AN89610

PSoC® 4 and PSoC 5LP ARM® Cortex® Code Optimization

Authors: Mark Ainsworth, Asha Ganesan, Mahesh Balan, Keith Mikoleit

Associated Project: No

Associated Part Family: All PSoC 4 and PSoC 5LP Parts

Software Version: PSoC Creator™ 3.0

Related Documents: For a complete list, click here.

To get the latest version of this application note, or the associated project file, please visit http://www.cypress.com/go/AN89610.

AN89610 shows how to optimize C and assembler code for the ARM Cortex CPUs in PSoC® 4 and PSoC 5LP. Coding techniques exist for improved CPU performance and effective use of the PSoC memory architecture, which can lead to increased efficiency and reduced power consumption. This application note covers both the gcc and Keil Microcontroller Development Kit (MDK) C compilers supported by PSoC Creator™.

Contents

1 Introduction

The ARM Cortex CPUs in the PSoC 4 and PSoC 5LP devices are designed to implement C code in a highly efficient manner. Thus, most of the time, you will not need any special knowledge to do C programming for PSoC 4 or PSoC 5LP. This application note helps you to solve more advanced, unique problems, typically around:

- Fitting an application into a small amount of flash or SRAM
- Time-constrained applications, that is, maximizing code speed and efficiency

A number of methods are provided to solve these types of problems.

This application note assumes that you know how to program embedded applications in the C language. Some knowledge of the gcc (GNU Compiler Collection) or Keil MDK (Microcontroller Development Kit) C compiler is recommended. Knowledge of the Thumb-2 assembly language used by the CPUs will also help.

You should also know how to use PSoC Creator, the integrated development environment for PSoC 3, PSoC 4, and PSoC 5LP. If you are new to PSoC 4 or PSoC 5LP, you can find introductions in AN79953, Getting Started with PSoC 4 and AN77759, Getting Started with PSoC 5LP. If you are new to PSoC Creator, see the PSoC Creator home page.

Note: Although many of the examples show code in Thumb-2, the Cortex assembly language, this application note is not intended to be a tutorial on this language. For details and tutorials on Thumb-2 assembler, see ARM Cortex Documentation.

For information on optimizing C code for the 8051 CPU in PSoC 3, see AN60630, PSoC 3 8051 Code and Memory Optimization.

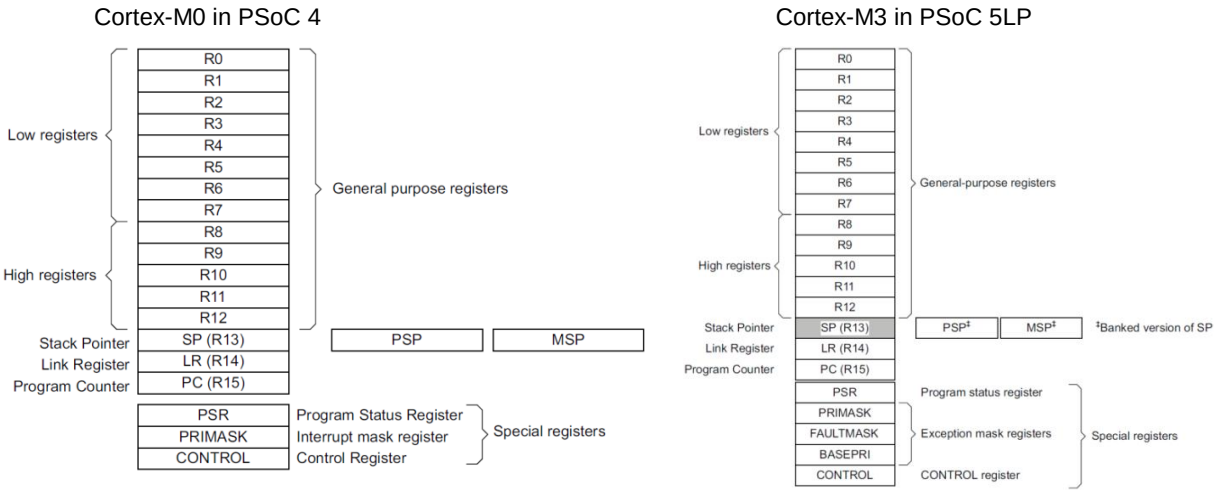
2 PSoC 4 and PSoC 5LP Architectures

To effectively use the methods described in this application note, it is important to understand the register and address architectures on which they are based. This section describes those architectures.

2.1 Register Set

The Cortex register set and instruction set are the basis for implementing highly efficient C code. The PSoC 4 Cortex-M0 and the PSoC 5LP Cortex-M3 registers are very similar, as Figure 1 shows.

Figure 1. Cortex CPU Architectures



All registers are 32-bit. There are 12 general-purpose registers (low registers R0 – R7 have more extensive support in the instruction set). Special registers include:

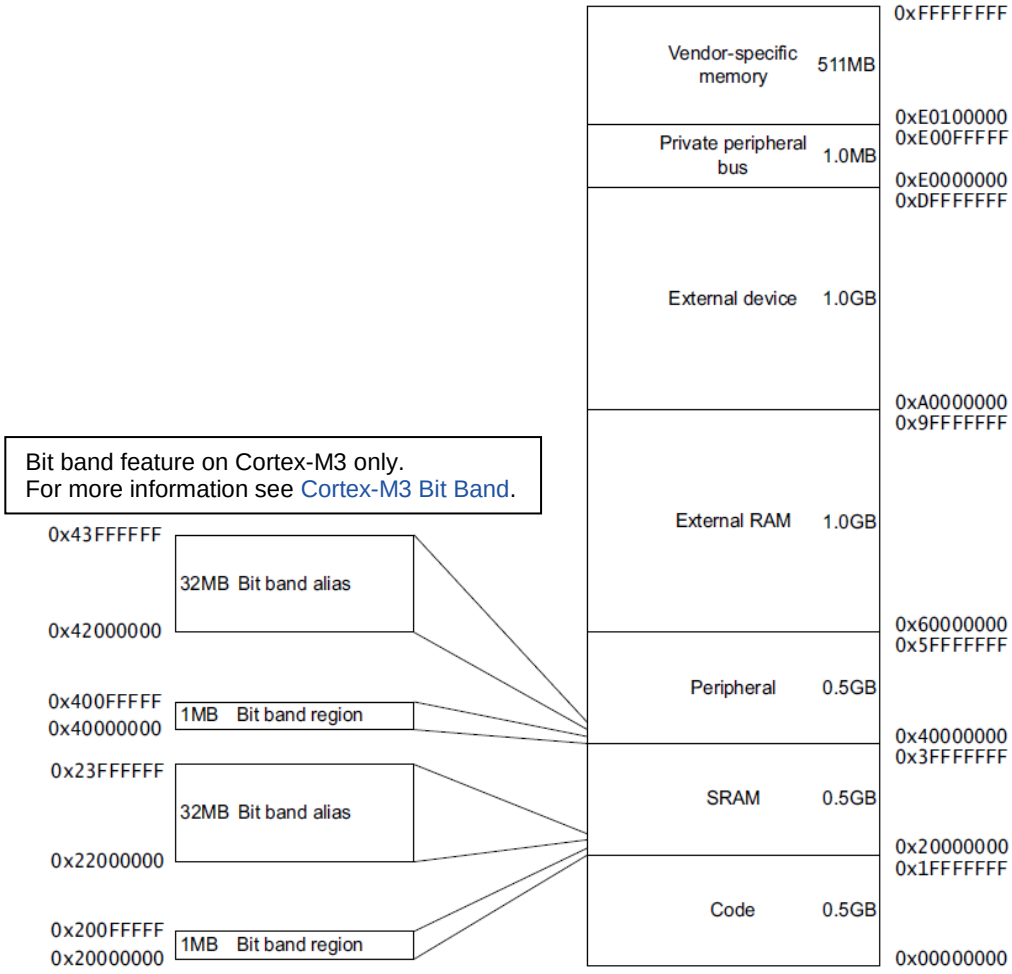
- Dual stack pointers (R13) for more efficient implementation of a real-time operating system (RTOS)
- Link register (R14) for fast return from function calls
- Program counter (R15)
- Program status register (PSR) contains instruction results such as zero and carry flags
- Interrupt mask register (Cortex-M0) / exception mask registers (Cortex-M3)
- Control register

The PSoC 5LP Cortex-M3 has more features in stack management, and in the PSR, interrupt, and control registers. The Cortex-M3 also has a more extensive instruction set, including divide (UDIV, SDIV), multiply and accumulate (MLA, MLS), saturate (USAT, SSAT), and bitfield instructions. See Special-Function Instructions for information on how to take advantage of these instructions.

2.2 Address Map

The Cortex-M0 and Cortex-M3 have a very similar address map, as Figure 2 shows.

Figure 2. Cortex Address Map



The address space is 4 Gbyte (32-bit addressing), and is divided into the access regions shown in Figure 2. The CPUs can execute instructions in the Code, SRAM, and External RAM regions; you can put code or data in any of these regions. The CPUs have a 3-instruction pipeline, which enables parallel fetch and execution of instructions.

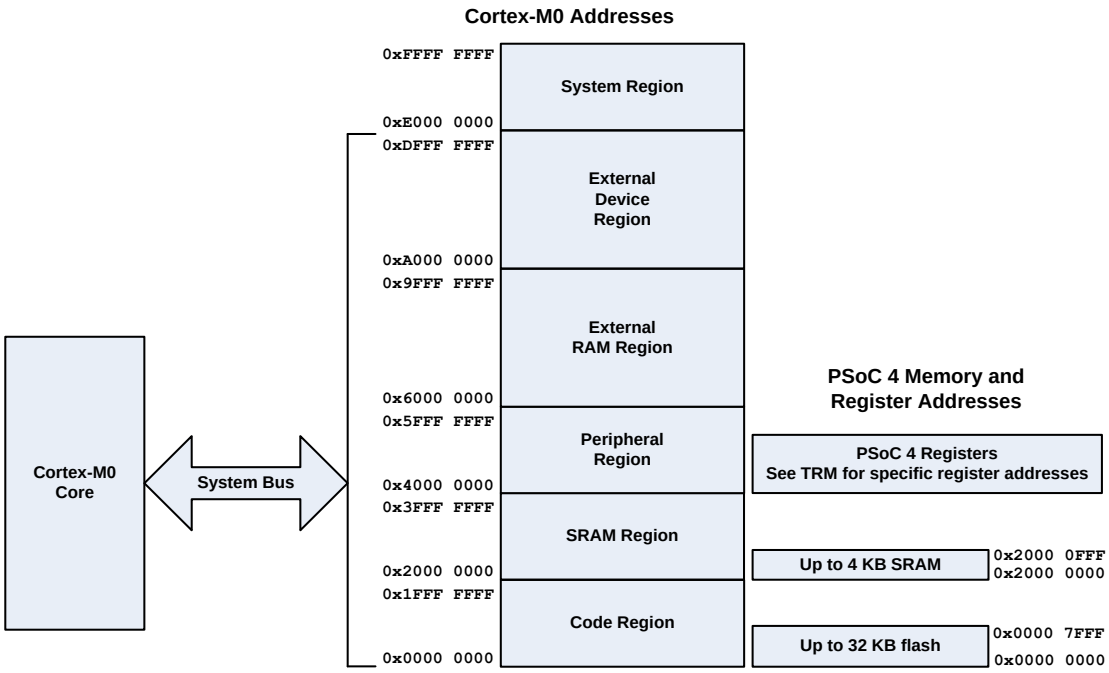
The PSoC 5LP Cortex-M3 has a bit band feature, where accessing an address in an alias region results in bit-level access in the corresponding bit band region. This lets you quickly set, clear or test a single bit in the bottom 1 Mbyte of the region. See Cortex-M3 Bit Band for more information.

Although the Cortex CPUs can access a 4 Gbyte address space, within the PSoC devices only a small fraction of these addresses access PSoC memory or registers. Following is an overview of where in the Cortex address space the PSoC memory and registers are located; for details see the memory maps in the device datasheets or Technical Reference Manuals (TRMs).

2.2.1 PSoC 4 Address Map

Figure 3 shows that a single Cortex-M0 bus, the System Bus, is used to access most of the regions in the address map.

Figure 3. PSoC 4 Address Map



The PSoC 4 memory and registers are addressed as follows:

- The PSoC 4 flash starts at address 0, in the Cortex Code region. The flash block includes a read accelerator; see a PSoC 4 device datasheet for details.
- The PSoC 4 SRAM starts at address 0x20000000, in the Cortex SRAM region.
- The PSoC 4 registers are addressed starting at 0x40000000, in the Cortex Peripheral region. See a PSoC 4 Technical Reference Manual (TRM) for specific register addresses.

All memory accesses are 32-bit.

Code can be placed in PSoC 4 SRAM; see Placing Code and Variables for details.

Note: Because PSoC 4 has only one bus, the speed and efficiency of code execution and data access depend solely on the speed of the memory occupying those regions. SRAM is usually faster than flash; however, the combination of the Cortex instruction pipeline and the flash read accelerator makes Code region accesses almost as fast as SRAM region accesses. It is possible to execute code from SRAM but significant performance gains may not necessarily be realized.

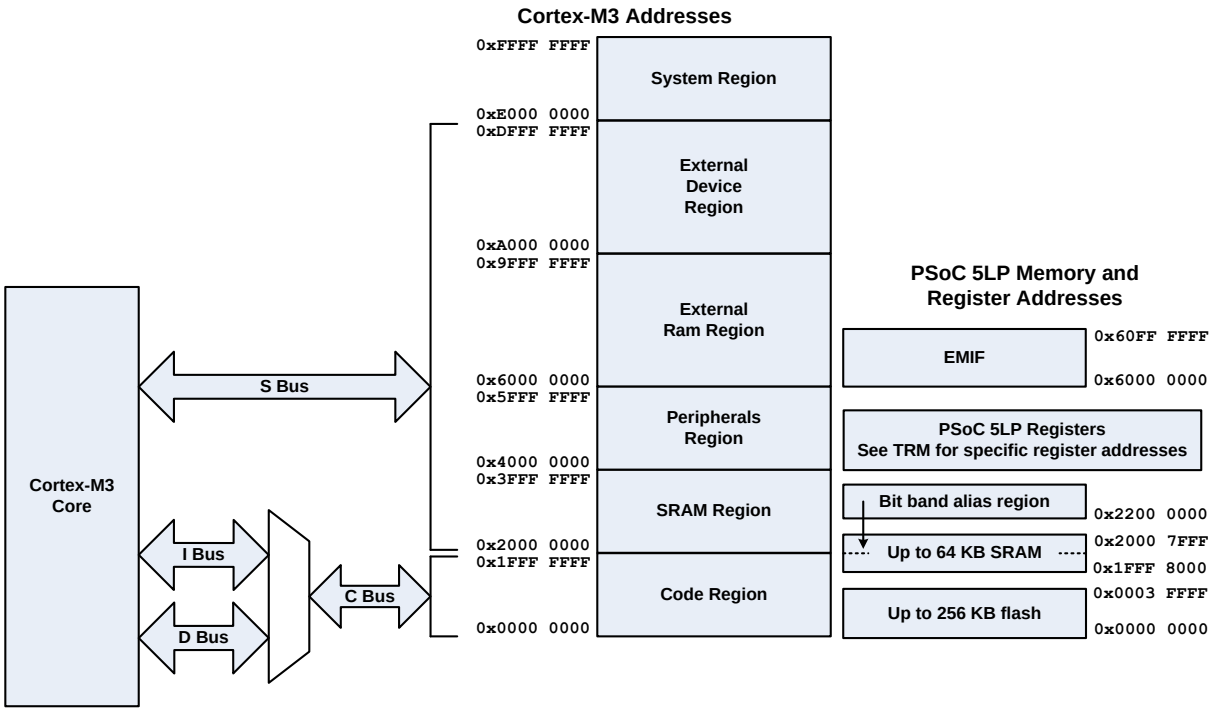
2.2.2 PSoC 5LP Address Map

The PSoC 5LP / Cortex-M3 architecture is more complex and has more features than that of the PSoC 4, as Figure 4 shows. The Cortex-M3 has three buses instead of one:

- I (instruction) Bus and D (data) Bus: for reading instructions and accessing data, respectively, from the Code region.
  In PSoC 5LP, the I and D Buses are multiplexed to a single C (code) Bus for accessing the Code region.
- S (system) Bus: for reading instructions and accessing data from the other regions

Because the C Bus and the S Bus are separate, the Cortex-M3 can do simultaneous parallel accesses of the Code region and the other regions, for more efficient operation.

Figure 4. PSoC 5LP Address Map



The PSoC 5LP memory and registers are addressed as follows:

- The PSoC 5LP flash starts at address 0, in the Cortex Code region. A flash cache is included; see a PSoC 5LP device datasheet for details.
- The PSoC 5LP SRAM is logically split in half, centered at address 0x20000000. For example, in a device with 64 KB SRAM, half of the SRAM, 32 KB, is addressed below 0x20000000 and the other half above 0x20000000. So the SRAM addresses range from 0x1FFF8000 to 0x20007FFF. The addresses in a device with 32 KB SRAM range from 0x1FFFC000 to 0x20003FFF.

  The lower half of SRAM, called **code SRAM**, is located in the Cortex Code region. The upper half, called **upper SRAM**, is located in the Cortex SRAM region. The two halves are accessed by different buses, as Figure 4 shows. Locating half of the SRAM in the Code region enables placement of code and data for possible faster access – see Placing Code and Variables for details.

Note: Within the PSoC 5LP, SRAM accesses are usually faster than flash accesses, however the combination of the Cortex instruction pipeline and the flash cache makes flash accesses almost as fast as SRAM accesses. It is possible to execute code from either code SRAM or upper SRAM but significant performance gains may not necessarily be realized.

Note that only upper SRAM is in the Cortex-M3 bit band region.

- The PSoC 5LP registers are addressed starting at 0x40000000, in the Cortex Peripheral region. See a PSoC 5LP Technical Reference Manual (TRM) for specific register addresses.
- The PSoC 5LP External Memory Interface (EMIF) addresses start at 0x60000000, in the Cortex External RAM region. For more information on PSoC 5LP EMIF see the device datasheet, TRM, or the EMIF Component datasheet.

All memory accesses are 32-bit except EMIF, which can be set to either 8-bit or 16-bit.

The PSoC 5LP also includes a direct memory access (DMA) controller. It shares bandwidth with the CPU as dual bus masters, using bus arbitration techniques. For more information, see AN52705, Getting Started with PSoC DMA. See also DMA Addresses in this application note.

2.3 Interrupts

Both Cortex CPUs offer sophisticated support for rapid and deterministic interrupt handling. For more information see a device datasheet or TRM, the PSoC Creator Interrupt Component datasheet, or AN54460, PSoC Interrupts.

3 Compiler General Topics

Before we begin in-depth examination of the gcc and MDK compilers, let us examine a few general compiler topics.

Note: All of the C code examples shown in this application note are designed for use with the C compilers supported by PSoC Creator 3.0: gcc 4.7.3 and the Keil Microcontroller Development Kit (MDK) version 5.03. The gcc 4.7.3 compiler is included free with your PSoC Creator installation. MDK must be purchased however an object-size-limited evaluation version, MDK-Lite, is available free from Keil. Compiler optimizations are turned off (the PSoC Creator default) except where noted.

All of the C code examples in this application note use ANSI standard C except for compiler-specific extensions.

3.1 Compiler Predefined Macros

It is a best practice to write C code that can be directly ported between as many different compilers as possible. However there are cases where this is not possible and you must write multiple versions of the same code, to be used with multiple compilers. If you need to do this you can use **predefined macros**, provided with most compilers, to identify the compiler being used. This allows you to compile only the code for the compiler being used, for example:

```
#if defined(MY_COMPILER_MACRO)
  /* put your compiler-unique code here */
#endif
```

To apply this technique to PSoC Creator projects, use the following macros that are included with the gcc and MDK compilers, respectively. Note that for MDK you are checking just for whether `__ARMCC_VERSION` is defined, indicating that that compiler is being used. You do not necessarily need to care about its actual value, i.e., the compiler version.

```
#if defined(__GNUC__)
  /* put your gcc unique code here */
#elif defined(__ARMCC_VERSION)
  /* put your MDK unique code here */
#endif
```

3.2 Viewing Compiler Output

To understand how a compiler performs under different conditions, you must review the output assembler code. There are two ways to do that in PSoC Creator.

1. Open the list file corresponding to the compiled C file (*filename.lst*), as Figure 5 shows.

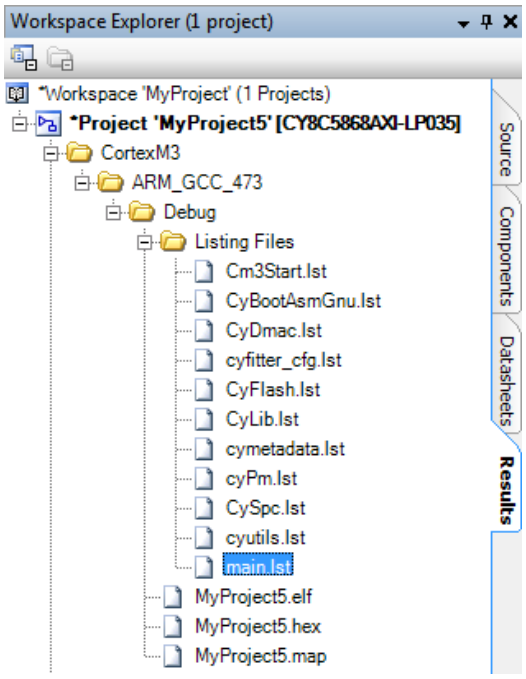
   The default PSoC Creator project build setting is to create a list file; see menu item Project > Build Settings > Compiler > General.

