C2000™ C28x Optimization Guide

v1.2

Online HTML version available here

CONTENTS

This guide describes a systematic approach to improving the performance of applications executing on the TMS320C28x CPU in C2000™ MCUs. The guide assumes the reader is familiar with application development on C2000 MCUs. For training material on developing applications and support information, refer to C2000 real-time control MCUs - Support & training. For an overview of the key software packages associated with C2000 software development, refer to the C2000 Software Guide.

CHAPTER

ONE

INTRODUCTION

Note: The online HTML version of this guide is available at https://software-dl.ti.com/C2000/docs/optimization_guide/index.html.

This chapter introduces a software development flow that can be used to improve the performance of C code executing on the TMS320C28x CPU in C2000™ MCUs.

1.1 Software development flow

Software development for the C28x CPU can be split into the following phases:

Phase 1

Write, compile and debug the application on a C2000 device. During this phase, compiler optimizations are disabled to provide the best debug experience. The focus of this phase is on functionality and correctness. However, there are some rules to keep in mind at this stage to generate efficient C2000 code and avoid later rework. Refer to *Initial Development* for details.

Phase 2

Profile the application to determine the regions of code where the application spends a majority of its run time. In some cases, it may be clear that the application spends most of its time in one or two ISRs. In this scenario, profiling can help determine which functions in the ISR account for a majority of the ISR's runtime.

Profiling is used to focus optimization efforts on the functions which account for a majority of the runtime. There are different approaches to profiling, refer to section *Profiling* for details.

Phase 3

Optimize the application to meet performance and code size constraints. Typical steps include:

- Placing the most commonly executed functions and associated data in RAM
- Enabling the appropriate compiler options:

  - Options to take advantage of optimization passes within the compiler - optimization levels, inlining etc.
  - Options to take advantage of hardware features (FPU, TMU, etc.)
- Where possible, use optimized libraries from TI (e.g. Digital Control Library)
- Provide more information to the compiler to help its optimizations (pragmas, restrict, etc.)
- Use the CLA

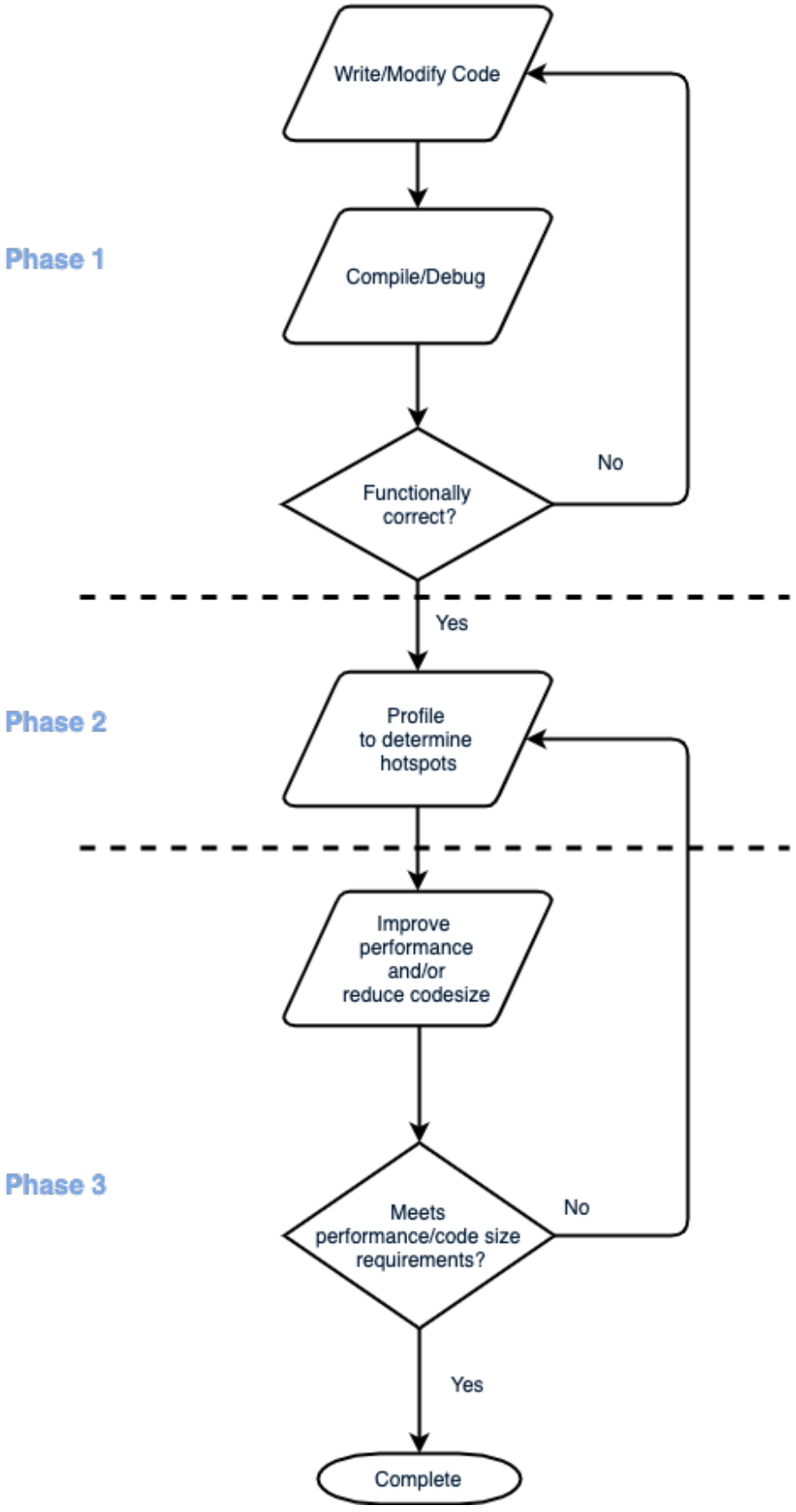

Fig. 1.1: Software Development - Profiling and Optimization

For details. refer to *Improving performance*.

Phases 2 and 3 are iterative. Try an optimization, measure performance/code-size and repeat. It is advisable to set up a self checking application so its correctness can be checked during optimizations.

1.2 Processing elements

C28x CPU The C28x CPU is a 32-bit fixed-point processor. It incorporates RISC features such as single-cycle instruction execution and register-to-register operations. The modified Harvard architecture of the CPU enables instruction and data fetches to be performed in parallel.

Floating-Point Unit

FPU The FPU extends the capabilities of the C28x fixed-point CPU by adding registers and instructions to support IEEE single-precision floating point operations.

FPU64 The FPU64 extends the capabilities of the C28x fixed-point CPU by adding registers and instructions to support both IEEE single-precision and double-precision floating point operations.

Trigonometric Math Unit

TMU The TMU extends the capabilities of a C28x+FPU by adding instructions and leveraging existing FPU instructions to speed up the execution of common trigonometric and arithmetic operations.

Viterbi, Complex Math and CRC Unit

VCU The VCU processor extends the capabilities of the C28x CPU by adding registers and instructions to support the following algorithm types: Viterbi decoding, cyclic redundancy check (CRC), complex math.

Further information about the C28x CPU, FPU, TMU and VCU can be found in the following document(s):

- TMS320C28x CPU and Instruction Set Reference Guide
- TMS320C28x Extended Instruction Sets Technical Reference Manual
- Accelerators: Enhancing the Capabilities of the C2000 MCU Family Technical Brief
- TMS320C28x FPU Primer

Control Law Accelerator (CLA) The Control Law Accelerator is a 32-bit floating point math accelerator that is common on most C2000 MCUs. It aids in the concurrent processing of fast control algorithms. For details on the CLA, refer to the CLA chapter in the device TRM.

CHAPTER

TWO

INITIAL DEVELOPMENT

This chapter outlines various factors to consider when starting development on C2000 MCUs.

2.1 Application Binary Interface (ABI)

Prior to release 18.12.0.LTS of TI's C28x Compiler Tools, the one and only ABI for C28x was the original COFF-based ABI.

C2000Ware Release 2.0 and 18.12.0.LTS of the TI Compiler Tools introduced a new ABI called the C28x EABI. It is based on the ELF object file format. It is derived from industry standard models, including the IA-64 C++ ABI and the System V ABI for ELF and Dynamic Linking. The processor-specific aspects of the ABI, such as data layout and calling conventions, are largely unchanged from the COFF ABI, although there are some differences. The COFF ABI and the EABI are incompatible - all of the code linked into an application binary must follow the same ABI.

TI's supports both the new EABI and the older COFF ABI on F2837x and F28004x. Migration to EABI for new software development on F2837x and F28004x is encouraged. Device support software and libraries available in C2000Ware for F2838x and F28002x are EABI only.

Key distinctions between EABI and COFFABI:

- The `double` type is 64 bits. The size of double changes from 32 bits to 64 bits when you migrate from COFF ABI to EABI.
- There is no leading underscore on symbols. COFF ABI adds a leading underscore to symbol names, but EABI does not. Assembly file references to symbols need special handling.

Additional Resources

- A migration guide from COFF to EABI is available here.
- For more details on C28x EABI, refer to the C28x Embedded Application Binary Interface application report.

2.2 Bitfield vs. driverlib

Bitfield and driverlib are two approaches to implementing a hardware abstraction layer for C2000 MCUs. Refer to the application report, Programming TMS320x28xx and TMS320x28xxx Peripherals in C/C++ for a comparison of the approaches.

Note: All of the code examples in this chapter were built with C28x compiler version 18.12.2 LTS using the EABI application binary interface.

2.3 Initial set of compiler options

This section documents some of the commonly used compiler options. For details on using the compiler, refer to TMS320C28x Optimizing C/C++ Compiler User's Guide, Chapter 2, Using the C/C++ Compiler.

