Application Report
SLAA801–October 2017

Optimizing C Code for Size With MSP430™ MCUs: Tips and Tricks

Katie Pier *MSP430 Applications*

ABSTRACT

When choosing a microcontroller (MCU), the amount of code space or nonvolatile memory in the device is often a key consideration. To efficiently fit as much functionality as possible into a device, there are considerations that can be given when writing and building code to get the most optimized code size. This application note outlines a number of optimization settings for Code Composer Studio™ (CCS) and IAR Embedded Workbench® (IAR EW430) compilers that can make a big impact on code size, as well as coding techniques for user code to build with optimal size.

Contents

1 Introduction 2
2 C Compiler Optimization 2
2.1 CCS 2
2.2 IAR 6
3 Coding Techniques 9
3.1 Use Smallest Possible Types for Variables and Constants 9
3.2 Avoid Multiply and Divide 10
3.3 Use Lookup Tables Instead of Calculating 10
3.4 Use Word Accesses to Registers 10
3.5 Write to Registers Only Once (Where Possible) 10
3.6 Use the __even_in_range() Intrinsic 11
3.7 Use Functions Judiciously and Write for Reuse and Commonality 11
4 Summary 12
5 References 12

Trademarks

1 Introduction

Changing just a few key lines of code and a few compiler settings can make a big difference when it comes to code size. The code sizes used as an example in this application report are using the code example Msp430fr211x_euscia0_uart_03.c, which can be built for the MSP430FR2000 MCU, which contains only 0.5KB of FRAM. Code sizes listed were built with CCS version 7.3 with compiler 16.9.4.LTS or IAR EW430 version 7.10.4.

2 C Compiler Optimization

While working in C is typically preferred over assembly for its easy readability, writing in C can add some overhead that starts to become non-trivial when using a device with limited code space. However, through careful usage of compiler settings and features like global variables, programming in C can become close to assembly programming in efficiency (and still allowing for the compiler to do the heavy code optimization work rather than hand-optimizing assembly code).

No matter which IDE is used, the compiler already includes many tools for optimizing C code. Understanding the settings available and what they mean allows the user to work with the compiler to get the best code optimization results.

2.1 CCS

2.1.1 Optimization Settings

The main control of the compiler optimization is through the optimization settings accessible within the IDE. The settings allow the user to select how aggressive they want the compiler to apply optimization (which kinds of optimizations it is allowed to use) and the desired balance of optimizing for code size vs execution speed.

In CCS, the optimization settings are found in Project > Properties > Build > MSP430 Compiler > Optimization. There are two main optimization settings: Optimization level, and Speed vs size trade-offs (see Figure 1).

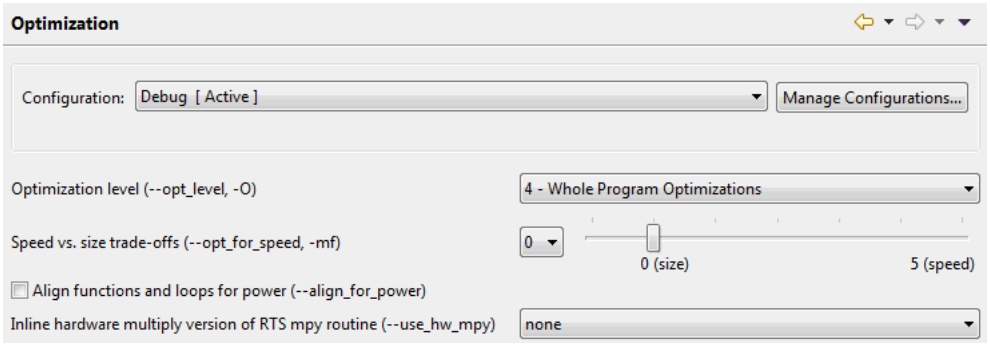


Figure 1. CCS Optimization Settings

The optimization level determines what types of optimizations the compiler is allowed to make. Speed vs size setting tells the compiler if trade-offs should be made more in the favor of size or speed, with 0 being optimizing with the most focus on size, and level 5 being the most focus on speed. Settings between 0 and 5 instruct the compiler to take a more balanced approach. For more information, see the *MSP430 Optimizing C/C++ Compiler User's Guide*.

CCS provides a tool called the Optimizer Assistant (View > Optimizer Assistant) that can be used to decide the optimal set of compiler settings for a particular project to fit in its target device (see Figure 2).

