word that represents the IEEE exceptions that were raised during the last floating point operation. By default, the floating point library sets these values but does not handle any of these exceptions.

The following exceptions are supported:

- `IEEE_Inexact`
- `IEEE_Divide_By_Zero`
- `IEEE_Underflow`
- `IEEE_Overflow`
- `IEEE_Signaling_Nan`

See the IEEE standard for the exact description of these exceptions.

Following is an example of how to use the exception status word:

Example 7-3. Using the exception status word

```
#include <fltmath.h>
float x,y;
. . .
x = x*y;
if (IEEE_Exceptions & IEEE_Overflow)
{
<handle overflow>
}
```

7.3.1.4 EnableFPExceptions

This is a bit field mask. Setting a flag enables raising an SIGFPE signal if the last FP operation raised this exception. For example:

Example 7-4. Setting a signal for exceptions

```
#include <fltmath.h>
#include <signal.h>
void SigFPHandler(int x)
{
switch (IEEE_Exceptions)
{
case IEEE_Overflow:
. . .
case IEEE_Divide_by_zero:
. . .
}
}
float x,y;
. . .
EnableFPExceptions = IEEE_Overflow | IEEE_Divide_by_zero;
signal(SIGFPE, SigFPHandler)
x = x*y /*This will raise SIGFPE if overflow or divide by zero occur */
```

This example installs a signal for handling overflow and divide by zero exceptions.

Note: Because the signal handling installs the handler address into the interrupt table, this example works only if the interrupt vector table is located in RAM. If the call to SIGNAL is not able to install the new handler, SIG_ERR is returned.

7.4 Integer Characteristics (limits.h)

The contents of `limits.h` are listed in Table 7-6.

Table 7-6. Contents of File limits.h

Constant	Value	Purpose
CHAR_BIT	8	Width of `char` type, in bits
CHAR_MAX	127	Maximum value for `char`
CHAR_MIN	-128	Minimum value for `char`
INT_MAX	2147483647	Maximum value for `int`
INT_MIN	(-2147483647-1)	Minimum value for `int`
UINT_MAX	4294967295u	Maximum value for `unsigned int`
LONG_MAX	2147483647	Maximum value for `long int`
LONG_MIN	(-2147483647-1)	Minimum value for `long int`
ULONG_MAX	4294967295uL	Maximum value for `unsigned long int`
MB_LEN_MAX	2	Maximum number of bytes in a multibyte character
SCHAR_MAX	127	Maximum value for `signed char`
SCHAR_MIN	-128	Minimum value for `signed char`
UCHAR_MAX	255	Maximum value for `unsigned char`
SHRT_MAX	32767	Maximum value for `short int`
SHRT_MIN	-32768	Minimum value for `short int`
USHRT_MAX	65535u	Maximum value for `unsigned short int`

7.5 Locales (locale.h)

Table 7-7 lists the locales functions that the compiler supports.

Table 7-7. Locale Functions

Function	Purpose
`localeconv(void)`	
`setlocale(int category, const char* locale)`	

Note: These functions are supported for compatibility purposes, and have no effect.

7.6 Floating Point Math (math.h)

The `math.h` library contains the following function types:

- Trigonometric functions
- Hyperbolic functions
- Exponential and logarithmic functions
- Power functions
- Other functions

The compiler runtime environment fully implements the `math.h` library using floating point emulation.

7.6.1 Trigonometric Functions

Table 7-8 lists the trigonometric functions that the compiler supports.

Table 7-8. Trigonometric Functions

Function	Purpose
`double acos(double)`	arc cosine
`double asin(double)`	arc sine
`double atan(double)`	arc tangent
`double atan2(double, double)`	arc tangent2
`double cos(double)`	cosine
`double sin(double)`	sine
`double tan(double)`	tangent

7.6.2 Hyperbolic Functions

Table 7-9 lists the hyperbolic functions that the compiler supports.

Table 7-9. Hyperbolic Functions

Function	Purpose
`double cosh(double)`	Hyperbolic cosine
`double sinh(double)`	Hyperbolic sine
`double tanh(double)`	Hyperbolic tangent

7.6.3 Exponential and Logarithmic Functions

Table 7-10 lists the exponential and logarithmic functions that the compiler supports.

Table 7-10. Exponential and Logarithmic Functions

Function	Purpose
`double exp(double)`	Exponential
`double frexp(double, int*)`	Splits floating point into fraction and exponent
`double ldexp(double, int)`	Computes value raised to a power
`double log(double)`	Natural logarithm
`double log10(double)`	Base ten (10) logarithm
`double modf(double, double*)`	Splits floating point into fraction and integer

7.6.4 Power Functions

Table 7-11 lists the power functions that the compiler supports.

Table 7-11. Power Functions

Function	Purpose
`double pow(double, double)`	Raises value to a power
`double sqrt(double)`	Square root

7.6.5 Other Functions

Table 7-12 lists the other functions that the compiler supports.

Table 7-12. Other Functions

Function	Purpose
`double ceil(double)`	Ceiling
`double fabs(double)`	Floating point absolute number
`double floor(double)`	Floor
`double fmod(double, double)`	Floating point remainder

7.7 Nonlocal Jumps (setjmp.h)

Table 7-13 lists the nonlocal jumps that the compiler supports.

Table 7-13. Nonlocal Jumps

Function	Purpose
`typedef unsigned int jmp_buf[32]`	Buffer used to save the execution context
`void longjmp(jmp_buf, int)`	Nonlocal jump
`int setjmp(jmp_buf)`	Nonlocal return

7.8 Signal Handling (signal.h)

Table 7-14 lists the signal handling that the compiler supports.

Table 7-14. Signal Handling (signal.h)

Function	Purpose
`int raise(int)`	Raises a signal
`void(*signal(int, void (*)( int ))) (int)`	Installs a signal handler

7.9 Variable Arguments (stdarg.h)

Table 7-15 lists the variable arguments that the compiler supports.

Table 7-15. Variable Arguments (stdarg.h)

Function	Purpose
`va_arg(_ap, _type) (*(_type*)((_ap) -= sizeof(_type)))`	Returns next parameter in argument list
`va_end(_ap) (void)0`	Performs cleanup of argument list
`va_list`	Type declaration of variable argument list
`va_start(_ap, _parmN) (void)(_ap = (char*)&_parmN)`	Performs initialization of argument list

7.10 Standard Definitions (stddef.h)

Table 7-16 lists the standard definitions that the compiler supports.

Table 7-16. Standard Definitions (stddef.h)

Function	Purpose
`NULL((void*)0)`	Null pointer constant
`offsetof(type, member)`	Field offset in bytes from start of structure
`typedef int ptrdiff_t`	Signed integer type resulting from the subtraction of two pointers
`typedef unsigned int size_t`	Unsigned integer type that is the data type of the `sizeof` operator
`typedef unsigned short wchar_t`	Wide character type, as defined in ISO C

7.11 I/O Library (stdio.h)

The `stdio.h` library contains the following function types:

- Input functions
- Stream functions
- Output functions
- Miscellaneous I/O functions

7.11.1 Input Functions

Table 7-17 lists the input functions that the compiler supports.

Table 7-17. Input Functions

Function	Purpose
`char* fgets(char*, int, FILE*)`	Reads characters to the specified stream
`int fgetc(FILE*)`	Inputs a single character if available from specified stream
`size_t fread(void*, size_t, size_t, FILE*)`	Inputs a size number of characters from `stdin`
`int fscanf(FILE*, const char*, ...)`	Inputs text from the specified stream
`int getc(FILE*)`	Inputs a single character if available from specified stream
`int getchar(void)`	Inputs a single character if available from `stdin`
`int scanf(const char*, ...)`	Inputs text from `stdin`
`int sscanf(const char*, const char*, ...)`	Inputs text from specified string

7.11.2 Stream Functions

Table 7-18 lists the stream functions that the compiler supports.

