PSoC™ Arm® Cortex® code optimization

About this document

Scope and purpose

This document 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 that can lead to increased efficiency and reduced power consumption. This document provides information and recommendations on how to optimize the flash on PSoC™ 4 devices.

Intended audience

This document is intended for customers who want to optimize C and assembler code for the Arm® Cortex® CPUs in PSoC™ 4 and PSoC™ 5LP device code and to PSoC™ Creator and ModusToolbox™ software.

Table of contents

Table of 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 do 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), Keil MDK (Microcontroller Development Kit), or IAR C compiler is recommended. Knowledge of the Thumb-2 assembly language used by the CPUs will also help.

For PSoC™ 4 devices, you should know how to use either ModusToolbox™ software or PSoC™ Creator. If you are new to PSoC™ 4, you can find introductions to the device in AN79953 - Getting started with PSoC™ 4 MCU. If you are new to ModusToolbox™ software, see the ModusToolbox™ software. If you are new to PSoC™ Creator, see the PSoC™ Creator.

For PSoC™ 5LP devices, you should know how to use PSoC™ Creator, the integrated development environment for PSoC™ 3, PSoC™ 4 and PSoC™ 5LP. If you are new to PSoC™ 5LP, you can find introductions to the device in 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 shown in Figure 1. Note that the Cortex®-M0+ registers in PSoC™ 4100PS/PSoC™ 4000S/4100S devices are similar to Cortex®-M0 registers below. See the "Processor Core Registers Summary" section in Cortex®-M0+ technical reference manual for details.

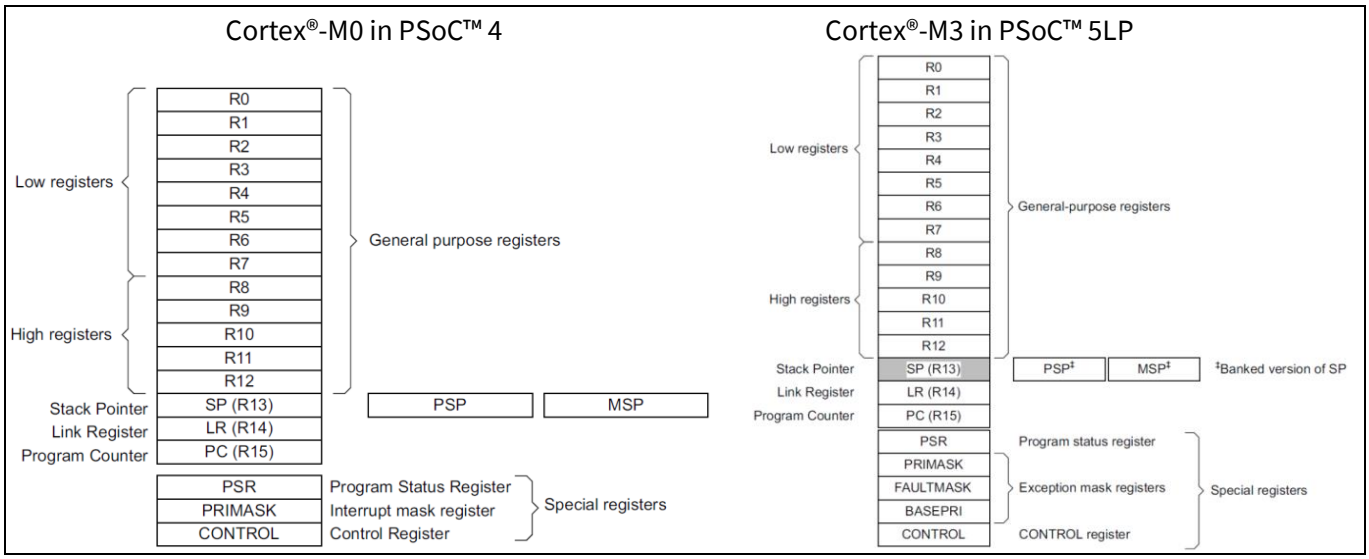

Figure 1 Cortex® CPU Architectures

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/M0+ and Cortex®-M3 have a very similar address map, as shown in Figure 2.