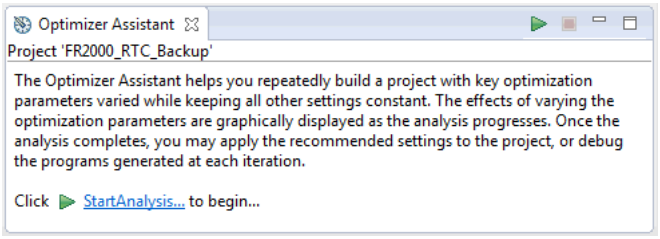


Figure 2. Optimizer Assistant

After clicking *Start Analysis*, select which build option to vary: speed vs size trade-offs or optimization level. The analysis then runs, varying the selected build option accordingly. The other build option uses whatever is currently in the project settings and remains constant through the test. For example, if the tool varies size vs speed setting, it uses whichever optimization level is currently selected in the project settings for all of the builds (see Figure 3).

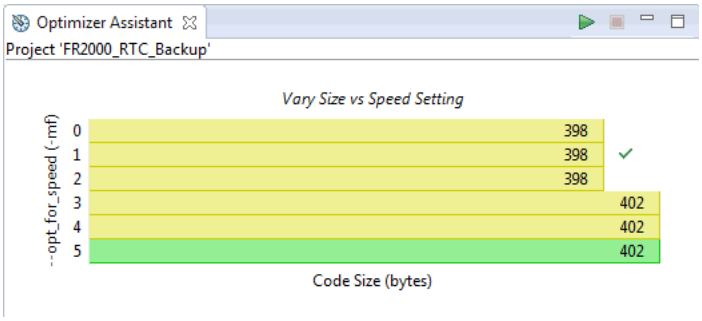


Figure 3. Varying Size vs Speed in Optimizer Assistant

The Optimizer Assistant displays the code size for the different settings. Red indicates if the code cannot fit in the selected device for the project, yellow indicates that the code fits the device but a better option is available, and green indicates the recommended option for best execution speed performance while still fitting into the target device memory. The check mark indicates the current selection in the project settings. For more information about using Optimizer Assistant, see http://processors.wiki.ti.com/index.php/Optimizer_Assistant

NOTE: After running the Optimizer Assistant, set the desired optimization settings in Project > Properties > Build > MSP430 Compiler > Optimization to apply them in subsequent builds.

2.1.2 Code and Data Model

MSP430™ microcontrollers have a 16-bit architecture. However, larger MSP430 devices have code space that extends to addresses 10000h and beyond, requiring 20 bits to store the full address. The MSP430X CPU architecture featured on these devices includes an extended instruction set to support operations on these 20-bit addresses. These extended instructions can take additional CPU cycles and increased program space due to requiring an extension word for double-operand instructions (see the appropriate family user's guide chapter on CPUX for more information on the extended instruction set). Therefore, on small memory devices where no addresses above 10000h exist in the device, it is important to ensure that only the base 16-bit instruction set is used to build with the smallest possible code size. This can be controlled by selecting the correct code and data model in the IDE project settings.

In CCS, go to Project > Properties > Build > MSP430 Compiler > Processor Options. Then select Small Code Memory Model and Small Data Memory Model (see Figure 4).

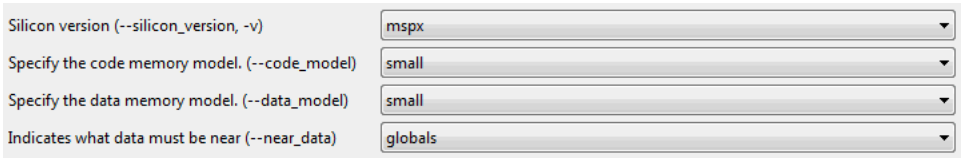


Figure 4. Code and Data Memory Model in CCS

The build takes several minutes the very first time after changing the code and data model, because the runtime support (RTS) library is built. Subsequent builds are much faster, because the library is not rebuilt. Changing code and data model can have a huge effect on the code size. Building the msp430fr211x_euscia0_uart_03.c with large code model and full optimization for code size builds to 928 bytes (see Figure 5).

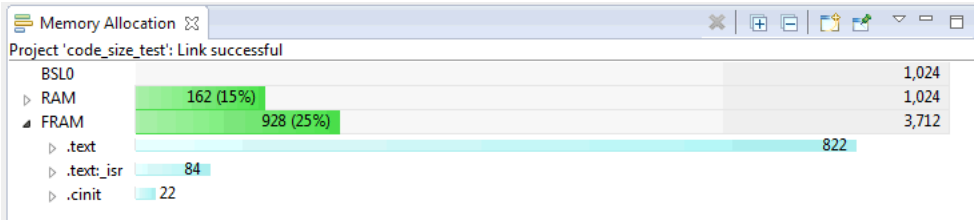


Figure 5. msp430fr211x_euscia0_uart_03.c With Large Memory Model

After changing the code and data model to small, the code size builds to 572 bytes - a 38% reduction (see Figure 6).