Table 7-18. Stream Functions

Function	Purpose
`void clearerr(FILE*)`	Clears the EOF and error indicators for the specified stream
`int fclose(FILE*)`	Flushes the specified stream and closes the file associated with it
`int feof(FILE*)`	Tests the EOF indicator for the specified stream
`int ferror(FILE*)`	Tests the error indicator for the specified stream
`int fgetpos(FILE*, fpos_t*)`	Stores the current value of the file position indicator for the specified stream
`FILE* freopen(const char*,const char*,FILE*)`	Opens the specified file in the specified mode, using the specified stream
`int fseek(FILE*, long int, int)`	Sets the file position indicator for the specified stream
`int fsetpos(FILE*, const fpos_t*)`	Sets the file position indicator for the specified stream to the specified value
`long int ftell(FILE*)`	Retrieves the current value of the file position indicator for the current stream
`int remove(const char*)`	Makes the specified file unavailable by its defined name
`int rename(const char*, const char*)`	Assigns to the specified file a new filename
`void rewind(FILE*)`	Sets the file position indicator for the specified stream to the beginning of the file
`void setbuf(FILE*, char*)`	Defines a buffer and associates it with the specified stream. A restricted version of `setvbuf()`
`int setvbuf(FILE*, char*, int, size_t)`	Defines a buffer and associates it with the specified stream
`stderr`	Standard error stream (Value = 3)
`stdin`	Standard input stream (Value = 1)
`stdout`	Standard output stream (Value = 2)
`FILE* tmpfile(void)`	Creates a temporary file
`char* tmpnam(char*)`	Generates a valid filename, meaning a filename that is not in use, as a string

7.11.3 Output Functions

Table 7-19 lists the output functions that the compiler supports.

Table 7-19. Output Functions

Function	Purpose
`int fprintf(FILE*, const char*, ...)`	Outputs the specified text to the specified stream
`int fputc(int, FILE*)`	Outputs a single character to the specified stream
`int fputs(const char*, FILE*)`	Outputs a string to the specified stream
`size_t fwrite(const void*, size_t, size_t, FILE*)`	Outputs a size number of characters to `stdout`
`char* gets(char*)`	Reads characters into the user's buffer
`void perror(const char*)`	Outputs an error message
`int printf(const char*, ...)`	Outputs the specified text to `stdout`
`int putc(int, FILE*)`	Outputs a single character to the specified stream
`int putchar(int)`	Outputs a single character
`int puts (const char*)`	Outputs the string to `stdout`, followed by a newline
`int sprintf(char*, const char*, ...)`	Outputs the specified text to the specified buffer
`int vfprintf(FILE*, const char*, va_list)`	Outputs the variable arguments to the specified stream
`int vprintf(const char*, va_list)`	Outputs the variable arguments to `stdout`
`int vsprintf(char*, const char*, va_list)`	Outputs the variable arguments to the specified buffer

7.11.4 Miscellaneous I/O Functions

Table 7-20 lists the miscellaneous I/O functions that the compiler supports.

Table 7-20. Miscellaneous I/O Functions

Function	Purpose
`int fflush(FILE*)`	Causes the output buffers to be emptied to their destinations
`FILE* fopen(const char*, const char*)`	Associates a stream with a file
`int ungetc(int, FILE*)`	Moves the character back to the head of the input stream

7.12 General Utilities (stdlib.h)

The `stdlib.h` library contains the following function types:

- Memory allocation functions
- Integer arithmetic functions
- String conversion functions
- Searching and sorting functions
- Pseudo random number generation functions
- Environment functions
- Multibyte functions

7.12.1 Memory Allocation Functions

Table 7-21 lists the memory allocation functions that the compiler supports.

Table 7-21. Memory Allocation Functions

Function	Purpose
`void free(void*)`	Returns allocated space to heap
`void* calloc(size_t, size_t)`	Allocates heap space initialized to zero
`void* malloc(size_t)`	Allocates heap space
`void* realloc(void*, size_t)`	Allocates a larger heap space and returns previous space to heap

7.12.2 Integer Arithmetic Functions

Table 7-22 lists the integer arithmetic functions that the compiler supports.

Table 7-22. Integer Arithmetic Functions

Function	Purpose
`int abs(int)`	Absolute value
`div_t div(int, int)`	Quotient and remainder
`long labs(long int)`	Computes absolute value and returns as `long`
`ldiv_t ldiv(long int, long int)`	Quotient and remainder of `long int`

7.12.3 String Conversion Functions

Table 7-23 lists the string conversion functions that the compiler supports.

Table 7-23. String Conversion Functions

Function	Purpose
`double atof(const char*)`	String to float
`int atoi(const char*)`	String to int
`long int atol(const char*)`	Long
`double strtod(const char*, char**)`	Double
`long int strtol(const char*, char**, int)`	Long
`unsigned long int strtoul(const char*, char**, int)`	Unsigned long

7.12.4 Searching and Sorting Functions

Table 7-24 lists the searching and sorting functions that the compiler supports.

Table 7-24. Searching and Sorting Functions

Function	Purpose
`void *bsearch(const void*, const void*, size_t, size_t, int(*)(const void*, const void*))`	Binary search
`void *qsort(void*, size_t, size_t, int(*)(const void*, const void*))`	Quick sort

7.12.5 Pseudo Random Number Generation Functions

Table 7-25 lists the pseudo random number generation functions that the compiler supports.

Table 7-25. Pseudo Random Number Generation Functions

Function	Purpose
`int rand(void)`	Random number generator
`void srand(unsigned int)`	Initializes the random number generator

7.12.6 Environment Functions

Table 7-26 lists the environment functions that the compiler supports.

Table 7-26. Environment Functions

Function	Purpose
`void abort(void)`	Causes an abnormal termination
`int atexit(void (*)(void))`	Registers a function to be called at normal termination
`void exit(int)`	Causes a normal termination
`char *getenv(const char *name)`[1]	Gets environment variable
`int system(const char *string)`[1]	Passes command to host environment

1. This function is supported for compatibility purposes and has no effect.

7.12.7 Multibyte Character Functions

Table 7-27 lists the multibyte character functions that the compiler supports.

Table 7-27. Multibyte Character Functions

Function	Purpose
`int mblen(const char*, size_t)`	Multibyte string length
`size_t mbstowcs(wchar_t*, const char*, size_t)`	Converts multibyte string to wide character string
`int mbtowc(wchar_t*, const char*, size_t)`	Converts multibyte to wide character
`int wctomb(char*, wchar_t)`	Converts wide character to multibyte
`size_t wcstombs (char*, const wchar_t*, size_t)`	Converts wide character string to multibyte string

7.13 String Functions (string.h)

The `string.h` library contains the following function types:

- Copying functions
- Concatenation functions
- Comparison functions
- Search functions
- Other functions

7.13.1 Copying Functions

Table 7-28 lists the copying functions that the compiler supports.

Table 7-28. Copying Functions

Function	Purpose
`void* memcpy(void*, const void*, size_t)`	Copies data
`void* memmove(void*, const void*, size_t)`	Swaps data
`char* strcpy(char*, const char*)`	Copies a string
`char* strncpy(char*, const char*, size_t)`	Copies a string of a maximum length

7.13.2 Concatenation Functions

Table 7-29 lists the concatenation functions that the compiler supports.

Table 7-29. Concatenation Functions

Function	Purpose
`char* strcat(char*, const char*)`	Concatenates a string to the end of another string
`char* strncat(char*, const char*, size_t)`	Concatenates a string of specified maximum length to the end of another string

7.13.3 Comparison Functions

Table 7-30 lists the comparison functions that the compiler supports.

Table 7-30. Comparison Functions

Function	Purpose
`int memcmp(const void*, const void*, size_t)`	Compares data
`int strcmp(const char*, const char*)`	Compares strings
`int strcoll(const char*, const char*)`	Compares strings based on locale
`int strncmp(const char*, const char*, size_t)`	Compares strings of maximum length
`size_t strxfrm(char*, const char*, size_t)`	Transforms a string into a second string of the specified size

7.13.4 Search Functions

Table 7-31 lists the search functions that the compiler supports.

Table 7-31. Search Functions

Function	Purpose
`void* memchr(const void*, int, size_t)`	Searches for a value in the first number of characters
`char* strchr(const char*, int)`	Searches a string for the first occurrence of `char`
`size_t strcspn(const char*, const char*)`	Searches a string for the first occurrence of `char` in string set and returns the number of characters skipped
`char strpbrk(const char*, const char*)`	Searches a string for the first occurrences of `char` in string set and returns a pointer to that location
`char* strrchr(const char*, int)`	Searches a string for the last occurrence of `char`
`size_t strspn(const char*, const char*)`	Searches a string for the first occurrence of `char` not in string set.
`char* strstr(const char*, const char*)`	Searches a string for the first occurrence of string
`char* strtok(char*, const char*)`	Separates a string into tokens

7.13.5 Other Functions

Table 7-32 lists the other functions that the compiler supports.