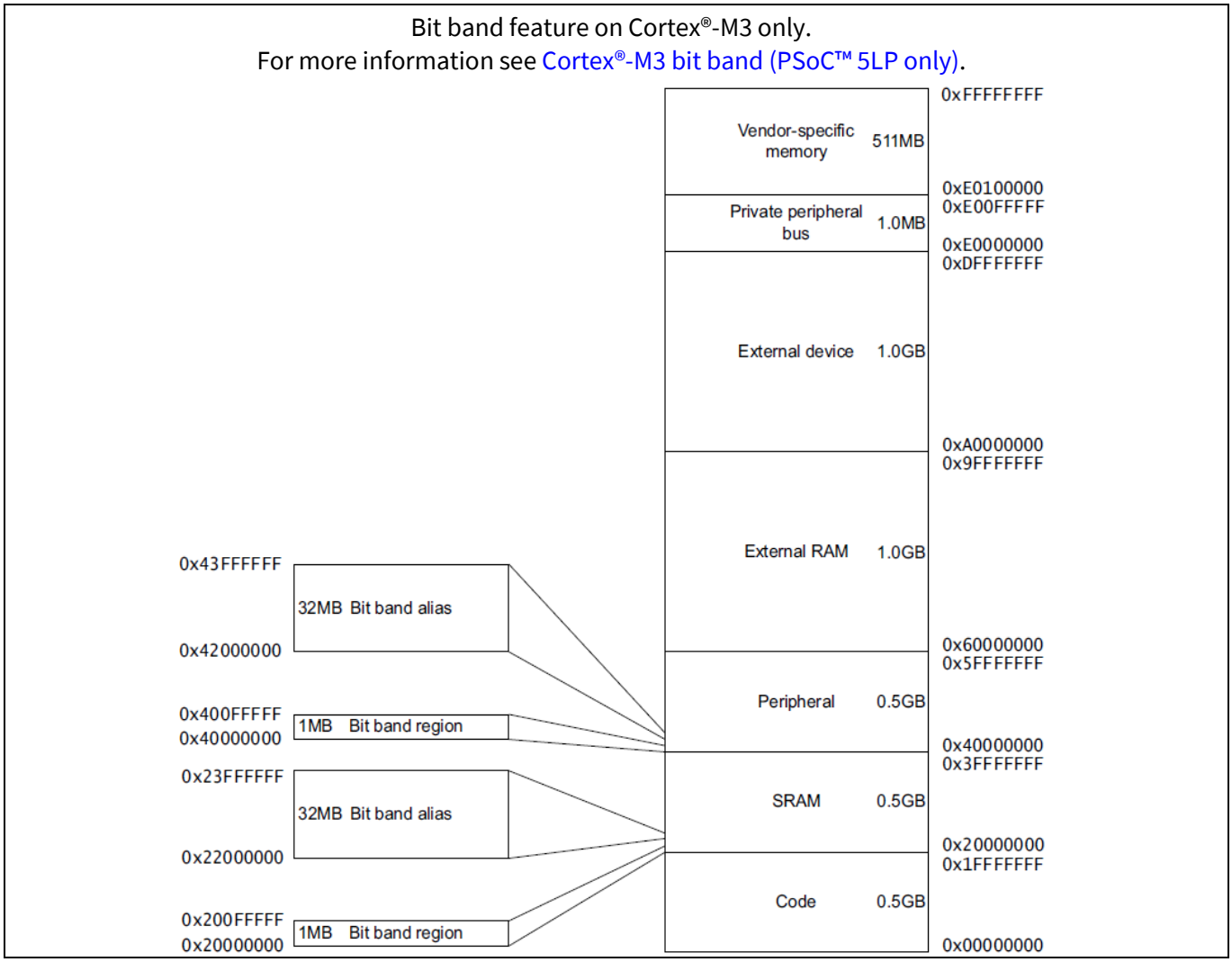

Figure 2 Cortex® address map

The address space is 4 GB (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 MB of the region. See Cortex®-M3 bit band (PSoC™ 5LP only) for more information.

Although the Cortex® CPUs can access a 4-GB address space, within the PSoC™ devices, only a small fraction of these addresses access PSoC™ memory or registers. The following is an overview of the location of the PSoC™ memory and registers in the Cortex® address space; for details, see the memory maps in the device datasheets or Reference Manuals (RMs).