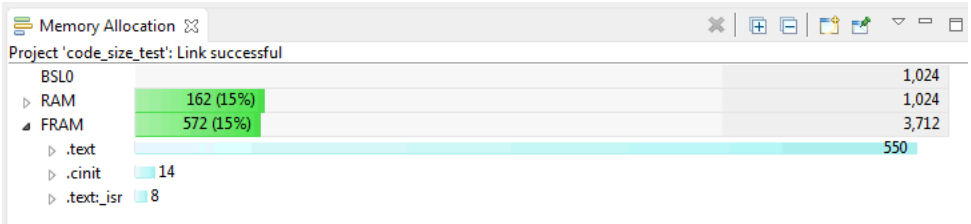


Figure 6. msp430fr211x_euscia0_uart_03.c With Small Memory Model

2.1.3 Global Variables and Initialization Settings

The C-compiler inserts some initialization code from the runtime support (RTS) library that runs every time the device starts up. This code does prepares the C environment by setting up the stack and initializing variables in RAM. The initialization routines used by the C-compiler by default use a compressed table of global variable initialization data, which makes sense for large projects with large numbers of global variables, large arrays, and other data that need to be initialized in the RAM at device start-up. However, for small devices that contain simple code containing a small number of global variables, this code is no longer space efficient. The crossover point is where the C-initialization code takes up more space than directly initializing the variables with user code (not using compression).

To understand how much space the C-start-up code is taking up in the part, look at the .map file (found in the Debug folder after building). Under the *Section Allocation Map* portion of the linker file, find ".text". Here, you can see the start address and length for all functions in your project, including functions inserted by the compiler from the RTS library. Functions from the RTS library are all marked as from rts430x_xx_xx_eabi.lib (the xx differs depending on code and data memory model selection). The boxed functions in Figure 7 are all used for global variable initialization – copy tables and decompression code including multiplication functions. These functions require significant code space if working with a very code-limited part, such as the MSP430FR2000 or MSP430G2001 MCU with only 512 bytes of FRAM or flash.

```
.text      0    0000f10e    00000226
                0000f10e    0000007e    rts430x_sc_sd_eabi.lib : autoinit_wdt.obj (.text:_auto_init_hold_wdt)
                0000f18c    0000006e                           : copy_decompress_lzss.obj (.text:decompress:lzss:__TI_decompress_lzss)
                0000f1fa    00000058                           : cpy_tbl.obj (.text:copy_in)
                0000f252    00000054    msp430fr211x_euscia0_uart_03.obj (.text:main)
                0000f2a6    00000032    msp430fr211x_euscia0_uart_03.obj (.text:USCI_A0_ISR)
                0000f2d8    0000001a    rts430x_sc_sd_eabi.lib : boot_special.obj (.text:_c_int00_noargs_noexit)
                0000f2f2    00000014                           : mult16.obj (.text)
                0000f306    00000012                           : copy_decompress_none.obj (.text:decompress:none:__TI_decompress_none)
                0000f318    00000010                           : memcpy.obj (.text:memcpy)
                0000f328    00000006                           : exit.obj (.text:abort)
                0000f32e    00000004                           : pre_init.obj (.text:_system_pre_init)
                0000f332    00000002                           : startup.obj (.text:_system_post_cinit)
```

Figure 7. Default Global Variable Initialization in CCS

Therefore, for small devices in which code size is a concern, the following methods should be employed to control the initialization of global variables.

Option 1: If it is possible in the application, simply eliminate use of global variables. This eliminates the overhead for initializing global variables at start-up.

Option 2: Use a limited number of global variables. Set up these variables such that they are not preinitialized by the RTS library, using compiler settings. Instead, initialize the global variables in main() with user code. Section 2.1.3.1 includes steps to address this.

2.1.3.1 Controlling Global Variable Initialization

Controlling global variable initialization to save code space involves both modifications to user code and using some compiler settings available in the IDE to tell the compiler not to automatically initialize the variables.

First, move the initialization of the global variables into main(). In the example msp430fr211x_euscia0_uart_03.c, there are only two global variables: RXData and TXData, that are both initialized. Figure 8 shows an example of moving the initialization for these variables to main().

```
#include <msp430.h>

//unsigned char RXData = 0;
//unsigned char TXData = 1;
unsigned char RXData;
unsigned char TXData;

int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;                 // Stop watchdog timer

    //Initialize globals
    RXData = 0;
    TXData = 1;
```

Figure 8. Moving Global Variable Initialization to main() in CCS

From our optimization settings and small data model usage before, the code built at 572 bytes. By simply moving the initialization into main, the code size shrinks to 468 bytes, or an 18% reduction in code size.