Table 7-32. Other Functions

Function	Purpose
`void* memset(void*, int, size_t)`	Copies a value into each number of characters
`char* strerror(int)`	Returns string for associated error condition
`size_t strlen(const char*)`	Returns size of string

7.14 Time Functions (time.h)

Table 7-33 lists the time functions that the compiler supports.

Table 7-33. Time Functions

Function	Purpose
`char *asctime(const struct tm *timeptr)`	Converts time to ASCII representation
`clock_t clock()`	Returns processor time
`typedef unsigned long clock_t`	Type used for measuring time
`char *ctime (const time_t *timer)`	Converts time to ASCII representation
`double difftime(time_t time1, time_t time0)`	Returns difference in seconds
`time_t mktime(struct tm *timeptr)`	Converts `struct tm` to `time_t`
`size_t strftime (char *s, size_t maxsize, const char *format, const struct tm *timeptr)`	Converts an ASCII string to `time_t`
`time_t time(time_t *timer)`	Returns processor time (same as `clock`)
`typedef unsigned long time_t`	Type used for measuring time
`struct tm *gmtime(const time_t *timer)`	Returns time in GMT time zone
`struct tm *localtime(const time_t *timer)`	Returns time in local time zone

7.14.1 Time Constant

Table 7-34 shows the time constant that the compiler supports.

Table 7-34. Time Constant

Constant	Value	Purpose
`CLOCKS_PER_SEC`	TBD	

7.14.2 Process Time

The `clock` function returns the current value of the system timer. This function must be configured to match the actual system timer configuration. The timer is started and set for a maximum period during the initialization of any C program that references the `clock` function, and is used only by this function. The return value of `clock` has type `clock_t`, which is `unsigned long`.

The following example shows how to use the `clock` function to time your application:

Example 7-5. Timing an application

```
#include <time.h>
clock_t start, end, elapsed;
/* . . . application setup . . . */
start = clock( );
/* . . . application processing . . . */
end = clock( );
elapsed = end - start; /* Assumes no wrap-around */
printf("Elapsed time: %Lu * 2 cycles. \n", elapsed);
```

7.15 Built-in Intrinsic Functions (prototype.h)

The compiler supports a set of built-in intrinsic functions that enable fractional operations to be implemented using integer data types, by mapping directly to SC100 assembly instructions.

Table 7-35 lists these built-in intrinsic functions.

Table 7-35. Built-in Intrinsic Functions

Function	Purpose
`short abs_s(short var1)`	Short absolute value of `var1`. For example, the result of `abs_s(-32768)`is `+32767`.
`short add(short var1,short var2)`	Short add. Performs the addition `var1+var2` with overflow control and saturation. The 16-bit result is set at `+32767` when overflow occurs, or at `-32768` when underflow occurs.
`BitReverseUpdate`	Increments the iterator with bit reverse.
`Word64 D_add(Word64 D_var1, Word64 D_var2)`	Double precision add. Performs the addition `D_var1+D_var2` with overflow control and saturation.
`short D_cmpeq(Word64 D_var1, Word64 D_var2)`	Double precision compare equal. Compares two 64-bit values and returns a 16-bit result containing the value ??? if the values are equal, or ??? if they are not.
`short D_cmpgt(Word64 D_var1, Word64 D_var2)`	Double precision compare greater than. Compares two 64-bit values and returns a 16-bit result containing ???.
`long D_extract_h(Word64 D_var1)`	Double precision extract high. Returns the 32 MSB of the 64-bit value `D_var1`.
`unsigned long D_extract_l (Word64 D_var1)`	Double precision extract low. Returns the 32 LSB of the 64-bit value `D_var1` as an unsigned 32-bit value.
`Word64 D_mac(Word64 D_var3, long L_var1,long L_var2)`	Double precision multiply accumulate. Multiplies `L_var1` by `L_var2` and shifts the result left by 1. Adds the 64-bit result to `L_var3` with saturation, and returns a 64-bit result. For example: `D_mac(D_var3,L_var1,L_var2) = D_add(D_var3,D_mult(L_var1,L_var2))`.
`Word64 D_msu(Word64 D_var3, long L_var1,long L_var2)`	Double precision multiply subtract. Multiplies `L_var1` by `L_var2` and shifts the result left by 1. Subtracts the 64-bit result from `D_var3` with saturation, and returns a 64-bit result. For example: `D_msu(D_var3,L_var1,L_var2) = D_sub(D_var3,D_mult(L_var1,L_var2))`.
`Word64 D_mult(long L_var1,long L_var2)`	Double precision multiply. The 64-bit result of the multiplication of `L_var1` by `L_var2` with one shift left, for example: `D_mult(L_var1,L_var2) = D_shl((L_var1*L_var2),1)`.
`long D_round(Word64 D_var1)`	Double precision round. Rounds the lower 32 bits of the 64-bit `D_var1` into the MS 32 bits with saturation. Shifts the resulting bits right by 32 and returns the 32-bit value.
`Word64 D_sat(Word64 D_var1)`	Double precision saturation. Saturates a 64-bit value.

Table 7-35. Built-in Intrinsic Functions (Continued)

Function	Purpose
`Word64 D_set(long L_var1, unsigned long L_var2)`	Double precision concatenation. Concatenates two 32-bit values, `L_var1` and unsigned `L_var2`, into one 64-bit value.
`Word64 D_sub(Word64 D_var1, Word64 D_var2)`	Double precision subtract. 64-bit subtraction of the two 64-bit variables `(D_var1-D_var2)` with overflow control and saturation.
`void debug()`	Generates assembly instruction to enter `Debug` mode.
`void debugev()`	Generates assembly instruction to issue `Debug` event.
`void di()`	Generates assembly instruction to disable interrupts.
`short div_s(short var1,short var2)`	Short divide. Produces a result which is the fractional integer division of `var1` by `var2`; `var1` and `var2` must be positive, and `var2` must be greater or equal to `var1`. The result is positive (leading bit equal to 0) and truncated to 16 bits. If `var1 = var2` then `div(var1,var2) = 32767`.
`void ei()`	Generates assembly instruction to enable interrupts.
`EndBitReverse`	Frees bit reverse iterator.
`short extract_h(long L_var1)`	Long extract high. Returns the 16 MSB of `L_var1`.
`short extract_l(long L_var1)`	Long extract low. Returns the 16 LSB of `L_var1`.
`void illegal()`	Generates assembly instruction to execute `illegal` exception.
`InitBitReverse`	Allocates a bit reverse iterator.
`long L_abs(long L_var1)`	Long absolute value of `L_var1`. Saturates in cases where the value is `-214783648`.
`long L_add(long L_var1,long L_var2)`	Long add. 32-bit addition of the two 32-bit variables `(L_var1+L_var2)` with overflow control and saturation. The result is set at `+2147483647` when overflow occurs, or at `-2147483648` when underflow occurs.
`long L_deposit_h(short var1)`	Deposit short in MSB. Deposits the 16-bit `var1` into the 16 MS bits of the 32-bit output. The 16 LS bits of the output are zeroed.
`long L_deposit_l(short var1)`	Deposit short in LSB. Deposits the 16-bit `var1` into the 16 LS bits of the 32-bit output. The 16 MS bits of the output are sign extended.
`long L_mac(long L_var3,short var1, short var2)`	Multiply accumulate. Multiplies `var1` by `var2` and shifts the result left by 1. Adds the 32-bit result to `L_var3` with saturation, and returns a 32-bit result. For example: `L_mac(L_var3,var1,var2) = L_add(L_var3,L_mult(var1,var2))`.
`long L_max(long L_var1,long L_var2)`	Compares the values of two 32-bit variables and returns the higher value of the two.
`long L_min(long L_var1,long L_var2)`	Compares the values of two 32-bit variables and returns the lower value of the two.