2.3.1 ABI

The ABI is selected through the –abi option as follows:

- COFF ABI (`--abi=coffabi`). This is the default.
- EABI (`--abi=eabi`). Use this option to select the C28x Embedded Application Binary Interface (EABI).

Refer to section *Application Binary Interface (ABI)* for details.

2.3.2 Enable FPU

On supported C28x CPUs, use the `--float_support[=fpu32|fpu64]` to enable 32-bit or 64-bit hardware floating-point support. FPU64 is supported only when using EABI.

Note: It is beneficial to enable the FPU even if the application does not use floating-point. This enables the compiler to leverage instructions such as the `Repeat Block (RPTB)` to significantly reduce branching overhead in certain loops.

2.3.3 Enable TMU

On supported C28x CPUs, use the `----tmu_support[=tmu0|tmu1]` to enable support for the Trigonometric Math Unit (TMU). Using this option automatically enables *FPU* support (as with the `--float_support=fpu32` option). When TMU support is enabled, intrinsics are available to perform trigonometric instructions on the TMU. In relaxed floating point mode, RTS library calls are replaced with the corresponding TMU hardware instructions for the following floating point operations: floating point division, sqrt, sin, cos, atan, and atan2. Additionally, if the `--tmu_support=tmu1` option is used with `-- fp_mode=relaxed`, special versions of the following 32-bit RTS math functions are used: exp2f(), espf(), log2f(), logf(), and powf().

2.3.4 Enable IDIV support

On supported C28x CPUs, use the `--idiv_support=idiv0` to enable support for fast integer division using hardware extensions to provide a set of instructions to accelerate integer division. When this option is enabled, the built-in integer division and modulo operators ("/" and "%") use the appropriate faster instructions.

For details on enabling FPU, TMU and IDIV, refer to the TMS320C28x Optimizing C/C++ Compiler User's Guide Section 2.3.4 Run-Time Model Options.

2.3.5 Unified memory model

Unified memory model means that any memory block can be used as either program memory or data memory. All memory blocks including SARAM, flash, ROM, OTP and XINTF memory is unified on the C28x MCUs. Peripheral registers are typically only mapped to data space.

If the applications memory map is configured as a single unified space, specifying unified memory model in the build options (`--unified_memory`) is important because it allows the compiler to generate efficient instructions for memcpy calls and structure assignments.

Even under unified memory, memory for some peripherals and some RAM associated with those peripherals is allocated only in data memory. If `--unified_memory` is enabled, program memory address access to specific symbols such as peripheral registers can be prevented by declaring those symbols as volatile.

Refer to *Data allocation for instructions with two memory operands* for an example of using `--unified_memory` to enable the compiler to generate the MAC instruction.

2.3.6 Enable debugging

During initial development, it is recommended to use the following options for debuggability:

- -g (`--symdebug:dwarf`)
- `--opt_level=off`

Setting `--opt_level=off` improves the debug experience. In general, the higher the level of optimization that is applied, the harder it is to debug the program. This is because higher levels of optimization enable more transformations and apply to broader granularities or scopes of the program. Optimizations at levels 0 and 1 are applied to individual statements or blocks of code within functions, level 2 enables optimizations across blocks of code within a function, level 3 enables optimizations across functions within a file, and level 4 enables optimizations across files. Since transformations that occur at higher levels are usually more widespread, it is harder for the debugger to map the resulting code to the original source program.

For details, refer to the TI E2E blog on Debugging and Optimization

2.4 Code considerations

2.4.1 Struct size

When declaring structs, keep overall size to less than 64 words.

In direct addressing mode, the 6-bit offset value is concatenated with the 16-bit DP register. The offset value enables 0 to 63 words to be addressed relative to the current DP register value. Structs larger that 64 words will require DP to the updated before accessing fields that are 64-words apart, resulting in less efficient code due to the extra DP update instructions.

Table 2.1 compares 2 structs - `Test1` is larger than 64 words and `Test2` is smaller than 64 words. Table 2.2 compares the generated assembly for functions with accesses to each struct. The larger struct requires an extra `MOVW` instruction to set the DP before accessing field `b`.

Table 2.1: Structure size and efficiency of generated code

struct Test1 is larger than 64 words	struct Test2 is smaller than 64 words
`typedef struct` `{` `  int a;` `  int array[63];` `  int b;` `} Test1;` `Test1 t1;` `void test1()` `{` `  t1.a = t1.b;` `  t1.b = 42;` `}`	`typedef struct` `{` `  int a;` `  int array[32];` `  int b;` `} Test2;` `Test2 t2;` `void test2()` `{` `  t2.a = t2.b;` `  t2.b = 42;` `}`

Table 2.2: Structure size and efficiency of generated code

```
||test1||:
        MOVW      DP,#||t1||+64
        MOV       AL,@||t1||+64
        MOVW      DP,#||t1||
        MOV       @||t1||,AL
        MOVW      DP,#||t1||+64
        MOVB      @||t1||+64,#42,
↪UNC
        LRETR
```

```
||test2||:
        MOVW      DP,#||t2||+33
        MOV       AL,@||t2||+33
        MOV       @||t2||,AL
        MOVB      @||t2||+33,#42,
↪UNC
        LRETR
```

2.4.2 Grouping global variables

Group global variables into structures can potentially enable the compiler to generate efficient direct addressing using the DP and minimize the number of updates to the DP between variable accesses.

Table 2.3 illustrates grouping global variables into fields in a struct.

Table 2.3: Global variables - grouping and efficiency

Global variables	Global variables grouped into a struct
`int16_t global0;` `int16_t global1;` `int16_t global2;` `int16_t global3;` `int16_t global4;` `int16_t global5[32];` `int16_t foo()` `{` `  return global0 + global1 +` `  global2 +` `  global3 + global4 +` `  global5[0];` `}`	`typedef struct {` `  int16_t global0;` `  int16_t global1;` `  int16_t global2;` `  int16_t global3;` `  int16_t global4;` `  int16_t global5[32];` `} Globals;` `Globals g;` `int16_t bar()` `{` `  return g.global0 + g.global1` `  + g.global2 +` `  g.global3 + g.global4` `  + g.global5[0];` `}`

Table 2.4 illustrates the improvement in the generated assembly from grouping global variables - there are fewer updates to the DP between accesses - 3 vs. 1 for the 5 accesses.

Table 2.4: Efficiency improvements from grouping global variables into a struct

```
||foo||:
        MOVW    DP,#||global0||
        MOV     AL,@||global0||
        MOVW    DP,#||global5||
        ADD     AL,@||global5||
        MOVW    DP,#||global1||
        ADD     AL,@||global1||
        ADD     AL,@||global2||
        ADD     AL,@||global3||
        ADD     AL,@||global4||
        LRETR
```

```
||bar||:
        MOVW    DP,#||g||+1
        MOV     AL,@||g||+1
        ADD     AL,@||g||
        ADD     AL,@||g||+2
        ADD     AL,@||g||+3
        ADD     AL,@||g||+4
        ADD     AL,@||g||+5
        LRETR
```

Refer to TMS320C28x Optimizing C/C++ Compiler User's Guide, Section 3.11, Data Page (DP) Pointer Load Optimization for details.

2.4.3 Local variables

Local variables in a function are placed on the stack. The compiler uses the SP register to access these local variables. The stack frame for a function consists of local variables and other compiler generated data. For frames that exceed 63 words in size (the maximum reach of the SP offset addressing mode), the compiler uses XAR2 as a frame pointer (FP).

To take advantage of SP-relative addressing, keep the local frame less than 64 words. Table 2.5 illustrates the impact of stack frame size on the efficiency of generated code. The only difference between the two code snippets is that in the example on the left, the stack frame is larger than 64 words. In this example, volatile is used to ensure the compiler reads the local variables a and b from memory vs. allocating them to registers

Table 2.5: Stack frame size and efficiency of generated code

Stack frame >= 64 words	Stack frame < 64 words
`int16_t local_variables1()` `{` `  volatile int16_t a;` `  int16_t array[64];` `  volatile int16_t b;` `  a = 42;` `  b = 44;` `  update(array, 64);` `  return a + b;` `}`	`int16_t local_variables2()` `{` `  volatile int16_t a;` `  int16_t array[32];` `  volatile int16_t b;` `  a = 42;` `  b = 44;` `  update(array, 32);` `  return a + b;` `}`

Table 2.6 illustrated how the compiler is able to use the more efficient SP-relative addressing when the frame size is < 64 words.

Table 2.6: Structure size and efficiency of generated code

Frame size >= 64 requires use of FP (AR2)	Frame size < 64 uses SP relative addressing
`` ||local_variables1||: MOVL *SP++,XAR1 MOVL *SP++,XAR2 MOVZ AR2,SP SUBB FP,#6 ADDB SP,#66 MOVZ AR4,SP MOVB *+FP[7], ↪#42,UNC MOVB AL,#64 SUBB XAR4,#64 MOVB *+FP[6], ↪#44,UNC MOVZ AR4,AR4 LCR #||update|| MOV AL,*+FP[6] ADD AL,*+FP[7] SUBB SP,#66 MOVL XAR2,*--SP MOVL XAR1,*--SP LRETR ```	``` ||local_variables2||: ADDB SP,#34 MOVZ AR4,SP MOVB *-SP[33],#42, ↪UNC MOVB AL,#32 SUBB XAR4,#32 MOVB *-SP[34],#44, ↪UNC MOVZ AR4,AR4 LCR #||update|| MOV AL,*-SP[34] ADD AL,*-SP[33] SUBB SP,#34 LRETR ```

2.4.4 Saturation in C

To perform efficient saturation in C on the C28x, use the ternary ?: operator.