Looking at the .map file, observe how some of the RTS library functions have been eliminated. However, others like copy_zero_init have been added (and some others like mult16 and memcpy are still there) (see Figure 9).

```
.text      0    0000f10a    000001c2
                0000f10a    0000007e    rts430x_sc_sd_eabi.lib : autoinit_wdt.obj (.text:_auto_init_hold_wdt)
                0000f188    0000005c    msp430fr211x_euscia0_uart_03.obj (.text:main)
                0000f1e4    00000058    rts430x_sc_sd_eabi.lib : cpy_tbl.obj (.text:copy_in)
                0000f23c    00000032    msp430fr211x_euscia0_uart_03.obj (.text:USCI_A0_ISR)
                0000f26e    0000001a    rts430x_sc_sd_eabi.lib : boot_special.obj (.text:_c_int00_noargs_noexit)
                0000f288    00000014                           : copy_zero_init.obj (.text:decompress:ZI:__TI_zero_init)
                0000f29c    00000014                           : mult16.obj (.text)
                0000f2b0    00000010                           : memcpy.obj (.text:memcpy)
                0000f2c0    00000006                           : exit.obj (.text:abort)
                0000f2c6    00000004                           : pre_init.obj (.text:_system_pre_init)
                0000f2ca    00000002                           : startup.obj (.text:_system_post_cinit)
```

Figure 9. RTC Library Global Variable Zero-Initialization

By default, projects built in the EABI format automatically initialize any uninitialized global variables to 0. This is to protect users from using a variable before it has a real value in it, which would cause a read of random values from RAM. However, as long as all global variables are initialized before usage, this problem does not occur. Because the code performs this initialization in main, zero-initialization can be turned off. To disable zero-initialization in CCS, click Project > Properties > MSP430 Linker > Advanced Options > Miscellaneous, and set the *Zero initialize ELF uninitialized sections* option to *Off* (see Figure 10).

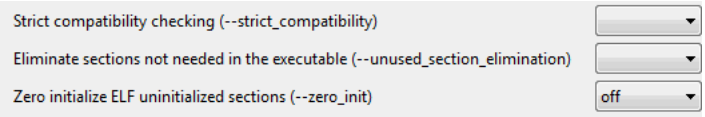


Figure 10. Disable Zero-Initialization in CCS

Before disabling zero-initialization, the code built at 468 bytes. After disabling zero-initialization, the code builds at 178 bytes – **a huge 62% reduction, saving more than half of the memory size of a 512 byte device!**

2.2 IAR

2.2.1 Optimization Settings

The main control of the compiler optimization is through the optimization settings accessible within the IDE. The settings allow the user to select how aggressively the compiler should apply optimization (which kinds of optimizations it is allowed to use) and the desired balance of optimizing for code size vs execution speed.

In IAR, the optimization settings are found in the Project > Options > C/C++ Compiler > Optimizations tab. There are two main optimization settings: optimization level, and speed vs size trade-offs (see Figure 11).

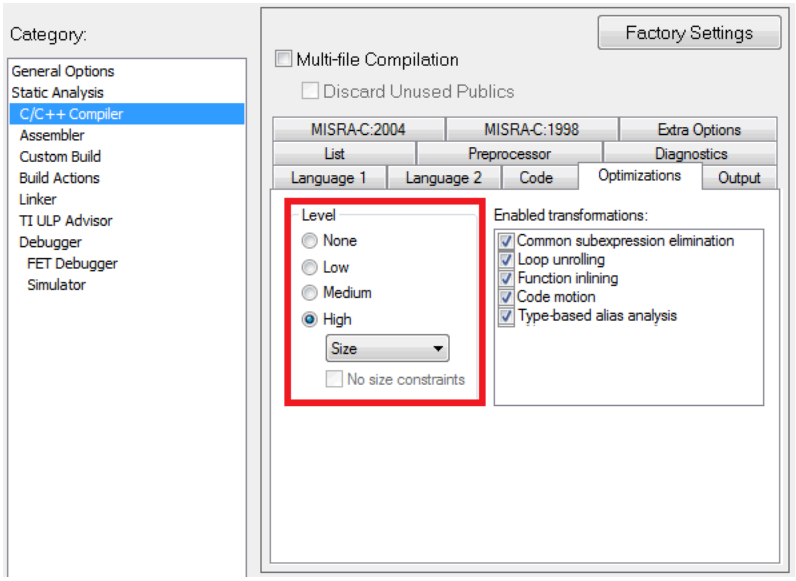


Figure 11. Optimization Settings in IAR

The optimization level determines what types of optimizations the compiler is allowed to make. Speed vs size setting tells the compiler if trade-offs should be made more in the favor of size or speed, with speed, size, and balanced options. There are also options for what types of optimizations to allow under *enabled transformations*. For more information, see the *IAR C/C++ Compiler User's Guide* included with IAR under Help.