Table 7-35. Built-in Intrinsic Functions (Continued)

Function	Purpose
`long L_msu(long L_var3,short var1, short var2)`	Multiply subtract. Multiplies `var1` by `var2` and shifts the result left by 1. Subtracts the 32-bit result from `L_var3` with saturation, and returns a 32-bit result. For example: `L_msu(L_var3,var1,var2) = L_sub(L_var3,L_mult(var1,var2))`.
`long L_mult(short var1,short var2)`	Long multiply. The 32-bit result of the multiplication of `var1` by `var2` with one shift left, for example: `L_mult(var1,var2)= L_shl((var1*var2),1)` and `L_mult(-32768,-32768) = 2147483647`.
`long L_negate(long L_var1)`	Long negate. Negates the 32-bit variable `L_var1` with saturation. Saturates in cases where the value is `-2147483648(0x8000 0000)`.
`long L_rol(long L_var1)`	Long rotate left. Rotates the 32-bit variable `L_var1` left into a 40-bit value, and returns a 32-bit result..
`long L_ror(long L_var1)`	Long rotate right. Rotates the 32-bit variable `L_var1` right into a 40-bit value, and returns a 32-bit result.
`long L_sat(long L_var1)`	Saturates a 32-bit value.
`long L_shl(long L_var1,short var2)`	Long shift left. Arithmetically shifts the 32-bit `L_var1` left `var2` positions. Zero fills the `var2` LSB of the result. If `var2` is negative, arithmetically shifts `L_var1` right by `var2` with sign extension. Saturates the result in cases where underflow or overflow occurs.
`long L_shr(long L_var1,short var2)`	Long shift right. Arithmetically shifts the 32-bit `L_var1` right `var2` positions with sign extension. If `var2` is negative, arithmetically shifts `L_var1` left by `var2` and zero fills the `var2` LSB of the result. Saturates the result in cases where underflow or overflow occurs.
`long L_shr_r(long L_var1,short var2)`	Long shift right and round. Same as `L_shr(L_var1,var2)` but with rounding. Saturates the result in cases where underflow or overflow occurs.
`long L_sub(long L_var1,long L_var2)`	Long subtract. 32-bit subtraction of the two 32-bit variables `(L_var1-L_var2)` with overflow control and saturation. The result is set at `+2147483647` when overflow occurs or at `-2147483648` when underflow occurs.
`short mac_r(long L_var3,short var1, short var2)`	Multiply accumulate and round. Multiplies `var1` by `var2` and shifts the result left by 1. Adds the 32-bit result to `L_var3` with saturation. Rounds the LS 16 bits of the result into the MS 16 bits with saturation and shifts the result right by 16. Returns a 16-bit result.
`void mark()`	Generates assembly instruction to write program counter to trace buffer, if trace buffer enabled.
`short max(short var1, short var2)`	Compares the values of two 16-bit variables and returns the higher value of the two.
`short min(short var1, short var2)`	Compares the values of two 16-bit variables and returns the lower value of the two.
`long mpyuu(long L_var1,long L_var2)`	Multiplies the 16 LSB of two 32-bit variables, treating both variables as unsigned values, and returns a 32-bit result.

Table 7-35. Built-in Intrinsic Functions (Continued)

Function	Purpose
`long mpyus(long L_var1,long L_var2)`	Multiplies the 16 LSB of the 32-bit variable `L_var1` , treated as an unsigned value, by the 16 MSB of the 32-bit variable `L_var2`, treated as a signed value. Returns a 32-bit result.
`long mpysu(long L_var1,long L_var2)`	Multiplies the 16 MSB of the 32-bit variable `L_var1` , treated as a signed value, by the 16 LSB of the 32-bit variable `L_var2`, treated as an unsigned value. Returns a 32-bit result.
`short msu_r(long L_var3,short var1, short var2)`	Multiply subtract and round. Multiplies `var1` by `var2` and shifts the result left by 1. Subtracts the 32-bit result from `L_var3` with saturation. Rounds the LS 16 bits of the result into the MS 16 bits with saturation and shifts the result right by 16. Returns a 16-bit result.
`short mult(short var1,short var2)`	Short multiply. Performs the multiplication of `var1` by `var2` and gives a 16-bit result which is scaled, for example: `mult(var1,var2) = extract_l(L_shr((var1 * var2),15))` and `mult(-32768,-32768) = 32767.`
`short mult_r(short var1,short var2)`	Multiply and round. Same as `mult` with rounding, for example: `mult_r(var1,var2) = extract_l (L_shr(((var1*var2)+16384),15))` and `mult_r(-32768,-32768) = 32767`.
`short negate(short var1)`	Short negate. Negates `var1` with saturation. Saturates in cases where the value is `-32768`, for example: `negate(var1) = sub(0,var1).`
`short norm_l(long L_var1)`	Normalizes any long fractional value. Produces the number of left shifts needed to normalize the 32-bit variable `L_var1` for positive values on the interval with minimum of `1073741824` and maximum of `2147483647`, and for negative values on the interval with minimum of `-2147483648` and maximum of `-1073741824`. In order to normalize the result, the following operation must be executed: `norm_L_var1 = L_shl(L_var1,norm_l(L_var1)).`
`short norm_s(short var1)`	Normalizes any fractional value. Produces the number of left shifts needed to normalize the 16-bit variable `var1` for positive values on the interval with minimum of `16384` and maximum of `32767`, and for negative values on the interval with minimum of `-32768` and maximum of `-16384`. In order to normalize the result, the following operation must be executed: `norm_var1 = shl(var1,norm_s(var1)).`
`short round(long var1)`	Round. Rounds the lower 16 bits of the 32-bit number into the MS 16 bits with saturation. Shifts the resulting bits right by 16 and returns the 16-bit number, for example: `round(L_var1) = extract_h(L_add(L_var1,32768)).`
`short saturate(short var1)`	Saturates a 16-bit value.
`setcnvrm()`	Sets rounding mode to convergent rounding mode.

Table 7-35. Built-in Intrinsic Functions (Continued)

Function	Purpose
`set2crm()`	Sets rounding mode to two's-complement rounding mode.
`void setnosat()`	Clears saturation mode bit in status register.
`void setsat32()`	Sets saturation mode bit in status register.
`short shl(short var1,short var2)`	Short shift left. Arithmetically shifts the 16-bit `var1` left `var2` positions. Zero fills the `var2` LSB of the result. If `var2` is negative, arithmetically shifts `var1` right by `var2` with sign extension. Saturates the result in cases where underflow or overflow occurs.
`short shr(short var1,short var2)`	Short shift right. Arithmetically shifts the 16-bit `var1` right `var2` positions with sign extension. If `var2` is negative, arithmetically shifts `var1` left by `var2` with sign extension. Saturates the result in cases where underflow or overflow occurs.
`short shr_r(short var1,short var2)`	Short shift right and round. Same as `shr(var1,var2)` but with rounding. Saturates the result in cases where underflow or overflow occurs.
`void stop()`	Generates assembly instruction to enter `Stop` low power mode.
`short sub(short var1,short var2)`	Performs the subtraction with overflow control and saturation.The 16-bit result is set at `+32767` when overflow occurs or at `-32768` when underflow occurs.
`void trap()`	Generates assembly instruction to execute `Trap` exception.
`void wait()`	Generates assembly instruction to enter `Wait` low power mode.
`Word40 X_abs(Word40 X_var1)`	40-bit absolute value of `X_var1`.
`Word40 X_add(Word40 X_var1, Word40 X_var2)`	Extended precision add. Performs the addition `X_var1+X_var2` without saturation. .
`short X_cmpeq(Word40 X_var1, Word40 X_var2)`	Extended precision compare equal. Compares two 40-bit values and returns a 16-bit result containing the value ??? if the values are equal, or ??? if they are not.
`short X_cmpgt(Word40 X_var1, Word40 X_var2)`	Extended precision compare greater than. Compares two 40-bit values and returns a 16-bit result containing ???.
`Word40 X_extend(long L_var1)`	Sign extend 32-bit value to 40-bit value.
`short X_extract_h(Word40 X_var1)`	Extended precision extract high. Returns the 16 MSB of the 40-bit value `X_var1`.
`short X_extract_l(Word40 X_var1)`	Extended precision extract low. Returns the 16 LSB of the 40-bit value `X_var1`.
`Word40 X_mac(Word40 X_var3, short var1,short var2)`	Extended precision multiply accumulate. Multiplies `var1` by `var2` and shifts the result left by 1. Adds the 40-bit result to `X_var3` without saturation, and returns a 40-bit result. For example: `X_mac(X_var3,var1,var2) = X_add(X_var3,X_mult(var1,var2)).`

Table 7-35. Built-in Intrinsic Functions (Continued)