2. Use the disassembly window in the debugger (menu item Debug > Windows > Disassembly). Right-click in that window to bring up options to show mixed source and assembler.

   Of course, this method has the disadvantage that you must have working target hardware and a PSoC Creator project that builds correctly before you can use the debugger.

Note: The free evaluation version of MDK, MDK-Lite, does not include assembler in the *.lst* file, so method 2 must be used to see the output assembler code.

Figure 5. Listing Files



3.3 Compiler Optimizations

Turning on optimization options makes the compiler attempt to improve the C code's performance and/or size, at the expense of compilation time and possibly the ability to debug the program.

PSoC Creator allows you to set compiler optimizations for an entire project, under Project > Build Settings > Compiler > Optimization. The optimizations offered by PSoC Creator for both gcc and MDK are just for "speed" or "size". (This is different from the 11 levels of optimization offered for the Keil 8051 compiler.)

The optimization option selected in the Build Settings dialog applies to all C files in the project. You can also apply optimizations to individual C files – in the Workspace Explorer window right-click on the file and select Build Settings. With gcc you can't set optimization levels for individual functions except for using certain function attributes. With MDK you can use #pragma to set optimization for an individual function. For more information see C Documentation.

It is strongly recommended that after compiling C code with optimizations you carefully review the assembler output and confirm that it is doing what you expect. Stepping through the assembler code in the debugger may also be helpful. One best practice is to get your C code working without optimizations, then rebuild with optimizations and repeat your tests. You can do this using the Debug and Release configurations in your PSoC Creator project build settings.

For specific examples of how various optimization options work, see Appendix A.

3.4 Attributes

An extension to the C language that is supported by both gcc and MDK is to apply attributes to functions, variables, and structure types. Attributes can be used, for example, to control:

- specific function optimizations
- how structures occupy memory (see Packed and Unpacked Structures)
- function and variable location in memory (see Placing Code and Variables)

The syntax is (two underscore characters before and after the "attribute"):

```
__attribute__ ((<attribute-list>))
```

Specific attributes are described in detail in subsequent sections in this application note. For more information, see C Documentation.

4 Accessing Variables

When reviewing compiler output, one of the first areas to examine is how variables (and arrays and structures) are read and written. In their assembly language output, the gcc and MDK compilers both implement certain techniques for accessing:

- global and static variables
- automatic (local) variables
- function arguments and function result

Let us examine how each of these is done.

4.1 Global and Static Variables

The Thumb-2 assembly language used by both Cortex CPUs does not generally support loading 32-bit immediate values into a register. (There are exceptions; for example small immediate values can be loaded and sign-extended.) This makes it difficult to load the address of a global or static variable, or in general to load any address. Table 1 shows two methods for handling this problem, for the following example C code:

```
/* loading a global variable */
uint32 myVar;
. . .
myVar = 7;
```

Table 1. Example Methods for Loading Addresses into CPU Registers

Method 1: two 16-bit immediate loads	Method 2: PC-relative load

`; rx = address of myVar`
`; load the lower and upper halves of the`
`; address`
`movw  rx, #<LS word> ; 32-bit instruction`
`movt  rx, #<MS word> ; 32-bit instruction`
`. . .`
`movs  ry, #7`
`str   ry, [rx, #0] ; myVar = 7`
	`; rx = address of myVar`
	`; load the value stored in flash, below`
	`ldr   rx, [pc, #<offset>] ; 16-bit instruction`
	`. . .`
	`movs  ry, #7`
	`str   ry, [rx, #0] ; myVar = 7`
	`. . .`
	`; address value stored after the end of`
	`; the function`
	`.word  <address of myVar> ; 32-bit value`

In general method 2 is preferred for size-limited applications because it uses two fewer bytes (one 16-bit word). However, method 1 may execute faster due to the Cortex instruction pipeline. Note that with PSoC, instruction execution speed also depends on the flash cache (PSoC 5LP) or accelerator (PSoC 4) and thus is not necessarily deterministic. See Memory Map for details.

Different compiler optimizations implement one or the other of these two methods; for detailed examples see Appendix A.

It is a coding best practice to minimize use of global variables. Doing so with the PSoC Cortex CPUs may also act to reduce code size by reducing the loading of memory and PSoC register addresses.

4.2 Automatic Variables

In C, automatic variables are variables that are defined within (local to) a function. Depending on the size and complexity of the function, and the compiler optimization setting, an automatic variable may be assigned to a CPU register or it may be saved on the stack, as Table 2 shows:

Table 2. Example Methods for Using Automatic Variables

C Code	Assembler Code

`void MyFunc(void)`
`{`
`  uint8 i = 3;`
`  uint8 j = 10;`
`  . . .`
`}`
	`; use rx as i, do NOT save it on the stack`
	`movs  rx, #3  ; initialize i`
	
	`; store j on the stack; use ry to temporarily`
	`; hold the initial value`
	`movs  ry, #10  ; initialize j`
	`strb  ry, [sp, #<offset>] ; on the stack`

Both size and speed optimizations tend to reduce stack usage for automatic variables; see Appendix A for examples.

4.3 Function Arguments and Result

The Procedure Call Standard for ARM Architecture allocates registers R0 – R3 for passing arguments to a function, and R0 for passing a function result. If the number of arguments is greater than four, the first four arguments are placed in the registers and the rest are pushed onto the stack.

Within the function, the arguments may be maintained in their respective registers, transferred to other registers, or saved on the stack. Given that a function's automatic variables may also be stored on the stack (Table 2), stack management may become complex. To handle this complexity two sequences of instructions, known as **prolog** and **epilog**, may be included in a compiled function, as the example in Table 3 shows:

Table 3. Example Function With Prolog and Epilog Instructions

C Code	Assembler Code

`/* Function with 6 arguments and a return value */`
`uint32 MyFunc(uint32 a, uint32 b, uint32 c, uint32 d, uint32 e, uint32 f)`
`{`
`  return a + b + c + d + e + f;`
`}`
	`; function prolog`
	`push  {r7}  ; make room on the stack`
	`sub  sp, sp, #20  ; for the arguments`
	`add  r7, sp, #0  ; use r7 as a base pointer`
	`str  r0, [r7, #12] ; save the arguments`
	`str  r1, [r7, #8]  ; on the stack`
	`str  r2, [r7, #4]`
	`str  r3, [r7, #0]`
	
	`; function body`
	`ldr  r2, [r7, #12] ; build the sum`
	`ldr  r3, [r7, #8]  ; in r2 and r3`
	`adds  r2, r2, r3`
	`ldr  r3, [r7, #4]`
	`adds  r2, r2, r3`
	`ldr  r3, [r7, #0]`
	`adds  r2, r2, r3`
	`ldr  r3, [r7, #24] ; argument e`
	`adds  r2, r2, r3`
	`ldr  r3, [r7, #28] ; argument f`
	`adds  r3, r2, r3`
	`mov  r0, r3  ; return value in r0`
	
	`; function epilog`
	`add  r7, r7, #20  ; restore stack and r7`
	`mov  sp, r7`
	`pop  {r7}`
	`bx  lr  ; return`

To minimize code size and maximize speed, you should limit the number of function arguments to four.

Depending on the size and complexity of the function, the size optimization tries to reduce the function prolog and epilog code; see Appendix A for examples.

4.4 LDR and STR instructions

These instructions are used to read to and write from memory and PSoC registers. They are quite powerful and offer many flexible options. Variants of the instructions support byte and halfword (16-bit) accesses, zero and sign extensions, and immediate and register offsets. The offset options are particularly useful for handling pointer offsets and for accessing members of arrays and structures, as Table 4 shows.

Table 4. Example Usage of LDR and STR Instructions

C Code	Assembler Code

`/* loading an array member */`
`uint8 myArray[100];`
`. . .`
`myArray[6] = 7;`
	`ldr  rx, [pc, #<offset>] ; rx = address of myArray`
	
	`movs  ry, #7`
	`strb  ry, [rx, #6]`

`uint8 myArray[100];`
`. . .`
`uint8 i;`
`. . .`
`myArray[i] = 7;`
	`ldr  rx, [pc, #<offset>] ; rx = address of myArray`
	
	`ldrb  ry, [sp, #<offset>] ; ry = i (automatic variable)`
	`movs  rz, #7`
	`strb  rz, [rx, ry]`

Note that the LDR and STR instructions are always register-relative, so before an LDR or STR instruction there is always another instruction to load a register with the target address; see Table 1. Variations and options for these instructions are different in the Cortex-M0 (PSoC 4) and the Cortex-M3 (PSoC 5LP). For more information, see ARM Cortex Documentation.

5 Mixing C and Assembler Code

One of the most effective ways to make your code shorter, faster, and more efficient is to write it in assembler. Using assembler may also enable you to take advantage of special-function instructions that are supported by the CPU but are not used by the C compiler; see Special-Function Instructions. However, coding in assembler is a daunting task for all but the smallest applications, and once written the code is not easy to maintain or port to other compilers or CPUs. That is why most code is written in C, and if assembler is used at all it is used only for a few critical functions.

Another problem with assembler is that it must be written in its own file, separate from the C files with which it must coexist. This can cause difficulties integrating and maintaining the code.

A solution to both of these problems is an extension to the C language called **inline assembler**, where assembler code can be placed directly in C files and is treated as just another C statement. This lets you use assembler only where it's needed to increase efficiency, and makes it easier to mix C and assembler. The gcc and MDK compilers both support inline assembler. In addition, MDK supports a similar feature called **embedded assembler**, where a function is written entirely in assembler but is included in a C file.

This section shows how to use combined C and assembler code, for both the gcc and MDK compilers. To effectively use the methods described in this section, it is important to understand the register architectures on which they are based – see CPU Register Architectures for details.

Note: The following examples show assembler for the Cortex-M3; the Cortex-M0 uses a more limited subset of the Cortex-M3 instructions. For details see ARM Cortex Documentation.

Note: Most assembler instructions act on the Cortex registers (see Register Set). The Procedure Call Standard for ARM Architecture requires that some of these registers be preserved by functions. If needed, use the PUSH and POP instructions to save and restore registers on the stack.

5.1 Syntax

The gcc syntax for inline assembler is:

```
asm("assembler instruction");
```

which adds a single line of assembler code to the C code. For example, the following increments the R0 register:

```
asm("ADD  r0, r0, #1")

; /* R0 = R0 + 1 */
```

The syntax for multi-line inline assembler is:

```
asm("line 1\n"
    "line 2\n"
    "line 3\n"
    . . .
    "line n");
```

For example:

```
/* R0 = R0 + 1; R1 = R0 */
asm("ADD  r0, r0, #1\n"
    "MOV  r1, r0");
```

Note: The keyword `__asm__` can be used instead of `asm`; see C Documentation for details.

Note: You can add the keyword `volatile` to prevent the statement from being optimized out by the compiler:

```
asm volatile(" ... ");
```

The MDK syntax for inline assembler is the same as that for gcc, except that the "asm" is preceded by two underscore characters:

```
__asm("assembler instruction");

__asm("line 1\n"
      "line 2\n"
      "line 3\n"
      . . .
      "line n");
```

The syntax for the MDK embedded assembler is:

```
__asm return-type
function-name(argument-list)
{
  /* This is a C comment */
  instruction  ; assembler comment
  ...
  instruction
}
```

For example:

```
__asm int DoSum(int x, int y)
{
  ADD r0, r0, r1
  BX  lr
}
```

5.2 Automatic Variables

With gcc, to access an automatic (or local) variable from inline assembler, you must first force the variable to occupy a register Rx. Declare the variable as follows:

```
register int foo asm("r0"); /* foo occupies register R0 */
```

Note: gcc actually supports a complex language of C expression operands for the `asm` keyword. A tutorial on this language is beyond the scope of this application note. Details can be found in C Documentation, especially section 6.41 of "Using the GNU Compiler Collection".

As an example, let us define two automatic variables, 'foo' and 'bar', and do a simple math operation between them:

```
void main()
{
    register int foo asm("r0") = 5L; /* register variables can be initialized */
    register int bar asm("r1");

    bar = foo + 1; /* C code version */

    asm("ADD r1, r0, #1"); /* bar = foo + 1 */
}
```

In the above example, the C code and the inline assembler do the same operation. However, the compiled C code (no optimization) uses an intermediate register and consequently produces 3x the instructions using 2x the flash memory, as this excerpt from the *.lst* file shows:

```
20:.\main.c      ****  bar = foo + 1;
42 0008 0346      mov  r3, r0
43 000a 03F10103  add  r3, r3, #1
44 000e 1946      mov  r1, r3

22:.\main.c      ****  asm("ADD r1, r0, #1"); /* bar = foo + 1 */
47 0010 00F10101  ADD  r1, r0, #1
```

Depending on the function size and complexity it may be possible to eliminate the intermediate register by using a compiler optimization option.

With MDK, there is no need to force an automatic (local) variable to occupy a register. Instead, you can access the variables directly:

```
void main()
{
    /* no need to declare variables in registers */
    int foo = 5;
    int bar;

    bar = foo + 1; /* C code version */

    __asm("ADDS bar, foo, #1"); /* bar = foo + 1 */
}
```

In this example, the C code and the inline assembler do the same operation and produce the exact same code.

5.3 Global and Static Variables

The previous methods can also be used with global and static variables (“globals”). Note that before accessing a global you must load a register with the address of the variable – see Global and Static Variables.

With gcc, use the following syntax to load an address:

```
LDR  rx, =variable_name
LDR  ry, =0x1FFF9000 ; or any
address
```

Let us repeat the previous example using globals instead of automatic variables:

```
int foo = 5L;
int bar;

void main()
{
    bar = foo + 1;

    /* bar = foo + 1 */
    asm("LDR r0, =foo\n"
        "LDR r1, =bar\n"
        "LDR r2, [r0]\n"
        "ADD r2, r2 #1\n"
        "STR r2, [r1]");
}
```

Again, the C code and the inline assembler do the same operation but due to different address load methods (see Table 1) the compiled C code (no optimizations) produces two more instructions and uses four more bytes of memory than the inline assembler (for the Cortex-M3), as the following debugger snip shows:

```
bar = foo + 1;
    F248130C  movw    r3, #810c
    F6C173FF  movt    r3, #1fff
    681B      ldr     r3, [r3, #0]
    F1030201  add.w   r2, r3, #1
    F248132C  movw    r3, #812c
    F6C173FF  movt    r3, #1fff
    601A      str     r2, [r3, #0]

/* bar = foo + 1 */
asm("LDR r0, =foo\n"
    "LDR r1, =bar\n"
    "LDR r2, [r0]\n"
    "ADD r2, r2, #1\n"
    "STR r2, [r1]");
    4804      ldr     r0, [pc, #10]
    4905      ldr     r1, [pc, #14]
    6802      ldr     r2, [r0, #0]
    F1020201  add.w   r2, r2, #1
    600A      str     r2, [r1, #0]
```

The above results may be different if compiler optimizations are used.

With MDK there are two methods to access global variables. The first method, inline assembler, is similar to that for accessing automatic variables:

```
/* bar = foo + 1 */
__asm("ADDS bar, foo, #1");
```

However, the assembler output is quite different:

```
4805      ldr   r0, [pc, #14]
6800      ldr   r0, [r0, #0]
1C40      adds  r0, r0, #1
4905      ldr   r1, [pc, #14]
6008      str   r0, [r1, #0]
```

The additional instructions are required for loading the variable addresses; see Global and Static Variables for more information. In this case the inline assembler is effectively a pseudoinstruction, generating five actual assembler instructions. The output is the same as if it were written in C, so in this case there is no advantage to using inline assembler.

The embedded assembler method looks like this:

```
__asm void AddGlobals(void)
{
    extern foo
    extern bar

    LDR  r0, =foo
    LDR  r1, =bar
    LDR  r0, [r0]
    ADD  r0, r0, #1
    STR  r0, [r1]
    BX   lr
}
```

In this case the resultant code is the same as for the inline assembler method.

5.4 Function Arguments

As noted in Function Arguments and Result, the Procedure Call Standard for ARM Architecture allocates registers R0 – R3 for passing arguments to a function, and R0 for passing a function result. If the number of arguments is greater than four, the first four arguments are placed in the registers and the rest are pushed onto the stack.

So if you limit the number of arguments to four, you can write assembler to directly access the registers that have those arguments. The following example shows multiple ways to implement a function to calculate the sum of four arguments:

```
uint32 addFunc(uint32 a, uint32 b, uint32 c, uint32 d)
{
    return a + b + c + d;
}
```

gcc:

```
uint32 addFunc(uint32 a, uint32 b, uint32 c, uint32 d)
{
    /* define return value in R0 */
    /* does not overwrite input argument 'a' in R0 */
    register uint32 rtnval asm("r0");

    /* the arguments are in registers R0 - R3 */
    asm volatile ("add r0, r0, r1\n"
                  "add r0, r0, r2\n"
                  "add r0, r0, r3");

    return rtnval; /* return value in R0 */
}
```

MDK embedded assembler:

```
__asm uint32 addFunc(uint32 a, uint32 b, uint32 c, uint32 d)
{
    ; the arguments are in registers R0 - R3
    ADD r0, r0, r1
    ADD r0, r0, r2
    ADD r0, r0, r3

    ; return value in R0
    BX  lr
}
```

6 Special-Function Instructions

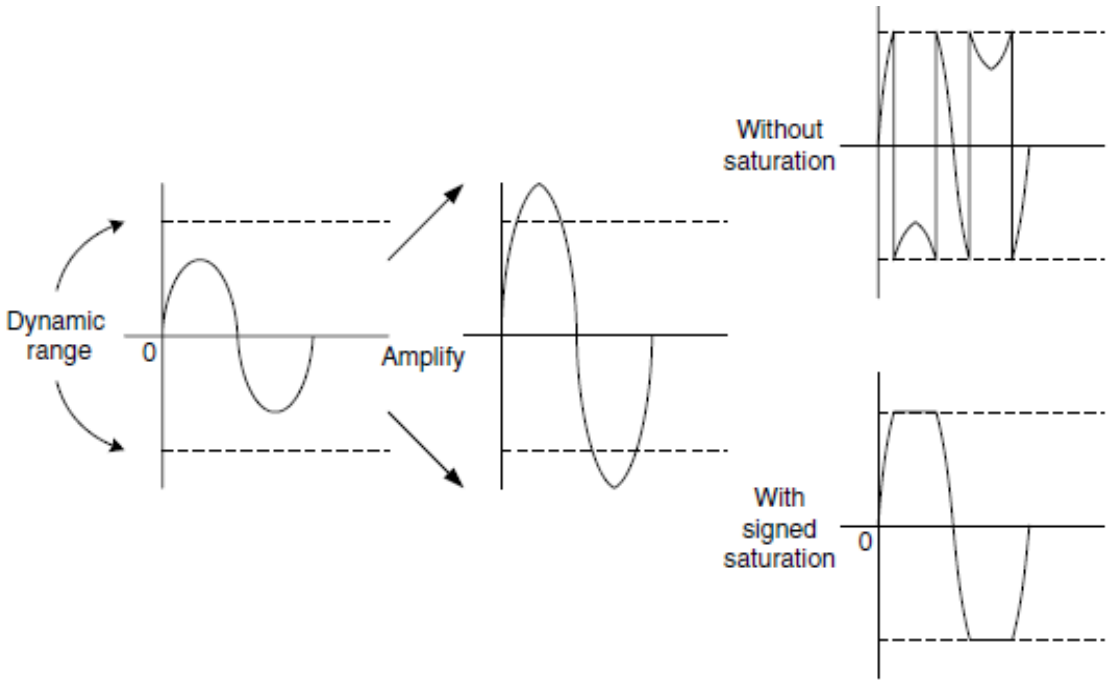
Some CPUs have special-function instructions which are not normally used by C compilers; this type of instruction can be accessed with C intrinsic functions or in some cases can only be accessed using assembler. This section explains how to use C intrinsic functions and mixed C and assembler to more easily gain access to these instructions.

Let us look at the PSoC 5 Cortex-M3 saturation instructions as an example; for other special-function instructions see C Documentation.

6.1 Saturation Instructions

Saturation is commonly used in signal processing, for example when a signal is amplified, as Figure 6 shows. Suppose we are using a 16 bit ADC and are interested in just the 12 LS bits. After amplification, if the value is adjusted by simply removing the unused MS bits, overflow may seriously distort the resulting signal. Saturation avoids overflow and reduces distortion.

