Please note that Cypress is an Infineon Technologies Company.
The document following this cover page is marked as “Cypress” document as this is the company that originally developed the product. Please note that Infineon will continue to offer the product to new and existing customers as part of the Infineon product portfolio.

Continuity of document content
The fact that Infineon offers the following product as part of the Infineon product portfolio does not lead to any changes to this document. Future revisions will occur when appropriate, and any changes will be set out on the document history page.

Continuity of ordering part numbers
Infineon continues to support existing part numbers. Please continue to use the ordering part numbers listed in the datasheet for ordering.

AN60486

PSoC® 1 M8C ImageCraft C Code Optimization

Author: Archana Yarlagadda
Associated Project: No
Associated Part Family: CY8C2xxxx (All PSoC 1 devices)
Software Version: PSoC Designer™ 5.4
Related Application Notes: AN75320, AN60630, AN2017

If you have a question, or need help with this application note, contact msur@cypress.com.

AN60486 shows you how to optimize PSoC 1 C code to be faster and smaller and covers PSoC Designer project and ImageCraft compiler settings that help you to do so. In addition, it offers several guidelines for efficient coding. This application note assumes that you are familiar with PSoC 1, the PSoC Designer Integrated Design Environment (IDE), and programming in C. For introductory information, refer to AN75320 – Getting Started with PSoC 1.

Contents

Introduction 1
Project-Level Optimization 2
- Setting 1: Relocatable Code Start Address 2
- Setting 2: Configuration Initialization 3
- Setting 3: Sublimation and Condensation 4
- Setting 4: Treat const as ROM Versus Treat const as RAM 6

ImageCraft Pro Compiler Options 6
Tips and Guidelines.....6
- Guideline 1: Avoiding Function Calls in Interrupt Service Routines 6
- Guideline 2: Limiting Math Functions 8
- Guideline 3: Using Array Indexing Versus Pointer.....10
- Guideline 4: Using Switch Statement Versus If-Else Statement.....11
- Guideline 5: Writing Part of Code in Assembler 12
- Guideline 6: Manipulating Bits in PSoC 1.....12
- Guideline 7: Calculating C Code Flash Usage and Execution Time.....13

Conclusion 15

Introduction

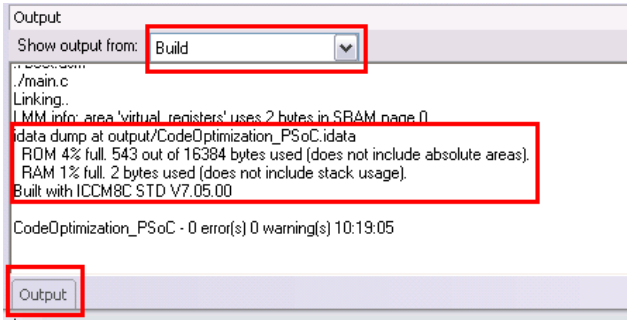
There are several ways to reduce the amount of flash (ROM) used in a PSoC Designer project. This can sometimes allow you to use a smaller PSoC device, which reduces cost.

The ImageCraft compiler Standard version is included with PSoC Designer. You can also buy the Pro version, which offers more optimization, on the ImageCraft website. The code snippets included in this application note can be used with both compilers. The examples shown were compiled with the Standard version; the additional optimizations that can be obtained using the Pro version are also covered.

To use the techniques explained in this document, you should know how to develop and build PSoC Designer projects and have a basic knowledge of C programming. Refer to AN75320 to get started with PSoC 1.

While developing a PSoC Designer project, if you want to find out how much ROM space it is using, look at the **Build** tab in the **Output** status window, as shown in Figure 1. The code space used in this example is 543 bytes.

Figure 1. PSoC Designer Build Message Showing ROM and RAM Usage



The following sections discuss project-level optimization settings, followed by some coding guidelines.

Project-Level Optimization

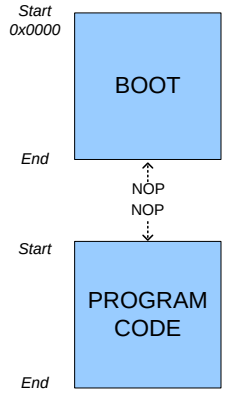
Several PSoC Designer code optimization settings are available at **Project > Settings**.

Setting 1: Relocatable Code Start Address

When a PSoC Designer project is built, the ImageCraft compiler converts the C files into assembly files. The assembler then converts them into relocatable object files. Finally, the linker combines the relocatable object files to produce the executable *.hex* file.

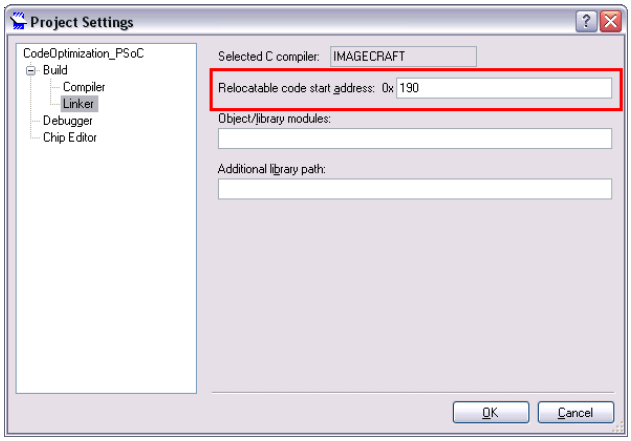
In addition to your code, the *.hex* file includes auto-generated initialization code (boot code). This is followed by a set of NOP instructions that act to reserve some flash bytes for possible expansion of boot code. See Figure 2. You can save flash by forcing your code to be located immediately after the boot code.

Figure 2. Flash Map for PSoC Designer Project



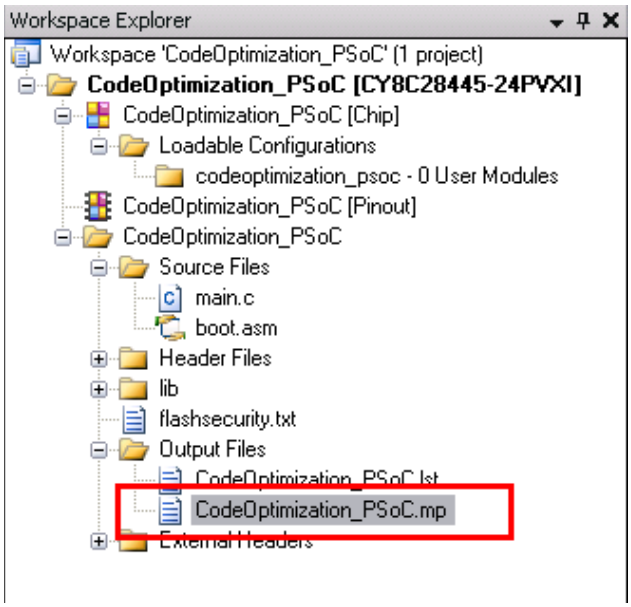
PSoC Designer lets you specify the start address of your portion of the code by choosing **Project > Settings**. In the window that pops up, select “Linker,” as shown in Figure 3.

Figure 3. Relocatable Start Code Address Selection



The address you place in the text box is based on the size of the boot code, which you can find in the map file (*.mp*), as shown in Figure 4.