Function	Purpose
`Word40 X_msu(Word40 X_var3, short var1,short var2)`	Extended precision multiply subtract. Multiplies `var1` by `var2` and shifts the result left by 1. Subtracts the 40-bit result from `var3` without saturation, and returns a 40-bit result. For example: `X_msu(X_var3,var1,var2) = X_sub(X_var3,X_mult(var1,var2)).`
`Word40 X_mult(short var_1,short var_2)`	Extended precision multiply. The 40-bit result of the multiplication of `var1` by `var2` with one shift left, for example: `X_mult(var1,var2) = X_shl((var1*var2),1).`
`short X_norm(Word40 X_var1)`	Normalizes a 40-bit fractional value.
`Word40 X_or(Word40 X_var1, Word40 X_var2)`	Performs logical OR on two 40-bit values.
`Word40 X_rol(Word40)`	Rotates left a 40-bit value.
`Word40 X_ror(Word40)`	Rotates right a 40-bit value.
`short X_round(Word40 X_var1)`	Extended precision round. Rounds the lower 16 bits of the 40-bit number into the MS 16 bits without saturation. Shifts the resulting bits right by 16 and returns the 16-bit number.
`long X_sat(Word40 X_var1)`	Extended precision saturation. Saturates a 40-bit value.
`Word40 X_set(char var1, unsigned long L_var2)`	Extended precision concatenation. Concatenates an 8-bit character value and an unsigned 32-bit value into one 40-bit value.
`Word40 X_shl(Word40 X_var1,short var2)`	Extended shift left. Arithmetically shifts the 40-bit `X_var1` left `var2` positions. Zero fills the `var2` LSB of the result. If `var2` is negative, arithmetically shifts `X_var1` right by `var2` with sign extension.
`Word40 X_shr(Word40 X_var1,short var2)`	Extended shift right. Arithmetically shifts the 40-bit `X_var1` right `var2` positions with sign extension. If `var2` is negative, arithmetically shifts `X_var1` left by `var2` and zero fills the `var2` LSB of the result.
`Word40 X_sub(Word40 X_var1, Word40 X_var1)`	Extended precision subtract. 40-bit subtraction of the two 40-bit variables `(X_var1-X_var2)` without saturation.
`long X_trunc(Word40 X_var1)`	Truncates 40-bit value into 32-bit value.

Appendix A
Migrating from Other Environments

The SC100 C Compiler provides header files that make it easy to migrate C code developed for certain other compilers. The compilation and its results may be affected in various ways by the differences between specific compiler environments and the compiler. The effects may include, for example, assembler errors for inlined code that is not supported, or loss of efficiency for functions that are supported, but implemented in a different way.

This Appendix contains the following sections:

- Section A.1, “Code Migration Overview,” provides general guidelines for migrating code from another environment to the compiler.
- Section A.2, “Migrating Code Developed for DSP56600,” describes the issues to be considered when migrating code developed for the DSP56600 compiler family.
- Section A.3, “Migrating Code Developed for TI6xx,” describes the differences to take into account when migrating code developed for the TI6xx family of compilers.

A.1 Code Migration Overview

In most circumstances, the compiler can successfully compile standard ANSI code that:

- Does not use compiler-specific extensions
- Does not rely implicitly on the sizes of data types
- Does not rely on system-specific features, such as memory maps or peripherals
- Does not rely on undefined compiler behavior

The compiler runtime libraries include a header file for each environment for which code is accepted, as follows:

- DSP56600 compilers: `port566toSC1.h` header file
- TI6xx compilers: `portc6xtoSC1.h` header file

The features used in the specified environment are defined in the relevant header file with correct values, to ensure that the code is not affected and compiles successfully.

To use these definitions, just include the appropriate header file to your source code. For example, when migrating code from the DSP56600 compiler environment, include the `port566toSC1.h` header file, as shown in Example A-1.

Example A-1. Migrating code from other environments

```
#include <port566toSC1.h>
void main()
{
}
```

A.2 Migrating Code Developed for DSP56600

When using the SC100 C Compiler with code developed for the DSP56600 family of compilers, the following differences should be taken into account:

- **Integer data types:** The DSP56600 and SC100 compilers map certain integer data types to different sizes. Table A-1 lists the data type size discrepancies that relate to integers:

Table A-1. DSP56600 Integer Data Type Differences

Data Type	DSP56600 Compiler	SC100 C Compiler
char unsigned char	Saved in memory as 16 bits. Some operations are performed with 16 bits, others with 8.	8 bits
packed char	8 bits	Not supported
int unsigned int	16 bits	32 bits
enum	16 bits	32 bits

- **Fractional data types:** DSP56600 compilers use built-in data types for declaring fractional variables. The SC100 C Compiler uses standard integer types for both fractional and integer values. Table A-2 lists the fractional data type differences:

Table A-2. DSP56600 Fractional Data Type Differences

Data Type	DSP56600 Compiler	SC100 C Compiler
16-bit fraction	_fract	Word16
32-bit fraction	long_fract	Word32
40-bit accumulator	long_fract	Word40
64-bit fraction	Not supported	Word64
Complex fractions	_complex	Not supported directly

- **Floating point data types:** DSP56600 compilers represent floating point data types according to a 32-bit proprietary format. The SC100 C Compiler maps fractional data types to a single-precision IEEE-754 type, using 32 bits. As a result, there may be differences in the numerical accuracy of floating point calculations.

- **Pointers:** The difference in pointer size between the two compilers is shown in Table A-3:

Table A-3. DSP56600 Pointer Size Differences

Data Type	DSP56600 Compiler	SC100 C Compiler
pointer to `char`	16 bits	32 bits
pointer to `short`	16 bits	32 bits, even addresses only
pointer to `long`	16 bits	32 bits, quad addresses only

In most circumstances, the difference in pointer size is unlikely to have any impact, since the relevant addresses are usually mapped to different numerical values on different processors.

- **Fractional arithmetic:** DSP56600 compilers support fractional arithmetic using integer-like operators, such as + and *. The SC100 C Compiler implements fractional operations through the use of intrinsic functions. Table A-4 lists the DSP56600 fractional operations and shows the equivalent SC100 C Compiler intrinsic functions:

Table A-4. DSP56600 Fractional Arithmetic Differences

Fractional Operation	DSP56600 Compiler	SC100C Compiler
Addition	+	Word16 add Word32 L_add
Subtraction	-	Word16 sub Word32 L_sub
Absolute value	_fabs _lfabs	Word16 abs_s Word32 L_abs
Multiplication	*	Word16 mult Word32 L_mult Word16 mult_r
Shift right	>>	Word16 shr Word32 L_shr
Shift left	<<	Word16 shl Word32 L_shl
Negate	-	Word16 negate Word32 L_negate
Round	_fract_round	Word16 round
Divide	_pdiv	Word16 div_s
Normalize	Can be implemented using `_asm`	Word16 norm_s Word16 norm_l
Saturation control	Can be implemented using `_asm`	void setnosat void setsat32

The SC100 C Compiler supports many more fractional operations, including 40-bit and 64-bit fractional functions, which are not supported in the DSP56600 environment.

- **Inlined assembly and C code:** DSP56600 compilers use `_inline and _asm` to designate a C routine for inlining, and to define the instructions, operands and modifiers for inlined assembly statements. The SC100 C Compiler uses the pragma `#pragma inline` to specify an inlined function. See Chapter 4, “Interfacing C and Assembly Code,” for more information.

- **Intrinsic functions:** The SC100 C Compiler library routines support a number of DSP56600 intrinsic functions, as shown in Table A-5:

Table A-5. DSP56600 Intrinsic Function Differences

Description	DSP56600 Compiler	SC100 C Compiler
Bit field operations	_bfchg() _bfclr() _bfset() _bftsth() _bftstl()	Can be implemented by library routines
Cache control	_cache_get_start() _cache_get_end() _pflush() _pflushun() _pfree() _plock() _punlock()	Not available
Fraction to integer coercion	_fract2int() _lfract2long	Not needed (both represented by integers)
Integer to fraction coercion	_intt2fract() _long2lfract()	Not needed (both represented by integers)
Extend byte in accumulator	_ext()	Not applicable
Fractional square root	_fsqrt()	Can be implemented by a library routine
String copy (inlined)	_strcmp()	Supported as a library routine (`strcmp`)
Absolute of long integer	_labs	labs()
Insert NOP instruction	_nop()	_asm(“nop”)
STOP instruction	_stop()	stop()
Software interrupt	_swi()	trap()
WAIT instruction	_wait()	wait()
Viterbi operation	_vsl	Can be implemented by a library routine

- **Pragmas:** The functions of the DSP56600 inlined assembly pragmas `asm`, `asm_noflush` and `endasm` are supported by the SC100 C Compiler using a function qualifier. The SC100 C Compiler loop optimization pragma #pragma `loop_count` is the equivalent of the DSP56600 pragmas `iterate_at_least_once` and `no_iterate_at_least_once`.