Figure 6. Saturation Operation



Saturation can be done in C using multiple comparison and if-else statements, but the Cortex-M3 has two assembler instructions that make the process far more efficient: SSAT and USAT for signed and unsigned, respectively. These instructions work as follows:

6.1.1 SSAT Instruction

The SSAT instruction saturates to the signed range $-2^{n-1} \le x \le 2^{n-1}-1$:

- if the value to be saturated is less than -2^{n-1}, the result returned is -2^{n-1}
- if the value to be saturated is greater than $2^{n-1}-1$, the result returned is $2^{n-1}-1$
- otherwise, the result returned is the same as the value to be saturated.

6.1.2 USAT Instruction

The USAT instruction saturates to the unsigned range $0 \le x \le 2^{n-1}$:

- if the value to be saturated is less than 0, the result returned is 0
- if the value to be saturated is greater than 2^{n-1}, the result returned is 2^{n-1}
- otherwise, the result returned is the same as the value to be saturated

6.1.3 Syntax

op Rd, #n, Rm

where:

- op is one of the following:
  - SSAT saturates a signed value to a signed range
  - USAT saturates a signed value to an unsigned range
- Rd is the destination register.
- n specifies the bit position to saturate to:
  - n ranges from 1 to 32 for SSAT
  - n ranges from 0 to 31 for USAT
- Rm is the register containing the value to saturate.

Note: The SSAT and USAT instructions operate on a 32-bit value in the input register. The corresponding C variable should be of type int, int32 or uint32. Sign extension may be required before executing the saturation instruction.

6.2 Intrinsic Functions

In C, an intrinsic function has the appearance of a function call but is replaced during compilation by a specific sequence of one or more assembler instructions. The ARM Cortex Microcontroller Software Interface Standard (CMSIS) library includes a set of intrinsic functions for most of the Cortex special-function assembler instructions. After a PSoC Creator project is built, you can find these functions in the Workspace Explorer window in the folder Generated Source > PSoCx > cyboot > *core_cmInstr.h*.

The saturation intrinsics look like this:

```
__SSAT(ARG1, ARG2)
__USAT(ARG1, ARG2)
```

where ARG1 is the input value to be saturated and ARG2 is the bit position to saturate to. Call the functions as follows:

```
int data_unsat = -1L;
int data_sat = __USAT(data_unsat, 8);
```

In this example, we saturate data_unsat to 8 bits, unsigned. If the value of data_unsat exceeds 255 (0xFF), the result is saturated to 255 (0xFF) and is stored in data_sat. If the value of data_unsat is negative, 0 is stored in data_sat.

6.3 Assembler

You can also use the techniques described in Mixing C and Assembler Code to insert special-function instructions, as Table 5 shows.

Table 5. Using Saturation Instructions in Mixed C and Assembler

gcc Example	MDK Example

`void main()`
`{`
`  register int data_unsat asm("r0");`
`  register int data_sat asm("r3");`
 	
`  asm("ssat r3, 8, r0");`
`  . . .`
`}`
	`void main()`
	`{`
	`  int data_unsat;`
	`  int data_sat;`
	
	`  __asm("ssat data_sat, 8, data_unsat");`
	`  . . .`
	`}`

In this example, we saturate data_unsat to 8 bits, signed. With 8-bit signed saturation the value can range from -128 to +127. So if the value is less than -128, the result is -128 and if the value is greater than +127, the result is +127. So the result is saturated to 0x7F in the positive direction and 0x80 in the negative direction.

We can use the saturation instructions to saturate a value to the required number of saturation bits. USAT can be used with an ADC configured in single ended mode and SSAT can be used with an ADC configured in differential mode.

7 Packed and Unpacked Structures

In most embedded systems, data is transmitted in a byte-by-byte fashion, for example with a UART or I^2C port. (The SPI protocol is an exception, for details see one of the PSoC Creator SPI Component datasheets.) With 8-bit CPUs complex data structures can be transmitted and received byte by byte and the result will exactly match the original. However with larger CPUs (16-bit, 32-bit, etc.) this is not necessarily true. Let us examine in detail why this is so, using the PSoC 4 Cortex-M0 and PSoC 5LP Cortex-M3 CPUs as examples.

The 32-bit Cortex CPUs in PSoC access memory as 32-bit words, and therefore they work most efficiently when data is stored on 32-bit boundaries, that is, where the two LS address bits are zero. If for example a 16-bit or 32-bit variable is saved starting at an odd address, where the LS bit of the address is 1, then two 32-bit memory reads are required to read it, and two read-modify-write cycles are required to write it. This can significantly impact execution speed.

Unfortunately, in C it is easy to create structures where words are on odd boundaries. Consider the following example:

```
struct myStruct
{
    uint8  m1; /* stored on a 32-bit boundary, address = ...xx00 */
    uint32 m2; /* stored on an 8-bit boundary, address = ...xx01 */
}
```

The above is called a **packed** structure, because the structures are placed in memory byte-by-byte regardless of address boundary considerations. Compilers for 8-bit CPUs usually generate packed structures. The gcc and MDK compilers as a default for the Cortex CPUs save structures in **unpacked** format, where the address is determined by the size of the structure member. For example:

```
struct myUnpackedStruct
{
    uint8  m1; /* stored on a 32-bit boundary, address = ...xx00 */
    /* 3 unused filler bytes */
    uint32 m2; /* stored on a 32-bit boundary, address = ...yy00 */
}
```

An unpacked structure can be accessed more efficiently but is larger, which may be a problem with devices with limited SRAM. But a more serious problem can occur when for example an unpacked structure is transmitted byte-by-byte and the receiver saves the bytes in a packed structure – the data becomes corrupted causing hard-to-find system-level defects.

There are several ways to correct this problem in code; the easiest is to simply optimize the order of structure member declarations. For example, we could reorder the original structure as:

```
struct myStruct
{
    uint32 m2; /* stored on a 32-bit boundary, address = ...xx00 */
    uint8  m1; /* stored on a 32-bit boundary, address = ...yy00 */
}
```

Now the structure is packed and each of its members' addresses are on 32-bit boundaries.

7.1.1 Compiler Considerations

For both gcc and MDK compilers, by default, structures are unpacked according to the following rules:

- A char or uint8 (one byte) is 1-byte aligned
- A short or uint16 (two bytes) is 2-byte aligned; LS address bit is 0
- A long or uint32 (four bytes) is 4-byte aligned; two LS address bits are 00
- A float (four bytes) is 4-byte aligned; two LS address bits are 00
- Any pointer, e.g., char *, int * (four bytes) is 4-byte aligned; two LS address bits are 00.

It is possible to force a structure to be packed, using the following syntax:

gcc:

```
struct myStruct
{
    . . .
} __attribute__ ((packed));
```

MDK:

```
__packed struct myStruct
{
    . . .
};
```

Note: It is recommended to use the `packed` statement in structure definitions only. It should not be used in declarations of actual structure variables nor should it be used in typedef declarations.

8 Compiler Libraries

For both gcc and MDK compilers, replacing C standard library function calls with equivalent C statements can significantly reduce memory usage. For example, consider the following C fragment:

```
#include <math.h>
uint32 a, b;
a = 5;
b = pow(a,3);
```

Table 6 shows the Flash and SRAM memory consumption for both compilers for PSoC 5LP and PSoC 4:

Table 6. Memory Consumption With a pow() Function Call

PSoC 5LP	gcc	MDK
Flash	8939	7696
SRAM	405	196

PSoC 4	gcc	MDK
Flash	14374	7700
SRAM	364	312

If the call to the pow() library function is replaced with the following equivalent code, you use a lot less memory, as Table 7 shows:

```
b = a * a * a;
```

Table 7. Memory Consumption Without Using a pow() Function Call

PSoC 5LP	gcc	MDK
Flash	1582 (-82.3%)	1444 (-81.2%)
SRAM	301 (-25.7%)	200 (-32.4%)

PSoC 4	gcc	MDK
Flash	1198 (-91.7%)	1004 (-87.0%)
SRAM	252 (-30.8%)	216 (-30.8%)

The reason for the size reduction is that by ANSI C definition the pow() function takes arguments of type double and returns a type double. When you call this function with integers they are automatically cast to the proper type before and after the function call, and this requires a lot of code to implement.

With PSoC Creator 3.0, a choice of gcc libraries is available: newlib and newlib-nano. The newlib-nano library cuts some less-used features from the standard C library functions, to reduce memory usage.

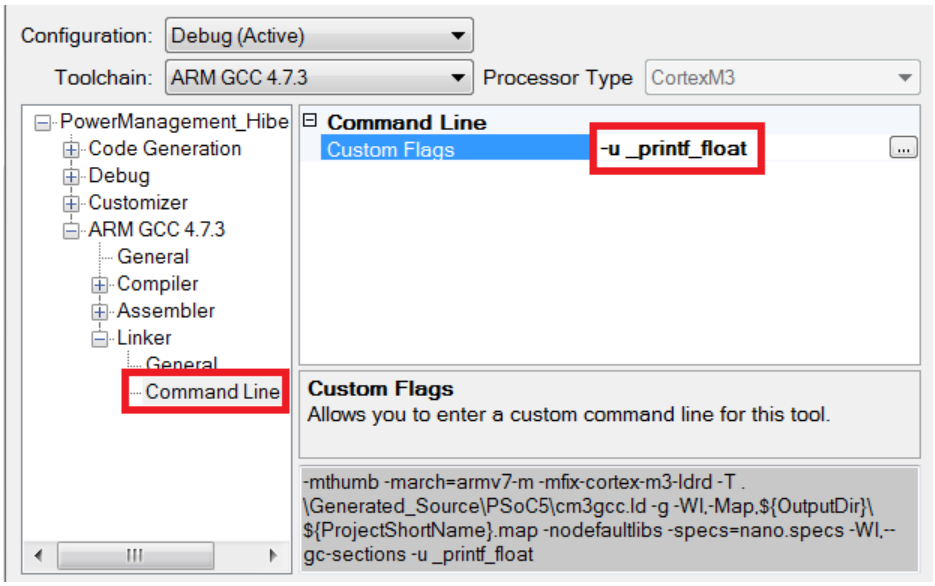
Note: One of features removed from newlib-nano is floating-point support in printf(), which may cause problems if you intend to display floating-point values. For example, consider the following code fragment:

```
char My_String[30];
float My_Float = 3.14159;
sprintf(My_String, "Value of pi is: %.2f to 2dp", My_Float);
```

With newlib-nano, the string `"Value of pi is: to 2dp"` is created in My_String; the expected value 3.14 is not included. There are two possible work-arounds:

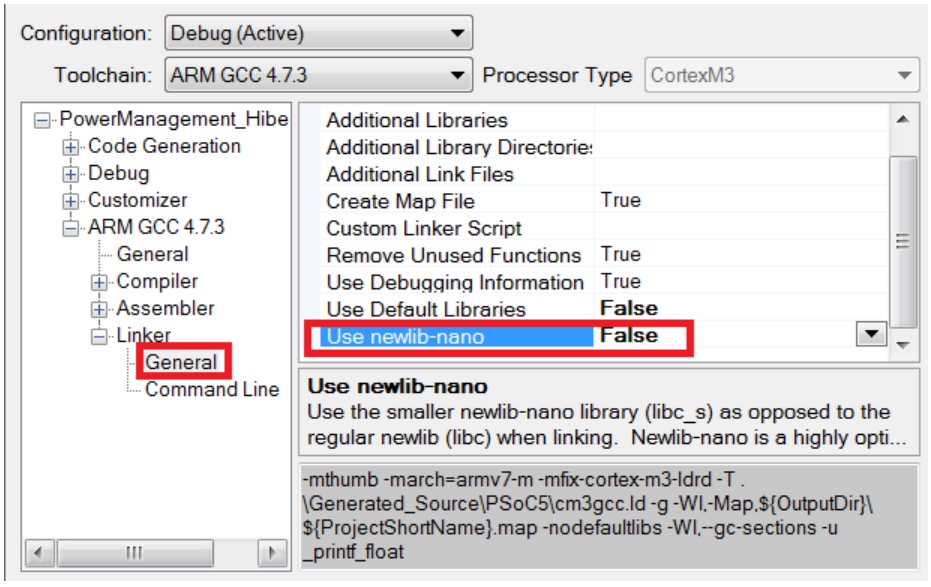
1. Enable floating-point formatting support in newlib-nano, as Figure 7 shows. This feature is disabled as a default. Enabling it increases flash usage by 10K to 15K bytes.

Figure 7. Enable floating-point formatting support in newlib-nano



2. Change the library to the full-featured newlib, as Figure 8 shows. Note that the **Use Default Libraries** option must also be set to False. The default is to use newlib-nano; changing to newlib increases flash usage by 25K to 30K bytes, and increases SRAM usage by approximately 2K bytes.

Figure 8. Disabling newlib-nano



MDK also offers a reduced-function library called Microlib. For more information see the MDK documentation.

9 Placing Code and Variables

This section shows how to place C code and variables into custom locations in memory. There are a number of reasons to do this, see Define Custom Locations for examples.

To effectively use the methods described in this section, it is important to understand the CPU architectures on which they are based – see Address Map for details.

9.1 Linker Script Files

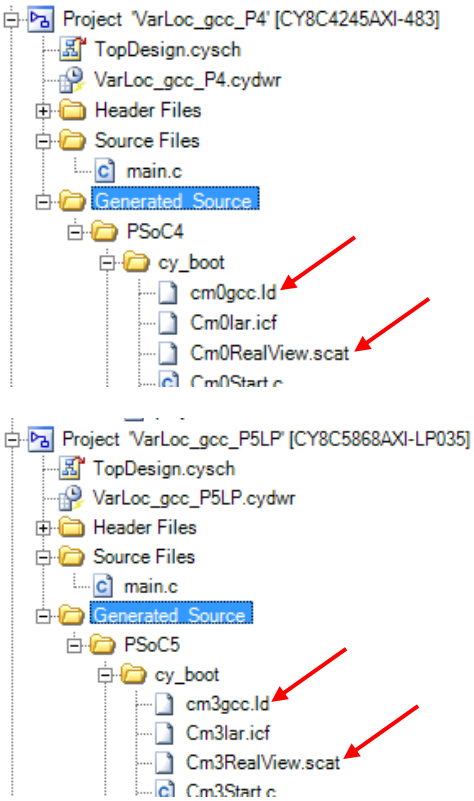
To place code and variables in custom locations, you must know how to modify linker script files. This section shows the basics of how linker script files control the use of memory in PSoC 4 and PSoC 5LP. Details can be found in your gcc or MDK documentation.

After your PSoC Creator project is built, the default linker script files can be found in the Generated Source folder, as Figure 9 shows. For gcc the linker script file is of type *.ld*, and for MDK the linker script file is of type *.scat* (for "scatter").

Note: Linker script files are automatically generated by PSoC Creator at project build time, and changes that you make to those files may be overwritten on the next build. You can instruct PSoC Creator to use a custom script file, using the PSoC Creator menu Project > Build Settings > Linker > General > Custom Linker Script.

If you use a custom linker script file, it is a best practice to add it to the project (menu Project > Existing Item…) and save it in the project folder. A custom *.scat* file must be saved in the PSoCx folder under Generated Source.

Figure 9. PSoC Creator Linker Script Files



9.1.1 Linker Script File for gcc

An *.ld* file has two major commands: `MEMORY {}` and `SECTIONS {}`. The `MEMORY` command describes the type, location and usage of all physical memory in the PSoC. For example, for a PSoC 4 with 32 KB flash and 4 KB SRAM:

```
MEMORY
{
  rom (rx) : ORIGIN = 0x0, LENGTH = 32768
  ram (rwx) : ORIGIN = 0x20000000, LENGTH = 4096
}
```

The region `rom` describes the PSoC flash and the region `ram` describes the PSoC SRAM. The letters "`rwx`" are memory attribute indicators: read, write, and execute, respectively. All origin and length units are in bytes, and the values can be in decimal or hexadecimal. PSoC 5LP is similar; the `ram ORIGIN` value describes the SRAM crossing the Cortex-M3 Code / SRAM region boundary (Figure 4). For example, for a PSoC 5LP with 256 KB flash and 64 KB SRAM:

```
MEMORY
{
  rom (rx) : ORIGIN = 0x0, LENGTH = 262144
  ram (rwx) : ORIGIN = 0x20000000 - (65536 / 2), LENGTH = 65536
}
```

The `SECTIONS` command lists all of the sections in address order, for example:

```
SECTIONS
{
  .text: { ... }
  .rodata: { ... }
  .ramvectors: { ... }
  .noinit: { ... }
  .data: { ... }
  .bss: { ... }
  .heap: { ... }
  .stack: { ... }
}
```

(Not all of the sections are shown above, only the major ones.) Figure 10 shows where these sections are placed in PSoC 4 and PSoC 5LP flash and SRAM:

- .text: executable code
- .rodata: `const` variables; initialization data
- .ramvectors: Cortex exception vectors table
- .noinit: variables that are not initialized
- .data: variables that are explicitly initialized
- .bss: variables that are initialized to 0
- heap
- stack

A closer examination of the linker script file shows that many of the sections end with a **region** statement. This statement tells the linker the memory region in which to place the section, for example:

```
.text: { ... } >rom
.data: { ... } >ram AT>rom
.heap: { ... } >ram
```

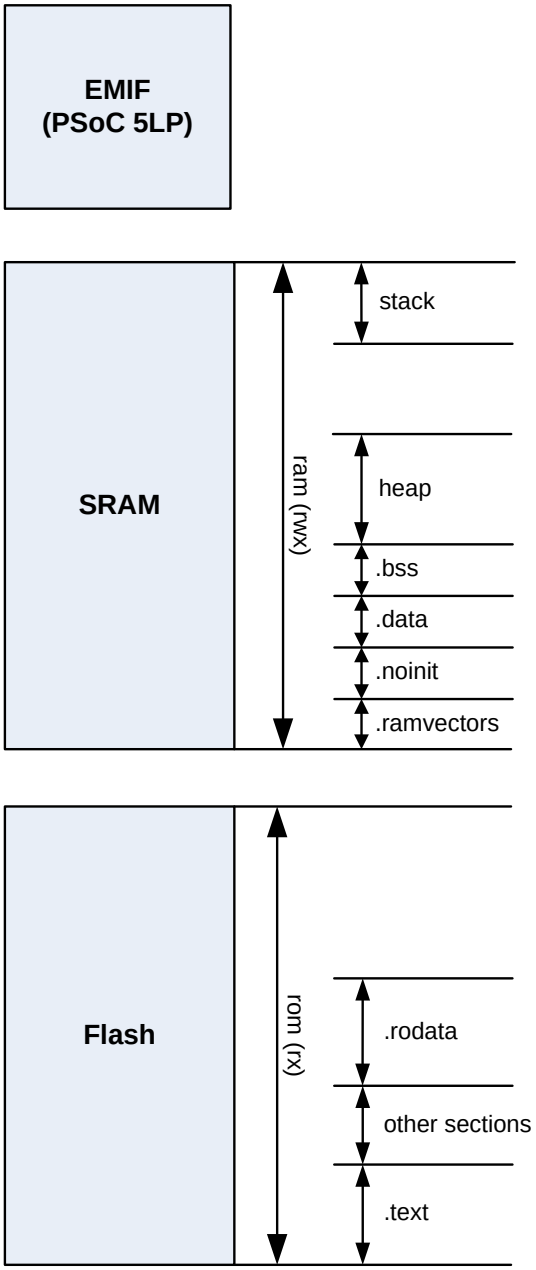
The `AT` statement enables explicit initialization of variables; see Variable Initialization.

Note: For PSoC 5LP, the placement of the sections in SRAM are indeterminate relative to position of the code SRAM / upper SRAM boundary (0x20000000; see Figure 4.

Note: For most applications it can be assumed that the stack is in upper SRAM and the other sections are all in code SRAM. This can be changed; see Modify the Linker Script File.

Complete documentation of *.ld* file usage can be found in your gcc documentation.