Using `if` results in the following code with `-O3`:

Table 2.7: Saturation with if - inefficient

C Source	Generated Assembly
`int saturate(int sum, int max, int min)` `{` `  if( sum > max )` `  sum = max;` `  if( sum < min )` `  sum = min;` `  return sum;` `}`	`||saturate||:` `  MOVZ  AR6,AL` `  MOV  AL,AR4` `  CMP  AH,AR6` `  MOV  AR6,AH,LT` `  CMP  AL,AR6` `  MOV  AR6,AL,GT` `  MOV  AL,AR6` `  LRETR`

Using `?:` generates the much more efficient MAX/MIN instructions:

Table 2.8: Saturation with ?: - efficient

C Source	Generated Assembly
`int saturate_opt(int sum, int max, int min)` `{` `  sum = (sum > max) ? max : sum;` `  sum = (sum < min) ? min : sum;` `  return sum;` `}`	`||saturate_opt||:` `  MIN  AL,AH` `  MAX  AL,AR4` `  LRETR`

2.4.5 Float vs. double

On C2000 devices without *FPU64* hardware support, there is significant overhead when application compiled for EABI performs operations on double types.

In EABI, the double type is mapped to 64-bit double-precision floating point. Refer to *Application Binary Interface (ABI)* for details. An application compiled for EABI (`--abi=eabi`) can introduce double precision floating point operations in the following ways:

1. Explicit use of the `double` type. Avoid using the `double` type unless the additional accuracy/range is required by the application.

2. Implicitly via floating point constants. Use the `f` suffix when the constant can be treated as a single precision floating point constant.

   Floating point constants without the `f` suffix are treated as double-precision by the compiler. This behavior is mandated by the C standard. This can lead to unexpected introduction of double precision operations in the application. Table 2.10 illustrates this inefficiency - the constant is treated as double precision, leading to conversion of `x` from float to double precision. The multiply is a double precision operation and the result is converted back to single precision.

Table 2.9: Single and double-precision floating point constants

Constant (treated as double)	Constant with `f` suffix (treated as float)
`float foo1(float x)` `{` `  return x * 42.2;` `}`	`float foo2(float x)` `{` `  return x * 42.2f;` `}`

Table 2.10: Comparison of generated assembly

Operations performed in double precision	Operations performed in single precision
`` ||foo1||: ADDB SP,#8 MOVZ AR4,SP SUBB XAR4,#8 MOVZ AR6,AR4 LCR #||__ ↪c28xabi_ftod|| ; call occurs [#||_ ↪_c28xabi_ftod||] MOVZ AR4,SP MOVZ AR6,SP MOVL XAR5,#$C ↪$FL1 SUBB XAR4,#8 SUBB XAR6,#4 MOVZ AR4,AR4 MOVZ AR6,AR6 LCR #||__ ↪c28xabi_mpyd|| ; call occurs [#||_ ↪_c28xabi_mpyd||] MOVZ AR4,SP SUBB XAR4,#4 MOVZ AR4,AR4 LCR #||__ ↪c28xabi_dtof|| ; call occurs [#||_ ↪_c28xabi_dtof||] SUBB SP,#8 LRETR ; return occurs ```	``` ||foo2||: MOVIZ R1H, ↪#16936 MOVXI R1H, ↪#52429 MPYF32 R0H,R1H, ↪R0H LRETR ; return occurs ```

3. Math routines such as `sqrt` from the C Standard Library. In COFF, since `float` and `double` are both 32-bit single precision, there is no distinction between `sqrt` and `sqrtf`. However, with EABI, `sqrtf` operates on a `float` argument, `sqrt` operates on `double`.

Detecting double precision operations

The compiler option `--float_operations_allowed=32` can be used to detect if an application is inadvertently using double precision operations.

This option restricts the type of floating point operations allowed in the application. The default

is all. If set to none, 32, or 64, the application is checked for operations that will be performed at runtime. For example, if `--float_operations_allowed=32` is specified on the command line, the compiler issues an error if the application contains double precision operations. For details, refer to TMS320C28x Optimizing C/C++ Compiler User's Guide.

2.4.6 Intrinsics

The C28x compiler provides intrinsics, special functions that map directly to inlined C28x/*FPU*/*FPU64*/*TMU* instructions. Intrinsics are used to express operations that cannot be easily expressed in C/C++ code. Intrinsics are used like functions - C/C++ variables can be used with intrinsics. Intrinsics are specified with a leading double underscore.

An example is the `__flip`, `__flip32` and `__flip64` set of intrinsics to reverse the order of bits in the source. The compiler maps all 3 intrinsics to an appropriate sequence of C28x `FLIP` instructions.

```
int32_t x;
int32_t y;

...
x = __flip32(y);
```

For details on the set of intrinsics available in the C28x compiler and their description, refer to TMS320C28x Optimizing C/C++ Compiler User's Guide, Section 7.6, Using Intrinsics to Access Assembly Language Statements.

2.4.7 Data allocation for instructions with two memory operands

Many instructions on the C2000 ALU take memory operands, meaning they can operate directly on data in memory without having to load to and store back from registers.

For instructions taking 2 memory operands, the second memory operand (*XAR7) uses the program memory bus. The C2000 RAM blocks only support one access to a memory block in a single pipeline cycle. To avoid a pipeline stalls, data arrays should be allocated to different physical RAM blocks. The physical RAM blocks can be found in the memory map of the device in its data manual.

The following instructions use the program memory bus for a second memory access via *XAR7:

- MAC
- IMACL
- QMACL
- DMAC

- MACF32 (*FPU* only)
- PREAD

Table 2.11 shows the C source for multiplying 2 arrays and accumulating the result. With `-O3 --unified_memory`, the compiler generates a RPT in parallel with a MAC instruction for the loop. The MAC instruction has 2 memory operands, corresponding to `array_1` and `array_2`.

Table 2.11: MAC instruction with 2 memory operands

C Source	Generated Assembly
`int32_t mac(int16_t* array1, int16_t* array2, int16_t M)` `{` `  _nassert(M > 0);` `  int j;` `  int32_t sum = 0;` `  for (j=0; j < M; j++)` `  sum += array1[j] * array2[j];` `  return sum;` `}`	`||mac||:` `  MOVL  XAR7,XAR5` `  ADDB  AL,#-1` `  MOVZ  AR5,AL` `  MOV  P,#0` `  MOVB  ACC,#0` `  RPT  AR5` `||  MAC  P,*XAR4++, *XAR7++` `  ADDL  ACC,P` `  LRETR`

Table 2.4.7 compares performance on F28004x when the arrays are placed in same and different memory blocks. Both scenarios use the same compiler options: `-v28 --abi=eabi --unified_memory --ramfunc=on -O3 --opt_for_speed=5`. Listing 2.1 illustrates how to place the 2 arrays in different physical memory blocks.

Scenario	Cycles
Linker cmd file places both arrays in same memory block	141
Linker cmd file places both arrays in different memory blocks	77

Listing 2.1: Linker command file

```
SECTIONS
{
   .array_a             : > RAMGS2
   .array_b             : > RAMGS3
}
```

2.4.8 Use of volatile

Shared Data

- Any global variable that is read/written by main() and one or more ISRs must be annotated volatile
- Volatile indicates to the compiler that the variable might be modified by something external to the obvious flow of the program such as an ISR
- This ensures the compiler preserves the number of volatile reads and writes to the global variable exactly as written in C/C++ code. The compiler will not:
  - Eliminate redundant reads or writes
  - Re-order accesses

Table 2.12 illustrates the need for volatile when optimizations are enabled. Without the `volatile` qualifier on `flag`, the compiler will remove the `if` block in `main()` because its analysis indicates that `flag` is always 0 and the `if` condition is always false. `volatile` indicates to the compiler that something outside of `main()`, in this case the ISR, can update `flag`.

Table 2.12: Use of volatile

Main application	Interrupt Service Routine
`volatile int flag;` `int x;` `int main()` `{` `  flag = 0;` `  ...` `  if (flag == 1)` `  x++;` `  ...` `}`	`extern int flag;` `interrupt void ISR(void)` `{` `  ...` `  flag = 1;` `  ...` `}`

Peripheral access

- The volatile keyword must be used when accessing memory locations that represent memory mapped peripherals.
- Such memory locations might change value in ways that the compiler cannot predict.
- This ensures the compiler preserves reads and writes to memory exactly as written in the C code.
- A missing volatile qualifier can result in the compiler incorrectly optimizing away or re-ordering reads/writes.

Listing 2.2: Using volatile for peripheral register access

```
1   static inline void
2   GPIO_writePin(uint32_t pin, uint32_t outVal)
3   {
4       volatile uint32_t *gpioDataReg;
5       uint32_t pinMask;
6
7       //
8       // Check the arguments.
9       //
10      ASSERT(GPIO_isPinValid(pin));
11
12      gpioDataReg = (uint32_t *)GPIODATA_BASE +
13                    ((pin / 32U) * GPIO_DATA_REGS_STEP);
```

(continues on next page)

(continued from previous page)

```
14
15          pinMask = (uint32_t)1U << (pin % 32U);
16
17          if(outVal == 0U)
18          {
19              gpioDataReg[GPIO_GPxCLEAR_INDEX] = pinMask;
20          }
21          else
22          {
23              gpioDataReg[GPIO_GPxSET_INDEX] = pinMask;
24          }
25      }
```

2.4.9 Other considerations

Atomic access

- 16-bit reads/writes are atomic.
- 32-bit float reads/writes are atomic except: writing a 32-bit float constant is only atomic if performed with a single opcode.
- 32-bit integer reads/writes:
- 32-bit reads/writes that use a single opcode are atomic.
- Atomic accesses within an ISR: By default, accesses within an ISR are atomic. The INTM bit is automatically set (disable interrupts) by the hardware during the context switch. The exception would be if the application re-enables interrupts within the ISR in order to nest interrupts.
- If possible, group atomic accesses together or create a function to perform the sequence disable-interrupts/atomic-accesses/enable-interrupts.
- For writes to global variables larger than 32 bits (64 bit long double, structures) disable/re-enable interrupts around the write. This ensures the writer updates the entire variable before the reader accesses it and avoids leaving the variable in an inconsistent or incomplete state.