2.2.1 PSoC™ 4 address map

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

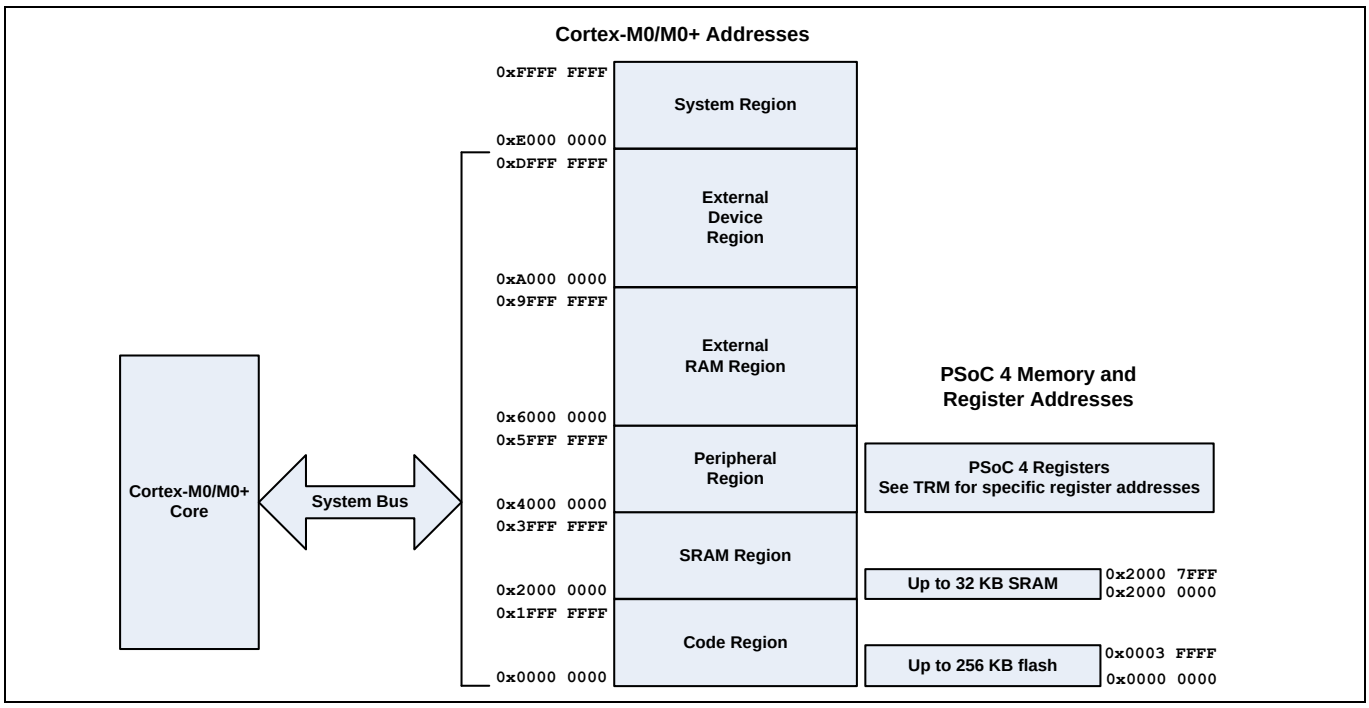

Figure 3 PSoC™ 4 address map

The PSoC™ 4 memory and registers are addressed as follows:

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

All memory accesses are 32-bit.

Code can be placed in 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 shown in Figure 4. 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, Cortex®-M3 can do simultaneous parallel accesses of the Code region and the other regions, for more efficient operation.