Figure 4. Map File in PSoC Designer



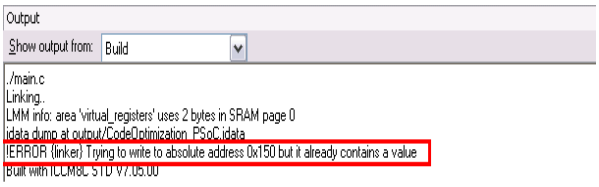
The map file shows the start and end addresses of different areas of code, as shown in Figure 5. The boot code is in area “TOP,” and your code is in area “lit.” You can set the **Relocatable code start address** in Figure 3 to the end address of the “TOP” area to use the flash space most efficiently. For example, the default setting is 0x190 (start of “lit” area). If you change it to 0x151 (end of “TOP” area), you can gain 63 (0x190 – 0x151 = 0x3F) more bytes in ROM.

Figure 5. “lit” and “TOP” Area in *.mp* File

```
Project:      output/CodeOptimization_PSoC
Build Number: 0
Date:         Mon Mar 19 10:19:03 2012
Compiler:     ICCM8C
Version:      7.05.00

Area                              Addr   Size   Decimal Bytes (Attributes)
--------------------------------  ----   ----   ------- ----- ------------
                                  lit    0190   00F2 =    242. bytes (rel,con,rom,lit)

      Addr  Global Symbol
     -----  --------------------------------
      0190  LoadConfigTBL_codeoptimization_psoc_Bank0
      01F3  LoadConfigTBL_codeoptimization_psoc_Bank1
      0282  __lit_end

Area                              Start  End    Decimal Bytes (Attributes)
--------------------------------  ----   ----   ------- ----- ------------
                                  TOP    0000   0151 =    337. bytes (abs,ovr,rom,code)

      Addr  Global Symbol
     -----  --------------------------------
      0080  __Start
      014E  _bGetPowerSetting
      014E  bGetPowerSetting
```

If the relocatable code start address is set to a value less than the end of “TOP,” an error message is displayed. For example, if the value is set to 0x150 and the boot code ends at 0x151, then an error will be displayed, as Figure 6 shows.

Figure 6. Error When Relocatable Code Address Is Less Than Boot Code Size



Setting 2: Configuration Initialization

Another option for code optimization is controlling how PSoC registers are initialized at startup. Register values can be set using two methods: **Loop** or **Direct write**, as shown in Figure 7. This option is available by choosing **Project > Settings** and then selecting “Chip Editor” in the window that appears.

With the **Loop** method, a table with register addresses and values is created, and a function writes the values in the table to the respective addresses. With the **Direct write** method, register writes are done directly, with one MOV instruction for each register.

The selected method is coded in the auto-generated file *PSOCConfigTBL.asm*. The code for the two methods is shown in Figure 8.

Figure 7. Configuration Initialization Selection

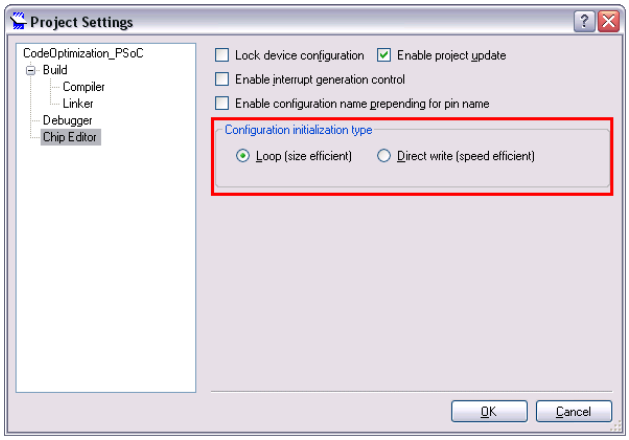


Figure 8. Code Difference Between Loop and Direct Write

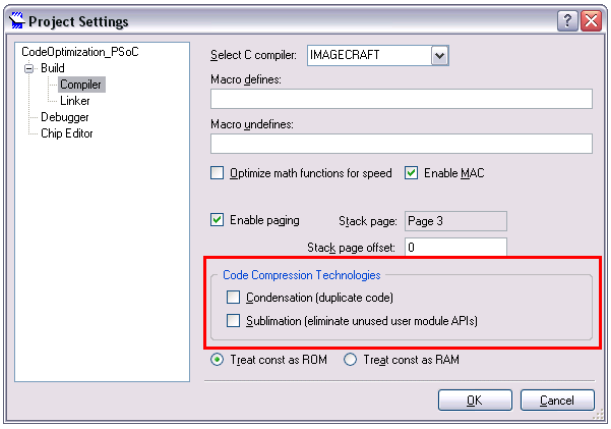


In Figure 8 you can see that for each register write, the **Direct write** method uses one more byte (MOV reg[expr], expr = 3 bytes) than the **Loop** method uses (addr, value = 2 bytes). But the **Loop** method also includes the table read function, which uses 94 bytes. Therefore, if there are more than 94 registers to be loaded (this is usually the case in all but the simplest PSoC Designer projects), the **Loop** selection uses less flash. The number of registers initialized using the configuration table shown in Figure 8 depends on the number of resources (user modules) used in the design. The greater the number, the more registers will be initialized. So, if the design includes many user modules to save flash and you are willing to sacrifice some time at startup, use the **Loop** method. But if you do not have many user modules in the design or desire a faster device startup, select the **Direct write** method.

Setting 3: Sublimation and Condensation

Sublimation and condensation are compression techniques provided by the ImageCraft Standard version. These can be set by choosing **Project > Settings** and then selecting “Compiler” in the pop-up window, as shown in Figure 9.

Figure 9. Sublimation and Condensation Selection



Sublimation

When you select the **Sublimation** option, the compiler deletes the unused functions in the user module APIs (interface code), which saves space.

For example, you can place a PGA and a PWM user module in a project and call only the “Start” function for both user modules. As shown in Figure 10 and Figure 11, you can save 142 bytes of flash by eliminating the other unused functions in those user module APIs.

Figure 10. ROM Usage Without Sublimation

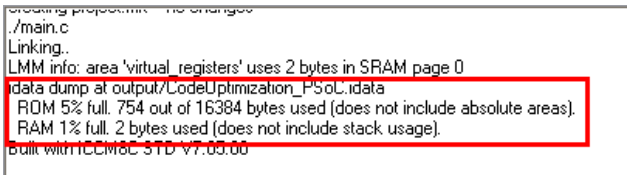
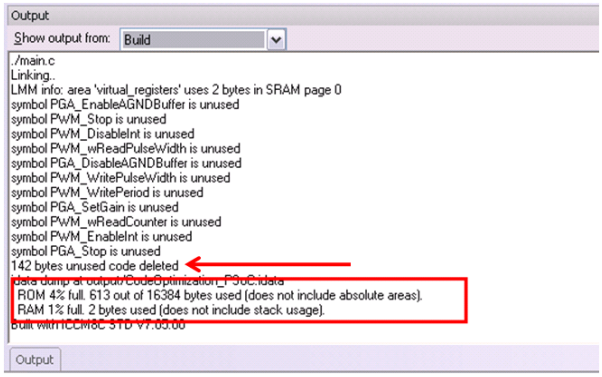
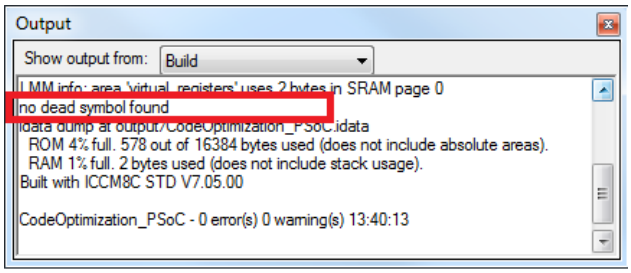


Figure 11. ROM Usage with Sublimation



If all functions in all user module APIs are used in a project, then a “no dead symbol found” message is displayed, as Figure 12 shows.