2.2.2 Code and Data Model

MSP430 microcontrollers have a 16-bit architecture. However, larger MSP430 devices have code space that extends to addresses 10000h and beyond, requiring 20 bits to store the full address. The MSP430X CPU architecture featured on these devices includes an extended instruction set to support operations on these 20-bit addresses. These extended instructions can take additional CPU cycles and increased program space due to requiring and extension word for double-operand instructions (see the appropriate family user's guide chapter on CPUX for more information on the extended instruction set). Therefore, on small memory devices where no addresses above 10000h exist in the device, it is important to ensure that only the base 16-bit instruction set is used to build with the smallest possible code size. This can be controlled by selecting the correct code and data model in the IDE project settings.

In IAR, go to Project > Options > General Options > Target tab. Then select Small for Code Model and Small for Data Model (see Figure 12).

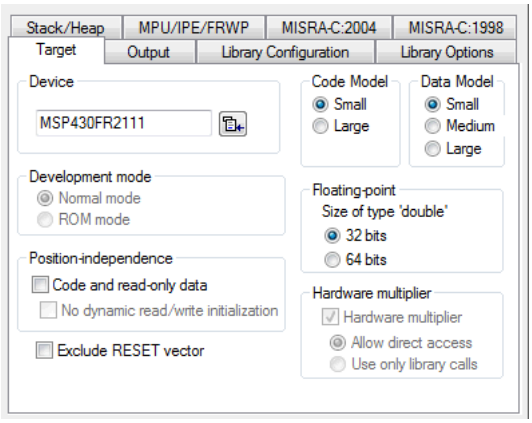


Figure 12. Small Code Model in IAR

Changing code and data model can have an effect on the code size in some cases. Often, the IAR compiler already accounts for this, but it is still good practice if no upper memory is available or needed. The example msp430fr211x_euscia0_uart_03.c builds to 239 bytes in IAR for both large or small code model, but for some devices or code there can be a difference.

2.2.3 Global Variables and Initialization Settings

The C-compiler inserts initialization code from the support library that runs every time the device starts up. This code prepares the C environment by setting up the stack and initializing variables in RAM. The initialization routines used by the C-compiler by default use initialization methods that make sense for large projects or for the general case. However, for very small devices that contain simple code containing a small number of global variables, this code is not always the most space efficient. The crossover point is where the C-initialization code takes up more space than directly initializing the variables with user code.

To understand how much space the C-start-up code uses, look at the .map file (found in the Output folder after building). Open Project > Options > Linker > List and check *Module Summary*. Then in the .map file, under the *Module Summary* portion near the end of the linker file, see the modules listed. This section lists he start address and length for the code and the functions inserted by the compiler from the library. Functions from the library are all marked starting with ?. The boxed functions in Figure 13 are all used for global variable initialization – memcpy, zero-initialization of memory, C-start-up environment initialization, and exit routines. These functions require significant code space if working with a very code-limited part, such as the MSP430FR2000 or MSP430G2001 MCU with only 512 bytes of FRAM or flash.

```
                ****************************************
                *                                      *
                *            MODULE SUMMARY            *
                *                                      *
                ****************************************

Module                          CODE      DATA      CONST
------                          ----      ----      -----
                               (Rel)   (Rel)   (Abs)   (Rel)
?__dbg_break                      2
?__exit                          20
?_exit                            4
?cstart                          40
?exit                             4
?memcpy                          18
?memzero                         20
?reset_vector                     2
msp430fr211x_euscia0_uart_03    130       2      26       1
  + common                      106
N/A (command line)                        160
-----------                     ---     ---     --       -
Total:                          240     162      26       1
  + common                      106
```

Figure 13. Default Global Variable Initialization in IAR

Therefore, for small devices where code size is a concern, the following methods should be employed to control the initialization of global variables.

Option 1: If it is possible in the application, simply eliminate use of global variables. This eliminates the overhead for initializing global variables at start-up.

Option 2: Use a limited number of global variables. Set up these variables such that they are not preinitialized by the library, using compiler settings. Instead, initialize the global variables in main() with user code. Section 2.2.3.1 includes steps to address this.

2.2.3.1 Controlling Global Variable Initialization

Controlling global variable initialization to save code space involves both modifications to user code, as well as using some compiler settings available in the IDE to tell the compiler not to automatically initialize the variables.