Figure 10. SECTIONS Command and PSoC Memory



9.1.2 Linker Script File for MDK

A *.scat* (for "scatter") file has no commands; instead it defines regions and sections. It has a single **load region** called `APPLICATION {}`. The load region contains several **execution regions**, which in turn contain one or more **section attributes**, for example:

```
APPLICATION ...  // load region
{
  CODE ...       // execution
region
  {
    * (+RO)      // section
attribute
  }
  ISRVECTORS ...
  {
    * (.ramvectors)
  }
  NOINIT_DATA ...
  {
    * (.noinit)
  }
  DATA ...
  {
    .ANY (+RW, +ZI)
  }
  ARM_LIB_HEAP ... { }
  ARM_LIB_STACK ... { }
}
```

(Not all of the execution regions and section attributes are shown above, only the major ones.) Figure 11 shows where the regions and sections are placed in PSoC 4 and PSoC 5LP flash and SRAM. Special sections RO, RW, and ZI are defined as follows:

- RO: all code, `const` variables, and initialization bytes for the RW section
- RW: all variables that are explicitly initialized; see Variable Initialization
- ZI: all variables that are initialized to zero; see Variable Initialization

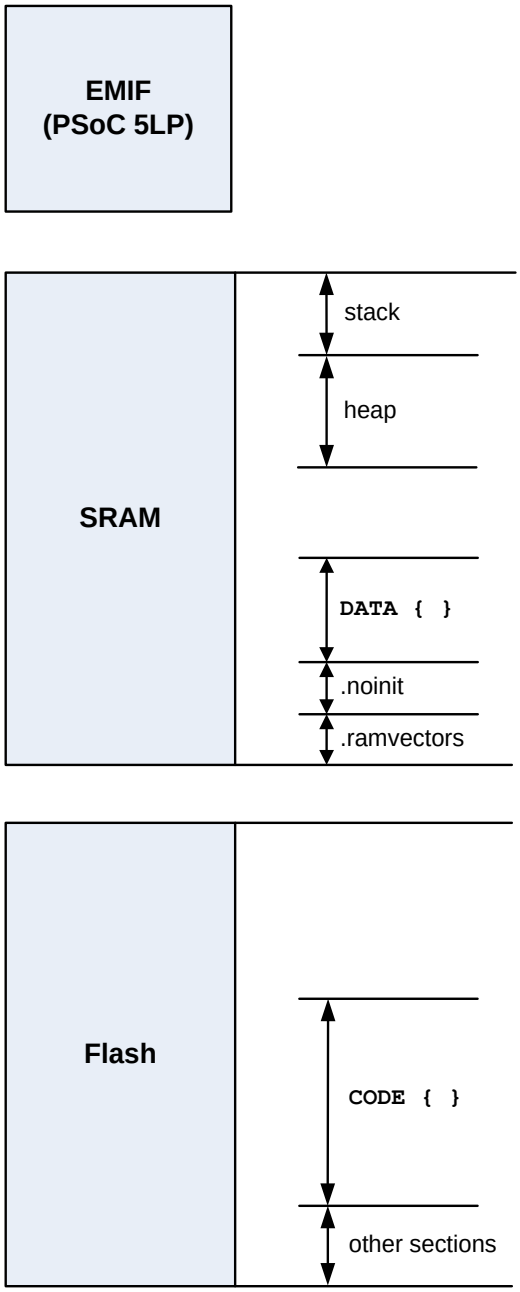
Other attributes such as `.noinit` cause placement of code or variables that match that attribute; see Define Custom Locations.

Note: For PSoC 5LP, the placement of the regions and sections in SRAM is indeterminate relative to the position of the code SRAM / upper SRAM boundary (0x20000000; see Figure 4.

Note: . For most applications it can be assumed that the stack and heap are in upper SRAM and the other sections are all in code SRAM. This can be changed; see Modify the Linker Script File.

Complete documentation of *.scat* file usage can be found in your MDK documentation.

Figure 11. *.scat* File Sections and PSoC Memory



The following points are valid for both linker file types:

1. Only global and C static variables are included. C automatic variables are handled differently; see Automatic Variables.
2. The size of the heap and stack are defined in the PSoC Creator project DWR window, **System** tab. The stack pointer is initialized to the highest SRAM address plus 1, and the stack grows downward. The heap, which is used by C functions such as malloc() and free(), grows upward from its base.
3. Although the heap has a defined size in the DWR window, in practice it can use all of SRAM between the sections in SRAM and the current value of the stack pointer. Similarly, the stack can grow downward beyond the defined stack section. If the stack starts to overlap the memory regions below it, hard-to-find defects can occur. One way to detect stack overflow is to add code to each function to check the current value of the stack pointer.

9.1.3 Variable Initialization

When placing global and static variables in custom locations, it is important to understand how they are initialized. For example, consider these global variable definitions:

```
uint8 foo = 5;
uint16 myArray[10] = {1234, 12, ... };
```

Because these variables are located in SRAM, their values are undefined when the PSoC is powered up. To properly initialize them the values are saved in flash and the C startup code, i.e., the code that is executed before main(), copies the values from flash to SRAM. The values in flash are in the `.rodata` section for gcc (Figure 10) or the `CODE (RO)` section for MDK (Figure 11). They are copied into the variables in the `.data` section for gcc or the `DATA (RW)` section for MDK – explicitly initialized variables must be located in these sections.

Global and static variables that are not explicitly initialized are set to zero by the C startup code. They must be located in the `.bss` section (gcc) or the `DATA (ZI)` section (MDK).

Global and static variables that are located outside the above sections are not initialized and their initial values are undefined – they must be initialized in your code. Explicit initializations are ignored.

9.1.4 Map File

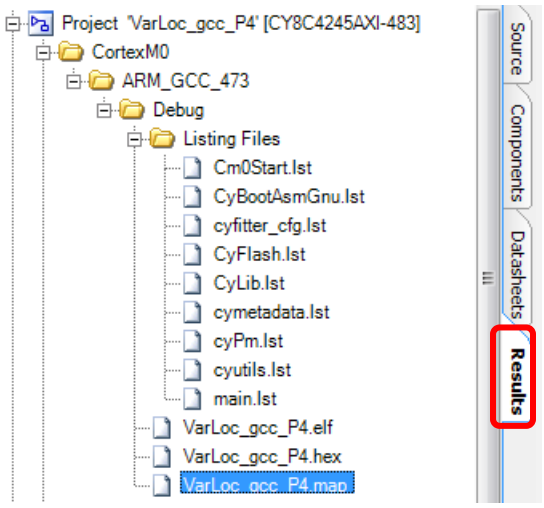
The gcc and MDK linkers both have an option to produce a *.map* file when a project is built. You can find the file in the **Results** tab in the Workspace Explorer window, as Figure 12 shows.

The *.map* file shows where in memory all code modules and variables have been placed by the linker. You should review it after a build operation and confirm that:

- All code and variables have been placed in the expected locations, and
- There are no section overlaps.

For MDK, the linker's default is to produce a *.map* file with no symbols, which makes it difficult to determine where your code and variables have been placed. To include the symbols, add `--symbols` to the linker command line, using the PSoC Creator menu Project > Build Settings > Linker > Command Line > Custom Flags.

Figure 12. *.map* File in PSoC Creator



Now that we have seen the basics of how linker script files work, we can examine how to use them to place code or variables in custom locations in PSoC memory.

9.2 Placement Procedure

To place C functions or variables (including arrays and structures) in custom locations, do the following:

1. Define the custom locations.
2. In the C source code, declare the functions and variables that are to be located, along with their custom sections.
3. Build the PSoC Creator project. Copy the generated linker script file to a custom file, then modify it to add and locate the sections from step 2.
4. Rebuild the PSoC Creator project. Review the *.map* file and confirm that the custom locations have been filled correctly and that there are no section overlaps.

Let us examine each of these steps in detail.

9.2.1 Define Custom Locations

Before using custom locations, you should understand clearly the reasons why you want to use them. For example, do you want to:

- Place a function, or a variable of type `const`, in a custom location in flash?
- Place a function in SRAM, for possible faster execution? If so, note that in PSoC 4 and PSoC 5LP flash accesses are almost as fast as SRAM accesses, so significant performance gains may not be realized. See Cortex-M0 in PSoC 4 and Cortex-M3 in PSoC 5LP for details.
- Place a variable such that it is not initialized by C startup code? This is typically used to maintain a variable's state through a device reset (except for a power-cycle reset).
- Place variables in PSoC 5LP upper SRAM, for bit band access?
- Place variables in other custom locations in SRAM?
- Place variables in PSoC 5LP EMIF memory?

Your answers to the above questions will help you to determine the addresses of your custom locations.

9.2.2 Declare Functions and Variables

Once you have determined the custom location addresses, declare your functions and variables. In the declarations, add the sections in which they will reside, using the `__attribute__` keyword (two underscore characters before and after the "attribute"):

```
uint8 foo __attribute__ ((section(".MY_section")));
```

This keyword can be used with both gcc and MDK. PSoC Creator provides a convenient macro `CY_SECTION` to simplify the above statement. Following are some examples of its usage:

```
uint8 foo CY_SECTION(".MY_section"); /* no explicit initialization, = 0 */

uint8 foo CY_SECTION(".MY_section") = 10; /* explicit initialization */

/* declare a function's section in the prototype only, and not in the actual
function */
uint16 MyFunction(char *x) CY_SECTION(".MY_section");

/* CYISR is a PSoC Creator macro to define an interrupt handler function.
   See Related Documents for more information. */
CYISR(MyFunction) CY_SECTION(".MY_section");
```

PSoC Creator also provides a convenient macro `CY_NOINIT`, which places a variable in the `.noinit` section; see Figure 10 and Figure 11:

```
/* no initialization by C startup code, initial value is undefined */
uint8 foo CY_NOINIT;
```

9.2.3 Modify the Linker Script File

The final task is to modify your project's linker script file, to declare where you want the previously defined sections to be placed.

For gcc, modify the linker script *.ld* file – change the `SECTIONS {}` command and possibly the `MEMORY {}` command. A common way to modify it would be to add statements for EMIF memory, or to split the SRAM, for example:

```
MEMORY
{
  rom (rx) : ORIGIN = 0x0, LENGTH = 262144
  coderam (rwx) : ORIGIN = 0x20000000 - (65536 / 2), LENGTH = (65536 / 2)
  upperram (rwx) : ORIGIN = 0x20000000, LENGTH = (65536 / 2)
  EMIF (rwx) : ORIGIN = 0x60000000, LENGTH = 0x1000000
}
```

Note: Changing the SRAM region names will cause errors in some section definitions in the default file, so you must change each section definition as needed. For example, change `.stack: { ... } >ram` to `.stack: { ... } >upperram`.

To locate a section, add a section definition to the `SECTIONS {}` command. The syntax for a section definition is:

```
.MY_section <address> <(NOLOAD)> : <alignment>
{
  *(.MY_section)
} ><memory region>
```

Note that the section definition name and the name of the section within that section can be the same. Having the first character of the name be a period "." is not required but is a common convention.

Use a memory region name from the `MEMORY {}` command, as described previously.

You can also just place your section within an existing section, for example:

```
.data:
{
  ...
  *(.MY_section)
  ...
} >ram AT>rom
```

For MDK, modify the linker script *.scat* file. The procedure is similar to that for gcc but simpler – there is no `MEMORY {}` or `SECTIONS {}` command to change. Instead, just add your execution region, for example:

```
MY_REGION <address> <UNINIT> <length>
{
  * (.MY_section)
}
```

You can also just place your section within an existing section, for example:

```
DATA
{
  * (.MY_section)
  .ANY (+RW, +ZI)
}
```

9.3 Example

As noted previously, there are several different applications for custom locations. Let us examine one of them as an example of Placement Procedure. In this example we will place an array in PSoC 5LP upper SRAM so that it can be accessed by the Cortex-M3 bit band feature – see Cortex-M3 Bit Band.

First, we define the array to occupy a section that we will call ".bitband":

```
uint8 myArray[10] CY_SECTION(".bitband");
```

Then we must modify the linker script file to tell it that we want to place the .bitband section in upper SRAM, starting at address 0x20000000. Table 8 shows how to do this for gcc (*.ld* file) or MDK (*.scat* file):

Table 8. Example Modifications of Linker Script Files

gcc Example, *.ld* File

```
/* put our .bitband section between the
   .heap section in code SRAM and the
   .stack section in upper SRAM */
.heap (NOLOAD) :
{
  . = _end;
  . += 256;
  __cy_heap_limit = .;
} >ram

.bitband 0x20000000 (NOLOAD) :
{
  *(.bitband)
} >ram

.stack (__cy_stack - 256) (NOLOAD) :
{
  __cy_stack_limit = .;
  . += 256;
} >ram
```

MDK Example, *.scat* File

```
/* put our BITBAND execution region between the
   DATA exection region in code SRAM and the
   heap in upper SRAM */
DATA +0
{
  .ANY (+RW, +ZI)
}

BITBAND 0x20000000 UNINIT
{
  * (.bitband)
}

ARM_LIB_HEAP (0x20000000 +
              (65536 / 2) - 256 - 256) EMPTY 256
{
}

ARM_LIB_STACK (0x20000000 + (65536 / 2)) EMPTY -256
{
}
```

Build the project, then check the *.map* file and confirm that the array has been located correctly and that there are no section overlaps.

Note the initial values of myArray are undefined; they must be initialized in your code. See Variable Initialization for details.

See Cortex-M3 Bit Band for details on how to do bit-level access of variables located in the bit band region.

9.4 General Considerations

When declaring code and variables in custom sections, keep the following in mind:

- Explicitly initialized variables must be placed in the `.data` section (gcc, Figure 10) or the `DATA` execution region (MDK, Figure 11). See Variable Initialization for details.

  Similarly, variables for which there is no explicit initialization and which you expect to be auto-initialized to zero must be placed in the `.bss` section (gcc) or the `DATA` execution region (MDK). (The MDK compiler automatically gives variables an `RW` or `ZI` section attribute, depending on whether or not they're explicitly initialized.)

  Variables that are not placed in the above sections are not initialized. Explicit initializations are ignored.

- Functions that are to be located in SRAM must be placed in the `.data` section (gcc, Figure 10) or the `DATA` execution region (MDK, Figure 11).

- It is more efficient to have all constant data, for example fixed data tables in arrays, be located in flash, however the default is to place them in SRAM. To force placement of a variable in flash, set its type to const and explicitly initialize it, for example:

  ```
  uint32 const var_in_flash = 0x12345678;
  ```

- If you are using custom locations in flash, note that the PSoC Creator bootloader uses the top one or two rows of flash to store information about bootloadable files. For more information see the Bootloader Component datasheet.
- With MDK, the easiest way to put a variable at any specified address is to use the `__attribute__((at(address)))` variable attribute. For example:

```
uint32 const var_in_flash[] __attribute__((at(0x300))) = { . . . };
```

The linker defines a special section and places the variable at the desired address, adjusting the placement of other code and variables as needed, as the following *.map* file snippet shows:

```
Symbol Name                 Value        Ov Type  Size  Object(Section)
. . .
.text                       0x000002f4   Section     0  indicate_semi.o(.text)
.text                       0x000002f4   Section     0  exit.o(.text)
.ARM.__AT_0x00000300        0x00000300   Section    12
main.o(.ARM.__AT_0x00000300)
.text                       0x0000030c   Section     0  init_alloc.o(.text)
.text                       0x00000394   Section     0  h1_free.o(.text)
. . .
```

For more information, see C Documentation.

- The `.ramvectors` section should always be at the bottom of SRAM and the stack section should always be at the top of SRAM.

Complete documentation of *.ld* file usage can be found in your PSoC Creator installation folder, typically:
C:\Program Files\Cypress\PSoC Creator\3.0\PSoC Creator\import\gnu_cs\arm\4.7.3\share\doc\gcc-arm-none-eabi\pdf\ld.pdf

Complete documentation of *.scat* file usage can be found in your MDK installation folder, typically:
C:\Keil\ARM\Hlp\armlink.chm and *C:\Keil\ARM\Hlp\armlinkref.chm*

9.5 EMIF Considerations (PSoC 5LP Only)

It is possible to place variables in the external memory (EMIF) supported by PSoC 5LP, using the techniques described previously, but there are some restrictions:

- You must have an EMIF Component placed on your PSoC Creator project schematic. Note that the external memory address, data and control lines can use a significant number of device pins – plan your design accordingly. See the EMIF Component datasheet for details.
- You cannot access the external memory until the EMIF API function EMIF_Start() is called. So you can't initialize EMIF variables in C startup; you must initialize them after the code reaches main() and EMIF_Start() is called.
- The EMIF supports 8-bit and 16-bit memories; placement and access of different size variables may be a consideration. It is recommended to align 16-bit and 32-bit variables and structure members on 2-byte and 4-byte boundaries, respectively.
- Code can be executed from EMIF, but only with 16-bit external memories. The code executes much more slowly than from device internal flash or SRAM. It is also difficult to initialize code in external memory. In general, having code in external memory is not recommended.

10 Cortex-M3 Bit Band (PSoC 5LP Only)

As indicated in Address Map, Figure 2 and Figure 4, the PSoC 5LP Cortex-M3 has a bit band feature, where accessing an address in an **alias region** results in bit-level access in the corresponding bit band region. This lets you quickly set, clear or test a single bit in the first 1 Mbyte of the region. For a given bit in a given byte address, the formula for the corresponding alias address is:

alias_address = 0x22000000 + 32 * (byte_address – 0x20000000) + 4 * bit_number

So for example if you want to set bit 5 in address 0x20000001, write a 1 to address:

0x22000000 + 32 * 1 + 4 * 5 = 0x22000034.

Similarly, to clear the bit, write a 0 to the alias address. To test the bit, read the alias address and test bit 0.

Note: In addition to the SRAM region, the Cortex-M3 supports bit band for the peripheral region, in which all of the PSoC 5LP registers are located. However, peripheral region bit band is not supported in the PSoC 5LP. Writing to the peripheral region's bit band alias region (0x42000000 – 0x43FFFFFF) may give unpredictable results in the PSoC 5LP registers and is not recommended.

To use the bit band feature with a variable, first place the variable in upper SRAM using the techniques described in Placing Code and Variables. Then, define macros to calculate and use the corresponding addresses in the bit band alias region:

```
#define  BIT_BAND_ALIAS_BASE  0x22000000
/* 'byte' should be a number 0x20000000 to 0x200FFFFF
   'bit' should be a number 0 to 7 */
#define  BIT_BAND_ALIAS_ADDR(byte, bit)   ((BIT_BAND_ALIAS_BASE + \
                                           32 * ((uint32)(byte) - \
                                                 CYREG_SRAM_DATA_MBASE) + \
                                            4 * (uint8)(bit))
/* 'a' should be an address (uint32 *) */
#define  GET_BIT(a, bit)          *(uint32 *)BIT_BAND_ALIAS_ADDR(a, bit)
/* 'val' should be 0 or 1 */
#define  SET_BIT(a, bit, val)     GET_BIT(a, bit) = (uint32)(val)
#define  TEST_BIT(a, bit, val)   (GET_BIT(a, bit) == (uint32)(val))
```

You can then use the macros to set or test a bit:

```
SET_BIT(&foo, 5, 1); /* set bit 5 of foo */
if (TEST_BIT(&foo, 5, 1)) { ... } /* test bit 5 */
```

In general, it is more efficient to set or clear a bit with the bit band technique than by reading, modifying and writing the variable, as Table 9 shows. When bit band is used, the read-modify-write cycle is done internally by the CPU, thereby saving one instruction.

Table 9. Assembly Language for Bit Band vs Direct Set

C Code	Assembler Code

`/* direct set bit */`
`foo |= (1 << 5);`
	`; R3 = address of foo`
	`ldr  r2, [r3]`
	`orr  r2, r2, #32`
	`str  r2, [r3]`

`/* use bit band */`
`SET_BIT(&foo, 5, 1);`
	`; R3 = bit band alias address for foo bit 5 */`
	`mov  r2, #1`
	`str  r2, [r3]`

Note that there is no efficient way use bit banding to toggle a bit. It is possible to do:

```
SET_BIT(&foo, 5,
        GET_BIT(&foo, 5) ^ 1);
```

However it is simpler and just as efficient to do:

```
foo ^= (1 << 5);
```

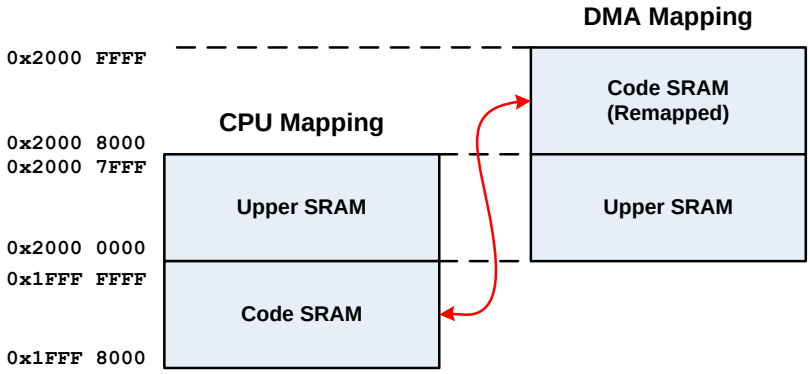
11 DMA Addresses (PSoC 5LP only)

This section assumes that you know how to use the direct memory access (DMA) controller in PSoC 5LP. The DMA controller can transfer data from a source to a destination with no CPU intervention. This allows the CPU to handle other tasks while the DMA does data transfers, thereby achieving a “multiprocessing” environment.

The DMA controller is highly flexible and capable of doing complex transfers of data between PSoC memory and on-chip peripherals including ADCs, DACs, the Digital Filter Block (DFB), USB, UART, and SPI. There are 24 independent DMA channels. For more information, see AN52705, Getting Started with PSoC DMA.

In PSoC 5LP, the DMA shares the Cortex-M3 S Bus (Figure 4) with the CPU. However, because the S Bus does not access the Code region, the DMA cannot directly access code SRAM (0x1FFF8000 to 0x1FFFFFFF). PSoC 5LP handles this by implementing remapping so that the DMA can access the code SRAM by accessing corresponding addresses 0x20008000 to 0x2000FFFF, as Figure 13 shows.

Figure 13. DMA Remapping of Code SRAM



Since the DMA is a 16-bit subsystem, when it increments an address only the lower 16 bits are incremented, with rollover. Therefore, the next DMA address following 0x2000FFFF (which is mapped to 0x1FFFFFFF) is 0x20000000. This means that the SRAM still functions as a contiguous 64-KB block of memory for DMA. This is also true for devices with less than 64K SRAM because the SRAM is always centered around 0x20000000.