Figure 12. Sublimation with “no dead symbol found” Message



Condensation

When you select the **Condensation** option, a function is formed for segments of code that are repeated in a project, and each instance of that code is replaced with a function call. In the simple example shown in Code 1, the same code is repeated three times in various locations of the project. When condensation is enabled, a single instance of the code is converted into a function, and wherever the instance occurs, it is replaced by a function call to that function, as shown in Figure 13. All such created functions/procedures are placed at the end of the code and below the label “<created procedures>,” as shown in Figure 13. As you can see, the created procedure for Code 1 starts at the label “<created procedures>”; therefore, wherever the instance is used, it is replaced with a call to the label “<created procedures>.” The functions at location “0520” and “052B” below “<created procedures>” are other duplicate codes in the project that are converted to callable functions.

Code 1

```
/* Instance one */
shadowRegs[PORT_2] |= 0x01;
PRT2DR = shadowRegs[PORT_2];

/* Instance two */
shadowRegs[PORT_2] |= 0x01;
PRT2DR = shadowRegs[PORT_2];

/* Instance three */
shadowRegs[PORT_2] |= 0x01;
PRT2DR = shadowRegs[PORT_2];
```

Figure 13. Condensation for Code 1 in *.lst* File

```
1984        0454: 90 C2     CALL   <created procedures>
1985   (0029)
1986   (0030)  /* Instance one */
1987   (0031)  shadowRegs[PORT_2] |= 0x01;
1988   (0032)  PRT2DR = shadowRegs[PORT_2];
1989   (0033)

2050        04AD: 90 69     CALL   <created procedures>
2051   (0060)
2052   (0061)
2053   (0062)  /* Instance two */
2054   (0063)  shadowRegs[PORT_2] |= 0x01;
2055   (0064)  PRT2DR = shadowRegs[PORT_2];

2139   (0108)      /* Instance three */
2140   (0109)      shadowRegs[PORT_2] |= 0x01;
2141        0505: 62 D0 00 MOV    REG[0xD0],0x0
2142        0508: 90 0E     CALL   <created procedures>
2143   (0110)      PRT2DR = shadowRegs[PORT_2];

2218   <created procedures>:
2219        0518: 2E 02 01 OR     [shadowRegs+2],0x1
2220        051B: 51 02     MOV    A,[shadowRegs+2]
2221        051D: 60 08     MOV    REG[0x8],A
2222        051F: 7F        RET
2223        0520: 62 D0 00 MOV    REG[0xD0],0x0
2224        0523: 26 02 FE AND    [shadowRegs+2],0xFE
2225        0526: 51 02     MOV    A,[shadowRegs+2]
2226        0528: 60 08     MOV    REG[0x8],A
2227        052A: 7F        RET
2228        052B: 71 10     OR     F,0x10
2229        052D: 43 08 01 OR     REG[0x8],0x1
2230        0530: 41 09 FE AND    REG[0x9],0xFE
2231        0533: 70 CF     AND    F,0xCF
2232        0535: 7F        RET
```

With the **Condensation** option, you can save a considerable amount of ROM space, as Figure 17 shows. To apply condensation to the code, the code in the "text" area (located in the *.mp* file, as shown in Figure 14) should be more than 256 bytes. If the "text" area is less than 256 bytes, the message "program code in 'text' area too small for worthwhile code compression" is displayed (Figure 15). If no instances of duplicate code are found for condensation, the message "no worthwhile duplicate found" is displayed (Figure 16).

Note The information displayed in Figure 15, Figure 16, and Figure 17 is for different codes to emulate different scenarios during code condensation.

Figure 14. "text" Area in *.mp* File

```
80
81  Area                                  Addr   Size   Decimal Bytes (Attributes)
82  --------------------------------      ----   ----   ------- ----- ------------
83                                        text   0445   0143 =   323. bytes (rel,con,rom,code)
84
85        Addr  Global Symbol
86       -----  --------------------------------
87        0445  __text_start
88        0445  _main
89        0588  __xidata_start
90        0559  <created procedures>
91        0588  __text_end
92        058E  __xidata_end
93
```

Figure 15. Condensation When "text" Area Is Less Than 256 Bytes

```
Output
Show output from: Build
Starting MAKE...
creating project.mk -- no changes
./main.c
Linking..
LMM info: area 'virtual_registers' uses 2 bytes in SRAM page 0
program code in 'text' area too small for worthwhile code compression
idata dump at output/CodeOptimization_PSoC.idata
  ROM 6% full. 863 out of 16384 bytes used (does not include absolute areas).
  RAM 1% full. 2 bytes used (does not include stack usage).
Built with ICCM8C STD V7.05.00
```

Figure 16. Condensation When No Duplicate Code Is Found

```
Output
Show output from: Build
Starting MAKE...
creating project.mk -- no changes
./main.c
Linking..
LMM info: area 'virtual_registers' uses 2 bytes in SRAM page 0
no worthwhile duplicate found
idata dump at output/CodeOptimization_PSoC.idata
  ROM 7% full. 1055 out of 16384 bytes used (does not include absolute areas).
  RAM 1% full. 2 bytes used (does not include stack usage).
Built with ICCM8C STD V7.05.00
```

Figure 17. Condensation on Code 1 (Code 1 is part of a bigger project whose text area is more than 256 bytes.)

```
Output
Show output from: Build
Starting MAKE...
creating project.mk -- no changes
./main.c
Linking..
LMM info: area 'data' item of 5 bytes allocated in SRAM page 0
LMM info: area 'virtual_registers' uses 4 bytes in SRAM page 0
374 bytes before Code Compression, 323 after. 12% reduction.
idata dump at output/CodeOptimization_PSoC.idata
  ROM 7% full. 1063 out of 16384 bytes used (does not include absolute areas).
  RAM 2% full. 9 bytes used (does not include stack usage).
Built with ICCM8C STD V7.05.00
```

Note Though condensation reduces ROM space considerably, it replaces duplicate codes with function calls, thus adding delay to the program execution.

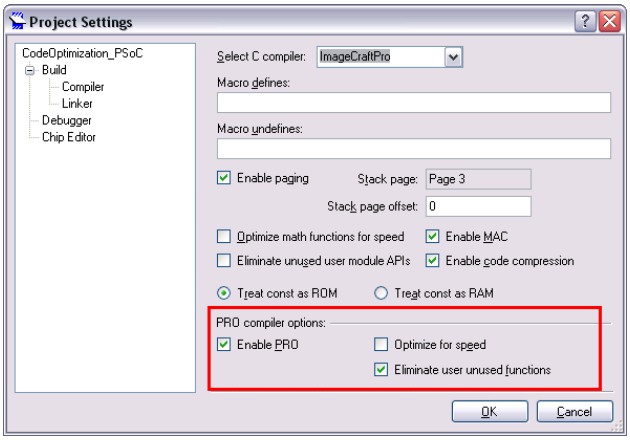
Setting 4: Treat const as ROM Versus Treat const as RAM

This option (see Figure 9) reduces RAM use by leaving constants in flash (ROM); it does not significantly reduce flash use. Use the **Treat const as ROM** option for most projects. The **Treat const as RAM** option is mainly used for backward compatibility with previous versions of the ImageCraft compiler.