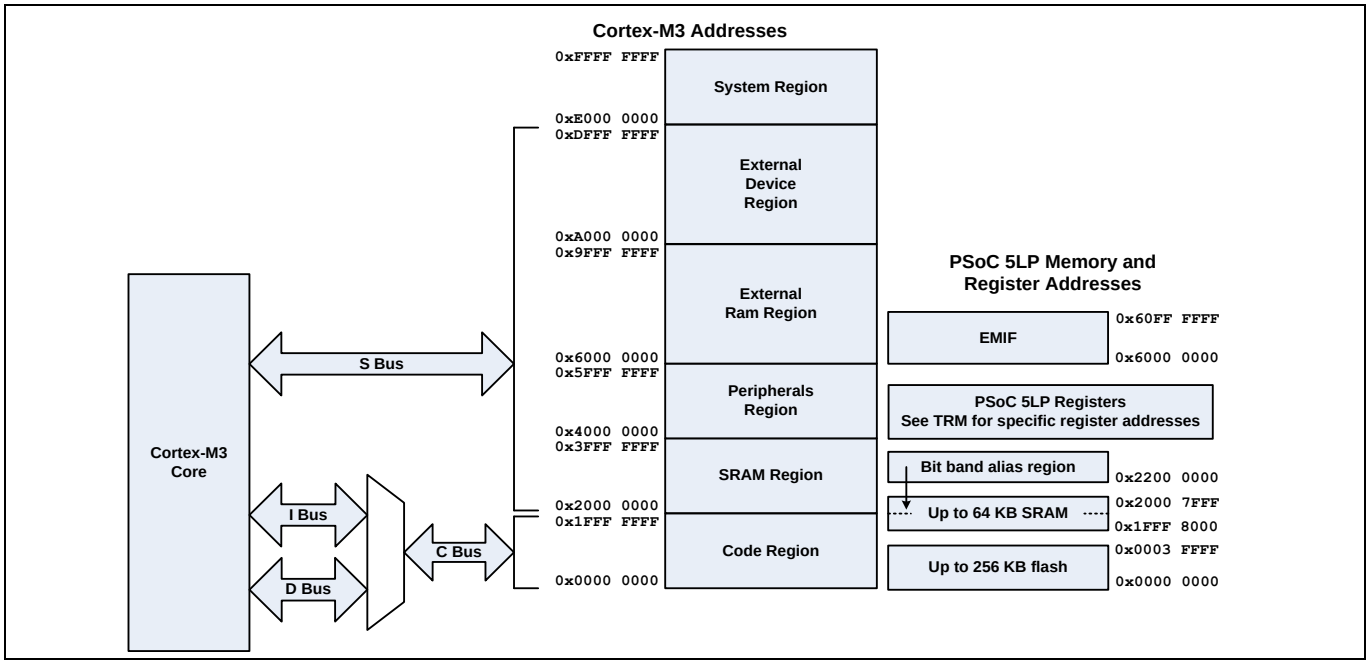

Figure 4 PSoC™ 5LP address map

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 shown in Figure 4. 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.

- 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.
- 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, Reference manual, or the External Memory Interface (EMIF) Component datasheet.

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

PSoC™ 5LP and some PSoC™ 4 devices 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 - PSoC™ 3 and PSoC™ 5LP - Getting started with 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™ 3 and PSoC™ 5LP interrupts/AN90799 – PSoC™ 4 interrupts.

3 Compiler general topics

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

Note: *All C code examples shown in this application note are designed for use with the C compilers supported by PSoC™ Creator 4.3: GCC 5.4, ModusToolbox: GCC 10.3.1, the Keil Microcontroller Development Kit (MDK) version 5.03, and IAR 8.50.4. The GCC 5.4 compiler is included free with your PSoC™ Creator installation, while the GCC 11.3.1 compiler is included free with your ModusToolbox™ installation. MDK must be purchased; however, an object-size-limited evaluation version, MDK-Lite, is available free from Keil. IAR must also be purchased; however, several free limited-feature evaluation versions are also available from IAR Systems. Compiler optimizations are turned off 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 good 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 ModusToolbox™ projects, use the following macros that are included with the MDK, GCC, and IAR compilers, respectively. In the ModusToolbox™ software project Makefile, change the **Toolchain** field to reflect your selection. Also, ensure that the `CY_COMPILER_PATH` field points to the .bin file directory of your compiler.

Note that for MDK, GCC, and IAR compilers, you are checking just whether `__ARMCC_VERSION`, `__GNUC__`, `__ICCARM__` is defined respectively, indicating that compiler is being used. You do not necessarily need to care about its actual value, i.e., the compiler version.

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

To apply this technique to PSoC™ Creator projects, you can use the same macros as shown above.

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 ModusToolbox™ software:

1. You can observe the generated assembly in the debugger disassembly window as shown in Figure 5.
2. You can modify the `CFLAGS` or `CXXFLAGS` fields in the Makefile to include list file generation options in your compiler command. These flags vary depending on the toolchain used. For the default GCC compiler, an example of these options looks like this:
   `CFLAGS=-Wa,-adhlns=$@.lst`