The remapping is taken into account by the PSoC Creator DMA Component API functions used to set up a DMA channel. If you do not use the API, always set the upper 16 bits of a DMA address for SRAM to 0x2000 regardless of the actual address.

12 Summary

This application note has presented a number of methods to increase the efficiency of your C code for the Cortex CPUs in PSoC 4 and PSoC 5LP. The gcc and MDK compilers supported by PSoC Creator work well for most applications without using these techniques; they are needed for special problems in meeting code size or execution speed requirements.

The methods presented, in no particular order, are:

- Limit the number of function arguments to no more than 4. See Function Arguments and Result.
- Minimize the number of global and static variables. Not only is this a coding best practice but it may reduce code size by reducing the number of address load operations. See Global and Static Variables.
- Use inline or embedded assembler to maximize efficiency in critical sections; see Mixing C and Assembler Code. You can also use this technique, or intrinsic functions, to take advantage of special instructions, especially in the Cortex-M3; see Special-Function Instructions.

  Also see Appendix A for examples of how to write efficient assembler code.
- Be careful when using standard compiler libraries, as they may use a lot of memory; consider using inline code instead. See Compiler Libraries.

- When using structures, pay careful attention to whether they should be packed or unpacked – there are advantages and disadvantages for each. See Packed and Unpacked Structures.
- Place speed critical code in SRAM; see Placing Code and Variables. Note that speed gains using this technique may not be realized.
- Place variables to take advantage of the bit band feature in the PSoC 5LP Cortex-M3; see Cortex-M3 Bit Band and Example.

12.1 Use All of the Resources in Your PSoC

There is one final method available for reducing code size. It is based on the fact that PSoC is designed to be a flexible device that enables you to build custom functions in programmable analog and digital blocks. For example, in PSoC 5LP you have the following peripherals that can act as “co-processors”:

- DMA Controller. Note that the most common CPU assembler instructions are MOV, LDR, and STR, which implies that the CPU spends a lot of cycles just moving bytes around. Let the DMA controller do that instead.
- Digital Filter Block (DFB) – a sophisticated 24-bit sum of products calculator
- Universal Digital Blocks (UDBs). There are as many as 24 UDBs, and each UDB has an 8-bit datapath that can add, subtract, and do bitwise operations, shifts, and cyclic redundancy check (CRC). The datapaths can be chained for word-wide calculations. Consider offloading CPU calculations to the datapaths.
- The UDBs also have programmable logic devices (PLDs) which can be used to build state machines, c.f. the Lookup Table (LUT) Component datasheet. LUTs can be an effective alternative to programming state machines in the CPU using C switch / case statements.
- Analog components including ADCs, DACs, comparators, opamps, as well as programmable switched capacitor / continuous time (SC/CT) blocks from which you can create programmable gain amplifiers (PGAs), transimpedance amplifiers (TIAs), and mixers. Consider doing your processing in the analog domain instead of the digital domain.

PSoC Creator offers a large number of Components to implement various functions in these peripherals. This allows you to develop an effective multiprocessing system in a single chip, offloading a lot of functionality from the CPU. This in turn can not only reduce code size, but by reducing the number of tasks that the CPU must perform, you can reduce CPU speed and thereby reduce power.

For example, with PSoC 5LP a digital system can be designed to control multiplexed ADC inputs, and interface with DMA to save the data in SRAM, to create an advanced analog data collection system with zero usage of the CPU.

Cypress offers extensive application note support for PSoC peripherals, as well as detailed data in the device datasheets and technical reference manuals (TRMs). For more information see Related Documents.

13 Related Documents

13.1 Application Notes

- AN77759 – Getting Started with PSoC 5LP
- AN79953 – Getting Started with PSoC 4
- AN52705 – Getting Started with PSoC DMA
- AN54460 – PSoC 3 and PSoC 5LP Interrupts
- AN90799 – PSoC 4 Interrupts
- AN60630 – PSoC 3 8051 Code and Memory Optimization

13.2 C Documentation

- gcc documentation can be found in your PSoC Creator installation folder.

  The compiler documentation can be found in:

  C:\Program Files\Cypress\PSoC Creator\3.0\PSoC Creator\import\gnu_cs\arm\4.7.3\share\doc\gcc-arm-none-eabi\pdf\gcc\gcc.pdf

  The linker script file documentation can be found in:

  C:\Program Files\Cypress\PSoC Creator\3.0\PSoC Creator\import\gnu_cs\arm\4.7.3\share\doc\gcc-arm-none-eabi\pdf\ld.pdf

- For MDK, the documentation can be found in your MDK installation folder, typically: *C:\Keil\ARM\Hlp*. Start with *armtools.chm*.

  For the compiler, see *armcc.chm* and *armccref.chm*.

  The linker script file documentation can be found in *armlink.chm* and *armlinkref.chm*.

13.3 ARM Cortex Documentation

ARM provides on their web site a wealth of information about the Cortex-M3 and the Cortex-M0 CPUs:

- Cortex-M0 Instruction Set
- Cortex-M3 Instruction Set
- Cortex Microcontroller Software Interface Standard (CMSIS) library
- ARM Related Books

About the Authors

Name:	Mark Ainsworth
Title:	Applications Engineer Principal
Background:	Mark Ainsworth has a BS in Computer Engineering from Syracuse University and an MSEE from University of Washington, as well as many years experience designing and building embedded systems.
Name:	Asha Ganesan
Title:	Applications Engineer
Background:	Asha Ganesan, a gold medalist from College of Engineering Guindy, India, earned her BE in Electronics and Communication Engineering. She is currently working on PSoC 3/4/5LP based projects and assisting PSoC users with their designs.
Name:	Mahesh Balan
Title:	Applications Engineer
Background:	Mahesh Balan earned his BTech in Electronics and Communication Engineering from Model Engineering College. He is currently working on PSoC 3/4/5LP based projects and assisting PSoC users with their designs.
Name:	Keith Mikoleit
Title:	Systems Engineer
Background:	Keith Mikoleit graduated from Western Washington University with a Bachelor's Degree in Electrical Engineering Technology.

A Appendix A: Compiler Output Details

This section shows in detail the assembler output for both compilers supported by PSoC Creator (gcc and MKD) and both PSoC CPUs (Cortex-M0 and Cortex-M3), with and without optimizations. The details are shown in several tables, which are organized as follows:

- Table 10. Compiler Output Details for gcc Compiler for Cortex-M3 CPU
- Table 11. Compiler Output Details for gcc Compiler for Cortex-M0 CPU
- Table 12. Compiler Output Details for MDK Compiler for Cortex-M3 CPU
- Table 13. Compiler Output Details for MDK Compiler for Cortex-M0 CPU

Although it may not be exactly what you get when you compile your C code, the assembler code in the tables can serve as useful examples that you can incorporate in your code. For details see Mixing C and Assembler Code.

The test program used to generate the tables can be found in Compiler Test Program.

A.1 Assembler Examples, gcc for Cortex-M3

Table 10 shows, for the gcc compiler for the Cortex-M3, examples of compiler output for different optimization options. The examples were extracted from the *.lst* files generated by the compiler.

See Function Arguments for details on register usage and stack usage in compiler functions.

Table 10. Compiler Output Details for gcc Compiler for Cortex-M3 CPU

C Code	gcc, Cortex-M3 No Optimization	gcc, Cortex-M3, Size Optimization	gcc, Cortex-M3, Speed Optimization

`// Calling a function`
`// with no arguments`
`LCD_Start();`
	`; do the function call`
	`bl  LCD_Start`
		`; same as for no optimization`	`; same as for no optimization`

`// Calling a function with`
`// one argument`
`LCD_PrintInt8(128);`
	`; R0 = first argument`
	`; conditional flags are NOT`
	`; updated by mov`
	`mov  r0, #80`
	`bl  LCD_PrintInt8`
		`; R0 = first argument`
		`; conditional flags ARE updated`
		`; by movs`
		`movs  r0, #80`
		`bl  LCD_PrintInt8`
			`; same as for size optimization`

`// Calling a function with`
`// two arguments`
`LCD_Position(0, 2);`
	`; R0 = first argument`
	`; R1 = second argument`
	`mov  r0, #0`
	`mov  r1, #2`
	`bl  LCD_Position`
		`; R0 = first argument`
		`; R1 = second argument`
		`movs  r1, #2`
		`movs  r0, #0`
		`bl  LCD_Position`
			`; same as for size optimization`

C Code	gcc, Cortex-M3 No Optimization	gcc, Cortex-M3, Size Optimization	gcc, Cortex-M3, Speed Optimization

`// For loop:`
`void ForLoop(uint8 i)`
`{`
`  for(i = 0; i < 10; i++)`
`  {`
`    LCD_PrintInt8(i);`
`  }`
`}`
	`; function prolog`
	`; i is saved on the stack`
	`sub   sp, sp, #16`
	`add   r7, sp, #0`
	`mov   r3, r0`
	`strb  r3, [r7, #15]`
	
	`; i = 0`
	`mov   r3, #0`
	`strb  r3, [r7, #15]`
	`b     .L2`
	
	`; do the function call with`
	`; i as the argument in R0`
	`.L3:`
	`ldrb  r3, [r7, #15]`
	`uxth  r3, r3 ; sign extend`
	`mov   r0, r3`
	`bl    LCD_PrintInt8`
	
	`; i++`
	`ldrb  r3, [r7, #15]`
	`add   r3, r3, #1`
	`strb  r3, [r7, #15]`
	
	`; check i  10, by comparing`
	`; it with 9`
	`.L2:`
	`ldrb  r3, [r7, #15]`
	`cmp   r3, #9`
	`bls   .L3`
	
	`; function epilog`
	`add   r7, r7, #16`
	`mov   sp, r7`
	`pop   {r7, pc} ; return`
		`; function prolog`
		`push  {r4, lr}`
		
		`; R4 = i`
		`movs  r4, #0`
		
		`.L2:`
		`; do the function call with`
		`; i as the argument in R0`
		`mov   r0, r4 ; sign extend`
		`adds  r4, r4, #1 ; i++`
		`uxtb  r4, r4`
		`bl    LCD_PrintInt8`
		
		`; check i not equal to 10`
		`cmp   r4, #10`
		`bne   .L2`
		
		`; function epilog`
		`pop   {r4, pc} ; return`
			`; function prolog`
			`push   {r3, lr}`
			
			`; unroll the loop`
			`; do the function call`
			`; 10 times`
			`; i as the argument in R0`
			`movs  r0, #0`
			`bl    LCD_PrintInt8`
			
			`movs  r0, #1`
			`bl    LCD_PrintInt8`
			
			`. . .`
			
			`movs  r0, #9`
			`pop   {r3, lr}`
			`; function returns back to`
			`; caller of this function`
			`b     LCD_PrintInt8`

C Code	gcc, Cortex-M3 No Optimization	gcc, Cortex-M3, Size Optimization	gcc, Cortex-M3, Speed Optimization

// While loop
// i is type automatic, see
// Accessing Automatic Variables
// for details
uint8 i = 0;
 	
while (i < 10)
{
  LCD_PrintInt8(i);
  i++;
}
	; prolog not shown
	; i = 0
	mov r3, #0
	strb r3, [r7, #7]
	b .L5
	
	.L6:
	; LCD_PrintInt8(i)
	ldrb r3, [r7, #7]
	uxth r3, r3
	mov r0, r3
	bl LCD_PrintInt8
	
	; i++
	ldrb r3, [r7, #7]
	add r3, r3, #1
	strb r3, [r7, #7]
	
	.L5:
	; while(i 10)
	ldrb r3, [r7, #7]
	cmp r3, #9
	bls .L6
	
	; epilog not shown
		; prolog not shown
		; i = 0
		movs r4, #0
		
		.L6:
		mov r0, r4
		adds r4, r4, #1 ; i++
		uxtb r4, r4
		; LCD_PrintInt8(i)
		bl LCD_PrintInt8
		
		; check i not equal to 10
		cmp r4, #10
		bne .L6
		
		; epilog not shown
			; function prolog
			push {r3, lr}
			
			; unroll the loop
			; do the function call
			; 10 times
			; i as the argument in R0
			movs r0, #0
			bl LCD_PrintInt8
			
			movs r0, #1
			bl LCD_PrintInt8
			
			. . .
			
			movs r0, #9
			pop {r3, lr}
			; function returns back to
			; caller of this function
			b LCD_PrintInt8

// Conditional statement
void Conditional(uint8 i, uint8
j)
{
  if(j == 1)
  {
  LCD_PrintInt8(i);
  }
  else
  {
  LCD_PrintInt8(i + 1);
  }
}
	; prolog not shown
	; if(j == 1)
	ldrb r3, [r7, #6]
	cmp r3, #1
	bne .L5
	
	; LCD_PrintInt8(i)
	ldrb r3, [r7, #7]
	uxth r3, r3
	mov r0, r3
	bl LCD_PrintInt8
	b .L4
	
	.L5:
	; LCD_PrintInt8(i + 1)
	ldrb r3, [r7, #7]
	uxth r3, r3
	add r3, r3, #1
	uxth r3, r3
	mov r0, r3
	bl LCD_PrintInt8
	
	.L4:
	; epilog not shown
		; no prolog
		; if(j == 1)
		cmp r1, #1
		beq .L10
		
		; LCD_PrintInt8(i)
		adds r0, r0, #1
		uxtb r0, r0
		
		.L10:
		; function returns back to
		; caller of this function
		b LCD_PrintInt8
			; same as for size optimization

C Code	gcc, Cortex-M3 No Optimization	gcc, Cortex-M3, Size Optimization	gcc, Cortex-M3, Speed Optimization

`// Switch case statements`
`void SwitchCase(uint8 j)`
`{`
`  switch(j)`
`  {`
`    case 0:`
`    LCD_PrintInt8(1);`
`    break;`
 	
`    case 1:`
`    LCD_PrintInt8(2);`
`    break;`
 	
`    default:`
`    LCD_PrintInt8(0);`
`    break;`
`  }`
`}`
	`; prolog not shown`
	`; switch(j)`
	`ldrb  r3, [r7, #7]`
	`cmp   r3, #0`
	`beq   .L9`
	`cmp   r3, #1`
	`beq   .L10`
	`b     .L12`
	
	`.L9:          ; case 0`
	`mov   r0, #1`
	`bl    LCD_PrintInt8`
	`b     .L7    ; break`
	
	`.L10:         ; case 1`
	`mov   r0, #2`
	`bl    LCD_PrintInt8`
	`b     .L9    ; break`
	
	`.L12:         ; default`
	`mov.w r0, #0`
	`bl    LCD_PrintInt8`
	`nop           ; break`
	
	`.L7:`
	`; epilog not shown`
		`; no prolog`
		`; switch(j)`
		`cbz   r0, .L13`
		`cmp   r0, #1`
		`bne   .L15`
		
		`movs  r0, #2 ; case 1`
		`b     .L16`
		
		`.L13:`
		`movs  r0, #1 ; case 0`
		`b     .L16`
		
		`.L15:`
		`movs  r0, #0 ; default`
		
		`.L16:`
		`; no epilog`
		`b     LCD_PrintInt8`
			`; no prolog`
			`; switch(j)`
			`cbnz  r0, .L12`
			`movs  r0, #1 ; case 0`
			`; no epilog`
			`b     LCD_PrintInt8`
			
			`.L12:`
			`cmp   r0, #1`
			`beq   .L13`
			`movs  r0, #0 ; default`
			`; no epilog`
			`b     LCD_PrintInt8`
			
			`movs  r0, #2 ; case 1`
			`; no epilog`
			`b     LCD_PrintInt8`

`// Ternary operator`
`void Ternary(uint8 i)`
`{`
`  LCD_PrintInt8(`
`    (i == 1) ? 80 : 100);`
`}`
	`; prolog not shown`
	`; check value of i`
	`ldrb  r3, [r7, #7]`
	`cmp   r3, #1`
	`bne   .L17`
	
	`mov   r3, #80`
	`b     .L18`
	
	`.L17:`
	`mov   r3, #100`
	
	`.L18:`
	`mov   r0, r3`
	`bl    LCD_PrintInt8`
	`; epilog not shown`
		`; no prolog`
		`; check value of i`
		`cmp   r0, #1`
		
		`; “ite” stands for if-then-`
		`; else instruction`
		`; “ne” condition checks`
		`; if the previous compare`
		`; instruction has cleared the`
		`; “equal to” flag`
		`ite    ne`
		
		`; mov if the result of the`
		`; previous “ite” instruction is`
		`; “not equal”`
		`movne  r0, #100`
		
		`; mov if the result of the`
		`; previous “ite” instruction is`
		`; “equal”`
		`moveq  r0, #80`
		
		`; no epilog`
		`b      LCD_PrintInt8`
			`; same as for size optimization`

C Code	gcc, Cortex-M3 No Optimization	gcc, Cortex-M3, Size Optimization	gcc, Cortex-M3, Speed Optimization

`// Addition operation`
`int DoAdd(int x, int y)`
`{`
`  return x + y;`
`}`
	`; prolog not shown`
	`ldr  r2, [r7, #4]`
	`ldr  r3, [r7, #0]`
	`adds  r3, r2, r3`
	`mov  r0, r3 ; return value`
	`; epilog not shown`
		`; no prolog`
		`adds  r0, r0, r1`
		`bx  lr ; return with result`
			`; same as for size optimization`

`// Subtraction operation`
`int DoSub(int x, int y)`
`{`
`  return x - y;`
`}`
	`; prolog not shown`
	`ldr  r2, [r7, #4]`
	`ldr  r3, [r7, #0]`
	`subs  r3, r2, r3`
	`mov  r0, r3 ; return value`
	`; epilog not shown`
		`; no prolog`
		`subs  r0, r0, r1`
		`bx  lr ; return with result`
			`; same as for size optimization`

`// Multiplication`
`int DoMul(int x, int y)`
`{`
`  return x * y;`
`}`
	`; prolog not shown`
	`ldr  r3, [r7, #4]`
	`ldr  r2, [r7, #0]`
	`mul  r3, r2, r3`
	`mov  r0, r3 ; return value`
	`; epilog not shown`
		`; no prolog`
		`muls  r0, r1, r0`
		`bx  lr ; return with result`
			`; same as for size optimization`

`// Division`
`int DoDiv(int x, int y)`
`{`
`  return x / y;`
`}`
	`; prolog not shown`
	`ldr  r2, [r7, #4]`
	`ldr  r3, [r7, #0]`
	`sdiv  r3, r2, r3`
	`mov  r0, r3 ; return value`
	`; epilog not shown`
		`; no prolog`
		`sdiv  r0, r0, r1`
		`bx  lr ; return with result`
			`; same as for size optimization`

`// Modulo operator`
`int DoMod(int x, int y)`
`{`
`  return x % y;`
`}`
	`; prolog not shown`
	`ldr  r3, [r7, #4]`
	`ldr  r2, [r7, #0]`
	`; truncated quotient`
	`sdiv  r2, r3, r2`
	`; quotient * divisor`
	`ldr  r1, [r7,#0]`
	`mul  r2, r1, r2`
	`; remainder = dividend -`
	`; (quotient * divisor)`
	`subs  r3, r3, r2`
	`mov  r0, r3 ; return value`
	`; epilog not shown`
		`; no prolog`
		`sdiv  r3, r0, r1`
		`; multiply and subtract instruction`
		`; implements remainder =`
		`; dividend - (quotient * divisor)`
		`mls  r0, r3, r1, r0`
		`bx  lr ; return with result`
			`; same as for size optimization`

C Code	gcc, Cortex-M3 No Optimization	gcc, Cortex-M3, Size Optimization	gcc, Cortex-M3, Speed Optimization

`// Pointer`
`void Pointer(uint8 x, uint8 *ptr)`
`{`
`  *ptr = *ptr + x;`
`  ptr++;`
`  LCD_PrintInt8(*ptr);`
`}`
	`; *ptr = *ptr + x`
	`ldr  r3, [r7, #0] ; ptr`
	`ldrb  r2, [r3, #0]`
	`ldrb  r3, [r7, #7] ; x`
	`adds  r3, r2, r3`
	`uxtb  r2, r3`
	`ldr  r3, [r7, #0] ; ptr`
	`strb  r2, [r3, #0]`
	
	`; ptr++`
	`ldr  r3, [r7, #0] ; ptr`
	`add  r3, r3, #1`
	`str  r3, [r7, #0]`
	
	`; LCD_PrintInt8(*ptr)`
	`ldr  r3, [r7, #0]`
	`ldrb  r3, [r3, #0]`
	`mov  r0, r3`
	`bl  LCD_PrintInt8`
		`; *ptr = *ptr + x`
		`ldrb  r3, [r1, #0] ; R1 = ptr`
		`adds  r0, r0, r3  ; R0 = x`
		`strb  r0, [r1, #0]`
		
		`; ptr++`
		`; LCD_PrintInt8(*ptr)`
		`ldrb  r0, [r1, #1]`
		`b  LCD_PrintInt8`
			`; same as for size optimization`