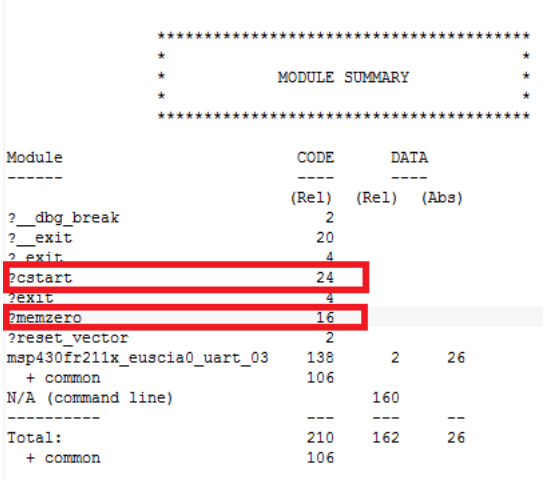
First, move the initialization of the global variables into main(). In the example msp430fr211x_euscia0_uart_03.c, there are only two global variables: RXData and TXData, that are both initialized. Figure 14 shows an example of moving the initialization for these variables to main().

```
#include <msp430.h>

//unsigned char RXData = 0;
//unsigned char TXData = 1;
unsigned char RXData;
unsigned char TXData;

int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;                // Stop watchdog timer

    //Initialize globals
    RXData = 0;
    TXData = 1;
```

Figure 14. Moving Global Variable Initialization to main() in IAR

From the optimization settings and small data model usage before, the code built at 240 bytes. By simply moving the initialization into main, the code size shrinks to 210 bytes, or 12.5% reduction in code size.

In the .map file (see Figure 15), observe how some of the library functions (for example, memcpy) have been eliminated. However, others like memzero are still there. Further, the cstart module has become smaller.



```
                ****************************************
                *                                      *
                *            MODULE SUMMARY            *
                *                                      *
                ****************************************

Module                          CODE       DATA
------                          ----       ----
                               (Rel)   (Rel)   (Abs)
?__dbg_break                      2
?__exit                          20
?_exit                            4
?cstart                          24
?exit                             4
?memzero                         16
?reset_vector                     2
msp430fr211x_euscia0_uart_03    138       2     26
  + common                      106
N/A (command line)                      160
----------                      ---     ---     --
Total:                          210     162     26
  + common                      106
```

Figure 15. Global Variable Zero-initialization in IAR

By default, projects built in the EABI format automatically initialize any uninitialized global variables to 0. This is to protect users from using a variable before it has a real value in it, which would read random values from RAM. However, as long as all global variables are initialized before usage, this problem does not occur. Because the code performs this initialization in main, zero-initialization can be turned off. In IAR, this is done by using the __no_init keyword when declaring global variables (see Figure 16).

```
#include <msp430.h>

//unsigned char RXData = 0;
//unsigned char TXData = 1;
__no_init unsigned char RXData;
__no_init unsigned char TXData;
```

Figure 16. Setting Global Variables as No-Init in IAR

Before disabling zero-initialization, the code built at 210 bytes. After disabling zero-initialization, the code builds at 182 bytes – a further 13% reduction in code size.

3 Coding Techniques

While the C compiler can do a lot to optimize code for size, there are also things the user can do when writing code to help ensure smaller build size. Some things are essentially hand optimizations, while others provide hints to the compiler about properties of the code and ways that it is then further allowed to safely optimize.

3.1 Use Smallest Possible Types for Variables and Constants

Constants are stored in nonvolatile memory, just like code. Therefore, using the smallest possible type when defining constants helps to reduce wasted code space. For example, if a lookup table contains only values between 0 and 255, using an unsigned 8-bit type when declaring the constant table reduces the space by half compared to a 16-bit int type. This concept also applies to variables stored in FRAM using the PERSISTENT keyword. If a variable is stored in FRAM instead of RAM, it uses FRAM space that could otherwise be used as code. Therefore, using the smallest type is important in this case as well.

Code can even be written such that smaller numbers are used for the lookup table, provided that the precision is still sufficient for the application. One example is a constant array containing values for timer PWM output. If the timer is sourced from 32768 Hz, but the timer output is only 60 Hz, the highest count for the timer period or duty cycle could be only 32768 / 60 = 545. This value cannot be stored in an 8-bit variable. But if the timer source is divided by 4 using the internal clock dividers in the clock module or in the timer, the highest count for duty cycle would now be 8192 / 60 = 136, which is small enough to store in an 8-bit value. Providing that this still provides for enough precision in setting duty cycle, making this simple change halves the size of a const lookup table containing timer count settings. Intelligently choosing how values are stored can make a big difference in the size of arrays and lookup tables.

3.2 Avoid Multiply and Divide

Multiply and divide operations take many cycles to perform and and require more code to enable these operations. Therefore, finding ways to remove unnecessary multiplication or division from code can be a great way to save on both code space and execution time.

For multiplication or division by powers of 2 (for example, 2, 4, 8, 16, …), bit shifts can be used instead. To do a bit shift in C, use >> to right shift and << to left shift, then the number of bits to shift. This is much more efficient than a multiply or divide because there are assembly instructions and hardware in the CPU for bit shifts.

Another option is to determine if multiplications or divisions truly need to occur at runtime or not. For example, if multiplication is used on two constants with no variable, then this calculation could be done ahead and the result used instead of having multiplication or division code that runs on the device every time, wasting space and execution time.