For other atomic operations, there are two recommended approaches:

- Use an atomic compiler intrinsic if one is available. These are documented in the compiler user's guide (www.ti.com/lit/SPRU514). The description will say "in an atomic way".
- **Disable / enable interrupts around atomic operations using below intrinsics:**
  __disable_interrupts( ); __enable_interrupts( );

Listing 2.3 is a code snippet from the Digital Control Library in C2000Ware illustrating disabling interrupts around updates to a struct to ensure atomic updates to the entire structure. Refer to the TMS320C28x Optimizing C/C++ Compiler User's Guide, Table 7-6, TMS320C28x C/C++ Compiler Intrinsics for details on the `__enable_interrupt()` and `__disable_interrupt()` intrinsics.

Listing 2.3: Disable interrupts to ensure atomic struct update

```
1   uint16_t val = __disable_interrupts();
2
3   p->Kp = p->sps->Kp;
4   p->Ki = p->sps->Ki;
5   p->Kd = p->sps->Kd;
6   p->Kr = p->sps->Kr;
7   p->c1 = p->sps->c1;
8   p->c2 = p->sps->c2;
9   p->Umax = p->sps->Umax;
10  p->Umin = p->sps->Umin;
11  DCL_restoreInts(v);
12
13  // If interrupts were originally enabled, re-enable them
14  if (0U == (val & 0x1))
15      __enable_interrupts();
```

Calling asm functions from C code

Any ASM functions called from C code must follow the C calling and register conventions. Refer to the TMS320C28x Optimizing C/C++ Compiler User's Guide, Sections 7.2 Register Conventions, 7.3 Function Structure and Calling Conventions and 7.5 Interfacing C and C++ With Assembly Language.

Any violation of these conventions can result in application passing with -Ooff, but failing at higher optimization levels.

Uninitialized variables

- Using variables without initialization can lead to undefined behavior
- The behavior of an application with uninitialized variables can change with optimization levels, making debug difficult
- Local variables
  - Must be explicitly initialized in the application before any use
- Global variables

  - C standard specifies that global (extern) and static variables without explicit initializations must be initialized to 0 before the program begins running
  - C runtime initialization behavior differs across COFF and EABI
  - Refer to the TMS320C28x Optimizing C/C++ Compiler User's Guide for details - Sections 7.10.3 Automatic Initialization of Variables for COFF and 7.10.4 Automatic Initialization of Variables for EABI.

Interrupts

- RPT instructions are not interruptible, and can potentially delay or block interrupts from executing * For example, if there is a memcpy() instruction in a background function, and the compiler generates RPT instructions for this function, that section of code will be uninterruptible * If the compiler generates RPT instructions within an ISR, interrupts will be blocked, even if interrupt nesting is enabled * To avoid this issue, there are two compiler options available - –no_rpt which will tell the compiler not to generate RPT instructions, or –rpt_threshold which will limit the number of consecutive RPT instructions generated

CHAPTER

THREE

PROFILING

Profiling is used to focus optimization efforts on functions that account for a majority of the runtime.

There are different approaches to profiling:

- Code Composer Studio™ (CCS) Profile clock feature
- CPUTimer
- CPUTimer and Function entry/exit hooks
- Toggle GPIO pin

3.1 CCS Profile Clock

The Code Composer Studio Profile Clock feature can be used to count the number of cycles from one breakpoint to the next. This is a quick way to determine the cycles taken by an arbitrary region of code.

3.2 CPUTimer

Using the CPUTimer is a programmatic approach to determining the number of cycles between any two points in the code. For example, Listing 3.3 illustrates how to determine the number of cycles taken by the loop using the CPUTimer.

Listing 3.1: CPUTimer header file (cycle_counter.h)

```
1 #ifndef _CYCLE_COUNTER_H_
2 #define _CYCLE_COUNTER_H_
3
4 #include <stdint.h>
5
```

(continues on next page)

(continued from previous page)

```
void     CycleCounter_Init(void);
uint32_t CycleCounter_Read(void);
void     CycleCounter_Stop(void);

#endif
```

Listing 3.2: CPUTimer source file (cycle_counter.c)

```
#include "common/inc/cycle_counter.h"
#include "driverlib/cputimer.h"

#define CPUTIMER_MAX_PERIOD (0xFFFFFFFFUL)

void CycleCounter_Init(void)
{
    CPUTimer_clearOverflowFlag(CPUTIMER1_BASE);

    CPUTimer_setPeriod(CPUTIMER1_BASE, CPUTIMER_MAX_PERIOD);

    CPUTimer_setPreScaler(CPUTIMER1_BASE, 0UL);

    CPUTimer_reloadTimerCounter(CPUTIMER1_BASE);

    CPUTimer_stopTimer(CPUTIMER1_BASE);

    CPUTimer_startTimer(CPUTIMER1_BASE);
}

void CycleCounter_Stop(void)
{
    CPUTimer_stopTimer(CPUTIMER1_BASE);
}

// With higher levels of optimization, it's possible that the compiler␣
↪moves
// application code before the first call to CycleCounter_Read() or␣
↪after the
// second call to CycleCounter_Read(). This can result in the reported␣
↪cycle
// count being lower than the actual cycle count.
// Disabling inlining of CycleCounter_Read prevents this from␣
↪occurring.
#pragma FUNC_CANNOT_INLINE(CycleCounter_Read)
uint32_t CycleCounter_Read(void)
{
```

(continues on next page)

(continued from previous page)

```
34      return CPUTIMER_MAX_PERIOD - CPUTimer_getTimerCount(CPUTIMER1_
    ↪BASE);
35  }
```

Note:

- For simplicity, this implementation does not consider overflow.
- For details on the DriverLib CPU Timer functions, refer to the DriverLib User's Guide for the device. E.g. the User's Guide for F28004x is available at `<C2000Ware install dir>/device_support/f28004x/docs/F28004x_DriverLib_Users_Guide.pdf`

Listing 3.3: CPUTimer Example

```
1   CycleCounter_Init();
2
3   uint32_t start    = CycleCounter_Read();
4   uint32_t overhead = CycleCounter_Read() - start;
5
6   start = CycleCounter_Read();
7
8   int i;
9   for (i=0; i < 100; i++)
10      asm("\tNOP;");
11
12  uint32_t time = CycleCounter_Read() - start - overhead;
13
14  printf("Cycles: %ld\n", time);
```

3.3 CPUTimer with Function Entry/Exit Hooks

The CPUTimer can be combined with the Function Entry/Exit Hooks feature available in the compiler to generate a quick profiler.

When the Entry/Exit hooks feature is enabled using the `--entry_hook` and `--exit_hook` options, the compiler inserts a call to an entry hook on entry to each function in the program. The compiler also inserts a call to a exit hook on exit of each function. Refer to the TMS320C28x Optimizing C/C++ Compiler User's Guide for details - Section 2.14, Enabling Entry Hook and Exit Hook Functions.

Listing 3.4 illustrates using the entry and exit hooks along with the CPUTimer to implement a simple profiler.

Listing 3.4: Using extry and exit hooks to implement a profiler

```
#include "common/inc/cycle_counter.h"
#include "common/inc/profiling_hooks.h"
#include <stdio.h>

#define MAX_ENTRIES (64)

// Indicate if the timestamp is associated with function entry or exit
typedef enum { PD_ENTRY=0, PD_EXIT } PD_Mode;

// Struct for data associated with a single timestamp
typedef struct {
    uint32_t function_address;
    uint32_t timestamp;
    PD_Mode  mode;
} ProfileData;

// Array to store profile data
ProfileData table[MAX_ENTRIES];
int         index = 0;

// Entry hook function used to record cycle count on entry into function
void __entry_hook(void (*addr)())
{
    if (index >= MAX_ENTRIES) return;

    table[index].function_address = (uint32_t)addr;
    table[index].mode             = PD_ENTRY;
    table[index].timestamp        = CycleCounter_Read();
    index++;
}

// Exit hook function used to record cycle count on exit from function
void __exit_hook(void (*addr)())
{
    if (index >= MAX_ENTRIES) return;

    table[index].timestamp        = CycleCounter_Read();
    table[index].function_address = (uint32_t)addr;
    table[index].mode             = PD_EXIT;
    index++;
}
```

Files with functions that need to be profiled are compiled using the `--entry_hook`

`--entry_parm=address --exit_hook --exit_parm=address` options.

When an application is built with the entry/exit hooks in Listing 3.4, the table is populated with profile data. For example, the code snippet in Listing 3.5 results in the following table:

Listing 3.5: Example using the hook functions for profiling

```
int main()
{
    ProfileData_init();

    foo();

    ProfileData_print();

    return 0;
}

void foo()
{
    int i;
    for (i=0; i < 100; i++)
        asm("\tNOP;");

    bar();
    bar();
}

void bar()
{
    int i;
    for (i=0; i < 100; i++)
        asm("\tNOP;");
}
```

```
0x00a647, 0, 25
0x00a637, 0, 562
0x00a637, 1, 1090
0x00a637, 0, 1134
0x00a637, 1, 1662
0x00a647, 1, 1697
```

CHAPTER

FOUR

IMPROVING PERFORMANCE

This chapter describes various techniques to improve the performance of C code on the C28x CPU. One or more of the techniques described below or a combination of techniques can be used to improve performance. Techniques that are beneficial will depend on the nature of the application and can vary from application to application. For example:

- *Executing from RAM* illustrates improving performance by placing frequently executed functions in RAM.
- *Optimization levels* provides an overview of compiler optimization levels.
- *Inlining* can improve performance in an application with multiple levels of function calls.
- Compiler annotations such as *Pragmas*, *Assertions* and *Restrict* provide additional information to the compiler to help improve the performance of generated code.
- *Loop unrolling* is a technique to improve the performance of small loops.
- *Leveraging DMAC instructions* describes how to leverage the Dual Multiply Accumulate (`DMAC`) instructions from C source.