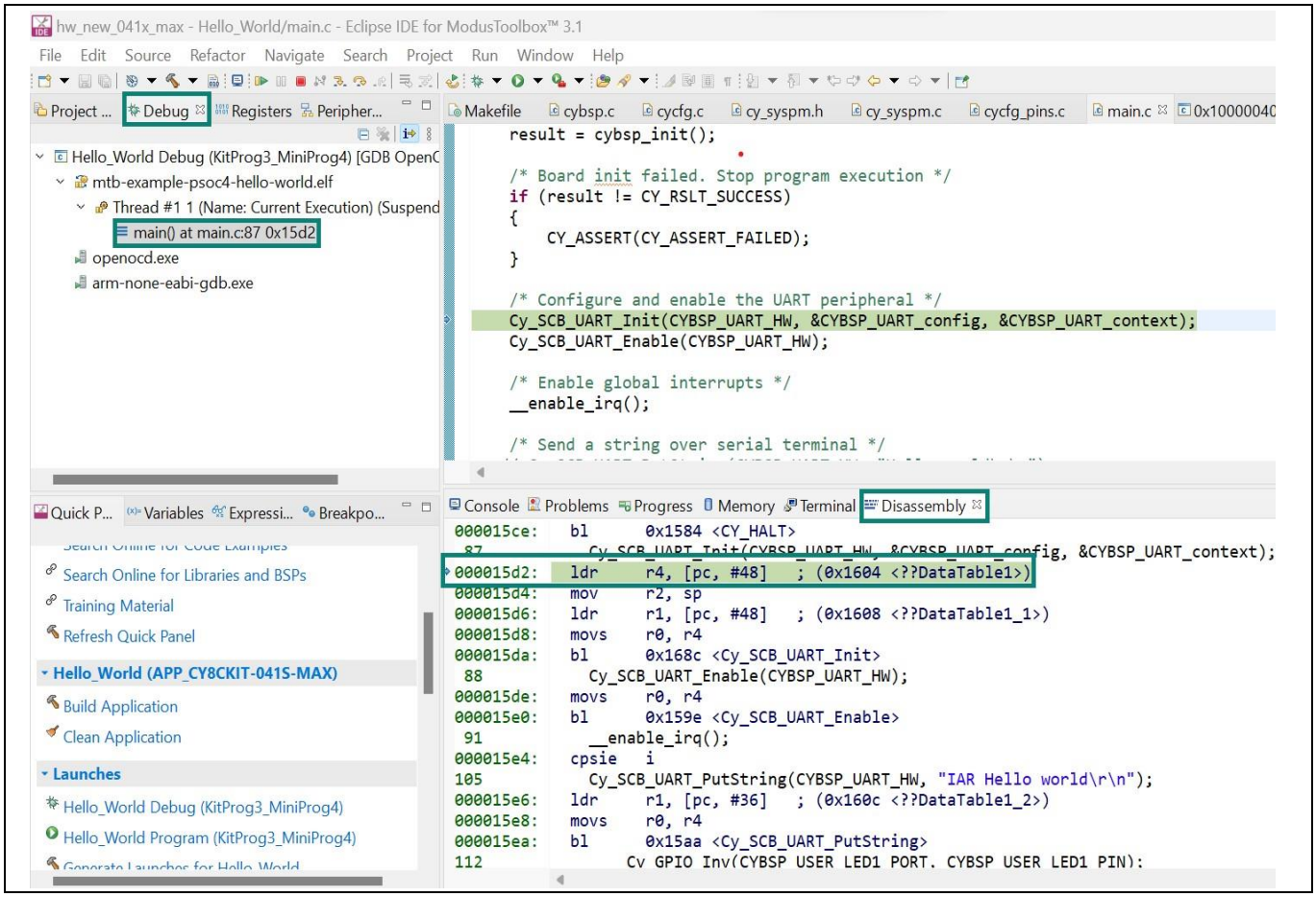

Figure 5 Eclipse IDE for ModusToolbox™ software disassembly window

There are two options to view the output assembler code in PSoC™ Creator:

1. Open the list file corresponding to the compiled C file (*filename.lst*), as shown in Figure 6. The default PSoC™ Creator project build setting is to create a list file; select **Project** > **Build Settings** > **Compiler** > **General**.
2. Use the disassembly window in the debugger (**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.