Even if a variable is part of the multiplication so it cannot all be calculated ahead, if that variable has a known range of values, a lookup table could be created to contain the possible results. A lookup table does not always build smaller, because the table must also reside in the nonvolatile memory of the device, so code should be built both ways to analyze which is the best for code size.

3.3 Use Lookup Tables Instead of Calculating

If a complex calculation must be repeatedly performed for different values, consider whether a lookup table may be a better alternative – this can be especially true for floating point calculations. If the variable input to the calculation has a known range of values, a lookup table can be created to contain the possible results. A lookup table does not always build smaller than using MSPMATHLIB, because the table must also reside in the nonvolatile memory of the device, so code should be built both ways to analyze which is the best for code size.

3.4 Use Word Accesses to Registers

Some registers in MSP430 devices have both byte and word versions. If the setting is allowed to be set at the same time instead of sequentially (that is, if the second write does not require the first write to happen first), then write both bytes at the same time by using the word form of the register.

A good example is port initialization. Instead of writing:

```
P1OUT = BIT0;
P2OUT = BIT7;
```

The following code can be used:

```
PAOUT = BIT15 | BIT0;
```

This sets both P1OUT and P2OUT at once instead of generating separate code for two separate sequential writes. This can save both code space and execution time.

Note how the BITx used for the upper byte needs to be adjusted so make sure to use the mnemonic for the 16-bit version of any bits for the upper byte of the word. For most registers that have both byte or word versions, byte-access versions of bits are denoted with _H or _L at the end, versus word-access versions of bit mnemonics not having this. Other common examples of using word instead of byte registers is using UCAxBRW instead of UCAxBR0 and UCAxBR1, or RTCCTL13 instead of RTCCTL1 and RTCCTL3.

3.5 Write to Registers Only Once (Where Possible)

Similarly to using word accesses to registers, combine multiple writes to the same register into a single instruction when possible. Identifying multiple writes to a register (especially when initializing a module) and combining them into one write can reduce the number of instructions for the application. Some code and some modules require certain bits to be set or cleared before they can (or should) be modified – for example, ADC12CTL0 bits that require ADC12ENC to be cleared before modification. However, in many cases, all of the bits of a register can be written at once without any logical problem.

Sometimes code clears bits and then sets different bits in the same register using two separate bit-wise register accesses:

```
TA0CCTL2 &= ~OUTMOD_7;
TA0CCTL2 |= OUTMOD_4;
```

But instead these could potentially be done as a single write with a full register write instead:

```
TA0CCTL2 = OUTMOD_4 | CCIE;
```

When using =, any other bits in the register that need to remain set should also be written; for example in this case, CCIE. Careful consideration should always be taken for what bits could be inadvertently cleared or changed when using = instead of bit-wise operations. Consult the device family user's guide for default register settings and be mindful of their reset state.

3.6 Use the __even_in_range() Intrinsic

The __even_in_range(x,NUM) intrinsic provides a hint to the compiler for switch statements that the value x will always be an even value in the range 0 to NUM inclusive. This allows the compiler to make more assumptions about the value of x and optimize further than it normally could. This is typically used for interrupt service routines (ISRs) because interrupt vectors always have a fixed range of values and are always even (see Figure 17).

```
#pragma vector=USCI_A0_VECTOR
__interrupt void USCI_A0_ISR(void)
{
    switch(__even_in_range(UCA0IV,USCI_UART_UCTXCPTIFG))
    {
        case USCI_NONE: break;
        case USCI_UART_UCRXIFG:
              UCA0IFG &=~ UCRXIFG;            // Clear interrupt
              RXData = UCA0RXBUF;             // Clear buffer
              if(RXData != TXData)            // Check value
              {
                  P1OUT |= BIT0;              // If incorrect turn on P1.0
                  while(1);                   // trap CPU
              }
              TXData++;                              // increment data byte
              __bic_SR_register_on_exit(LPM0_bits); // Exit LPM0 on reti
              break;
        case USCI_UART_UCTXIFG: break;
        case USCI_UART_UCSTTIFG: break;
        case USCI_UART_UCTXCPTIFG: break;
    }
}
```

Figure 17. Example ISR Using the __even_in_range() Intrinsic

__even_in_range() is not limited to ISRs, however. The intrinsic could also be used on other switch statements in code, as long as the code ensures that the control variable for the switch is always even and has an upper limit on its value. An example could be a byte counter variable that increments by 2 instead of by 1, and has a maximum value controlled by code so that it rolls over to 0 or 2.

For more information on the __even_in_range() intrinsic, see *MSP430 Optimizing C/C++ Compiler User's Guide* .