  The following DSP56600 pragmas have no equivalent in the SC100 C Compiler environment:

  - `cache_align_now`
  - `cache_sector_size`
  - `cache_region_start`
  - `cache_region_endpack_strings`
  - `nopack_strings`
  - `source`
  - `nosource`
  - `jumptable_memory`

- **Interrupt handlers:** The SC100 C Compiler pragma `interrupt` performs the function of both `_fast_interrupt` and `_long_interrupt` in the DSP56600 environment.
- **Storage specifiers:** The DSP56600 compilers support a number of storage specifiers, which are either not used in the SC100 environment, or are specified at link time, as shown in Table A-6:

Table A-6. DSP56600 Storage Specifiers

Storage	DSP56600 Compiler	SC100 C Compiler
X memory	_X	Not applicable
Y_memory	_Y	Not applicable
Program memory	_P	Not applicable
L memory	_L	Not applicable
Lowest 64 words in data memory	_near	Not applicable
Internal memory	_internal	Specified at link time
External memory	_external	Specified at link time
Absolute address for global variable	_at	Specified at link time in the application configuration file

- **Miscellaneous:** Table A-7 outlines some further differences between the two compilers:

Table A-7. DSP56600 Miscellaneous Differences

Description	DSP56600 Compiler	SC100 C Compiler
Wrap-around semantics for fractional data	_nosat	Not applicable
Force DSP56300 GNU calling convention	_compatible	Not applicable
Circular buffer support	_circ	Addressing calculations using the C modulo (%) operator

A.3 Migrating Code Developed for TI6xx

The following differences should be considered when using the compiler with code developed for the TI6xx family of compilers:

- **Data Types:** TI6xx compilers map the integer type `long` to 40 bits. The compiler defines the integer type `long` as 32 bits. C code that relies on the fact that type `long` is 40 bits wide must be modified before it can be migrated.
- **Keywords:** The TI6xx keywords `cregister`, `near` and `far` are not supported by the compiler. When including the migration header file, these keywords are accepted but have no effect on the compilation results.

  The TI6xx keywords `interrupt` and `inline` are supported, but are implemented differently, using `#pragma inline` and `#pragma interrupt`. As a result, no automatic translation is provided. The code must be modified to use the pragmas supported by the compiler. For further information, see Section 3.4.5, “Pragmas,” on page 3-52.
- **Pragmas:** TI6xx pragmas are ignored. Warnings are issued, but the correctness of the compilation is not affected.
- **Inlined assembly code:** By definition, inlined assembly code is not portable from one environment to another. The SC100 Assembler is unable to recognize inlined TI6xx assembly code, and issues errors.
- **Intrinsic functions:** The TI6xx intrinsic functions listed in the `portc6xtoSC1.h` header file are supported. These are functionally equivalent to their corresponding TI6xx intrinsic functions, but their performance may be significantly affected.

Index

A

abort environment function 7-17
abs integer arithmetic function 7-15
abs_s intrinsic function 3-47, 7-21
acos trigonometric function 7-9
add intrinsic function 3-47, 7-21
align #pragma 3-53
Alignment
    bit-fields 3-40
    variables 3-59
-ansi shell option 3-12, 3-20
Application configuration file 6-13
    binding section 6-16
    overlay section 6-17
    schedule section 6-14
Application entry point 6-3
-arch shell option 3-13
Arithmetic
    fixed point 3-40
    floating point 3-40
    fractional 3-42
    integer 3-42
asctime time function 7-20
asin trigonometric function 7-9
asm statement 4-2
Assembly functions 4-7
Assembly instruction inlining
    asm statement 4-2
Assembly instructions
    inlining sequence 4-2
    inlining single instruction 4-1
atan trigonometric function 7-9
atan2 trigonometric function 7-9
atexit environment function 7-17
atof string conversion function 7-16
atoi string conversion function 7-16
atol string conversion function 7-16

B

Bare board startup 6-2
Bare board startup code 6-1
Basic block 5-2, 5-26
-be shell option 3-13, 3-25, 3-38, 3-40, 3-41
Big-endian mode 3-25
Binding section
    application configuration file 6-16
Bit-fields 3-40
BitReverseUpdate intrinsic function 3-50, 7-21
Built-in intrinsic functions 7-21

C

C environment startup 6-3
C environment startup code 6-1
C language
    dialects 3-26
    extensions 3-26
    K&R 3-31
    PCC 3-31
C language options 3-20
-C shell option 3-11, 3-16
-c shell option 3-11, 3-14
Call tree 6-14
Calling convention
    stack-based 6-19
    stack-less 6-21
calloc memory allocation function 7-15
ceil function 7-10
-cfe shell option 3-11, 3-14
Character typing 7-2
clearerr stream function 7-13
clock time function 7-20
clock_t time function 7-20
Code
    linear 5-3
    migrating from other environments A-1
    parallelized 5-3
Command file 3-10, 3-15
Command line 3-9
    syntax 3-9
Common subexpression elimination 5-17
Comparison functions 7-18
Compatibility clause 6-17
Compilation process 1-4, 3-2
Composed variable loop 5-13
Concatenation functions 7-18
Conditional execution 5-26
Configuration
    memory map 6-10
    startup code 6-4
Constant folding 5-18
Control options 3-10
Conversion functions 7-3
Copying functions 7-18
cos trigonometric function 7-9
cosh hyperbolic function 7-9

Cross-file optimization 5-4, 5-7, 5-35
-crt shell option 3-13, 3-25
ctime time function 7-20
ctype.h library 7-1, 7-2

D

-D shell option 3-11, 3-17
D_add intrinsic function 3-48, 7-21
D_cmpeq intrinsic function 3-48, 7-21
D_cmpgt intrinsic function 3-48, 7-21
D_extract_h intrinsic function 3-48, 7-21
D_extract_l intrinsic function 3-48, 7-21
D_mac intrinsic function 3-48, 7-21
D_msu intrinsic function 3-48, 7-21
D_mult intrinsic function 3-48, 7-21
D_round intrinsic function 3-48, 7-21
D_sat intrinsic function 3-48, 7-21
D_set intrinsic function 3-48, 7-22
D_sub intrinsic function 3-48, 7-22
Data allocation
    static 6-10
Data types 3-36
    bit-fields 3-40
    character 3-37
    double precision fractional 3-45
    extended precision fractional 3-45
    floating point 3-39
    fractional long 3-45
    fractional representation 3-40
    fractional short 3-45
    integer 3-38
    pointers 3-40
-dc shell option 3-13, 3-22
-de shell option 3-12, 3-22
Dead assignment
    elimination 5-19
Dead code
    elimination 5-19
Dead storage
    elimination 5-19
debug intrinsic function 3-50, 7-22
debugev intrinsic function 3-50, 7-22
Delay slots 5-22
Dependencies
    between instructions 5-3
Dependency 5-8
di intrinsic function 3-50, 7-22
Dialects
    C language 3-26
difftime time function 7-20
div integer arithmetic function 7-15
div_s intrinsic function 3-47, 7-22
-dL shell option 3-13, 3-22
-dL1 shell option 3-13, 3-22
-dL2 shell option 3-13, 3-22
-dL3 shell option 3-13, 3-22
-dm shell option 3-13, 3-22
-do shell option 3-13, 3-22
Double precision 3-46
DSP56600 compiler
    differences A-2
    header file A-1, A-2
    migrating code A-1, A-2
-dx shell option 3-13, 3-22
Dynamic loop 5-11
Dynamic memory allocation 6-9

E

-E shell option 3-11, 3-14
ei intrinsic function 3-50, 7-22
Elimination
    dead assignment 5-19
    dead code 5-19
    dead storage 5-19
    jump-to-jump 5-19
    subexpression 5-17
EndBitReverse intrinsic function 3-50, 7-22
Entry points 6-14
Environment functions 7-17
Environment variables 3-9
Execution sets 5-3
    parallelized 5-8
Execution units 5-3
exit environment function 7-17
exp function 7-10
Exponential functions 7-10
Extended 3-49
Extended precision 3-45
Extensions 3-7, 3-18
    C language 3-26
external #pragma 3-53
External function 3-55
extract_h intrinsic function 3-48, 7-22
extract_l intrinsic function 3-48, 7-22