4.1 Memory

4.1.1 Executing from flash

The TMS320F28xxx family is designed for stand alone operation in embedded controller applications. The on-chip flash usually eliminates the need for external non-volatile memory and a host processor from which to boot-load. For details on running applications from internal flash memory, refer to the application note Running an Application from Internal Flash Memory on the TMS320F28xxx DSP.

Executing code from RAM is faster than executing it from flash. However, C2000 MCUs support code-prefetch and data caching while executing from flash to minimize overhead. For details on these features, refer to the "Flash and OTP Memory" section in the device Technical Reference Manual (TRM).

Note: Both code-prefetch and data caching are disabled at power-up. Application software must enable code-prefetch and configure the wait states appropriately. It also needs to enable the data cache. Refer to the `InitFlash()` function in C2000Ware for details. For example, `InitFlash()` for F28004x is defined in `<C2000Ware install directory>/device_support/f28004x/common/source/f28004x_sysctrl.c`.

Table 4.1 lists cycle counts for executing the loop in Listing 4.1 on flash and RAM. Compiler options used: `-O3 --opt_for_speed=5 --abi=eabi`. The `--ramfunc=on` option and corresponding linker command file was used to execute code from RAM.

Table 4.1: Comparing code execution cycles for flash vs. RAM

Description	Cycles on F28004x
flash without enabling code-prefetch	72006
flash with code-prefetch enabled	59996
RAM	54002
Cycles to execute the loop (calculated)	54000

Listing 4.1: Loop used to compare flash vs. RAM code execution

```
    int i;
    //                  RPT+4 NOPs      BANZ      iterations
    // Total cycles = (5 * 10       +   4     ) * 1000        = 54000␣
↪cycles
    for (i=0; i < 1000; i++)
    {
        asm("          RPT #3 || NOP;"); // 5 cycles - RPT + 4 NOPs
        asm("          RPT #3 || NOP;");
        asm("          RPT #3 || NOP;");
        asm("          RPT #3 || NOP;");
        asm("          RPT #3 || NOP;");
        asm("          RPT #3 || NOP;");
        asm("          RPT #3 || NOP;");
        asm("          RPT #3 || NOP;");
        asm("          RPT #3 || NOP;");
        asm("          RPT #3 || NOP;");
    }
```

4.1.2 Executing from RAM

Code

As seen from the data in Table 4.1, it is beneficial to copy time critical code from its load address in flash to RAM for execution.

The `ramfunc` attribute is a TI compiler feature which allows code to easily specify that a function will be placed in and executed out of RAM. The attribute is applied to a function with GCC attribute syntax, as follows:

```
__attribute__((ramfunc))
void f(void) { ... }
```

The `--ramfunc=on` option is equivalent to specifying the attribute on all functions in source files compiled with the option, with no source modification required.

Note: Fast branch instructions (SBF/BF) are generated for RAM functions. These instructions take advantage of dual prefetch queue on the C28x core that reduces the cycles for a taken branch from 7 to 4.

The `ramfunc` attribute and option is available C2000 compiler versions 15.6 and above. For older compilers that do not support this feature, the CODE_SECTION pragma may be used in combination with linker command file modifications.

```
#pragma CODE_SECTION(f, ".TI.ramfunc")
void f(void) { ... }
```

The linker command file is set up to create symbols corresponding to the load and run addresses for the `.TI.ramfunc` section.

Listing 4.2: Linker command snippet for `.TI.ramfunc`

```
   .TI.ramfunc      : LOAD = FLASH_BANK0_SEC1,
                      RUN = RAMLS0to7,
                      LOAD_START(RamfuncsLoadStart),
                      LOAD_SIZE(RamfuncsLoadSize),
                      LOAD_END(RamfuncsLoadEnd),
                      RUN_START(RamfuncsRunStart),
                      RUN_SIZE(RamfuncsRunSize),
                      RUN_END(RamfuncsRunEnd),
                       ALIGN(4)
```

Code in the application uses a `memcpy` to copy the `.TI.ramfunc` section from link address in flash to run address in RAM.

Listing 4.3: Copy .TI.ramfunc from flash to RAM

```
    memcpy(&RamfuncsRunStart, &RamfuncsLoadStart, (size_t)&
↪RamfuncsLoadSize);
```

Data

Constant arrays - if access to a constant array is time critical, then consider copying the array from its load address in flash to a RAM address to reduce access time.

4.1.3 Other considerations

- If code accesses data within the same physical memory, then performance will degrade due to resource conflicts. Place code and the data it accesses in separate blocks to improve performance.
- Wait states will degrade performance. Most SARAM is zero-wait on 28x MCUs. Always check the data manual to find the wait states for each physical block and whether it applies to program or data accesses.
- If code makes extensive use of two data buffers, putting each buffer in a different RAM block may improve performance. The goal is to reduce the pipeline stalls due to write and read occurring in the same cycle to different buffers. Refer to *Data allocation for instructions with two memory operands* for an example.

4.2 Optimization levels

The compiler can perform many optimizations to improve the execution speed and reduce the size of C and C++ programs. Table 4.2 lists the optimization levels available, the scope of each level and some examples of optimizations performed at each level.

Table 4.2: Optimization levels

Optimization level	Scope	Optimizations performed
`--opt_level=0`, `-Ooff`	None	None. This is the default setting for the C28x compiler.
`--opt_level=0`, `-O0`	Statement	• Allocates variables to registers • Performs loop rotation • *Simplifies expressions*, statements • Eliminates *unused assignments* • *Dead code elimination* • Expands calls to functions declared inline • Simplifies control code (if-else)
`--opt_level=1`, `-O1`	Block	• Performs all –opt_level=0 (-O0) optimizations, plus: • Performs local *constant propagation and folding*, copy propagation • Eliminates local common subexpressions
`--opt_level=2`, `-O2`	Function	• Performs all –opt_level=1 (-O1) optimizations, plus: • Loop optimizations, *Loop unrolling* • Eliminates global common subexpressions • Eliminates global unused assignments • Generates *auto incremented addresses*
`--opt_level=3`, `-O3`	File (i.e. across functions in a file)	• Performs all –opt_level=2 (-O2) optimizations, plus: • *Inlining* of small functions • Removes functions not called in the file
`--opt_level=4`, `-O4`	Program	Link time optimizations. Refer to TMS320C28x Optimizing C/C++ Compiler User's Guide, Section 3.6, Link-Time Optimization (–opt_level=4 Option).

Note: To generate efficient code, it is highly recommended to set the optimization level at `-O2` or

higher.

For descriptions of these optimizations, refer to TMS320C28x Optimizing C/C++ Compiler User's Guide, Section 3.16, What Kind of Optimization Is Being Performed?

4.2.1 Examples

Expression simplification

Listing 4.4: Example to illustrate expression simplification

```
int32_t test(int32_t a, int32_t b, int32_t c, int32_t d)
{
    int32_t tmp;

    if (d > 0)
        tmp = (a * b) + (a * c);
    else
        tmp = (a * b);

    return tmp;
}
```

There are 3 32-bit multiplies in the source code in Listing 4.4, which require the `IMPYL` instruction. At `-O2`, the compiler is able to simplify the expressions to generate **1** `IMPYL` instructions vs. **3** without optimizations.

Optimization level	Number of IMPYL in generated assembly
-Ooff	3
-O0, -O1	2
-O2	1

Constant propagation and folding

Listing 4.5: Example to illustrate constant propagation and folding

```
1 int32_t constant(int32_t c, int32_t d)
2 {
3     int32_t a = 42;
4     int32_t b = 10;
5     int32_t tmp;
```

(continues on next page)

(continued from previous page)

```
6
7       if (d > 0)
8           tmp = (a * b) + (a * c);
9       else
10          tmp = (a * b);
11
12      return tmp;
13  }
```

This optimization propagates the values of constants into expressions and precomputes the results of constant expressions.

At `-O2` and higher, the compiler replaces the expression with:

```
(d > 0L) ? (tmp = (c+10L)*42L) : (tmp = 420L);
```

I.e. it propagates the values of `a` and `b` into the expressions on lines 8, 10 and computes `a * b` on line 10, replacing the expression with the constant `420`.

Unused assignment removal

Listing 4.6: Example to illustrate unused assignment removal

```
1   int32_t unused_asg(int32_t a, int32_t b, int32_t c, int32_t d)
2   {
3       int32_t tmp = 42;
4
5       if (d > 0)
6           tmp = (a * b) + (a * c);
7       else
8           tmp = (a * b);
9
10      return tmp;
11  }
```

In Listing 4.6, the assignment to `tmp` on line 3 is not required because of the subsequent assignments to `tmp` on both the if and else paths on lines 6 and 8 respectively. At `-O0` and higher, the compiler removes the assignment.

This improves performance because expressions not required for correctness are removed, resulting in fewer cycles.