`// Function pointer`
`void FuncPtr(uint8 x,`
`  void *fptr(uint8))`
`{`
`  (*fptr)(x);`
`}`
	`; (*fptr)(x)`
	`ldrb  r2, [r7, #7] ; x`
	`ldr  r3, [r7, #0] ; fptr`
	`mov  r0, r2`
	`; in a blx instruction, the`
	`; LS bit of the register`
	`; must be 1 to keep the CPU`
	`; in Thumb mode, or an`
	`; exception occurs`
	`blx  r3`
		`; (*fptr)(x)`
		`; in a blx instruction, the LS`
		`; bit of the register must be`
		`; 1 to keep the CPU in Thumb`
		`; mode, or an exception occurs`
		`blx  r1`
			`; same as for size optimization`

C Code	gcc, Cortex-M3 No Optimization	gcc, Cortex-M3, Size Optimization	gcc, Cortex-M3, Speed Optimization

// Packed structures
struct FOO_P
{
  uint8 membera;
  uint8 memberb;
  uint32 memberc;
  uint16 memberd;
} __attribute__ ((packed));
 	
extern struct FOO_P myfoo_p;
 	
void PackedStruct(void)
{
  myfoo_p.membera = 5;
  myfoo_p.memberb = 10;
  myfoo_p.memberc = 15;
  myfoo_p.memberd = 20;
}
	; membera = 5
	movw r3, #:lower16:myfoo_p
	movt r3, #:upper16:myfoo_p
	mov r2, #5
	strb r2, [r3, #0]
	; memberb = 10
	movw r3, #:lower16:myfoo_p
	movt r3, #:upper16:myfoo_p
	mov r2, #10
	strb r2, [r3, #1]
	; memberc = 15
	movw r3, #:lower16:myfoo_p
	movt r3, #:upper16:myfoo_p
	mov r2, #0
	orr r2, r2, #15
	strb r2, [r3, #2]
	mov r2, #0
	strb r2, [r3, #3]
	mov r2, #0
	strb r2, [r3, #4]
	mov r2, #0
	strb r2, [r3, #5]
	; memberd = 20
	movw r3, #:lower16:myfoo_p
	movt r3, #:upper16:myfoo_p
	mov r2, #0
	orr r2, r2, #20
	strb r2, [r3, #6]
	mov r2, #0
	strb r2, [r3, #7]
		ldr r3, [pc, .L28]
		movs r2, #5
		movs r0, #10
		strb r2, [r3, #0] ; membera = 5
		strb r0, [r3, #1] ; memberb = 10
		movs r2, #0
		movs r1, #15
		movs r0, #20
		strb r1, [r3, #2] ; memberc = 15
		strb r2, [r3, #3]
		strb r2, [r3, #4]
		strb r2, [r3, #5]
		strb r0, [r3, #6] ; memberd = 20
		strb r2, [r3, #7]
		bx lr
		
		.L28:
		.word myfoo_p
			movw r3, #:lower16:myfoo_p
			movt r3, #:upper16:myfoo_p
			movs r1, #5
			movs r0, #10
			movs r2, #0
			strb r1, [r3, #0] ; membera = 5
			strb r0, [r3, #1] ; memberb = 10
			movs r1, #15
			movs r0, #20
			strb r1, [r3, #2] ; memberc = 15
			strb r2, [r3, #3]
			strb r2, [r3, #4]
			strb r2, [r3, #5]
			strb r0, [r3, #6] ; memberd = 20
			strb r2, [r3, #7]
			bx lr

// unpacked structures
struct FOO
{
  uint8 membera;
  uint8 memberb;
  uint32 memberc;
  uint16 memberd;
};
 	
extern struct FOO myfoo;
 	
void PackedStruct(void)
{
  myfoo.membera = 5;
  myfoo.memberb = 10;
  myfoo.memberc = 15;
  myfoo.memberd = 20;
}
	; membera = 5
	movw r3, #:lower16:myfoo
	movt r3, #:upper16:myfoo
	mov r2, #5
	strb r2, [r3, #0]
	; memberb = 10
	movw r3, #:lower16:myfoo
	movt r3, #:upper16:myfoo
	mov r2, #10
	strb r2, [r3, #1]
	; memberc = 15
	movw r3, #:lower16:myfoo
	movt r3, #:upper16:myfoo
	mov r2, #15
	str r2, [r3, #4]
	; memberd = 20
	movw r3, #:lower16:myfoo
	movt r3, #:upper16:myfoo
	mov r2, #20
	strh r2, [r3, #8]
		ldr r3, [pc, .L31]
		movs r2, #5
		strb r2, [r3, #0] ; membera = 5
		movs r0, #10
		movs r1, #15
		movs r2, #20
		strb r0, [r3, #1] ; memberb = 10
		str r1, [r3, #4] ; memberc = 15
		strh r2, [r3, #8] ; memberd = 20
		bx lr
		
		.L31:
		.word myfoo
			movw r3, #:lower16:myfoo
			movt r3, #:upper16:myfoo
			movs r2, #5
			strb r2, [r3, #0] ; membera = 5
			movs r0, #10
			movs r1, #15
			movs r2, #20
			strb r0, [r3, #1] ; memberb = 10
			str r1, [r3, #4] ; memberc = 15
			strh r2, [r3, #8] ; memberd = 20
			bx lr

A.2 Assembler Examples, gcc for Cortex-M0

Table 11 shows, for the gcc compiler for the Cortex-M0, examples of compiler output for different optimization options. The examples were extracted from the *.lst* files generated by the compiler.

See Function Arguments for details on register usage and stack usage in compiler functions.

Table 11. Compiler Output Details for gcc Compiler for Cortex-M0 CPU

C Code	gcc, Cortex-M0, No Optimization	gcc, Cortex-M0, Size Optimization	gcc, Cortex-M0, Speed Optimization

`// Calling a function`
`// with no arguments`
`LCD_Start();`
	`; do the function call`
	`bl LCD_Start`
		`; same as for no optimization`	`; same as for no optimization`

`// Calling a function with`
`// one argument`
`LCD_PrintInt8(128);`
	`; R0 = first argument`
	`; conditional flags are NOT`
	`; updated by mov`
	`mov r0, #128`
	`bl LCD_PrintInt8`
		`; same as for no optimization`	`; same as for no optimization`

`// Calling a function with`
`// two arguments`
`LCD_Position(0, 2);`
	`; R0 = first argument`
	`; R1 = second argument`
	`mov r1, #2`
	`mov r0, #0`
	`bl LCD_Position`
		`; same as for no optimization`	`; same as for no optimization`

C Code	gcc, Cortex-M0, No Optimization	gcc, Cortex-M0, Size Optimization	gcc, Cortex-M0, Speed Optimization

`// For loop:`
`void ForLoop(uint8 i)`
`{`
`  for(i = 0; i < 10; i++)`
`  {`
`    LCD_PrintInt8(i);`
`  }`
`}`
	`; function prolog`
	`; i is saved on the stack`
	`push  {r7, lr}`
	`sub   sp, sp, #16`
	`add   r7, sp, #0`
	`mov   r2, r0`
	`add   r3, r7, #7`
	`strb  r2, [r3]`
	
	`; i = 0`
	`mov   r3, r7`
	`add   r3, r3, #15`
	`mov   r2, #0`
	`strb  r2, [r3]`
	`b     .L2`
	
	`; do the function call with`
	`; i as the argument in R0`
	`.L3:`
	`mov   r3, r7`
	`add   r3, r3, #15`
	`ldrb  r3, [r3]`
	`mov   r0, r3`
	`bl    LCD_PrintInt8`
	
	`; i++`
	`mov   r3, r7`
	`add   r3, r3, #15`
	`mov   r2, r7`
	`add   r2, r2, #15`
	`ldrb  r2, [r2]`
	`add   r2, r2, #1`
	`strb  r2, [r3]`
	
	`; check i  10, by comparing`
	`; it with 9`
	`.L2:`
	`mov   r3, r7`
	`add   r3, r3, #15`
	`ldrb  r3, [r3]`
	`cmp   r3, #9`
	`bls   .L3`
	
	`; function epilog`
	`mov   sp, r7`
	`add   sp, sp, #16`
	`pop   {r7, pc}`
		`; function prolog`
		`push  {r4, lr}`
		
		`; R4 = i`
		`mov   r4, #0`
		
		`.L2:`
		`; do the function call with`
		`; i as the argument in R0`
		`mov   r0, r4 ; sign extend`
		`add   r4, r4, #1 ; i++`
		`uxtb  r4, r4`
		`bl    LCD_PrintInt8`
		
		`; check i not equal to 10`
		`cmp   r4, #10`
		`bne   .L2`
		
		`pop   {r4, pc} ; return`
			`; function prolog`
			`push  {r3, lr}`
			
			`; unroll the loop`
			`; do the function call`
			`; 10 times`
			`; i as the argument in R0`
			`mov   r0, #0`
			`bl    LCD_PrintInt8`
			
			`mov   r0, #1`
			`bl    LCD_PrintInt8`
			
			`. . .`
			
			`mov   r0, #9`
			`bl    LCD_PrintInt8`
			
			`pop   {r3, pc} ; return`

C Code	gcc, Cortex-M0, No Optimization	gcc, Cortex-M0, Size Optimization	gcc, Cortex-M0, Speed Optimization

`// While loop`
`// i is type automatic, see`
`// Accessing Automatic Variables`
`// for details`
`uint8 i = 0;`
 	
`while (i < 10)`
`{`
`  LCD_PrintInt8(i);`
`  i++;`
`}`
	`; prolog not shown`
	`; i = 0`
	`add  r3, r7, #7`
	`mov  r2, #0`
	`strb  r2, [r3]`
	`b  .L5`
	
	`.L6:`
	`; LCD_PrintInt8(i)`
	`add  r3, r7, #7`
	`ldrb  r3, [r3]`
	`mov  r0, r3`
	`bl  LCD_PrintInt8`
	
	`; i++`
	`add  r3, r7, #7`
	`add  r2, r7, #7`
	`ldrb  r2, [r2]`
	`add  r2, r2, #1`
	`strb  r2, [r3]`
	
	`.L5:`
	`; while(i  10)`
	`add  r3, r7, #7`
	`ldrb  r3, [r3]`
	`cmp  r3, #9`
	`bls  .L6`
	
	`; epilog not shown`
		`; prolog not shown`
		`; i = 0`
		`movs  r4, #0`
		
		`.L6:`
		`mov  r0, r4`
		`add  r4, r4, #1 ; i++`
		`uxtb  r4, r4`
		`; LCD_PrintInt8(i)`
		`bl  LCD_PrintInt8`
		
		`; check i not equal to 10`
		`cmp  r4, #10`
		`bne  .L6`
		
		`; epilog not shown`
			`; prolog not shown`
			`; unroll the loop`
			`; do the function call`
			`; 10 times`
			`; i as the argument in R0`
			`mov  r0, #0`
			`bl  LCD_PrintInt8`
			
			`mov  r0, #1`
			`bl  LCD_PrintInt8`
			
			`. . .`
			
			`mov  r0, #9`
			`bl  LCD_PrintInt8`
			
			`; epilog not shown`

C Code	gcc, Cortex-M0, No Optimization	gcc, Cortex-M0, Size Optimization	gcc, Cortex-M0, Speed Optimization

`// Conditional statement`
`void Conditional(uint8 i, uint8 j)`
`{`
`  if(j == 1)`
`  {`
`    LCD_PrintInt8(i);`
`  }`
`  else`
`  {`
`    LCD_PrintInt8(i + 1);`
`  }`
`}`
	`; prolog not shown`
	`; if(j == 1)`
	`add  r3, r7, #6`
	`ldrb  r3, [r3]`
	`cmp  r3, #1`
	`bne  .L8`
	
	`; LCD_PrintInt8(i)`
	`add  r3, r7, #7`
	`ldrb  r3, [r3]`
	`mov  r0, r3`
	`bl  LCD_PrintInt8`
	`b  .L7`
	
	`.L8:`
	`; LCD_PrintInt8(i + 1)`
	`add  r3, r7, #7`
	`ldrb  r3, [r3]`
	`add  r3, r3, #1`
	`uxtb  r3, r3`
	`mov  r0, r3`
	`bl  LCD_PrintInt8`
	
	`.L7:`
	`; epilog not shown`
		`; prolog not shown`
		`; if(j == 1)`
		`cmp  r1, #1`
		`beq  .L11`
		
		`; LCD_PrintInt8(i)`
		`add  r0, r0, #1`
		`uxtb  r0, r0`
		
		`.L11:`
		`bl  LCD_PrintInt8`
		
		`; epilog not shown`
			`; same as for size optimization`

`// Switch case statements`
`void SwitchCase(uint8 j)`
`{`
`  switch(j)`
`  {`
`    case 0:`
`    LCD_PrintInt8(1);`
`    break;`
 	
`    case 1:`
`    LCD_PrintInt8(2);`
`    break;`
 	
`    default:`
`    LCD_PrintInt8(0);`
`    break;`
`  }`
`}`
	`; prolog not shown`
	`; switch(j)`
	`add  r3, r7, #7`
	`ldrb  r3, [r3]`
	`cmp  r3, #0`
	`beq  .L12`
	`cmp  r3, #1`
	`beq  .L13`
	`b  .L15`
	
	`.L12  ; case 0`
	`mov  r0, #1`
	`bl  LCD_PrintInt8`
	`b  .L10  ; break`
	
	`.L13:  ; case 1`
	`mov  r0, #2`
	`bl  LCD_PrintInt8`
	`b  .L10  ; break`
	
	`.L15:  ; default`
	`mov  r0, #0`
	`bl  LCD_PrintInt8`
	`mov  r8, r8 ; break - nop`
	
	`.L10:`
	`; epilog not shown`
		`; prolog not shown`
		`; switch(j)`
		`cmp  r0, #0`
		`beq  .L14`
		`cmp  r0, #1`
		`bne  .L17`
		
		`mov  r0, #2 ; case 1`
		`b  .L18`
		
		`.L14:`
		`mov  r0, #1 ; case 0`
		`b  .L18`
		
		`.L17:`
		`mov  r0, #0 ; default`
		
		`.L18:`
		`bl  LCD_PrintInt8`
		`; epilog not shown`
			`; prolog not shown`
			`; switch(j)`
			`cmp  r0, #0`
			`bne  .L14`
			`mov  r0, #1 ; case 0`
			`bl  LCD_PrintInt8`
			
			`.L8:`
			`pop  {r3, pc} ; return`
			
			`.L14:`
			`cmp  r0, #1`
			`beq  .L15`
			`mov  r0, #0 ; default`
			`bl  LCD_PrintInt8`
			`b  .L8`
			
			`mov  r0, #2 ; case 1`
			`bl  LCD_PrintInt8`
			`b  .L8`

C Code	gcc, Cortex-M0, No Optimization	gcc, Cortex-M0, Size Optimization	gcc, Cortex-M0, Speed Optimization

`// Ternary operator`
`void Ternary(uint8 i)`
`{`
`  LCD_PrintInt8(`
`  (i == 1) ? 80 : 100);`
`}`
	`; prolog not shown`
	`; check value of i`
	`add  r3, r7, #7`
	`ldrb  r3, [r3]`
	`cmp  r3, #1`
	`bne  .L17`
	
	`mov  r3, #80`
	`b  .L18`
	
	`.L17:`
	`mov  r3, #100`
	
	`.L18:`
	`mov  r0, r3`
	`bl  LCD_PrintInt8`
	`; epilog not shown`
		`; prolog not shown`
		`mov  r3, #100`
		`cmp  r0, #1`
		`bne  .L20`
		`mov  r3, #80`
		
		`.L20:`
		`mov  r0, r3`
		`bl  LCD_PrintInt8`
		`; epilog not shown`
			`; prolog not shown`
			`mov  r3, #100`
			`cmp  r0, #1`
			`beq  .L19`
			
			`.L17:`
			`mov  r0, r3`
			`bl  LCD_PrintInt8`
			`pop  {r3, pc} ; return`
			
			`.L19:`
			`mov  r3, #80`
			`b  .L17`

`// Addition operation`
`int DoAdd(int x, int y)`
`{`
`  return x + y;`
`}`
	`; prolog not shown`
	`ldr  r2, [r7, #4]`
	`ldr  r3, [r7, #0]`
	`add  r3, r2, r3`
	`mov  r0, r3 ; return value`
	`; epilog not shown`
		`; no prolog`
		`add  r0, r0, r1`
		`bx  lr ; return with result`
			`; same as for size optimization`

`// Subtraction operation`
`int DoSub(int x, int y)`
`{`
`  return x - y;`
`}`
	`; prolog not shown`
	`ldr  r2, [r7, #4]`
	`ldr  r3, [r7, #0]`
	`sub  r3, r2, r3`
	`mov  r0, r3 ; return value`
	`; epilog not shown`
		`; no prolog`
		`sub  r0, r0, r1`
		`bx  lr ; return with result`
			`; same as for size optimization`

`// Multiplication`
`int DoMul(int x, int y)`
`{`
`  return x * y;`
`}`
	`; prolog not shown`
	`ldr  r3, [r7, #4]`
	`ldr  r2, [r7, #0]`
	`mul  r3, r2`
	`mov  r0, r3 ; return value`
	`; epilog not shown`
		`; no prolog`
		`mul  r0, r1`
		`bx  lr ; return with result`
			`; same as for size optimization`

`// Division`
`int DoDiv(int x, int y)`
`{`
`  return x / y;`
`}`
	`; prolog not shown`
	`ldr  r0, [r7, #4]`
	`ldr  r1, [r7, #0]`
	`bl  __aeabi_idiv`
	`mov  r3, r0`
	`mov  r0, r3 ; return value`
	`; epilog not shown`
		`push  {r3, lr}`
		`bl  __aeabi_idiv`
		`; return with result`
		`pop  {r3, pc}`
			`; same as for size optimization`

`// Modulo operator`
`int DoMod(int x, int y)`
`{`
`  return x % y;`
`}`
	`; prolog not shown`
	`ldr  r3, [r7, #4]`
	`mov  r0, r3`
	`ldr  r1, [r7]`
	`bl  __aeabi_idivmod`
	`mov  r3, r1`
	`mov  r0, r3 ; return value`
	`; epilog not shown`
		`push  {r3, lr}`
		`bl  __aeabi_idivmod`
		`; return with result`
		`mov  r0, r1`
		`pop  {r3, pc}`
			`; same as for size optimization`

C Code	gcc, Cortex-M0, No Optimization	gcc, Cortex-M0, Size Optimization	gcc, Cortex-M0, Speed Optimization

`// Pointer`
`void Pointer(uint8 x, uint8 *ptr)`
`{`
`  *ptr = *ptr + x;`
`  ptr++;`
`  LCD_PrintInt8(*ptr);`
`}`
	`; *ptr = *ptr + x`
	`ldr  r3, [r7]  ; ptr`
	`ldrb  r2, [r3]`
	`add  r3, r7, #7 ; x`
	`ldrb  r3, [r3]`
	`add  r3, r2, r3`
	`uxtb  r2, r3`
	`ldr  r3, [r7]  ; ptr`
	`strb  r2, [r3]`
	
	`; ptr++`
	`ldr  r3, [r7]  ; ptr`
	`add  r3, r3, #1`
	`str  r3, [r7]`
	
	`; LCD_PrintInt8(*ptr)`
	`ldr  r3, [r7]`
	`ldrb  r3, [r3]`
	`mov  r0, r3`
	`bl  LCD_PrintInt8`
		`; *ptr = *ptr + x`
		`ldrb  r3, [r1]  ; R1 = ptr`
		`add  r0, r0, r3 ; R0 = x`
		`strb  r0, [r1]`
		
		`; ptr++`
		`; LCD_PrintInt8(*ptr)`
		`ldrb  r0, [r1, #1]`
		`bl  LCD_PrintInt8`
			`; same as for size optimization`

`// Function pointer`
`void FuncPtr(uint8 x,`
`  void *fptr(uint8))`
`{`
`  (*fptr)(x);`
`}`
	`; (*fptr)(x)`
	`add  r3, r7, #7 ; x`
	`ldrb  r2, [r3]`
	`ldr  r3, [r7]  ; fptr`
	`mov  r0, r2`
	`; in a blx instruction, the`
	`; LS bit of the register`
	`; must be 1 to keep the CPU`
	`; in Thumb mode, or an`
	`; exception occurs`
	`blx  r3`
		`; (*fptr)(x)`
		`; in a blx instruction, the LS`
		`; bit of the register must be`
		`; 1 to keep the CPU in Thumb`
		`; mode, or an exception occurs`
		`blx  r1`
			`; same as for size optimization`

C Code	gcc, Cortex-M0, No Optimization	gcc, Cortex-M0, Size Optimization	gcc, Cortex-M0, Speed Optimization

`// Packed structures`
`struct FOO_P`
`{`
`  uint8  membera;`
`  uint8  memberb;`
`  uint32 memberc;`
`  uint16 memberd;`
`} __attribute__ ((packed));`
 	