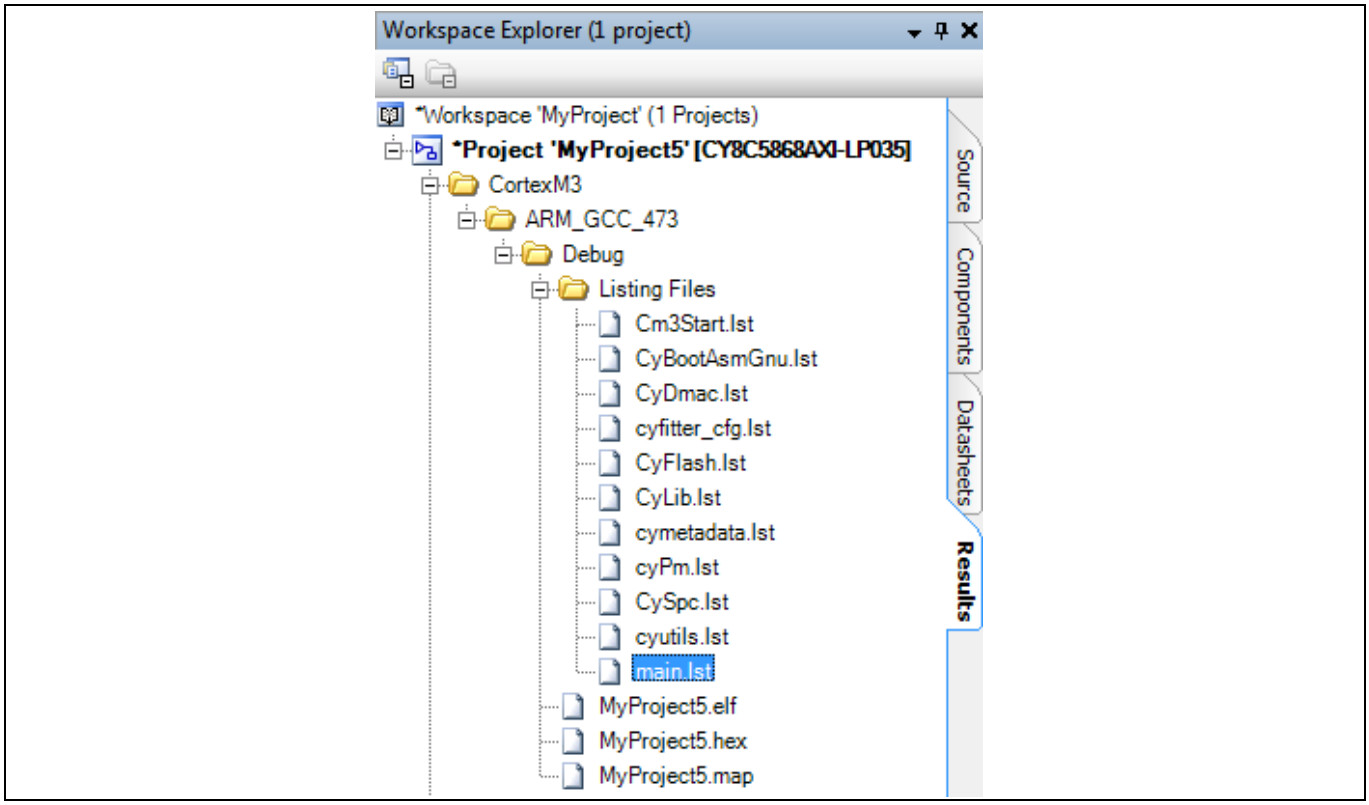

Figure 6 Listing files in PSoC™ Creator

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.

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.

ModusToolbox™ software allows you to set compiler optimizations for the entire project using the Makefile. Setting the `CONFIG` field to `Custom` and adding a compiler optimization option to the `CFLAGS (C)` or `CXXFLAGS (C++)` field will set the optimization. With IAR and MDK, you can use `#pragma` to set optimization for an individual function. For more information, see the C documentation.

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 GCC are **none**, **debug**, **minimal**, **high**, **speed**, and **size.** For MDK, PSoC™ Creator offers **none**, **size**, and **speed** optimizations. This is different from the 11 levels of optimization offered for the Keil 8051 compiler. IAR provides similar levels of optimization, offering **none**, **low**, **medium**, **high-balanced**, **high-speed**, and **high-size**.

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 cannot set optimization levels for individual functions except for using certain function Attributes.

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 good 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 or ModusToolbox™ software Makefile build settings.

See Appendix: Compiler output details for specific examples of how various optimization options work.