ImageCraft Pro Compiler Options

In addition to the optimization options that the ImageCraft Standard compiler offers, ImageCraft Pro includes a few other options, as Figure 18 shows.

Figure 18. ImageCraft Pro Compiler Options



The **Sublimation** and **Condensation** options offered in ImageCraft Standard are the same as the **Eliminate unused user module APIs** and **Enable code compression** options in ImageCraft Pro respectively.

ImageCraft Pro also offers an option to **Eliminate user unused functions**, which is very useful when your project is huge and user unused functions are difficult to spot. A simple scenario would be a project revision; unused APIs tend to remain in the code, consuming ROM space. When this option is enabled, the compiler will make sure that unused APIs are not placed in the code and the ROM space is available for other uses.

One more option available in ImageCraft Pro is **Optimize for speed**. When this option is enabled, the code is compiled for faster execution. ROM usage may increase or decrease, depending on where optimization has been applied. For instance, smaller loops in code are expanded to sequential instructions, increasing ROM usage. Redundant assignments or move operations are removed, reducing ROM usage.

Tips and Guidelines

Several coding techniques can make your firmware more efficient. These methods work with PSoC Designer projects and PSoC 1, and they can be applied to similar 8-bit processors, IDEs, and compilers, such as PSoC 3 and PSoC Creator™.

Guideline 1: Avoiding Function Calls in Interrupt Service Routines

When compiling C code for an interrupt service routine (ISR), the ImageCraft compiler pushes onto the stack all of the virtual registers (registers used by the compiler to store temporary values) that will be used by the ISR. If the ISR code includes a function call, the compiler cannot tell which registers will be modified by the called function. The ImageCraft C compiler uses up to 15 virtual registers to store temporary data on the stack.

In PSoC chips with RAM greater than 256 bytes, 4 page pointers are also stored and restored along with the 15 virtual registers. The example in Code 2 does not require any virtual registers to be saved by the compiler (Figure 19). The same functionality, when implemented using a function call, as shown in Code 3, uses an additional 15 virtual registers, as shown in Figure 20. Each register requires the following additional overhead code: MOV [2 bytes] + PUSH [1 byte] + POP [1 byte] + MOV [2 bytes] for a total of 6 bytes per register. Code 3 uses an additional 90 bytes and involves a delayed or longer ISR execution because of the many MOV/PUSH instructions.

Code 2

```
BYTE bVar1;

#pragma interrupt_handler
SleepTimerHandler;

void SleepTimerHandler(void)
{
    bVar1 = 1;
}
```

Figure 19. Code Generated for ISR Without Function Call (Code 2)

```
958  (0012) #pragma interrupt_handler SleepTimerHandler;
959  (0013)
960  (0014) void SleepTimerHandler(void)
961  (0015) {
962  _SleepTimerHandler:
963      0445: 71 C0     OR    F,0xC0
964      0447: 08        PUSH  A
965      0448: 5D D0     MOV   A,REG[0xD0]
966      044A: 08        PUSH  A
967  (0016)      bVar1 = 1;
968      044B: 62 D0 00  MOV   REG[0xD0],0x0
969      044E: 55 09 01  MOV   [bVar1],0x1
970      0451: 18        POP   A
971      0452: 60 D0     MOV   REG[0xD0],A
972      0454: 18        POP   A
973      0455: 7E        RETI
974  (0017) }
```

Code 3

```
BYTE bVar1;
void TestFunc()
{
    bVar1 = 1;
}

#pragma interrupt_handler
SleepTimerHandler;

void SleepTimerHandler(void)
{
    TestFunc();
}
```

Figure 20. Code Generated for ISR with Function Call (Code 3)

```
990  _SleepTimerHandler:
991      02C7: 08        PUSH  A
992      02C8: 51 10     MOV   A,[__r0]
993      02CA: 08        PUSH  A
994      02CB: 51 0F     MOV   A,[__r1]
995      02CD: 08        PUSH  A
996      02CE: 51 0E     MOV   A,[__r2]
997      02D0: 08        PUSH  A
998      02D1: 51 0D     MOV   A,[__r3]
999      02D3: 08        PUSH  A
1000     02D4: 51 0C     MOV   A,[__r4]
1001     02D6: 08        PUSH  A
1002     02D7: 51 0B     MOV   A,[__r5]
1003     02D9: 08        PUSH  A
1004     02DA: 51 0A     MOV   A,[__r6]
1005     02DC: 08        PUSH  A
1006     02DD: 51 09     MOV   A,[__r7]
1007     02DF: 08        PUSH  A
1008     02E0: 51 08     MOV   A,[__r8]
1009     02E2: 08        PUSH  A
1010     02E3: 51 07     MOV   A,[__r9]
1011     02E5: 08        PUSH  A
1012     02E6: 51 06     MOV   A,[__r10]
1013     02E8: 08        PUSH  A
1014     02E9: 51 05     MOV   A,[__r11]
1015     02EB: 08        PUSH  A
1016     02EC: 51 04     MOV   A,[__rX]
1017     02EE: 08        PUSH  A
1018     02EF: 51 03     MOV   A,[__rY]
1019     02F1: 08        PUSH  A
1020     02F2: 51 02     MOV   A,[__rZ]
1021     02F4: 08        PUSH  A
1022 (0032)      TestFunc();
1023     02F5: 9F CC     CALL  TestFunc|__text_start|_TestFunc
1024     02F7: 18        POP   A
1025     02F8: 53 02     MOV   [__rZ],A
1026     02FA: 18        POP   A
1027     02FB: 53 03     MOV   [__rY],A
1028     02FD: 18        POP   A
1029     02FE: 53 04     MOV   [__rX],A
1030     0300: 18        POP   A
1031     0301: 53 05     MOV   [__r11],A
1032     0303: 18        POP   A
1033     0304: 53 06     MOV   [__r10],A
1034     0306: 18        POP   A
1035     0307: 53 07     MOV   [__r9],A
1036     0309: 18        POP   A
1037     030A: 53 08     MOV   [__r8],A
1038     030C: 18        POP   A
1039     030D: 53 09     MOV   [__r7],A
1040     030F: 18        POP   A
1041     0310: 53 0A     MOV   [__r6],A
1042     0312: 18        POP   A
1043     0313: 53 0B     MOV   [__r5],A
1044     0315: 18        POP   A
1045     0316: 53 0C     MOV   [__r4],A
1046     0318: 18        POP   A
1047     0319: 53 0D     MOV   [__r3],A
1048     031B: 18        POP   A
1049     031C: 53 0E     MOV   [__r2],A
1050     031E: 18        POP   A
1051     031F: 53 0F     MOV   [__r1],A
1052     0321: 18        POP   A
1053     0322: 53 10     MOV   [__r0],A
1054     0324: 18        POP   A
1055     0325: 7E        RETI
```

Guideline 2: Limiting Math Functions

In many cases, a C compiler implements simple math calculations with inline assembler code. However, for complex calculations, the compiler may instead add one or more math library functions to your code. This may not be immediately obvious. Depending on the operation and variable type, even a simple C operator such as “+,” “-,” “*,” “/,” “%,” “>,” and “<” may require a library function to implement.