Auto incremented addressing

Listing 4.7: Example to illustrate auto incremented addressing

```
int32_t addressing(int32_t* array, int16_t N)
{
    int32_t sum = 0;
    int32_t i   = 0;

    _nassert (N > 0);
    for (i = 0; i < N; i++)
        sum += array[i];

    return sum;
}
```

At `-O2` and higher, the compiler generates the efficient auto incremented addressing mode for the loop in Listing 4.7, resulting in fewer instructions to execute the loop: **12** instructions at `-O1` vs. **8** instructions at `-O2`.

Table 4.3: Assembly generated for loop at various optimization levels

-O1	-O2 generates efficient *XARn++ addressing
||CL7||: ;*** g2: ;*** sum += array[i]; ;*** if ((++i) <(long)N) goto g2; MOVL ACC,XAR5 LSL ACC,1 ADDL ACC,XAR4 MOVL XAR6,ACC ADDB XAR5,#1 MOVL ACC,P ADDL ACC,*+XAR6[0] MOVL P,ACC MOV AL,AR7 MOV ACC,AL CMPL ACC,XAR5 B ||CL7||,GT	||CL7||: ;*** g2: ;*** sum += *U$7++; ;*** if ((--L$1) != (-1L)) goto g2; MOVL ACC,XAR6 SUBB XAR5,#1 ADDL ACC,*XAR4++ MOVL XAR6,ACC MOVB ACC,#0 SUBB ACC,#1 CMPL ACC,XAR5 B ||CL7||,NEQ

Dead code elimination

Listing 4.8: Example to illustrate dead code elimination

```
int32_t dce(int32_t a, int32_t b, int32_t c, int32_t d)
{
    int32_t tmp1 = a * b * c * d;
    int32_t tmp;
```

(continues on next page)

(continued from previous page)

```
    if (d > 0)
        tmp = (a * b) + (a * c);
    else
        tmp = (a * b);

    return tmp;
}
```

In Listing 4.8, the expression computed and assigned to `tmp1` is *dead* because `tmp1` is not used anywhere in the function. *Dead code elimination* is a compiler technique to remove unused expressions. At `-Ooff`, the generated assembly contains **6** `IMPYL` instructions, corresponding to each of the multiplies in the source. At `-O0`, the compiler is able to optimize the code and reduce the number of `IMPYL` generated to **2** using a combination of dead code elimination and expression simplification.

```
(d > 0L) ? (tmp = (b+c)*a) : (tmp = a*b);
```

4.2.2 Code size vs. speed tradeoffs

For details on code size vs. speed tradeoffs, refer to TMS320C28x Optimizing C/C++ Compiler User's Guide, Section 3.2, Controlling Code Size Versus Speed.

4.2.3 Optimization levels and debug

At higher levels of optimization, it gets progressively harder to debug (e.g. single-step) the application. This is because at higher optimization levels, the compiler makes transformations to the application to reduce its execution time, memory footprint, power consumption, or a combination of these. These transformations significantly change the layout of the code and make it difficult, or impossible, for the debugger to identify the source code that corresponds to a set of assembly instructions.

The best approach is to perform initial development and debug with optimization disabled and then enable optimizations. Refer to *Enable debugging* for details.

4.2.4 Optimizer interlist

Optimization makes normal source interlisting impractical, because the compiler extensively rearranges the program.

The `--src_interlist` option interlists compiler comments with assembly source statements. When this option is used with optimization enabled, the interlist feature does not run as a separate pass. Instead, the compiler inserts comments into the code, indicating how the compiler has rearranged and optimized the code. These comments appear in the assembly language file as comments starting with `;**`.

Table 4.4: Output of the `--src_interlist` option

C source

```
float fmac(float *farray, int N)
{
    int i;
    float sum = 0.0f;

    #pragma MUST_ITERATE(4, , 4)
    #pragma UNROLL(2)
    for (i = 1; i < N; i++)
        sum += farray[i] *␣
↪farray[i-1];

    return sum;
}
```

Interlist output in the assembly file

```
||fmac||:
;***           ------------------
 ↪----    U$13 = farray;
;***           ------------------
 ↪----    L$1 = (N>>1)-1;
;*** 31        ------------------
 ↪-----    sum = 0.0F;
;***           ------------------
 ↪----    #pragma MUST_ITERATE(2,
 ↪ 16382, 2)
;***           ------------------
 ↪----    #pragma UNROLL(1L)
;***           ------------------
 ↪----    // LOOP BELOW UNROLLED␣
 ↪BY FACTOR(2)
;***           ------------------
 ↪----    #pragma LOOP_
 ↪FLAGS(4103u)
;***         --------------------
 ↪--g2:
;*** 36        ------------------
 ↪-----    C$1 = U$13[1];
;*** 36        ------------------
 ↪-----    sum += *U$13++*C$1;
;*** 36        ------------------
 ↪-----    sum += U$13[1]*C$1;
;*** 35        ------------------
 ↪-----    ++U$13;
;*** 35        ------------------
 ↪-----    if ( (--L$1) != (-1)␣
 ↪) goto g2;
;*** 38        ------------------
 ↪-----    return sum;
```

From the listing in Table 4.4, it is clear that the loop has been unrolled 2x by the optimizer. The original pragmas from the source have also been updated to account for the unrolling. For details on loop unrolling, refer to *Loop unrolling*.

Warning: The `--c_src_interlist` option can have a negative effect on performance and code size because it can prevent some optimizations from crossing C/C++ statement boundaries. So, the `--src_interlist` is recommended when optimizations are enabled. In CCS,

the `--src_interlist` option is available in the "Source interlist" dropdown under Build -> C2000 Compiler -> Advanced Options -> Assember Options.

For details on the interlist option, refer to TMS320C28x Optimizing C/C++ Compiler User's Guide, Section 3.10, Using the Interlist Feature With Optimization.

4.3 Inlining

Inlining is the process of inserting code for a function at the point of call. Benefits:

- Saves the overhead of a function call.
- Allows the optimizer to optimize the function in the context of the surrounding code.

When an inline function is called, a copy of the C/C++ source code for the function is inserted at the point of the call. Inlining function expansion can speed up execution by eliminating function call overhead. This is particularly beneficial for very small functions that are called frequently or larger functions that are called very few times (once or twice). **Function inlining involves a tradeoff between execution speed and code size, because the code is duplicated at each function call site.**

Table 4.5 lists cycle counts for executing the function sequence in Listing 4.9 without and with inlining enabled.

`foo1` calls `foo2`, which calls `foo3` which in turn calls `foo4`. Using `static` enables the compiler to remove the function bodies after inlining. This reduces code size by removing the need to have more than one copy of the function.

Table 4.5: Comparing code execution times for Flash vs. RAM

Description	Cycles on F28004x
With –opt_level=3. Inlining disabled using –auto_inline=0	58
With –opt_level=3 (Inlining is enabled by default at this optimization level). The reduction in cycles is due to the elimination of call instructions and additional optimization opportunities from inlining.	19

Listing 4.9: Function call sequence used to illustrate benefits of inlining

```
1 float foo1(float f1, float f2)
2 {
3     return f1 * f2 + foo2(f1, f2);
```

(continues on next page)

(continued from previous page)

```
}

static float foo2(float f1, float f2)
{
    return f1 * 2.0f - foo3(f1, f2);
}

static float foo3(float f1, float f2)
{
    return f2 * 4.0f - foo4(f1, f2);
}

static float foo4(float f1, float f2)
{
    return f1 * (f2 - f1);
}
```

There are different approaches to controlling the scope of inlining to manage the execution speed - code size tradeoff.

- If the project is compiled with `--opt_level=3` (`-O3`) or higher.

  `-O3` has the side effect of enabling inlining across all the files in the project and can result in a significant code size increase. Use `--auto_inline=[size]` with `--opt_level=3` to place a limit on the size of the functions that are inlined. If required, inlining can be disabled at `-O3` using `--auto_inline=0` or `-oi0`. Refer to the TMS320C28x Optimizing C/C++ Compiler User's Guide, Section 3.5, "Automatic Inline Expansion (–auto_inline Option)" for details.

- If the project is compiled with `--opt_level=1` or `--opt_level=2`

  Use `static inline` on specific functions that would benefit from being inlined into call sites.

- To enforce inlining irrespective of optimization level, use either the attribute always_inline or the pragma FUNC_ALWAYS_INLINE.

Refer to the TMS320C28x Optimizing C/C++ Compiler User's Guide, Section 2.11, "Using Inline Function Expansion" for details.

4.4 Pragmas

Pragma directives tell the compiler how to treat a certain function, object, or section of code. This section covers pragmas that are relevant to improving performance of executed code. For the complete set of pragmas supported by the compiler, refer to Section 6.10, Pragma Directives in the TMS320C28x Optimizing C/C++ Compiler User's Guide.

4.4.1 MUST_ITERATE

```
#pragma MUST_ITERATE ( min, max, multiple )
```

The MUST_ITERATE pragma specifies to the compiler certain properties of a loop. The arguments `min` and `max` are programmer-guaranteed minimum and maximum trip counts. The trip count is the number of times a loop iterates. The trip count of the loop must be evenly divisible by `multiple`.

Listing 4.10: Using MUST_ITERATE pragma to avoid generating loop bounds checks and enable unrolling by 4.

```
1  int16_t sum(int16_t* input, int16_t count)
2  {
3      int16_t sum = 0;
4      int16_t i   = 0;
5
6      #pragma MUST_ITERATE(4, , 4)
7      for (i = 0; i < count; i++)
8          sum += input[i];
9
10     return sum;
11 }
```

The assembly code generated for Listing 4.10 when compiled with `--opt_level=3` is shown in Listing 4.11.