F

-F shell option 3-11, 3-15
fabs function 7-10
fclose stream function 7-13
feof stream function 7-13
ferror stream function 7-13
fflush I/O function 7-14
fgetc input function 7-12
fgetpos stream function 7-13
File extensions 3-7, 3-18
File types 3-7
Finalization code 6-1, 6-3

float.h library 7-1, 7-4
Floating point arithmetic 3-40
Floating point characteristics 7-4
Floating point math 7-9
floor function 7-10
fmod function 7-10
fopen I/O function 7-14
fprintf output function 7-14
fputc output function 7-14
fputs output function 7-14
Fractional
  arithmetic 3-42
  constants 3-46
  representation 3-40
  values 3-46
fread input function 7-12
free memory allocation function 7-15
freopen stream function 7-13
frexp function 7-10
fscanf input function 7-12
fseek stream function 7-13
fsetpos stream function 7-13
ftell stream function 7-13
Function inlining 5-16
Functions
  built-in intrinsic 7-21
  comparison 7-18
  concatenation 7-18
  conversion 7-3
  copying 7-18
  environment 7-17
  exponential 7-10
  external 3-55
  hyperbolic 7-9
  I/O 7-14
  input 7-12
  integer arithmetic 7-15
  intrinsic 3-45, 3-47, 7-21
  logarithmic 7-10
  memory allocation 7-15
  multibyte character 7-17
  output 7-14
  power 7-10
  pseudo random number generation 7-16
  search 7-19
  searching 7-16
  sorting 7-16
  stream 7-13
  string 7-17
  string conversion 7-16
  testing 7-3
  time 7-20
  trigonometric 7-9
fwrite output function 7-14

G

-g shell option 3-12, 3-20
General utilities 7-15
getc input function 7-12
getchar input function 7-12
getenv environment function 7-17
gets output function 7-14
Global variables 6-17
gmtime time function 7-20
Guidelines
  optimizer 5-35

H

-h shell option 3-11, 3-15
Hardware loops 6-24
Hardware registers
  initialization 6-2
Header file
  TI6xx compiler A-1
Heap 6-9
Hyperbolic functions 7-9

I

-I shell option 3-11, 3-17
I/O functions 7-14
I/O services
  low level 6-3
  termination 6-3
illegal intrinsic function 3-50, 7-22
Include files 3-17
InitBitReverse intrinsic function 3-50, 7-22
Initialization
  M registers 6-2
  status registers 6-2
  variables 3-25, 6-3
Initialization code 6-1, 6-3
Initializing variables with fractional values 3-46
inline #pragma 3-53
Inlining 3-54
  sequence of assembly instructions 4-2
  single assembly instruction 4-1
Input file extension 3-18
Input functions 7-12
Instruction scheduling 5-22
Integer arithmetic 3-42
Integer arithmetic functions 7-15
Integer characteristics 7-8
interrupt #pragma 3-53
Interrupt entry 6-16, 6-23
Interrupt handler 3-56, 6-14, 6-23
Interrupt vector 6-2, 6-16, 6-23
Interrupts 6-3
Intrinsic functions 3-45

architecture primitives 3-50
assembly instruction architecture primitives 3-50
bit reverse addressing 3-50
double precision fractional arithmetic 3-48
fractional arithmetic 3-47
fractional arithmetic with guard bits 3-49
long fractional arithmetic 3-48
Invariant code loop 5-17
isalnum testing function 7-3
isalpha testing function 7-3
iscntrl testing function 7-3
isdigit testing function 7-3
isgraph testing function 7-3
islower testing function 7-3
ISO libraries 7-1
isprint testing function 7-3
ispunct testing function 7-3
isspace testing function 7-3
isupper testing function 7-3
isxdigit testing function 7-3

J

Jump-to-jump elimination 5-19

K

K&R mode 3-31
-kr shell option 3-12, 3-20

L

L_abs intrinsic function 3-48, 7-22
L_add intrinsic function 7-22
L_deposit_h intrinsic function 3-48, 7-22
L_deposit_l intrinsic function 3-48, 7-22
L_mac intrinsic function 3-47, 7-22
L_max intrinsic function 3-48, 7-22
L_min intrinsic function 3-48, 7-22
L_msu intrinsic function 3-47, 7-23
L_mult intrinsic function 7-23
L_negate intrinsic function 3-48, 7-23
L_rol intrinsic function 3-50, 7-23
L_ror intrinsic function 3-50, 7-23
L_sat intrinsic function 3-48, 7-23
L_shl intrinsic function 3-48, 7-23
L_shr intrinsic function 3-48, 7-23
L_shr_r intrinsic function 3-48, 7-23
L_sub intrinsic function 3-48, 7-23
labs integer arithmetic function 7-15
L-add intrinsic function 3-48
ldexp function 7-10
ldiv integer arithmetic function 7-15
Libraries
ISO 7-1
non-ISO 7-1
limits.h library 7-1, 7-8
Linear code 5-3
Linker command file 6-3, 6-6
Listing files 3-22
Little-endian 3-25
Little-endian mode 3-41
Little-endian representation 3-38, 3-40
L-mult intrinsic function 3-48
locale.h library 7-1, 7-8
localeconv locales function 7-8
Locales functions 7-8
localtime time function 7-20
log function 7-10
log10 function 7-10
Logarithmic functions 7-10
Logical memory 6-11
Loop
composed variable 5-13
dynamic 5-11
multi-step 5-12
simple 5-10
square 5-13
transformations 5-10
Loop count 3-57
loop_count #pragma 3-53
Loops
hardware 6-24
Low level transformations (LLT) 5-20

M

M registers
initialization 6-2
value 6-24
-M shell option 3-11, 3-16
-ma shell option 3-13, 3-24
mac_r intrinsic function 3-47, 7-23
Machine configuration file 6-11
Macros 3-17
fractional values 3-46
predefined 3-61
preprocessor 3-17
Main entry point 6-14
malloc memory allocation function 7-15
mark intrinsic function 3-50, 7-23
math.h library 7-1, 7-9
max intrinsic function 3-47, 7-23
-mb shell option 3-13, 3-25
-mc shell option 3-13, 3-24
-mem shell option 3-13, 3-25
memchr search function 7-19
memcmp comparison function 7-18
memcpy copying function 7-18
memmove copying function 7-18
Memory

logical 6-11
mode 3-25, 6-6
physical 6-11
Memory allocation
dynamic 6-9
functions 7-15
Memory layout
default 6-7
Memory map
configuration 6-10
default values 6-8
initialization 6-3
Memory model
big 6-5
small 6-6
Memory space 6-11
memset function 7-19
Messages 3-22
-MH shell option 3-11, 3-16
Migrating code A-1
min intrinsic function 3-47, 7-23
mktime time function 7-20
Mode
K&R/PCC 3-31
modf function 7-10
mpysu intrinsic function 3-50, 7-24
mpyus intrinsic function 3-50, 7-24
mpyuu intrinsic function 3-50, 7-23
-mrom shell option 3-13
msu_r intrinsic function 3-47, 7-24
mult intrinsic function 3-47, 7-24
mult_r intrinsic function 3-47, 7-24
Multibyte character functions 7-17
Multiple execution units 5-3
Multi-step loop 5-12

N

-n shell option 3-13, 3-23
negate intrinsic function 3-47, 7-24
noinline #pragma 3-53
Non-cross file optimization 3-4
Non-ISO libraries 7-1
Nonlocal jumps 7-11
norm_l intrinsic function 3-48, 7-24
norm_s intrinsic function 3-47, 7-24

O

-o shell option 3-11, 3-19
-O0 shell option 3-12, 5-5
-O1 shell option 3-12, 5-5, 5-9
-O2 shell option 3-12, 5-5, 5-20
-Og shell option 3-12, 5-5, 5-35
Optimization
cross file 1-2, 3-4, 5-4, 5-7, 5-35
for size 5-7, 5-33
levels 5-4
non-cross file 3-4
options 5-4, 5-5
target independent 5-8, 5-9
target specific 5-8, 5-20
Optimizer
guidelines 5-35
invoking 5-6
Options
C language 3-20
control 3-10
extensions 3-18
messages 3-22
output files 3-19
shell 3-11
-Os shell option 3-12, 5-5
Output files 3-19
Output functions 7-14
Overlay section
application configuration file 6-17
Overlay specification 6-14

P

Parallelized code 5-3
Parallelized execution sets 5-8
Passing options 3-21
perror output function 7-14
Physical memory 6-11
Pipeline restrictions 5-23
Pointers 3-40
Post-increment detection 5-28
pow function 7-10
Power functions 7-10
Pragmas
#pragma align 3-53, 3-60
#pragma external 3-53, 3-55
#pragma inline 3-53
#pragma interrupt 3-56
#pragma loop count 3-53, 3-58
#pragma noinline 3-53, 3-54
#pragma profile 3-53, 3-56, 3-57
#pragma save 3-53
#pragma save_ctxt 3-54
placement 3-52
syntax 3-52
Predefined macros 3-61
Prefix grouping 5-33
Preprocessing options 3-16
Preprocessor macros 3-17
printf output function 7-14
Process time 7-20
profile #pragma 3-53