`extern struct FOO_P myfoo_p;`
 	
`void PackedStruct(void)`
`{`
`  myfoo_p.membera =  5;`
`  myfoo_p.memberb = 10;`
`  myfoo_p.memberc = 15;`
`  myfoo_p.memberd = 20;`
`}`
	`; membera = 5`
	`ldr  r3, [pc, .L32]`
	`mov  r2, #5`
	`strb  r2, [r3]`
	`; memberb = 10`
	`ldr  r3, [pc, .L32]`
	`mov  r2, #10`
	`strb  r2, [r3, #1]`
	`; memberc = 15`
	`ldr  r3, [pc, .L32]`
	`ldrb  r1, [r3, #2]`
	`mov  r2, #0`
	`and  r2, r1`
	`mov  r1, #15`
	`orr  r2, r1`
	`strb  r2, [r3, #2]`
	`ldrb  r1, [r3, #3]`
	`mov  r2, #0`
	`and  r2, r1`
	`strb  r2, [r3, #3]`
	`ldrb  r1, [r3, #4]`
	`mov  r2, #0`
	`and  r2, r1`
	`strb  r2, [r3, #4]`
	`ldrb  r1, [r3, #5]`
	`mov  r2, #0`
	`and  r2, r1`
	`strb  r2, [r3, #5]`
	`; memberd = 20`
	`ldr  r3, [pc, .L32]`
	`ldrb  r1, [r3, #6]`
	`mov  r2, #0`
	`and  r2, r1`
	`mov  r1, #20`
	`orr  r2, r1`
	`strb  r2, [r3, #6]`
	`ldrb  r1, [r3, #7]`
	`mov  r2, #0`
	`and  r2, r1`
	`strb  r2, [r3, #7]`
	
	`.L32:`
	`.word  myfoo_p`
		`ldr  r3, [pc, .L30]`
		`mov  r2, #5`
		`mov  r0, #10`
		`strb  r2, [r3]  ; membera = 5`
		`strb  r0, [r3, #1] ; memberb = 10`
		`mov  r2, #0`
		`mov  r1, #15`
		`mov  r0, #20`
		`strb  r1, [r3, #2] ; memberc = 15`
		`strb  r2, [r3, #3]`
		`strb  r2, [r3, #4]`
		`strb  r2, [r3, #5]`
		`strb  r0, [r3, #6] ; memberd = 20`
		`strb  r2, [r3, #7]`
		`bx  lr`
		
		`.L30:`
		`.word  myfoo_p`
			`; same as for size optimization`

C Code	gcc, Cortex-M0, No Optimization	gcc, Cortex-M0, Size Optimization	gcc, Cortex-M0, Speed Optimization

`// unpacked structures`
`struct FOO`
`{`
`  uint8  membera;`
`  uint8  memberb;`
`  uint32 memberc;`
`  uint16 memberd;`
`};`
 	
`extern struct FOO myfoo;`
 	
`void PackedStruct(void)`
`{`
`  myfoo.membera =  5;`
`  myfoo.memberb = 10;`
`  myfoo.memberc = 15;`
`  myfoo.memberd = 20;`
`}`
	`; membera = 5`
	`ldr   r3, [pc, .L35]`
	`mov   r2, #5`
	`strb  r2, [r3]`
	`; memberb = 10`
	`ldr   r3, [pc, .L35]`
	`mov   r2, #10`
	`strb  r2, [r3, #1]`
	`; memberc = 15`
	`ldr   r3, [pc, .L35]`
	`mov   r2, #15`
	`str   r2, [r3, #4]`
	`; memberd = 20`
	`ldr   r3, [pc, .L35]`
	`mov   r2, #20`
	`strh  r2, [r3, #8]`
	
	`.L35:`
	`.word  myfoo`
		`ldr   r3, [pc, .L33]`
		`mov   r2, #5`
		`strb  r2, [r3]     ; membera = 5`
		`mov   r0, #10`
		`mov   r1, #15`
		`mov   r2, #20`
		`strb  r0, [r3, #1] ; memberb = 10`
		`str   r1, [r3, #4] ; memberc = 15`
		`strh  r2, [r3, #8] ; memberd = 20`
		`bx    lr`
		
		`.L33:`
		`.word   myfoo`
			`; same as for size optimization`

A.3 Assembler Examples, MDK for Cortex-M3

Note: Table 12 shows, for the MDK compiler for the Cortex-M3, examples of compiler output for different optimization options. Since the free evaluation version of MDK, MDK-Lite, does not include assembler in the *.lst* file, the examples were extracted from the assembler-level debug window in PSoC Creator.

See Function Arguments for details on register usage and stack usage in compiler functions.

Table 12. Compiler Output Details for MDK Compiler for Cortex-M3 CPU

C Code	MDK, Cortex-M3, No Optimization	MDK, Cortex-M3, Size Optimization	MDK, Cortex-M3, Speed Optimization

`// Calling a function`
`// with no arguments`
`LCD_Start();`
	`; do the function call`
	`bl   LCD_Start`
		`; same as for no optimization`	`; same as for no optimization`

`// Calling a function with`
`// one argument`
`LCD_PrintInt8(128);`
	`; R0 = first argument`
	`movs  r0, #80`
	`bl    LCD_PrintInt8`
		`; same as for no optimization`	`; same as for no optimization`

`// Calling a function with`
`// two arguments`
`LCD_Position(0, 2);`
	`; R0 = first argument`
	`; R1 = second argument`
	`movs  r1, #2`
	`movs  r0, #0`
	`bl    LCD_Position`
		`; same as for no optimization`	`; same as for no optimization`

C Code	MDK, Cortex-M3, No Optimization	MDK, Cortex-M3, Size Optimization	MDK, Cortex-M3, Speed Optimization

`// For loop:`
`void ForLoop(uint8 i)`
`{`
`  for(i = 0; i < 10; i++)`
`  {`
`  LCD_PrintInt8(i);`
`  }`
`}`
	`; prolog`
	`push  {r4, lr}`
	`mov  r4, r0`
	
	`movs  r4, #0  ; i = 0`
	`b.n  <ForLoop+0x12>`
	
	`<ForLoop+0x8>:`
	`; do the function call with`
	`; i as the argument in R0`
	`mov  r0, r4`
	`bl  LCD_PrintInt8`
	
	`adds  r0, r4, #1 ; i++`
	`uxtb  r4, r0 ; sign extend`
	
	`<ForLoop+0x12>:`
	`cmp  r4, #a  ; i < 10`
	`blt.n  <ForLoop+0x8>`
	
	`pop  {r4, pc} ; return`
		`; prolog`
		`push  {r4, lr}`
		
		`movs  r4, #0  ; i = 0`
		
		`<ForLoop+0x4>:`
		`; do the function call with`
		`; i as the argument in R0`
		`mov  r0, r4`
		`bl  LCD_PrintInt8`
		
		`adds  r4, r4, #1 ; i++`
		`uxtb  r4, r4 ; sign extend`
		`cmp  r4, #a  ; i < 10`
		`bcc.n  <ForLoop+0x4>`
		
		`pop  {r4, pc} ; return`
			`; same as for size optimization`

`// While loop`
`// i is type automatic, see`
`// Accessing Automatic Variables`
`// for details`
`uint8 i = 0;`
 	
`while (i < 10)`
`{`
`  LCD_PrintInt8(i);`
`  i++;`
`}`
	`; prolog`
	`push  {r4, lr}`
	
	`movs  r4, #0  ; i = 0`
	`b.n  <WhileLoop+0x10>`
	
	`<WhileLoop+0x6>:`
	`; do the function call with`
	`; i as the argument in R0`
	`mov  r0, r4`
	`bl  LCD_PrintInt8`
	
	`adds  r0, r4, #1 ; i++`
	`uxtb  r4, r0 ; sign extend`
	
	`<WhileLoop+0x10>:`
	`cmp  r4, #a  ; i < 10`
	`blt.n  <WhileLoop+0x6>`
	
	`pop  {r4, pc}; return`
		`; prolog`
		`push  {r4, lr}`
		
		`movs  r4, #0  ; i = 0`
		
		`<WhileLoop+0x4>:`
		`; do the function call with`
		`; i as the argument in R0`
		`mov  r0, r4`
		`bl  LCD_PrintInt8`
		
		`adds  r4, r4, #1 ; i++`
		`uxtb  r4, r4 ; sign extend`
		
		`cmp  r4, #a  ; i < 10`
		`bcc.n  <WhileLoop+0x4>`
		
		`pop  {r4, pc}; return`
			`; same as for size optimization`

C Code	MDK, Cortex-M3, No Optimization	MDK, Cortex-M3, Size Optimization	MDK, Cortex-M3, Speed Optimization

`// Conditional statement`
`void Conditional(uint8 i,`
`uint8 j)`
`{`
`  if(j == 1)`
`  {`
`    LCD_PrintInt8(i);`
`  }`
`  else`
`  {`
`    LCD_PrintInt8(i + 1);`
`  }`
`}`
	`; prolog`
	`push    {r4, r5, r6, lr}`
	
	`mov    r4, r0`
	`mov    r5, r1`
	`cmp    r5, #1 ; j == 1`
	`bne.n  <Conditional+0x12>`
	
	`mov    r0, r4`
	`bl     LCD_PrintInt8`
	`b.n    <Conditional+0x1a>`
	
	`<Conditional+0x12>:`
	`; LCD_PrintInt8(i + 1)`
	`adds   r1, r4, #1`
	`uxtb   r0, r1 ; sign extend`
	`bl     LCD_PrintInt8`
	
	`<Conditional+0x1a>:`
	`; return`
	`pop    {r4, r5, r6, pc}`
		`; no prolog`
		`; if(j == 1)`
		`cmp    r1, #1`
		`beq.n  <Conditional+0x8>`
		
		`adds   r0, r0, #1 ; i + 1`
		`uxtb   r0, r0`
		
		`<Conditional+0x8>:`
		`; function returns back to`
		`; caller of this function`
		`b.w    LCD_PrintInt8`
			`; same as for size optimization`

`// Switch case statements`
`void SwitchCase(uint8 j)`
`{`
`  switch(j)`
`  {`
`    case 0:`
`    LCD_PrintInt8(1);`
`    break;`
 	
`    case 1:`
`    LCD_PrintInt8(2);`
`    break;`
 	
`    default:`
`    LCD_PrintInt8(0);`
`    break;`
`  }`
`}`
	`; prolog`
	`push   {r4, lr}`
	
	`; switch(j)`
	`mov    r4, r0`
	`cbz    r4, <SwitchCase+0xc>`
	`cmp    r4, #1`
	`bne.n  <SwitchCase+0x1c>`
	`b.n    <SwitchCase+0x14>`
	
	`<SwitchCase+0xc>:`
	`movs   r0, #1 ; case 0`
	`bl     LCD_PrintInt8`
	`b.n    <SwitchCase+0x24>`
	
	`<SwitchCase+0x14>:`
	`movs   r0, #2 ; case 1`
	`bl     LCD_PrintInt8`
	`b.n    <SwitchCase+0x24>`
	
	`<SwitchCase+0x1c>:`
	`movs   r0, #0 ; default`
	`bl     LCD_PrintInt8`
	`nop`
	
	`<SwitchCase+0x24>:`
	`nop`
	`pop    {r4, pc} ; return`
		`; prolog not shown`
		`; switch(j)`
		`cbz    r0, <SwitchCase+0xa>`
		`cmp    r0, #1`
		`beq.n  <SwitchCase+0xe>`
		
		`movs   r0, #0 ; default`
		`b.n    <SwitchCase+0x10>`
		
		`<SwitchCase+0xa>:`
		`movs   r0, #1 ; case 0`
		`b.n    <SwitchCase+0x10>`
		
		`<SwitchCase+0xe>:`
		`movs   r0, #2 ; case 1`
		
		`<SwitchCase+0x10>:`
		`; function returns back to`
		`; caller of this function`
		`b.w    LCD_PrintInt8`
			`; same as for size optimization`

C Code	MDK, Cortex-M3, No Optimization	MDK, Cortex-M3, Size Optimization	MDK, Cortex-M3, Speed Optimization

`// Ternary operator`
`void Ternary(uint8 i)`
`{`
`  LCD_PrintInt8(`
`  (i == 1) ? 80 : 100);`
`}`
	`; prolog`
	`push  {r4, lr}`
	
	`mov  r4, r0 ; i == 1`
	`cmp  r4, #1`
	`bne.n  <Ternary+0xc>`
	
	`movs  r1, #50`
	`b.n  <Ternary+0xe>`
	
	`<Ternary+0xc>:`
	`movs  r1, #64`
	
	`<Ternary+0xe>:`
	`mov  r0, r1`
	`bl  LCD_PrintInt8`
	
	`pop  {r4, pc} ; return`
		`; no prolog`
		`cmp  r0, #1`
		`beq.n  <Ternary+0xa>`
		
		`<Ternary+0x6>:`
		`movs  r0, #64  ; 0x64`
		`; function returns back to`
		`; caller of this function`
		`b.w  LCD_PrintInt8`
		
		`<Ternary+0xa>:`
		`movs  r0, #50  ; 0x50`
		`b.n  <Ternary+0x6>`
			`; same as for size optimization`

`// Addition operation`
`int DoAdd(int x, int y)`
`{`
`  return x + y;`
`}`
	`; no prolog`
	`mov  r2, r0`
	`adds  r0, r2, r1`
	`bx  lr ; return with result`
		`; no prolog`
		`add  r0, r1`
		`bx  lr ; return with result`
			`; same as for size optimization`

`// Subtraction operation`
`int DoSub(int x, int y)`
`{`
`  return x - y;`
`}`
	`; no prolog`
	`mov  r2, r0`
	`subs  r0, r2, r1`
	`bx  lr ; return with result`
		`; no prolog`
		`subs  r0, r0, r1`
		`bx  lr ; return with result`
			`; same as for size optimization`

`// Multiplication`
`int DoMul(int x, int y)`
`{`
`  return x * y;`
`}`
	`; no prolog`
	`mov  r2, r0`
	`mul.w  r0, r2, r1`
	`bx  lr ; return with result`
		`; no prolog`
		`muls  r0, r1`
		`bx  lr ; return with result`
			`; same as for size optimization`

`// Division`
`int DoDiv(int x, int y)`
`{`
`  return x / y;`
`}`
	`; no prolog`
	`mov  r2, r0`
	`sdiv  r0, r2, r1`
	`bx  lr ; return with result`
		`; no prolog`
		`sdiv  r0, r0, r1`
		`bx  lr ; return with result`
			`; same as for size optimization`

`// Modulo operator`
`int DoMod(int x, int y)`
`{`
`  return x % y;`
`}`
	`; no prolog`
	`mov  r2, r0`
	`; truncated quotient`
	`sdiv  r0, r2, r1`
	`; multiply and subtract instruction`
	`; implements remainder =`
	`; dividend - (quotient * divisor)`
	`mls  r0, r1, r0, r2`
	`bx  lr ; return with result`
		`; no prolog`
		`; truncated quotient`
		`sdiv  r2, r0, r1`
		`; multiply and subtract instruction`
		`; implements remainder =`
		`; dividend - (quotient * divisor)`
		`mls  r0, r1, r2, r0`
		`bx  lr ; return with result`
			`; same as for size optimization`

C Code	MDK, Cortex-M3, No Optimization	MDK, Cortex-M3, Size Optimization	MDK, Cortex-M3, Speed Optimization

`// Pointer`
`void Pointer(uint8 x, uint8 *ptr)`
`{`
`  *ptr = *ptr + x;`
`  ptr++;`
`  LCD_PrintInt8(*ptr);`
`}`
	`; prolog`
	`push  {r4, r5, r6, lr}`
	`mov   r5, r0`
	`mov   r4, r1`
	
	`; *ptr = *ptr + x;`
	`ldrb  r0, [r4, #0]`
	`add   r0, r5`
	`strb  r0, [r4, #0]`
	
	`adds  r4, r4, #1 ; ptr++;`
	
	`ldrb  r0, [r4, #0]`
	`bl    LCD_PrintInt8`
	
	`pop   {r4, r5, r6, pc} ; return`
		`; no prolog`
		`; *ptr = *ptr + x`
		`ldrb  r2, [r1, #0] ; R1 = ptr`
		`add   r0, r2   ; R0 = x`
		`strb  r0, [r1, #0]`
		
		`; ptr++`
		`; LCD_PrintInt8(*ptr)`
		`ldrb  r0, [r1, #1]`
		
		`; function returns back to`
		`; caller of this function`
		`b.w   LCD_PrintInt8`
			`; same as for size optimization`

`// Function pointer`
`void FuncPtr(uint8 x,`
`             void`
`*fptr(uint8))`
`{`
`  (*fptr)(x);`
`}`
	`; prolog`
	`push  {r4, r5, r6, lr}`
	`mov   r5, r0`
	`mov   r4, r1`
	
	`; (*fptr)(x)`
	`mov   r0, r5`
	`; in a blx instruction, the`
	`; LS bit of the register`
	`; must be 1 to keep the CPU`
	`; in Thumb mode, or an`
	`; exception occurs`
	`blx   r4`
	
	`pop   {r4, r5, r6, pc} ; return`
		`; (*fptr)(x)`
		`; in a bx instruction, the LS`
		`; bit of the register must be`
		`; 1 to keep the CPU in Thumb`
		`; mode, or an exception occurs`
		
		`; function returns back to`
		`; caller of this function`
		`bx   r1`
			`; same as for size optimization`

`// Packed structures`
`struct FOO_P`
`{`
`  uint8  membera;`
`  uint8  memberb;`
`  uint32 memberc;`
`  uint16 memberd;`
`} __attribute__ ((packed));`
 	
`extern struct FOO_P myfoo_p;`
 	
`void PackedStruct(void)`
`{`
`  myfoo_p.membera =  5;`
`  myfoo_p.memberb = 10;`
`  myfoo_p.memberc = 15;`
`  myfoo_p.memberd = 20;`
`}`
	`; no prolog`
	`; membera = 5`
	`movs   r0, #5`
	`ldr    r1, [pc, #14]`
	`strb   r0, [r1, #0]`
	`; memberb = 10`
	`movs   r0, #a`
	`strb   r0, [r1, #1]`
	`; memberc = 15`
	`movs   r0, #f`
	`; word unaligned access`
	`str.w  r0, [r1, #2]`
	`; memberd = 20`
	`movs   r0, #14`
	`strh   r0, [r1, #6]`
	`bx     lr  ; return`
	
	`.word  &myfoo_p`
		`; no prolog`
		`ldr    r0, [pc, #14]`
		`; membera = 5`
		`movs   r1, #5`
		`strb   r1, [r0, #0]`
		`; memberb = 10`
		`movs   r1, #a`
		`strb   r1, [r0, #1]`
		`; memberc = 15`
		`movs   r1, #f`
		`; word unaligned access`
		`str.w  r1, [r0, #2]`
		`; memberd = 20`
		`movs   r1, #14`
		`strh   r1, [r0, #6]`
		`bx     lr  ; return`
		
		`.word  &myfoo_p`
			`; same as for size optimization`

C Code	MDK, Cortex-M3, No Optimization	MDK, Cortex-M3, Size Optimization	MDK, Cortex-M3, Speed Optimization

`// unpacked structures`
`struct FOO`
`{`
`  uint8  membera;`
`  uint8  memberb;`
`  uint32 memberc;`
`  uint16 memberd;`
`};`
 	
`extern struct FOO myfoo;`
 	
`void PackedStruct(void)`
`{`
`  myfoo.membera =  5;`
`  myfoo.memberb = 10;`
`  myfoo.memberc = 15;`
`  myfoo.memberd = 20;`
`}`
	`; no prolog`
	`; membera = 5`
	`movs   r0, #5`
	`ldr    r1, [pc, #10]`
	`strb   r0, [r1, #0]`
	`; memberb = 10`
	`movs   r0, #a`
	`strb   r0, [r1, #1]`
	`; memberc = 15`
	`movs   r0, #f`
	`str    r0, [r1, #4]`
	`; memberd = 20`
	`movs   r0, #14`
	`strh   r0, [r1, #8]`
	`bx     lr  ; return`
	
	`.word  &myfoo`
		`; no prolog`
		`ldr    r0, [pc, #10]`
		`; membera = 5`
		`movs   r1, #5`
		`strb   r1, [r0, #0]`
		`; memberb = 10`
		`movs   r1, #a`
		`strb   r1, [r0, #1]`
		`; memberc = 15`
		`movs   r1, #f`
		`str    r1, [r0, #4]`
		`; memberd = 20`
		`movs   r1, #14`
		`strh   r1, [r0, #8]`
		`bx     lr  ; return`
		
		`.word  &myfoo`
			`; same as for size optimization`

A.4 Assembler Examples, MDK for Cortex-M0

Table 13 shows, for the MDK compiler for the Cortex-M0, examples of compiler output for different optimization options. Since the free evaluation version of MDK does not produce a usable *.lst* file, the examples were extracted from the assembler-level debug window in PSoC Creator.

See Function Arguments for details on register usage and stack usage in compiler functions.