Although they are very efficient and useful, library functions can occupy a lot of code space, which you may not anticipate. In many cases, you can avoid overusing library functions by carefully managing the data types and operations in your code.

When a program uses integer arithmetic, math library functions are added, depending on the size and type of the variables used (8-, 16- or 32-bit; signed or unsigned). Details about the byte usage of different functions are located in the *Libraries User Guide* at **Help > Documentation > Designer Specific Documents**.

The base arithmetic functions (that is, functions that are always used) include integer addition, subtraction, and shift. When other operations, such as multiplication, are used, code for those operations is also included.

Shift and Add Instead of Multiply or Divide

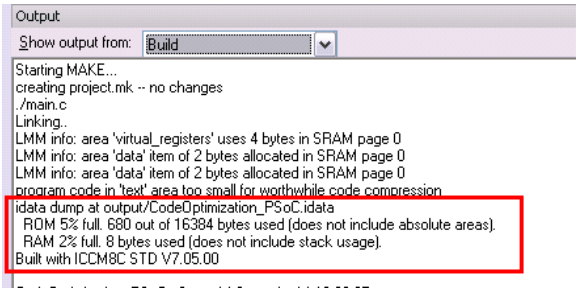
Tricks like bit shift and add, instead of a multiply or divide for unsigned integers, save code space. In unsigned integers, a single bitwise shift right is equivalent to division by 2, and shift left is equivalent to multiplication by 2. By using shift and add, as shown in the Code 5, you can avoid including multiplication and division library functions.

In the following two snippets of code, which give the same result, the Code 4 implementation uses 55 bytes more than Code 5, as shown in Figure 21 and Figure 22. This difference is due to the addition of a “__mul16” function to the code.

Code 4

```
unsigned int iTest1, iTest2;
void main(void)
{
    iTest1 = iTest2 *3;
}
```

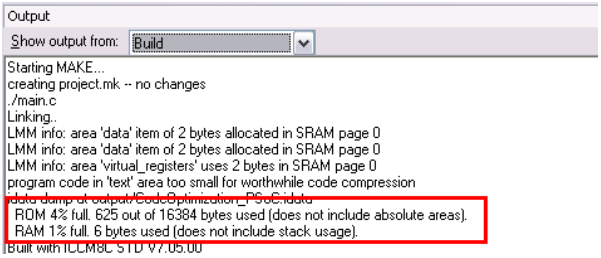
Figure 21. ROM Usage for Code 4



Code 5

```
unsigned int iTest1, iTest2;
void main(void)
{
    iTest1 = (iTest2 << 1) + iTest2;
}
```

Figure 22. ROM Usage for Code 5



Avoid Power Functions

Power functions also add math library functions. Power operations can be reduced to multiply or divide operations, for example, $4^2 = 4 \times 4$. As explained in the previous section, a multiply or divide can then be implemented using shift and add. Thus, you can save code space in programs that have limited power functions by reducing the implementation to shift and add where possible.

Avoid Floating-Point Math

Floating-point math with 8-bit processors almost always requires library functions. In addition to the function for the basic math operation, utility functions for rounding, normalization, and checking special conditions may also be added to the code. Refer to the *Arithmetic Libraries User Guide* for more information about floating-point libraries. To give you an estimate, the code space required for the floating-point functions follows (also available in the *Arithmetic Libraries User Guide*).

Comparison (*_fpcmp) = 109–125 bytes

Addition (*_fpadd) = 461–478 bytes

Subtraction (*_fpsub) = 468–485 bytes

Multiplication (*_fpmul) = 406–558 bytes

Division (*_fpdiv) = 432–449 bytes

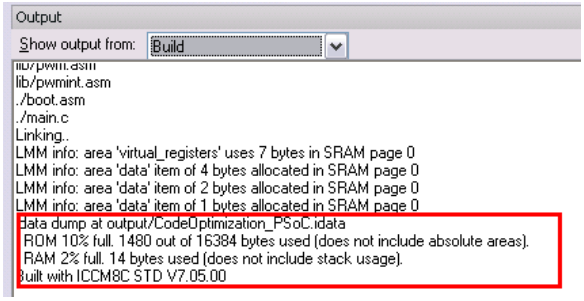
The floating-point math functions use the integer math libraries as the base. This means that the integer math libraries are also required when using floating-point math. Instead of using floating-point math, the variables can be scaled to use integer math directly in cases where the range of the variable is known.

For example, in the following two pieces of code, the range of the variable is known to be two digits after the decimal. Thus, if you multiply all of the floating-point numbers by 100, you can use integer math instead of floating-point math. In the example, the integer-math method in Code 7 uses 761 bytes less than the floating-point equivalent in Code 6, as shown in Figure 23 and Figure 24.

Code 6

```
int iTest2, iTest3;
float fTest1;
void main(void)
{
    fTest1 = iTest2 * 2.42;
    if(fTest1 > 7.5)
    {
        iTest3 = 2;
    }
    else
    {
        iTest3 = 1;
    }
}
```

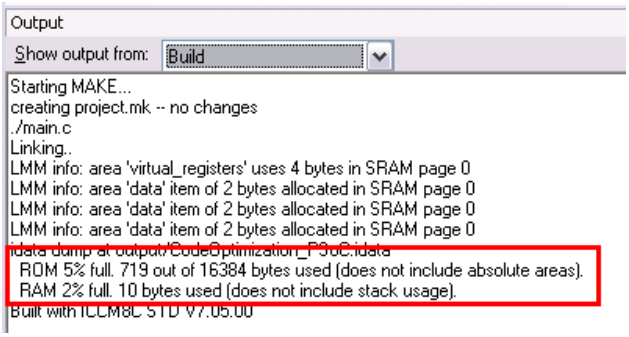
Figure 23. ROM Usage for Code 6



Code 7

```
int iTest1, iTest2, iTest3;
void main(void)
{
    iTest1 = iTest2 * 242;
    if(iTest1 > 750)
    {
        iTest3 = 2;
    }
    else
    {
        iTest3 = 1;
    }
}
```

Figure 24. ROM Usage for Code 7



Use a Lookup Table Instead of Calculation

Sometimes it may be more efficient to use a lookup table (LUT) instead of doing calculations. There are multiple tradeoffs, such as speed, accuracy, and code space. The choice is based on the type of application.

For example, the project given in AN2017 – PSoC 1 Temperature Measurement with Thermistor offers an option for the floating-point and LUT method implementation. The use of a LUT in place of floating-point math in this project saves 4187 bytes of memory (3779 bytes for the lookup table and 7966 bytes for the floating-point equation), but accuracy is the tradeoff.