Profile value 3-56
prototype.h 7-1
prototype.h library 7-1, 7-21
Pseudo random number generation functions 7-16
putc output function 7-14
putchar output function 7-14
puts output function 7-14

Q

-q shell option 3-13, 3-23

R

-r shell option 3-11, 3-19
rand pseudo random number generation function 7-16
realloc memory allocation function 7-15
remove stream function 7-13
rename stream function 7-13
Reporting 3-23
Reset interrupt vector 6-2
rewind stream function 7-13
round intrinsic function 3-47, 7-24
Runtime
    environment 6-1
    startup code 6-1

S

-S shell option 3-11, 3-14
saturate intrinsic function 3-47, 7-24
save_ctxt #pragma 3-53
-sc shell option 3-12, 3-20
scanf input function 7-12
Schedule section
    application configuration file 6-14
Search functions 7-19
Searching functions 7-16
set2cnvrm intrinsic function 3-50, 7-24
set2crm intrinsic function 3-50, 7-25
setbuf stream function 7-13
setjmp.h library 7-1, 7-11
setlocale locales function 7-8
setnosat intrinsic function 3-50, 7-25
setsat32 intrinsic function 3-50, 7-25
setvbuf stream function 7-13
Shell 1-3, 3-1
Shell command file 3-15
Shell command line 3-9
Shell options
    behavior control 3-11
    C language 3-12
    file extension override 3-11
    hardware model and configuration 3-13
    optimization pragma and code 3-12
    output filename and location 3-11
    output of listing files and messages control 3-12
    pass-through 3-12
    preprocessing 3-11
    stop processing 3-14
    summary 3-11
shl intrinsic function 3-47, 7-25
shr intrinsic function 3-47, 7-25
shr_r intrinsic function 3-47, 7-25
Signal handling 7-11
signal.h library 6-23, 7-1, 7-11
Simple loop 5-10
sin trigonometric function 7-9
sinh hyperbolic function 7-9
Software pipelining 5-23
Sorting functions 7-16
Space optimization 5-7, 5-33
Speculative execution 5-27
sprintf output function 7-14
sqrt function 7-10
Square loop 5-13
srand pseudo random number generation function 7-16
sscanf input function 7-12
Stack
    frame 6-22
    memory allocation 6-9
    pointer 6-9, 6-19
    space 6-22
    start address 6-3, 6-9
Stack-based
    calling convention 6-19
Stack-less
    calling convention 6-21
Standard definitions 7-12
Startup code 6-1
    bare board 6-1
    C environment 6-1
    configuration 6-4
Static data allocation 6-10
Status registers
    default settings 6-2
    initialization 6-2
stdarg.h library 7-1, 7-11
stddef.h library 7-1, 7-12
stderr stream function 7-13
stdin stream function 7-13
stdio.h library 7-1, 7-12
stdlib.h library 7-1, 7-15
stdout stream function 7-13
stop intrinsic function 3-50, 7-25
strcat concatenation function 7-18
strchr search function 7-19
strcmp comparison function 7-18
strcoll comparison function 7-18
strcpy copying function 7-18

strcspn search function 7-19
Stream functions 7-13
Strength reduction 5-10, 5-32
strerror function 7-19
strftime time function 7-20
String conversion functions 7-16
String functions 7-17
string.h library 7-1, 7-17
strlen function 7-19
strncat concatenation function 7-18
strncmp comparison function 7-18
strncpy copying function 7-18
strpbrk search function 7-19
strrchr search function 7-19
strspn search function 7-19
strstr search function 7-19
strtod string conversion function 7-16
strtok search function 7-19
strtol string conversion function 7-16
strtoul string conversion function 7-16
strxfrm comparison function 7-18
sub intrinsic function 3-47, 7-25
Subexpression elimination 5-17
Symbolic labels 4-10
System context 3-54

T

tan trigonometric function 7-9
tanh hyperbolic function 7-9
target architecture 3-24
Target-independent optimizations 5-9
Target-specific optimizations 5-20
Target-specificl optimizations 5-8
Target-specificl peephole 5-29
Task entry point 6-14
Termination
    I/O services 6-3
Testing functions 7-3
TI6xx compiler
    header file A-1
    migrating code A-1
Time constant 7-20
time function 7-20
Time functions 7-20
time.h library 7-1, 7-20
time_t time function 7-20
Timer 6-2
tolower conversion function 7-3
toupper conversion function 7-3
Transformations
    loop 5-10
trap intrinsic function 3-50, 7-25
Trigonometric functions 7-9

U

-U shell option 3-11, 3-17
ungetc I/O function 7-14
-usc shell option 3-20

V

-v shell option 3-13, 3-23
Variable arguments 7-11
Variables
    alignment 3-59
    initialization 6-3
vfprintf output function 7-14
vprintf output function 7-14
vsprintf output function 7-14

W

-w shell option 3-13, 3-23
wait intrinsic function 3-50, 7-25
-Wall shell option 3-13, 3-23
Warnings 3-23
-Wg shell option 3-13, 3-23
-Wj shell option 3-13, 3-23
WORD16 macro 3-46
WORD32 macro 3-46
Word40 extended precision fractional 3-45
Word64 double precision fractional 3-46

X

X_abs intrinsic function 3-49, 7-25
X_add intrinsic function 3-49, 7-25
X_cmpeq intrinsic function 3-49, 7-25
X_cmpgt intrinsic function 3-49, 7-25
X_extend intrinsic function 3-49, 7-25
X_extract_h intrinsic function 3-49, 7-25
X_extract_l intrinsic function 3-49, 7-25
X_mac intrinsic function 3-49, 7-25
X_msu intrinsic function 3-49, 7-26
X_mult intrinsic function 3-49, 7-26
X_norm intrinsic function 3-49, 7-26
X_or intrinsic function 3-49, 7-26
X_rol intrinsic function 3-49, 7-26
X_ror intrinsic function 3-49, 7-26
X_round intrinsic function 3-49, 7-26
X_sat intrinsic function 3-49, 7-26
X_set intrinsic function 3-49, 7-26
X_shl intrinsic function 3-49, 7-26
X_shr intrinsic function 3-49, 7-26
X_sub intrinsic function 3-49, 7-26
X_trunc intrinsic function 3-49, 7-26
-Xasm shell option 3-12, 3-21
-xasm shell option 3-11, 3-18
-xc shell option 3-11, 3-18
-Xlnk shell option 3-12, 3-21

-xobj shell option 3-11, 3-18

How to reach us:

USA/Europe/Locations Not Listed:

Motorola Literature Distribution
P.O. Box 5405
Denver, Colorado 80217
1 (800) 441-2447

Asia/Pacific
Motorola Semiconductors H.K. Ltd.,
Silicon Harbour Centre, 2 Dai King Street, Tai Po
Industrial Estate, Tai Po, N.T., Hong Kong
852-26668334

Japan
Motorola Japan, Ltd.,
SPS, Technical Information Center
3-20-1, Minami-Azabu. Minato-ku, Tokyo 106-8573
81-3-3440-3569

Motorola Fax Back System (Mfax™)
1 (800) 774-1848; RMFAX0@email.sps.mot.com

DSP Helpline
dsphelp@dsp.sps.mot.com

Technical Information Center
1 (800) 521-6274

Internet
http://www.motorola-dsp.com

Agere Systems Internet
http://www.agere.com

Email
docmaster@micro.lucent.com

N. America
Agere Systems, Inc.
1-800-372-2447, FAX 610-712-4106
In CANADA: 1-800-553-2448, FAX 610-712-4106

Asia/Pacific
Agere Systems Singapore Pte. Ltd., Singapore
Tel. (65) 778 8833, FAX (65) 777 7495

China
Agere Systems (Shanghai) Co., Ltd., Shanghai
Tel. (86) 21 5047 1212, Fax (86) 21 5047 2266

Japan
Agere Systems Japan, Ltd., Shinagawa-ku, Japan
Tel. (81) 3 5421 1600, FAX (81) 3 5421 1700

Europe Dataline
Tel. (44) 7000 582 368, FAX (44) 1189 328 148

MOTOROLA digital dna™

agere systems