3.7 Use Functions Judiciously and Write for Reuse and Commonality

Functions can make code much easier to read. However, functions can also add overhead in terms of execution speed and code space. If a function is only called once, especially if it is a small function, the extra overhead is typically not worth it. You can keep a small function for readability but not incur the extra overhead by inlining the function using the inline (or __inline) keyword with the function declaration. This declaration instructs the compiler to insert the contents of the function everywhere it is called, instead of doing an actual function call. More information on inlining functions is available in *MSP430 Optimizing C/C++ Compiler User's Guide* .

However, there are also cases where something needs to be done several times in the code. In this case, a function can make for a more code-size efficient solution, because it allows the same code to be reused in several different places. Writing code intentionally for reusability can make a big difference on code size as well.

4 Summary

Experimentation is often the best method of determining the most optimized coding solution. Using version control software and creating different branches or versions, or making copies or variants of a project, allows code to be built with different optimization strategies as outlined in this report. The results can then be compared against each other to find the optimal solution.

5 References

1. *MSP430 Optimizing C/C++ Compiler User's Guide*
2. *MSP430FR21xx, MSP430FR2000 Code Examples*

IMPORTANT NOTICE FOR TI DESIGN INFORMATION AND RESOURCES

Texas Instruments Incorporated ('TI") technical, application or other design advice, services or information, including, but not limited to, reference designs and materials relating to evaluation modules, (collectively, "TI Resources") are intended to assist designers who are developing applications that incorporate TI products; by downloading, accessing or using any particular TI Resource in any way, you (individually or, if you are acting on behalf of a company, your company) agree to use it solely for this purpose and subject to the terms of this Notice.

TI's provision of TI Resources does not expand or otherwise alter TI's applicable published warranties or warranty disclaimers for TI products, and no additional obligations or liabilities arise from TI providing such TI Resources. TI reserves the right to make corrections, enhancements, improvements and other changes to its TI Resources.

You understand and agree that you remain responsible for using your independent analysis, evaluation and judgment in designing your applications and that you have full and exclusive responsibility to assure the safety of your applications and compliance of your applications (and of all TI products used in or for your applications) with all applicable regulations, laws and other applicable requirements. You represent that, with respect to your applications, you have all the necessary expertise to create and implement safeguards that (1) anticipate dangerous consequences of failures, (2) monitor failures and their consequences, and (3) lessen the likelihood of failures that might cause harm and take appropriate actions. You agree that prior to using or distributing any applications that include TI products, you will thoroughly test such applications and the functionality of such TI products as used in such applications. TI has not conducted any testing other than that specifically described in the published documentation for a particular TI Resource.

You are authorized to use, copy and modify any individual TI Resource only in connection with the development of applications that include the TI product(s) identified in such TI Resource. NO OTHER LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE TO ANY OTHER TI INTELLECTUAL PROPERTY RIGHT, AND NO LICENSE TO ANY TECHNOLOGY OR INTELLECTUAL PROPERTY RIGHT OF TI OR ANY THIRD PARTY IS GRANTED HEREIN, including but not limited to any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information regarding or referencing third-party products or services does not constitute a license to use such products or services, or a warranty or endorsement thereof. Use of TI Resources may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

TI RESOURCES ARE PROVIDED "AS IS" AND WITH ALL FAULTS. TI DISCLAIMS ALL OTHER WARRANTIES OR REPRESENTATIONS, EXPRESS OR IMPLIED, REGARDING TI RESOURCES OR USE THEREOF, INCLUDING BUT NOT LIMITED TO ACCURACY OR COMPLETENESS, TITLE, ANY EPIDEMIC FAILURE WARRANTY AND ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT OF ANY THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

TI SHALL NOT BE LIABLE FOR AND SHALL NOT DEFEND OR INDEMNIFY YOU AGAINST ANY CLAIM, INCLUDING BUT NOT LIMITED TO ANY INFRINGEMENT CLAIM THAT RELATES TO OR IS BASED ON ANY COMBINATION OF PRODUCTS EVEN IF DESCRIBED IN TI RESOURCES OR OTHERWISE. IN NO EVENT SHALL TI BE LIABLE FOR ANY ACTUAL, DIRECT, SPECIAL, COLLATERAL, INDIRECT, PUNITIVE, INCIDENTAL, CONSEQUENTIAL OR EXEMPLARY DAMAGES IN CONNECTION WITH OR ARISING OUT OF TI RESOURCES OR USE THEREOF, AND REGARDLESS OF WHETHER TI HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You agree to fully indemnify TI and its representatives against any damages, costs, losses, and/or liabilities arising out of your non-compliance with the terms and provisions of this Notice.

This Notice applies to TI Resources. Additional terms apply to the use and purchase of certain types of materials, TI products and services. These include; without limitation, TI's standard terms for semiconductor products http://www.ti.com/sc/docs/stdterms.htm), evaluation modules, and samples (http://www.ti.com/sc/docs/sampterms.htm).

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