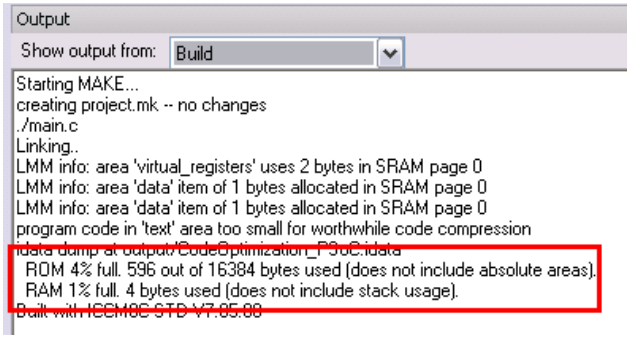
Maintain Consistency in Data Type

When choosing a data type for a variable, choose the smallest data type required for the function. For example, choose BYTE instead of int if the maximum value of the variable is not greater than 256. Similarly, choose unsigned variables instead of signed variables, as applicable. For example, consider Code 4 if the variables “iTest1” and “iTest2” do not exceed 255 in the program. Then you can declare them as “unsigned char” instead of “unsigned int,” as shown in Code 8. The ROM usage is shown in Figure 25. Code 8 consumes 84 bytes less than Code 4.

Code 8

```
unsigned char cTest1, cTest2;
void main(void)
{
    cTest1 = cTest2 *3;
}
```

Figure 25. ROM Usage for Code 8



When you know the positive and negative range of a signed variable, you can offset the variable to bring it to positive values and thus use unsigned math functions. Similarly, if the variable is expected to have a value between –10 and +10, offset the variable by 10, so that the code 0 corresponds to –10, and 20 corresponds to +10.

Maintain consistency in the data type used, because typecasting is expensive in terms of code space. If you absolutely need to use typecasting in the program, typecast the variables to use the same math libraries to save code space for the math libraries.

Guideline 3: Using Array Indexing Versus Pointer

In the array-indexing method, ImageCraft directly accesses each address using the index. No computation is needed because the address remains constant. In the case of pointer accessing, each access is based on the pointer, which is a variable. Thus, the compiler must perform more computation to get the address.

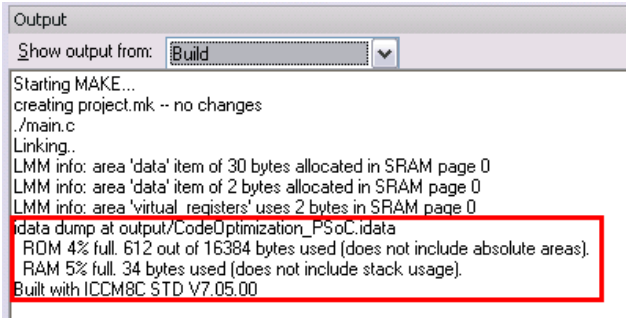
When the access method is used repeatedly, the difference in the type of access leads to a significant variation in memory use.

For example, consider the following two code examples. Code 9 uses 25 bytes more than Code 10, as shown in Figure 26 and Figure 27. So a careful observation of the type of access method used (array indexing versus pointer) is important for code optimization. There are numerous variations in the type of access and variable types. The key is to make sure the address is a constant, in which case the compiler directly replaces the address. When the address must be calculated during run time, the compiler uses more code space. In general, the “->” operator is expensive in terms of code space.

Code 9

```
typedef struct
{
    int iData;
    BYTE bData;
}sData;

typedef struct
{
    sData myArray[10];
}sArray;

sArray myTest;
sData* myPtr;

void main(void)
{
    myPtr = myTest.myArray;
    myPtr->iData = 100;
}
```

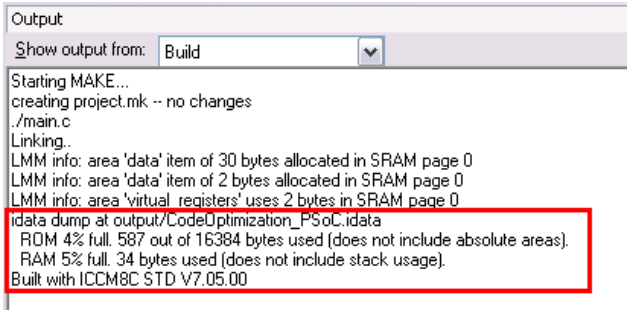
Figure 26. ROM Usage for Code 9



Code 10

```
typedef struct
{
    int iData;
    BYTE bData;
}sData;

typedef struct
{
    sData myArray[10];
}sArray;

sArray myTest;
sData* myPtr;

void main(void)
{
    myTest.myArray[1].iData = 100;
}
```

Figure 27. ROM Usage for Code 10



Guideline 4: Using Switch Statement Versus If-Else Statement

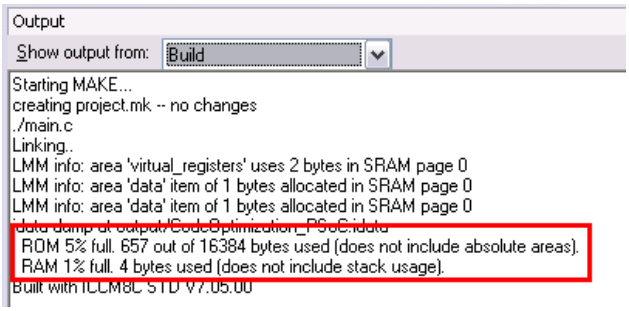
For case decisions, you can use either a switch statement or an if-else statement. The difference between the two is that the switch statement always does a 16-bit comparison, whereas the if-else statement bases its comparison on the variable type. In the case of a single-byte variable (BYTE), the ImageCraft compiler produces more efficient code using an if-else construct compared to a switch construct. This is because an 8-bit comparison needs less code space than a 16-bit comparison.

The switch statement uses 9 additional bytes, plus 5 bytes for each case. For example, a four-case switch statement with a default clause, as shown in Code 11, uses (9 + 4 × 4) = 25 more bytes than the equivalent Code 12, as shown in Figure 28 and Figure 29.

Code 11

```
BYTE bTest1, bTest2;
void main(void)
{
    switch(bTest1)
    {
        case 4:
        {
            bTest2 = 1;
            break;
        }
        case 3:
        {
            bTest2 = 2;
            break;
        }
        case 2:
        {
            bTest2 = 3;
            break;
        }
        default:
        {
            bTest2 = 4;
        }
    }
}
```

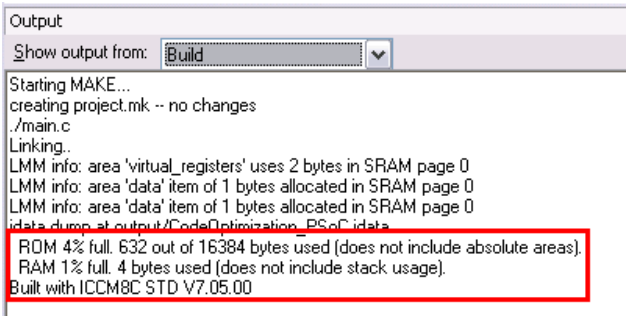
Figure 28. ROM Usage for Code 11



Code 12

```
BYTE bTest1, bTest2;
void main(void)
{
    if(bTest1 == 4)
    {
        bTest2=1;
    }
    else if(bTest1 == 3)
    {
        bTest2 = 2;
    }
    else if(bTest1 ==2)
    {
        bTest2 = 3;
    }
    else
    {
        bTest2=4;
    }
}
```

Figure 29. ROM Usage for Code 12



When the case decision is for a 2-byte variable (WORD), the resulting code size is nearly identical for either implementation.