Listing 4.11: Assembly generated with MUST_ITERATE pragma

```
||sum||:
        MOV       AH,AL
        MOVB      AL,#0
        ASR       AH,2
        ADDB      AH,#-1
        MOVZ      AR6,AH
||$C$L1||:
```

(continues on next page)

(continued from previous page)

```
        ADD         AL,*XAR4++
        ADD         AL,*XAR4++
        ADD         AL,*XAR4++
        ADD         AL,*XAR4++
        BANZ        ||$C$L1||,AR6--
        LRETR
```

Refer to the *Assertions* section for another approach to specifying loop information to the compiler.

Warning: When specifying a multiple via the MUST_ITERATE pragma, results of the program are undefined if the trip count is not evenly divisible by multiple. Also, results of the program are undefined if the trip count is less than the minimum or greater than the maximum specified.

4.4.2 UNROLL

```
#pragma UNROLL(n)
```

If possible, the compiler unrolls the loop so there are `n` copies of the original loop. The compiler only unrolls if it can determine that unrolling by a factor of n is safe. In order to increase the chances the loop is unrolled, the compiler needs to know certain properties:

- The loop iterates a multiple of n times. This information can be specified to the compiler via the multiple argument in the MUST_ITERATE pragma.
- The smallest possible number of iterations of the loop
- The largest possible number of iterations of the loop

In cases where the compiler is not able to analyze and determine these properties, the `MUST_ITERATE` pragma can be used.

Specifying `#pragma UNROLL(1)` asks that the loop not be unrolled. Automatic loop unrolling also is not performed in this case.

4.4.3 FUNC_ALWAYS_INLINE

```
#pragma FUNC_ALWAYS_INLINE ( func )
```

The pragma FUNC_ALWAYS_INLINE and the equivalent always_inline attribute force a function to be inlined (where it is legal to do so) unless `--opt_level=off`. That is, the pragma FUNC_ALWAYS_INLINE forces function inlining even if the function is not declared as inline and the `--opt_level=0` or `--opt_level=1`.

For a discussion on the benefits of inlining, refer to section *Inlining*.

4.5 Assertions

The `_nassert` intrinsic generates no code and so is not a typical compiler intrinsic. Instead, it tells the compiler that the expression declared with the assert function is true. It can be used to assert that certain conditions are true, which in turn can be used by the compiler during its optimizations.

Warning: Code can fail at runtime if the condition specified in the `_nassert` is not true.

Listing 4.12 illustrates an example of using `_nassert`. In this case, the programmer is **guaranteeing** that the loop executes at least once and `count` is a multiple of 4. This enables the compiler to avoid generating code to check for `count == 0` and peeled iterations during unrolling.

Listing 4.12: Using nassert to avoid generating unnecessary checks

```
1  #include <stdint.h>
2
3  int16_t sum(int16_t* input, int16_t count)
4  {
5      int16_t sum = 0;
6      int16_t i   = 0;
7
8      _nassert(count > 0 && count % 4 == 0);
9
10     for (i = 0; i < count; i++)
11         sum += input[i];
12
13     return sum;
14 }
```

Table 4.6 shows the assembly generated without and with the `_nassert`.

Table 4.6: Assembly comparison

Asm generated with _nassert (loop is unrolled 4x)

```
||sum||:
        MOV       AH,AL
        MOVB      AL,#0
        ASR       AH,2
        ADDB      AH,#-1
        MOVZ      AR6,AH
||$C$L1||:
        ADD       AL,*XAR4++
        ADD       AL,*XAR4++
        ADD       AL,*XAR4++
        ADD       AL,*XAR4++
        BANZ      ||$C$L1||,AR6--
        LRETR
```

Asm generated without _nassert (loop is unrolled 2x)

```
||sum||:
        MOV       AH,AL
        MOVB      XAR7,#0
        BF        ||$C$L4||,LEQ
        CMPB      AH,#2
        BF        ||$C$L1||,GEQ
        MOV       PL,#0
        BF        ||$C$L3||,UNC
||$C$L1||:
        AND       AH,AH,#0xfffe
        MOV       PL,AH
        MOVL      XAR5,XAR4
        MOV       AH,AL
        ASR       AH,1
        ADDB      AH,#-1
        MOVZ      AR6,AH
||$C$L2||:
        MOV       AH,AR7
        ADD       AH,*XAR5++
        ADD       AH,*XAR5++
        MOVZ      AR7,AH
        BANZ      ||$C$L2||,AR6--
||$C$L3||:
        TBIT      AL,#0
        BF        ||$C$L4||,NTC
        MOVL      ACC,XAR4
        SETC      SXM
        ADD       ACC,PL
        MOVL      XAR4,ACC
        MOV       AL,AR7
        ADD       AL,*+XAR4[0]
        MOVZ      AR7,AL
||$C$L4||:
        MOV       AL,AR7
        LRETR
```

4.6 Restrict

4.6.1 Overview

The `restrict` keyword is a qualifier for a pointer variable's type. By applying restrict to the type declaration of a pointer **p**, the programmer is making the following guarantee to the compiler:

Within the scope of the declaration of **p** *, only* **p** *or expressions based on* **p** *will be used to access the object pointed to by* **p**.

The compiler can take advantage of this guarantee to generate more efficient code.

Explanation of the guarantee:

1. Within the scope of the declaration of **p**

   p is is a pointer variable. Examples are `p1`, `s.p2`, `p3[i]`,and both `p4` and `p5` in `p4->p5[]`. The program region over which the restriction applies is the scope of **p**'s declaration.

2. only **p** or expressions based on **p**

   This refers to the pointer in such accesses as `*p`, `p[i]`, and `p[i+3]`.

3. will be used to access the object pointed to by **p**

   Only actual fetches and stores are accesses. `p[i]` is an access, but `&p[i]` and `p+i` are not.

Warning: Incorrect usage of `restrict` can lead to the compiler generating incorrect code. An example of incorrect usage is applying `restrict` to pointers that point to overlapping objects in memory. Refer to *Incorrect Usage* for an example.

4.6.2 Example

The comparison below illustrates the effectiveness of using `restrict`. Adding the `restrict` qualifier to the types for pointers `a1` and `b1` guarantees to the compiler that these pointers will not be used to access the same memory location as `t->sum1` or `t->sum2`. This enables the compiler to generate a more efficient sequence of instructions for the loop.

In Table 4.7, the loop executes 256 times. The cycle counts were measured on F280049C with code and data in RAM and with `-O3 --opt_for_speed=5`. With `restrict`, the cycle count reduces from **3618** to **1209** cycles.

Table 4.7: Effectiveness of restrict.

`#include <stdint.h>` `typedef struct` `{` `  float*  a;` `  float*  b;` `  float  sum1;` `  float  sum2;` `  int16_t N;` `} Test;` `void foo2(Test *t)` `{` `  float* a1 = t->a;` `  float* b1 = t->b;` `  int i;` `  for (i = 0; i < t->N; i++)` `  {` `  t->sum1 += a1[i] * b1[i];` `  t->sum2 += a1[i] * a1[i];` `  }` `}`	`#include <stdint.h>` `typedef struct` `{` `  float*  a;` `  float*  b;` `  float  sum1;` `  float  sum2;` `  int16_t N;` `} Test;` `void foo1(Test *t)` `{` `  float* restrict a1 = t->a;` `  float* restrict b1 = t->b;` `  int i;` `  for (i = 0; i < t->N; i++)` `  {` `  t->sum1 += a1[i] * b1[i];` `  t->sum2 += a1[i] * a1[i];` `  }` `}`
3618 cycles	1209 cycles

Note: `restrict` is effective only at `--opt_level=2` or higher.

4.6.3 Usage

Global variables

```
int *restrict p1;
int *restrict p2;
extern int A[];
```

Taken together, these file scope declarations of global variables guarantee to the compiler that if an object is accessed using any one of p1, p2, or A[] it will not be accessed using any of the others. Furthermore, since the file scope encompasses all other scopes, no accesses through local pointer variables can access the object pointed to by p1 or p2.

Function parameters

The parameters in a function declaration have function prototype scope, which terminates at the end of the declaration:

```
void foo(float *restrict v1, float *v2, int n);
```

In this function's definition, the parameters have the same block scope as i:

```
void foo(float *restrict v1, float *v2, int n)
{
    int i;
    ...
}
```

Restricting `v1` guarantees to the compiler that the object pointed to by `v1` does not overlap with objects pointed to by other pointers in the body of foo().

Note: Arrays are passed by reference in C. To restrict-qualify an array parameter, the restrict keyword should appear as follows:

```
void foo(short a[restrict 100]);
```

Local pointer variables

```
void foo(Test *t)
{
    float* restrict a1 = t->a;
    float* restrict b1 = t->b;

    ...
}
```

Adding restrict qualification to the pointer's type in local variables `a1` and `b1` enables the programmer to restrict the nature of the accesses made via the pointer within the smaller scope of the function.

4.6.4 Incorrect Usage

Listing 4.13 is an example of incorrect use of `restrict`. Pointers `p` and `q` are restrict-qualified. **However, the arguments to `copy` are such that the pointers overlap**. This can lead to the compiler generating invalid code.

Listing 4.13: Incorrect usage of restrict

```
void copy(int n, int *restrict p, int *restrict q)
{
    while (n-- > 0) *p++ = *q++;
}

void test(void)
{
    extern int d[100];
    copy(50, d+1, d); // Breaks the restrict guarantee!
}
```

4.7 Loop unrolling

Loop unrolling is a technique to improve performance. Small loops are expanded such that an iteration of the loop is replicated a certain number of times in the loop body. The number of times an iteration is replicated is known as the unroll factor.

4.7.1 Benefits

- **Reduce branch overhead** This is especially significant for small loops. For example, the loop in Listing 4.10 is unrolled by a factor of 4. From the assembly Listing 4.11, it is evident that the branch overhead is reduced by a factor of 4 (one `BANZ` for 4 iterations vs. 1 without unrolling).