Table 13. Compiler Output Details for MDK Compiler for Cortex-M0 CPU

C Code	MDK, Cortex-M0, No Optimization	MDK, Cortex-M0, Size Optimization	MDK, Cortex-M0, Speed Optimization

`// Calling a function`
`// with no arguments`
`LCD_Start();`
	`; do the function call`
	`bl   LCD_Start`
		`; same as for no optimization`	`; same as for no optimization`

`// Calling a function`
`with`
`// one argument`
`LCD_PrintInt8(128);`
	`; R0 = first argument`
	`movs  r0, #80`
	`bl    LCD_PrintInt8`
		`; same as for no optimization`	`; same as for no optimization`

`// Calling a function`
`with`
`// two arguments`
`LCD_Position(0, 2);`
	`; R0 = first argument`
	`; R1 = second argument`
	`movs  r1, #2`
	`movs  r0, #0`
	`bl    LCD_Position`
		`; same as for no optimization`	`; same as for no optimization`

C Code	MDK, Cortex-M0, No Optimization	MDK, Cortex-M0, Size Optimization	MDK, Cortex-M0, Speed Optimization

`// For loop:`
`void ForLoop(uint8 i)`
`{`
`  for(i = 0; i < 10; i++)`
`  {`
`    LCD_PrintInt8(i);`
`  }`
`}`
	`; prolog`
	`push  {r4, lr}`
	`mov   r4, r0`
	
	`movs  r4, #0  ; i = 0`
	`b.n   <ForLoop+0x12>`
	
	`<ForLoop+0x8>:`
	`; do the function call with`
	`; i as the argument in R0`
	`mov   r0, r4`
	`bl    LCD_PrintInt8`
	
	`adds  r0, r4, #1 ; i++`
	`uxtb  r4, r0 ; sign extend`
	
	`<ForLoop+0x12>:`
	`cmp   r4, #a  ; i < 10`
	`blt.n <ForLoop+0x8>`
	
	`pop   {r4, pc} ; return`
		`; prolog`
		`push  {r4, lr}`
		
		`movs  r4, #0  ; i = 0`
		
		`<ForLoop+0x4>:`
		`; do the function call with`
		`; i as the argument in R0`
		`mov   r0, r4`
		`bl    LCD_PrintInt8`
		
		`adds  r4, r4, #1 ; i++`
		`uxtb  r4, r4 ; sign extend`
		`cmp   r4, #a  ; i < 10`
		`bcc.n <ForLoop+0x4>`
		
		`pop   {r4, pc} ; return`
			`; same as for size optimization`

`// While loop`
`// i is type automatic, see`
`// Accessing Automatic Variables`
`// for details`
`uint8 i = 0;`
 	
`while (i < 10)`
`{`
`  LCD_PrintInt8(i);`
`  i++;`
`}`
	`; prolog`
	`push  {r4, lr}`
	
	`movs  r4, #0  ; i = 0`
	`b.n   <WhileLoop+0x10>`
	
	`<WhileLoop+0x6>:`
	`; do the function call with`
	`; i as the argument in R0`
	`mov   r0, r4`
	`bl    LCD_PrintInt8`
	
	`adds  r0, r4, #1 ; i++`
	`uxtb  r4, r0 ; sign extend`
	
	`<WhileLoop+0x10>:`
	`cmp   r4, #a  ; i < 10`
	`blt.n <WhileLoop+0x6>`
	
	`pop   {r4, pc}; return`
		`; prolog`
		`push  {r4, lr}`
		
		`movs  r4, #0  ; i = 0`
		
		`<WhileLoop+0x4>:`
		`; do the function call with`
		`; i as the argument in R0`
		`mov   r0, r4`
		`bl    LCD_PrintInt8`
		
		`adds  r4, r4, #1 ; i++`
		`uxtb  r4, r4 ; sign extend`
		
		`cmp   r4, #a  ; i < 10`
		`bcc.n <WhileLoop+0x4>`
		
		`pop   {r4, pc}; return`
			`; same as for size optimization`

C Code	MDK, Cortex-M0, No Optimization	MDK, Cortex-M0, Size Optimization	MDK, Cortex-M0, Speed Optimization

`// Conditional statement`
`void Conditional(uint8 i, uint8 j)`
`{`
`  if(j == 1)`
`  {`
`    LCD_PrintInt8(i);`
`  }`
`  else`
`  {`
`    LCD_PrintInt8(i + 1);`
`  }`
`}`
	`; prolog`
	`push    {r4, r5, r6, lr}`
	
	`mov    r4, r0`
	`mov    r5, r1`
	`cmp    r5, #1 ; j == 1`
	`bne.n  <Conditional+0x12>`
	
	`mov    r0, r4`
	`bl     LCD_PrintInt8`
	`b.n    <Conditional+0x1a>`
	
	`<Conditional+0x12>:`
	`; LCD_PrintInt8(i + 1)`
	`adds   r1, r4, #1`
	`uxtb   r0, r1 ; sign extend`
	`bl     LCD_PrintInt8`
	
	`<Conditional+0x1a>:`
	`; return`
	`pop    {r4, r5, r6, pc}`
		`; prolog`
		`push    {r4, lr}`
		
		`; if(j == 1)`
		`cmp    r1, #1`
		`beq.n  <Conditional+0xa>`
		
		`adds   r0, r0, #1 ; i + 1`
		`uxtb   r0, r0`
		
		`<Conditional+0xa>:`
		`bl     LCD_PrintInt8`
		
		`; return`
		`pop    {r4, pc}`
			`; same as for size optimization`

`// Switch case statements`
`void SwitchCase(uint8 j)`
`{`
`  switch(j)`
`  {`
`    case 0:`
`    LCD_PrintInt8(1);`
`    break;`
 	
`    case 1:`
`    LCD_PrintInt8(2);`
`    break;`
 	
`    default:`
`    LCD_PrintInt8(0);`
`    break;`
`  }`
`}`
	`; prolog`
	`push   {r4, lr}`
	
	`; switch(j)`
	`mov    r4, r0`
	`cmp    r4, #0`
	`beq.n  <SwitchCase+0xe>`
	`cmp    r4, #1`
	`bne.n  <SwitchCase+0x1e>`
	`b.n    <SwitchCase+0x16>`
	
	`<SwitchCase+0xe>:`
	`movs   r0, #1 ; case 0`
	`bl     LCD_PrintInt8`
	`b.n    <SwitchCase+0x26>`
	
	`<SwitchCase+0x14>:`
	`movs   r0, #2 ; case 1`
	`bl     LCD_PrintInt8`
	`b.n    <SwitchCase+0x26>`
	
	`<SwitchCase+0x1e>:`
	`movs   r0, #0 ; default`
	`bl     LCD_PrintInt8`
	`nop`
	
	`<SwitchCase+0x26>:`
	`nop`
	`pop    {r4, pc} ; return`
		`; prolog`
		`push   {r4, lr}`
		
		`; switch(j)`
		`cmp    r0, #0`
		`beq.n  <SwitchCase+0xe>`
		`cmp    r0, #1`
		`beq.n  <SwitchCase+0x12>`
		
		`movs   r0, #0 ; default`
		`b.n    <SwitchCase+0x14>`
		
		`<SwitchCase+0xe>:`
		`movs   r0, #1 ; case 0`
		`b.n    <SwitchCase+0x14>`
		
		`<SwitchCase+0x12>:`
		`movs   r0, #2 ; case 1`
		
		`<SwitchCase+0x14>:`
		`bl     LCD_PrintInt8`
		`pop    {r4, pc} ; return`
			`; same as for size optimization`

C Code	MDK, Cortex-M0, No Optimization	MDK, Cortex-M0, Size Optimization	MDK, Cortex-M0, Speed Optimization

`// Ternary operator`
`void Ternary(uint8 i)`
`{`
`  LCD_PrintInt8(`
`  (i == 1) ? 80 :`
`100);`
`}`
	`; prolog`
	`push  {r4, lr}`
	
	`mov   r4, r0 ; i == 1`
	`cmp   r4, #1`
	`bne.n  <Ternary+0xc>`
	
	`movs   r1, #50`
	`b.n   <Ternary+0xe>`
	
	`<Ternary+0xc>:`
	`movs   r1, #64`
	
	`<Ternary+0xe>:`
	`mov   r0, r1`
	`bl    LCD_PrintInt8`
	
	`pop   {r4, pc} ; return`
		`; prolog`
		`push  {r4, lr}`
		
		`cmp   r0, #1 ; i == 1`
		`beq.n  <Ternary+0xe>`
		
		`movs   r0, #64`
		
		`<Ternary+0x8>:`
		`bl    LCD_PrintInt8`
		`pop   {r4, pc} ; return`
		
		`<Ternary+0xe>:`
		`movs   r0, #50`
		`b.n   <Ternary+0x8>`
			`; same as for size optimization`

`// Addition operation`
`int DoAdd(int x, int y)`
`{`
`  return x + y;`
`}`
	`; no prolog`
	`mov   r2, r0`
	`adds  r0, r2, r1`
	`bx    lr ; return with result`
		`; no prolog`
		`adds  r0, r1`
		`bx    lr ; return with result`
			`; same as for size optimization`

`// Subtraction operation`
`int DoSub(int x, int y)`
`{`
`  return x - y;`
`}`
	`; no prolog`
	`mov   r2, r0`
	`subs  r0, r2, r1`
	`bx    lr ; return with result`
		`; no prolog`
		`subs  r0, r0, r1`
		`bx    lr ; return with result`
			`; same as for size optimization`

`// Multiplication`
`int DoMul(int x, int y)`
`{`
`  return x * y;`
`}`
	`; no prolog`
	`mov   r2, r0`
	`muls  r0, r2, r1`
	`bx    lr ; return with result`
		`; no prolog`
		`muls  r0, r1`
		`bx    lr ; return with result`
			`; same as for size optimization`

`// Division` `int DoDiv(int x, in`t y)` `{` ` return x / y;` `}`	`; prolog` `push  {r4, r5, r6, lr}` `mov   r4, r0` `mov   r5, r1` `mov   r1, r5` `mov   r0, r4` `bl    __aeabi_idiv` `; return with result` `pop   {r4, r5, r6, pc}`	`; prolog` `push  {r4, lr}` `bl    __aeabi_idiv` `; return with result` `pop   {r4, pc}`	`; same as for size optimization`

C Code	MDK, Cortex-M0, No Optimization	MDK, Cortex-M0, Size Optimization	MDK, Cortex-M0, Speed Optimization

`// Modulo operator`
`int DoMod(int x, int y)`
`{`
`  return x % y;`
`}`
	`; prolog`
	`push  {r4, r5, r6, lr}`
	`mov   r4, r0`
	`mov   r5, r1`
	`mov   r1, r5`
	`mov   r0, r4`
	`bl    __aeabi_idiv`
	`; return with result`
	`mov   r0, r1`
	`pop   {r4, r5, r6, pc}`
		`; prolog`
		`push  {r4, lr}`
		`bl    __aeabi_idiv`
		`; return with result`
		`mov   r0, r1`
		`pop   {r4, pc}`
			`; same as for size optimization`

`// Pointer`
`void Pointer(uint8 x,`
`uint8 *ptr)`
`{`
`  *ptr = *ptr + x;`
`  ptr++;`
`  LCD_PrintInt8(*ptr);`
`}`
	`; prolog`
	`push  {r4, r5, r6, lr}`
	`mov   r5, r0`
	`mov   r4, r1`
	
	`; *ptr = *ptr + x;`
	`ldrb  r0, [r4, #0]`
	`adds  r0, r0, r5`
	`strb  r0, [r4, #0]`
	
	`adds  r4, r4, #1 ; ptr++;`
	
	`ldrb  r0, [r4, #0]`
	`bl    LCD_PrintInt8`
	
	`pop   {r4, r5, r6, pc} ; return`
		`; prolog`
		`push  {r4, lr}`
		
		`; *ptr = *ptr + x`
		`ldrb  r2, [r1, #0] ; R1 = ptr`
		`adds   r0, r2, r0   ; R0 = x`
		`strb  r0, [r1, #0]`
		
		`; ptr++`
		`; LCD_PrintInt8(*ptr)`
		`ldrb  r0, [r1, #1]`
		`bl    LCD_PrintInt8`
		
		`pop   {r4, pc} ; return`
			`; same as for size optimization`

`// Function pointer`
`void FuncPtr(uint8 x,`
`  void`
`*fptr(uint8))`
`{`
`  (*fptr)(x);`
`}`
	`; prolog`
	`push  {r4, r5, r6, lr}`
	`mov   r5, r0`
	`mov   r4, r1`
	
	`; (*fptr)(x)`
	`mov   r0, r5`
	`; in a blx instruction, the`
	`; LS bit of the register`
	`; must be 1 to keep the CPU`
	`; in Thumb mode, or an`
	`; exception occurs`
	`blx   r4`
	
	`pop   {r4, r5, r6, pc} ; return`
		`; (*fptr)(x)`
		`; in a bx instruction, the LS`
		`; bit of the register must be`
		`; 1 to keep the CPU in Thumb`
		`; mode, or an exception occurs`
		
		`; function returns back to`
		`; caller of this function`
		`bx   r1`
			`; same as for size optimization`

C Code	MDK, Cortex-M0, No Optimization	MDK, Cortex-M0, Size Optimization	MDK, Cortex-M0, Speed Optimization

`// Packed structures`
`struct FOO_P`
`{`
`  uint8  membera;`
`  uint8  memberb;`
`  uint32 memberc;`
`  uint16 memberd;`
`} __attribute__`
`((packed));`
 	
`extern struct FOO_P`
`myfoo_p;`
 	
`void PackedStruct(void)`
`{`
`  myfoo_p.membera =  5;`
`  myfoo_p.memberb = 10;`
`  myfoo_p.memberc = 15;`
`  myfoo_p.memberd = 20;`
`}`
	`; prolog`
	`push  {r4, lr}`
	`; membera = 5`
	`movs  r0, #5`
	`ldr   r1, [pc, #18]`
	`strb  r0, [r1, #0]`
	`; memberb = 10`
	`movs  r0, #a`
	`strb  r0, [r1, #1]`
	`; memberc = 15`
	`adds  r1, r1, #2`
	`movs  r0, #f`
	`bl    __aeabi_uwrite4`
	`; memberd = 20`
	`movs  r1, #14`
	`ldr   r0, [pc,#8]`
	`strb  r1, [r0, #6]`
	`movs  r1, #0`
	`strb  r1, [r0, #7]`
	`pop   {r4, pc}  ; return`
	
	`.word  &myfoo_p`
		`; prolog`
		`push  {r4, lr}`
		`ldr   r4, [pc, #18]`
		`; membera = 5`
		`movs  r0, #5`
		`strb  r0, [r4, #0]`
		`; memberb = 10`
		`movs  r0, #a`
		`strb  r0, [r4, #1]`
		`; memberc = 15`
		`adds  r1, r4, #2`
		`movs  r0, #f`
		`bl    __aeabi_uwrite4`
		`; memberd = 20`
		`movs  r0, #14`
		`strb  r1, [r4, #6]`
		`movs  r0, #0`
		`strb  r1, [r4, #7]`
		`pop   {r4, pc}  ; return`
		
		`.word  &myfoo_p`
			`; same as for size optimization`

`// unpacked structures`
`struct FOO`
`{`
`  uint8  membera;`
`  uint8  memberb;`
`  uint32 memberc;`
`  uint16 memberd;`
`};`
 	
`extern struct FOO myfoo;`
 	
`void PackedStruct(void)`
`{`
`  myfoo.membera =  5;`
`  myfoo.memberb = 10;`
`  myfoo.memberc = 15;`
`  myfoo.memberd = 20;`
`}`
	`; no prolog`
	`; membera = 5`
	`movs   r0, #5`
	`ldr    r1, [pc, #10]`
	`strb   r0, [r1, #0]`
	`; memberb = 10`
	`movs   r0, #a`
	`strb   r0, [r1, #1]`
	`; memberc = 15`
	`movs   r0, #f`
	`str    r0, [r1, #4]`
	`; memberd = 20`
	`movs   r0, #14`
	`strh   r0, [r1, #8]`
	`bx     lr  ; return`
	
	`.word  &myfoo`
		`; same as for no optimization`	`; no prolog`
			`ldr    r0, [pc, #10]`
			`; membera = 5`
			`movs   r1, #5`
			`strb   r1, [r0, #0]`
			`; memberb = 10`
			`movs   r1, #a`
			`strb   r1, [r0, #1]`
			`; memberc = 15`
			`movs   r1, #f`
			`str    r1, [r0, #4]`
			`; memberd = 20`
			`movs   r1, #14`
			`strh   r1, [r0, #8]`
			`bx     lr  ; return`
			
			`.word  &myfoo`

A.5 Compiler Test Program

The following C code was used to generate the compiler output in the previous tables. It compiles for PSoC 4 and PSoC 5LP, for gcc and MDK, with no optimization and with size and speed optimization. It can be added to a PSoC Creator project; the following must also be done in the project:

- Add a Character LCD Component to the project schematic, and rename it to “LCD”.
- For PSoC 4, reduce the heap and stack size settings for these lower-memory parts. This is done in the Design-Wide Resource (DWR) window, System tab. Values of 0x100 and 0x400, for heap size and stack size respectively, are usually appropriate.

The code is in two files, main.c and test.c. This is main.c:

```
#include <project.h>
extern void ForLoop(uint8);
extern void WhileLoop(void);
extern void Conditional(uint8, uint8);
extern void SwitchCase(uint8);
extern void Ternary(uint8);
extern int  DoAdd(int, int);
extern int  DoSub(int, int);
extern int  DoMul(int, int);
extern int  DoDiv(int, int);
extern int  DoMod(int, int);
extern void Pointer(uint8, uint8 *);
extern void FuncPtr(uint8, void (*)(uint8));
extern void PackedStruct(void);
extern void UnpackedStruct(void);

struct FOO /* structures are unpacked by default */
{
      uint8  membera;
      uint8  memberb;
      uint32 memberc;
      uint16 memberd;
};

struct FOO_P /* packed structure */
{
      uint8  membera;
      uint8  memberb;
      uint32 memberc;
      uint16 memberd;
} __attribute__ ((packed));

uint8 myData = 6;
struct FOO_P myfoo_p;
struct FOO   myfoo;

int main()
{
    /* Place your initialization/startup code here (e.g. MyInst_Start()) */
    LCD_Start();

    /* CyGlobalIntEnable; */ /* Uncomment this line to enable global interrupts. */
    for(;;)
    {
        LCD_PrintInt8(128);
        LCD_Position(0, 2);
        ForLoop(9);
        WhileLoop();
```

```
        Conditional(3, 4);
        SwitchCase(4);
        Ternary(5);
        LCD_PrintNumber((uint16)DoAdd(5, 4));
        LCD_PrintNumber((uint16)DoSub(5, 4));
        LCD_PrintNumber((uint16)DoMul(5, 4));
        LCD_PrintNumber((uint16)DoDiv(5, 4));
        LCD_PrintNumber((uint16)DoMod(5, 4));
        Pointer(4, &myData);
        FuncPtr(3, &LCD_PrintInt8);
        PackedStruct();
        UnpackedStruct();
    } /* end of for(;;) */
} /* end of main() */
```

And this is test.c:

```
#include <project.h>

struct FOO /* structures are unpacked by default */
{
   uint8  membera;
   uint8  memberb;
   uint32 memberc;
   uint16 memberd;
};

struct FOO_P /* packed structure */
{
    uint8  membera;
    uint8  memberb;
    uint32 memberc;
    uint16 memberd;
} __attribute__ ((packed));

extern struct FOO_P myfoo_p;
extern struct FOO   myfoo;

void ForLoop(uint8 i)
{
    for(i = 0; i < 10; i++)
    {
        LCD_PrintInt8(i);
    }
}

void WhileLoop(void)
{
    uint8 i = 0;
    while(i < 10)
    {
        LCD_PrintInt8(i);
        i++;
    }
}

void Conditional(uint8 i, uint8 j)
{
    if(j == 1)
    {
```

```
            LCD_PrintInt8(i);
        }
        else
        {
            LCD_PrintInt8(i + 1);
        }
    }

    void SwitchCase(uint8 j)
    {
        switch(j)
        {
            case 0:
            LCD_PrintInt8(1);
            break;

            case 1:
            LCD_PrintInt8(2);
            break;

            default:
            LCD_PrintInt8(0);
            break;
        }
    }

    void Ternary(uint8 i)
    {
        LCD_PrintInt8((i == 1) ? 80 : 100);
    }

    int DoAdd(int x, int y)
    {
        return x + y;
    }

    int DoSub(int x, int y)
    {
        return x - y;
    }

    int DoMul(int x, int y)
    {
        return x * y;
    }

    int DoDiv(int x, int y)
    {
        return x / y;
    }

    int DoMod(int x, int y)
    {
        return x % y;
    }

    void Pointer(uint8 x, uint8 *ptr)
    {
        *ptr = *ptr + x;
        ptr++;
        LCD_PrintInt8(*ptr);
```

```
}

void FuncPtr(uint8 x, void *fptr(uint8))
{
    (*fptr)(x);
}

void PackedStruct(void)
{
    myfoo_p.membera =  5;
    myfoo_p.memberb = 10;
    myfoo_p.memberc = 15;
    myfoo_p.memberd = 20;
}

void UnpackedStruct(void)
{
    myfoo.membera =  5;
    myfoo.memberb = 10;
    myfoo.memberc = 15;
    myfoo.memberd = 20;
}
```

Document History

Document Title: PSoC® 4 and PSoC 5LP ARM Cortex Code Optimization - AN89610

Document Number: 001-89610

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	4275133	MKEA	02/07/2014	New application note
*A	4994599	MKEA	10/29/2015	Clarified that PSoC 5LP cannot execute code from an 8-bit EMIF memory. Added a reference to AN90799, PSoC 4 Interrupts.

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
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

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

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