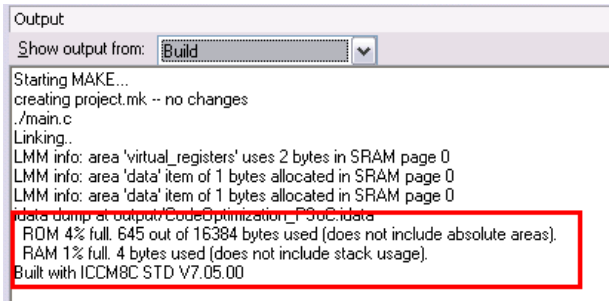
Guideline 5: Writing Part of Code in Assembler

Writing a program in assembler avoids compiler interpretations and allows complete user optimization. Though writing an entire program in assembler is tedious and cumbersome, converting only part of the code to assembly language can optimize the code size and performance. Consider Code 11 with assignment to the "bTest2" variable in assembly, as Code 13 shows. Code 13 consumes 12 bytes less than Code 11, as Figure 30 shows.

Code 13

```
BYTE bTest1, bTest2;
void main(void)
{
    switch(bTest1)
    {
        case 4:
        {
            asm("MOV [_bTest2], 0x1");
            break;
        }
        case 3:
        {
            asm("MOV [_bTest2], 0x2");
            break;
        }
        case 2:
        {
            asm("MOV [_bTest2], 0x3");
            break;
        }
        default:
        {
            asm("MOV [_bTest2], 0x4");
        }
    }
}
```

Figure 30. ROM Usage for Code 13



Guideline 6: Manipulating Bits in PSoC 1

No data type is defined for "bit" variables in PSoC 1. To manipulate bit/bits of a variable, a mask can be used with logical operators or by direct assignment to the variable. The mask can be a constant based on the manipulation requirement. Code 14 and Code 15 show how to manipulate bits in a variable.

Code 14

```
BYTE bTest1, bTest2, bTest3;
void main(void)
{
  /* To set first, sixth and eighth bit
  of the variable */
  bTest1 = (0xA1);

  /*  To  clear  the  second  and  the
  seventh bits the variable */
  bTest2 = bTest2 & (0xBD);

  /* To invert the second, fifth and
  eighth bits in the variable */
  bTest3 = bTest3 ^ (0x92);
}
```

If a variable has more than one bit definition and one or more bits must be manipulated without affecting other bits in the variable, then use the logical operators to manipulate the bits. These operations use XOR, AND, and OR instructions. The corresponding *.asm* code for Code 14 is generated in the *<projectname>.lst* file (**Workspace > Output Files**) as follows.

```
  /* To set first, sixth and eighth bit
  of the variable */
  bTest1 = bTest1 | (0xA1);
  __text_start|_main|_main:
  62 D0 00 MOV   REG[0xD0],0x0
  2E 03 A1 OR    [bTest1],0xA1

  /* To clear the second and the
  seventh bits the variable */
  bTest2 = bTest2 & (0xBD);
  62 D0 00 MOV   REG[0xD0],0x0
  26 02 BD AND   [bTest2],0xBD

  /* To invert the second, fifth and
  eighth bits in the variable */
  bTest3 = bTest3 ^ (0x92);
  62 D0 00 MOV   REG[0xD0],0x0
  51 04    MOV   A,[bTest3]
  31 92    XOR   A,0x92
  53 04    MOV   [bTest3],A
  address: 8F FF    JMP   address
```

Note The instruction `MOV REG[0xD0],0x0` sets the RAM page number of the global variable "bTest1/bTest2/bTest3."

If the size of a variable is a single bit or if all the bits defined in the variable are being assigned/manipulated simultaneously, then use direct assignment to manipulate them, as shown in Code 15. This uses the MOV instruction, which is one cycle faster than the XOR, AND, and OR instructions.

Note To find out how many cycles an instruction takes to execute, refer to the *Assembly Language User Guide,* Appendix A5, "Instruction Set Summary," in PSoC Designer at **Help > Documentation > Compiler and Programming Documents**.

Code 15

```
BYTE bTest1, bTest2;
void main(void)
{
  /* To clear the first and the second
  bits and set the third bit of a three
  bits variable */
  bTest1 = 0x04;

 /* To invert the variable */
 bTest2 = ~ (bTest2);
}
```

The corresponding *.asm* code follows:

```
BYTE bTest1, bTest2;
void main(void)
{
  /* To clear the first and the second
  bits and set the third bit of a three
  bits variable */
  bTest1 = 0x04;
  __text_start|_main|_main:
  62 D0 00 MOV   REG[0xD0],0x0
  55 02 04 MOV   [bTest1],0x4

  /* To invert the variable */
  bTest2 = ~ (bTest2);
  62 D0 00 MOV   REG[0xD0],0x0
}
  51 03    MOV   A,[bTest2]
  73       CPL   A
  53 03    MOV   [bTest2],A
  address: 8F FF    JMP   address
```

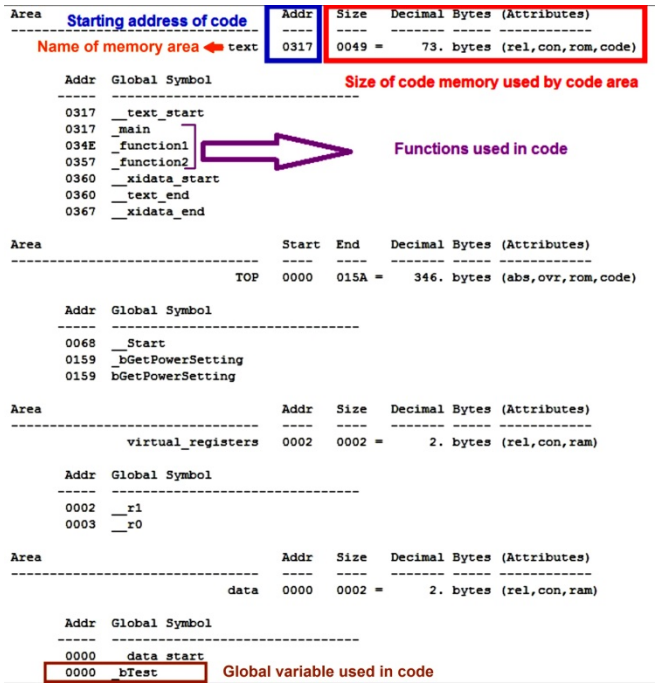
Note Instruction `MOV REG[0xD0],0x0` sets the RAM page number of the variable "bTest1/bTest2."

Guideline 7: Calculating C Code Flash Usage and Execution Time

When code memory usage and program execution time are a concern, knowing the flash usage of a function or a piece of code and its execution time can be useful in manually optimizing the code.

To find out the code size of a function, refer to the *<projectname>.mp* file in **Workspace > Output Files**. The *.mp* file provides details on code memory usage by code areas and the functions/constants defined in the areas. It also provides details on RAM usage by RAM areas and the address of global variables defined in those areas, as shown in Figure 31.

Figure 31. Address and Memory Details in *.mp* File



Note Refer to the *C Language Compiler User Guide*, section 6.9, and the *Assembly language User Guide,* section 5.1, for details on the code/RAM areas.