There are additional benefits on C28x CPUs with FPU support:

- **Generate RPTB for small loops** - loop unrolling increases the number of instructions in the loop body and enables the compiler to meet the minimum block size requirements for the `RPTB` instruction.
- **Improved floating-point performance** - loop unrolling can improve performance by providing the compiler more instructions to schedule across the unrolled iterations. This reduces the number of NOPs generated and also provides the compiler with a greater opportunity to generate parallel instructions.

Note: Loop unrolling will result in a code size increase because the compiler replicates the loop body. `#pragma UNROLL(1)` can be used to prevent the compiler from unrolling the loop.

4.7.2 Performing unrolling

There are two ways in which loop unrolling can be performed:

1. The compiler can automatically unroll the loop. Listing 4.12 is an example of a loop that is unrolled 4 times by the compiler.
2. The UNROLL pragma can be used to indicate to the compiler that the loop is a candidate for unrolling. Refer to *UNROLL* for details.

4.8 Leveraging DMAC instructions

Dual Multiply and Accumulate (DMAC) instructions perform multiply-accumulate operations on two adjacent signed integers (16-bit) simultaneously, optionally shifting the products. A multiply-accumulate operation multiplies two numbers and adds that product to an accumulator.

Table 4.8: Generating DMAC from C source

C source

```
long dmac(int* array1, int*␣
↪array2, int M)
{
    // Assert to the compiler␣
↪that both arrays are 32bit␣
↪aligned
    _nassert((long)array1 % 2 ==␣
↪0);
    _nassert((long)array2 % 2 ==␣
↪0);

    // Assert to the compiler␣
↪that M is even and > 0
    _nassert((M > 0) && (M % 2␣
↪== 0));

    int j;
    long sum = 0;
    for (j=0; j < M; j++)
        sum += (long)array1[j] *␣
↪array2[j];

    return sum;
}
```

Assembly output with DMAC

```
||dmac||:
        ASR     AL,1
        MOVL    XAR7,XAR5
        ADDB    AL,#-1
        MOVZ    AR5,AL
        MOV     P,#0
        MOVB    ACC,#0
        RPT     AR5
||      DMAC    ACC:P,*XAR4++,
↪*XAR7++
        ADDL    ACC,P
        LRETR
```

Table 4.8 illustrates an approach to generating the DMAC instruction from C source. Refer to the TMS320C28x Optimizing C/C++ Compiler User's Guide, Section 3.15, "Compiler Support for Generating DMAC Instructions" for details.

CHAPTER

FIVE

COMMON ISSUES WITH OPTIMIZATIONS

A potential scenario during development is that the application works when compiler optimizations are disabled (`-Ooff`), but fails with higher levels of optimization (`-O1, -O2, -O3 or -O4`). Typical reasons for this include:

- Access to shared data from main program and Interrupt Service Routines (ISRs)
  - Volatile qualifiers
  - Atomic updates
- Accessing memory mapped peripheral registers without volatile
- Calling asm functions from C code without following C conventions
- Uninitialized variables

5.1 Shared Data

- Any global variable that is read/written by main() and one or more ISRs must be annotated volatile
- Volatile indicates to the compiler that the variable might be modified by something external to the obvious flow of the program such as an ISR
- This ensures the compiler preserves the number of volatile reads and writes to the global variable exactly as written in C/C++ code. The compiler will not:
  - Eliminate redundant reads or writes
  - Re-order accesses

Table 2.12 illustrates the need for volatile when optimizations are enabled. Without the `volatile` qualifier on `flag`, the compiler will remove the `if` block in `main()` because its analysis indicates that `flag` is always 0 and the `if` condition is always false. `volatile` indicates to the compiler that something outside of `main()`, in this case the ISR, can update `flag`.

Table 5.1: Use of volatile

Main application	Interrupt Service Routine
`volatile int flag;` `int x;` `int main()` `{` `  flag = 0;` `  ...` `  if (flag == 1)` `  x++;` `  ...` `}`	`extern int flag;` `interrupt void ISR(void)` `{` `  ...` `  flag = 1;` `  ...` `}`

5.2 Peripheral access

- The volatile keyword must be used when accessing memory locations that represent memory mapped peripherals.
- Such memory locations might change value in ways that the compiler cannot predict.
- This ensures the compiler preserves reads and writes to memory exactly as written in the C code.
- A missing volatile qualifier can result in the compiler incorrectly optimizing away or re-ordering reads/writes.

Listing 5.1: Using volatile for peripheral register access

```
1   static inline void
2   GPIO_writePin(uint32_t pin, uint32_t outVal)
3   {
4       volatile uint32_t *gpioDataReg;
5       uint32_t pinMask;
6
7       //
8       // Check the arguments.
9       //
10      ASSERT(GPIO_isPinValid(pin));
11
12      gpioDataReg = (uint32_t *)GPIODATA_BASE +
```

(continues on next page)

(continued from previous page)

```
                        ((pin / 32U) * GPIO_DATA_REGS_STEP);

        pinMask = (uint32_t)1U << (pin % 32U);

        if(outVal == 0U)
        {
            gpioDataReg[GPIO_GPxCLEAR_INDEX] = pinMask;
        }
        else
        {
            gpioDataReg[GPIO_GPxSET_INDEX] = pinMask;
        }
    }
```

5.3 Atomic access

- 16-bit reads/writes are atomic.
- 32-bit float reads/writes are atomic except: writing a 32-bit float constant is only atomic if performed with a single opcode.
- 32-bit integer reads/writes:
- 32-bit reads/writes that use a single opcode are atomic.
- Atomic accesses within an ISR: By default, accesses within an ISR are atomic. The INTM bit is automatically set (disable interrupts) by the hardware during the context switch. The exception would be if the application re-enables interrupts within the ISR in order to nest interrupts.
- If possible, group atomic accesses together or create a function to perform the sequence disable-interrupts/atomic-accesses/enable-interrupts.
- For writes to global variables larger than 32 bits (64 bit long double, structures) disable/re-enable interrupts around the write. This ensures the writer updates the entire variable before the reader accesses it and avoids leaving the variable in an inconsistent or incomplete state.

For other atomic operations, there are two recommended approaches:

- Use an atomic compiler intrinsic if one is available. These are documented in the compiler user's guide (www.ti.com/lit/SPRU514). The description will say "in an atomic way".
- **Disable / enable interrupts around atomic operations using below intrinsics:**
  __disable_interrupts( ); __enable_interrupts( );

Listing 2.3 is a code snippet from the Digital Control Library in C2000Ware illustrating disabling interrupts around updates to a struct to ensure atomic updates to the entire structure. Refer to the

TMS320C28x Optimizing C/C++ Compiler User's Guide, Table 7-6, TMS320C28x C/C++ Compiler Intrinsics for details on the `__enable_interrupt()` and `__disable_interrupt()` intrinsics.

Listing 5.2: Disable interrupts to ensure atomic struct update

```
    uint16_t val = __disable_interrupts();

    p->Kp = p->sps->Kp;
    p->Ki = p->sps->Ki;
    p->Kd = p->sps->Kd;
    p->Kr = p->sps->Kr;
    p->c1 = p->sps->c1;
    p->c2 = p->sps->c2;
    p->Umax = p->sps->Umax;
    p->Umin = p->sps->Umin;
    DCL_restoreInts(v);

    // If interrupts were originally enabled, re-enable them
    if (0U == (val & 0x1))
        __enable_interrupts();
```

5.4 Calling asm functions from C code

Any ASM functions called from C code must follow the C calling and register conventions. Refer to the TMS320C28x Optimizing C/C++ Compiler User's Guide, Sections 7.2 Register Conventions, 7.3 Function Structure and Calling Conventions and 7.5 Interfacing C and C++ With Assembly Language.

Any violation of these conventions can result in application passing with -Ooff, but failing at higher optimization levels.

5.5 Uninitialized variables

- Using variables without initialization can lead to undefined behavior
- The behavior of an application with uninitialized variables can change with optimization levels, making debug difficult
- Local variables
  - Must be explicitly initialized in the application before any use
- Global variables

  - C standard specifies that global (extern) and static variables without explicit initializations must be initialized to 0 before the program begins running
  - C runtime initialization behavior differs across COFF and EABI
  - Refer to the TMS320C28x Optimizing C/C++ Compiler User's Guide for details - Sections 7.10.3 Automatic Initialization of Variables for COFF and 7.10.4 Automatic Initialization of Variables for EABI.

5.6 Interrupts

- RPT instructions are not interruptible, and can potentially delay or block interrupts from executing * For example, if there is a memcpy() instruction in a background function, and the compiler generates RPT instructions for this function, that section of code will be uninterruptible * If the compiler generates RPT instructions within an ISR, interrupts will be blocked, even if interrupt nesting is enabled * To avoid this issue, there are two compiler options available - –no_rpt which will tell the compiler not to generate RPT instructions, or –rpt_threshold which will limit the number of consecutive RPT instructions generated

CHAPTER

SIX

SUPPORT

To submit feedback on this guide or for C2000 related questions, please post to the C2000 microcontrollers forum in the TI E2E™ support forums.

CHAPTER

SEVEN

CHANGELOG

Table 7.1: Version History

Version	Date	Summary
v1.3	November 2023	Updated Interrupts and RPT instructions, Atomic access section
v1.2	April 2020	Added *Common issues with optimizations* section, updated *Profiling* section.
v1.1	March 2020	Added link for PDF download, updated Important Notice
v1.0	March 2020	Initial version of guide.

CHAPTER

EIGHT

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES "AS IS" AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI's products are provided subject to TI's Terms of Sale (https://www.ti.com/legal/termsofsale.html) or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI's provision of these resources does not expand or otherwise alter TI's applicable warranties or warranty disclaimers for TI products.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265

INDEX