To find out the code size of a function, search for the function name in the *.mp* file. If you have not placed the function in a custom area, by default all the user-defined functions are placed in the "text" code area (Figure 32). Figure 32 shows the start addresses of "main," "function1," and "function2" defined in Code 16. To find out the code size of "main," subtract the address of the following function ("function1" in this case) from the address of "main." The size of "main" is 55 bytes (0x034E – 0x0317 = 0x0037), "function1" is 9 bytes (0x0357 – 0x034E = 0x0009) and "function2" is 9 bytes (0x0360 – 0x0357 = 0x0009).

Code 16

```
int bTest;
void main(void)
{
bTest = ++bTest;
}

void function1(int a)
{     a++;
      return;
}

void function2(int b)
{     b--;
      return;
}
```

Figure 32. Flash Calculation Using *.mp* File

```
Area                                Addr   Size
--------------------------------    ----   ----
                            text    0317   0049

      Addr  Global Symbol
     -----  --------------------------------
      0317    _text_start
      0317  _main
      034E  _function1
      0357  _function2
      0360    _xidata_start
      0360  __text_end
      0367  __xidata_end
```

To calculate the code execution time, you need to know the execution time of individual instructions. To find out the execution time of individual instructions, refer to the *Assembly Language User Guide*, Appendix A5, "Instruction Set Summary," in PSoC Designer at **Help > Documentation > Compiler and Programming Documents.** The "CPU clock cycles" column of the table provides the number of CPU clocks required to execute the instruction. Multiply that value with the CPU clock period to find out the execution period of an instruction. The following examples illustrate the calculation of execution time with different codes.

Example 1: Take an empty "while (1)" loop as follows.

```
void main(void)
{
while (1)
      {
      }
}
```

The corresponding *.asm* code for the empty "while (1)" loop generated in the *<projectname>.lst* file follows.

```
while (1)
{
}
address: 8X XX    JMP   address
```

The JMP instruction takes five CPU cycles for execution. If the CPU clock is 24 MHz, then the execution of the JMP instruction takes:

$$5 \times \frac{1}{24}\ \mu s = 0.208\ \mu s$$

At this rate, the CPU executes approximately one (JMP) instruction per 0.208 µS = (1/0.208) = 4.8 million instructions per second.

Example 2:
Consider an example for simple addition as follows.

```
void main(void)
{
BYTE bTest1, bTest2;
while(1)
{
      bTest1 = bTest1 + bTest2;
}
}
```

The corresponding *.asm* code follows.

```
BYTE bTest1, bTest2;
while(1)
{
      bTest1 = bTest1 + bTest2;
    address1: 52 01    MOV   A,[X+1]
    address2: 05 00    ADD   [X+0],A
}
    address3: 8X XX    JMP   address1
```

The execution of the previous code takes a total of 6 (MOV) + 8 (ADD) + 5 (JMP) = 19 clock cycles. This is on average 19/3 = 6.33 cycles per instruction. If the CPU clock is 24 MHz, then the execution of the entire code takes:

$$\frac{19}{24{,}000{,}000} s = 0.792\ \mu s$$

At this rate, the CPU executes one instruction per 0.792/3 = 0.264 µS or (1/0.264) = 3.79 million instructions per second.

Conclusion

These guidelines can help you optimize your code. Some of them are specific to the ImageCraft compiler, and some are general. Following these suggestions during and after firmware development helps optimize the code space. All the code snippets and snapshots have been tested and captured for the CY8C28xxx device and ImageCraft Standard compiler version 7.0.5. The results will be similar for other PSoC 1 devices as well.

Besides following these guidelines, when working with a smaller flash-size device, a good practice is to watch the code size increase during project development. Whenever there is a huge increase in the code space, look at the map file (Figure 4) to see if any unexpected functions are being used by the latest part of the code. Knowing the exact part of the code that causes the flash increase makes it easier to optimize your code.

For further details on compiler and project optimization options, refer to the *ImageCraft C Compiler Guide* and the *ImageCraft Assembly Language Guide*.

About the Author

Name:	Archana Yarlagadda
Title:	Applications Engineer
Background:	Applications Engineer at Cypress with focus on PSoC
	Masters in Analog VLSI from University of Tennessee, Knoxville
Contact:	msur@cypress.com

Document History

Document Title: PSoC® 1 M8C ImageCraft C Code Optimization – AN60486

Document Number: 001-60486

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	2994124	YARA	07/27/2010	New Application Note.
*A	3330339	YARA	07/27/2011	Rewritten based on customer feedback
*B	3565784	MSUR	03/29/2012	Updated all the snapshots to latest Designer version Added a section to describe compiler options of ImageCraft Pro Changed 'Technique' word to 'Guideline' to match AN60630 Changed Code 1 to reflect a common use case of Condensation Added Codes to Guideline#6 and #2 (Maintaining Consistency in data type Updated all sections to comply with the latest PSoC Designer. Updated abstract. Updated template.
*C	4088730	MSUR	8/6/2013	Added link to AN75320 – Getting Started with PSoC 1 Added Figure 12 Updated the links on how to reach compiler settings in all sections Updated Guideline#2 (Avoid floating-point calculation) per the updated Arithmetic Libraries Guide Minor document updates based on feedback
*D	4429390	ASRI	07/03/2014	Removed Guideline #5: Initializing Global Variables. This requirement is already taken care of by updated version of the compiler. Added Guideline #6: Bit manipulation in PSoC 1 Added Guideline #7: Calculating flash usage and execution time of C code Added Figure 31. Address and Memory Details in *.mp* File Added Figure 32. Flash Calculation Using *.mp*
*E	4476944	MSUR	08/18/2014	No technical updates. Completing Sunset Review.

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

Automotive	cypress.com/go/automotive
Clocks & Buffers	cypress.com/go/clocks
Interface	cypress.com/go/interface
Lighting & Power Control	cypress.com/go/powerpsoc cypress.com/go/plc
Memory	cypress.com/go/memory
PSoC	cypress.com/go/psoc
Touch Sensing	cypress.com/go/touch
USB Controllers	cypress.com/go/usb
Wireless/RF	cypress.com/go/wireless

PSoC® Solutions

psoc.cypress.com/solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP

Cypress Developer Community

Community | Forums | Blogs | Video | Training

Technical Support

cypress.com/go/support

PSoC is a registered trademark of Cypress Semiconductor Corp. PSoC Creator and PSoC Designer are trademarks of Cypress Semiconductor Corp. All other trademarks or registered trademarks referenced herein are the property of their respective owners.

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709

Phone : 408-943-2600
Fax : 408-943-4730
Website : www.cypress.com

© Cypress Semiconductor Corporation, 2010-2014. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.
This Source Code (software and/or firmware) is owned by Cypress Semiconductor Corporation (Cypress) and is protected by and subject to worldwide patent protection (United States and foreign), United States copyright laws and international treaty provisions. Cypress hereby grants to licensee a personal, non-exclusive, non-transferable license to copy, use, modify, create derivative works of, and compile the Cypress Source Code and derivative works for the sole purpose of creating custom software and or firmware in support of licensee product to be used only in conjunction with a Cypress integrated circuit as specified in the applicable agreement. Any reproduction, modification, translation, compilation, or representation of this Source Code except as specified above is prohibited without the express written permission of Cypress.
Disclaimer: CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Cypress reserves the right to make changes without further notice to the materials described herein. Cypress does not assume any liability arising out of the application or use of any product or circuit described herein. Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress' product in a life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.
Use may be limited by and subject to the applicable Cypress software license agreement.