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>))
```

IAR supports a limited selection of GCC-style attributes using this syntax.

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

4 Accessing variables

When reviewing the 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, MDK, and IAR compilers 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 Address map for details.

Different compiler optimizations implement one or the other of these two methods; see Appendix: Compiler output details for detailed examples.

As a coding good practice, minimize the use of global variables. Doing so with the PSoC™ Cortex® CPUs may also act to reduce the 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 shown in Table 2.

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: Compiler output details 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 shown in Table 3.

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`

C code	Assembler code
	/ mov sp, r7 pop {r7} bx lr / return */

To minimize code size and maximize speed, 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: Compiler output details for examples.

4.4 LDR and STR instructions

These instructions are used to read from and write to 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 shown in Table 4.

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; 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; 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 code only where it is needed to increase efficiency, and makes it easier to mix C and assembler code. The GCC, MDK, and IAR compilers all 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 the GCC, MDK, and IAR compilers. To effectively use the methods described in this section, it is important to understand the register architectures on which they are based – see Register set for details.

Note: *The following examples show assembler code for Cortex®-M3; Cortex®-M0/M0+ uses a more limited subset of the Cortex®-M3 instructions. See Arm® Cortex® documentation.*

Note: *Most assembler instructions act on 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 the 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");
```

Mixing C and assembler code

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 the inline assembler is the same as that for GCC, except that “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
  }
```

IAR uses the keyword `__asm`, however `asm` is also available for use. Note that the keyword `asm` is not available when compiling with the option `-strict`. The syntax for the IAR inline assembler is the same as for GCC.

5.2 Automatic variables

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

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

Note: *xxxGCC 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. See 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.

Mixing C and assembler code

In IAR, the `register` keyword is not honored, and automatic variables are placed either in registers or on the stack, depending on register availability. IAR does provide a way to reference variables in the inline assembler using the syntax:

```
__asm("assembler instruction %output_operand, %input_operand1, …, %input_operandN"
      : "=constraint" (output_symbol)
      : "constraint1" (intput1_symbol), …, "constraintN" (inputn_symbol));
```

Using the constraint `"r"` causes a general-purpose register to be used in the expression. For example:

```
void main()
{
    /* variables initialized without register keyword */
    int foo = 5L;
    int bar;

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

    /* Assembly version */
    __asm("ADDS %0, %1, #1"
          : "=r" (bar)
          : "r" (foo));
}
```

This example compiles into:

```
/* variables initialized without register keyword */
int foo = 5L;
        MOVS     R1,#+5
int bar;

/* C code version */
bar = foo + 1;
        MOVS     R0,R1
  ADDS     R0,R0,#+1

/* Assembly version */
__asm("ADDS %0, %1, #1"
      : "=r" (bar)
      : "r" (foo));
  ADDS R0, R1, #1
```

The C code version in this example uses two instructions to accomplish what the inline assembly accomplishes in one. Note that it is possible that other compiler optimization levels could produce different outputs and could remove the extra instruction from the C code version.

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 variable, 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 /* an address */
```

Let us repeat the previous example using global 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 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]
```

These results may be different if compiler optimizations are used.

Mixing C and assembler code

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 pseudo instruction, 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.

In IAR, the inline assembler accesses global variables the same way as accessing automatic variables:

```
/* bar = foo + 1; */
__asm("ADDS %0, %1, #1"
      : "=r" (bar)
      : "r" (foo));
```

Similar to MDK, however, the output assembly is quite different:

```
__asm("ADDS %0, %1, #1"
      : "=r" (bar)
      : "r" (foo));

      LDR      R0,??main_0
      LDR      R0,[R0, #+0]
      ADDS     R0, R0, #1
      LDR      R1,??main_0+0x4
      STR      R0,[R1, #+0]
```

The additional instructions are required for loading the variable addresses. In this case, as with the MDK case, the inline assembly instruction is acting as a pseudo instruction that compiles into a series of instructions. The

C version of this statement compiles into the same instructions, meaning that there is no advantage to using the inline assembly.

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.

Thus, if you limit the number of arguments to four, you can write assembler code 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
}
```

Mixing C and assembler code

IAR:

```
uint32 addFunc(uint32 a, uint32 b, uint32 c, uint32 d)
{
    int32 rtnval;

    /* the arguments are in registers R0 - R3 */
    __asm volatile ("ADDS R0, R0, R1\n"
                    "ADDS R0, R0, R2\n"
                    "ADDS %0, R0, R3"
                    : "=r" (rtnval) : );

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

6 Special-function instructions

Some CPUs have special-function instructions which are not normally used by C compilers. These instructions 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 code to more easily gain access to these instructions.

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

6.1 Saturation instructions

Saturation is commonly used in signal processing, for example, when a signal is amplified, as shown in Figure 7. 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.

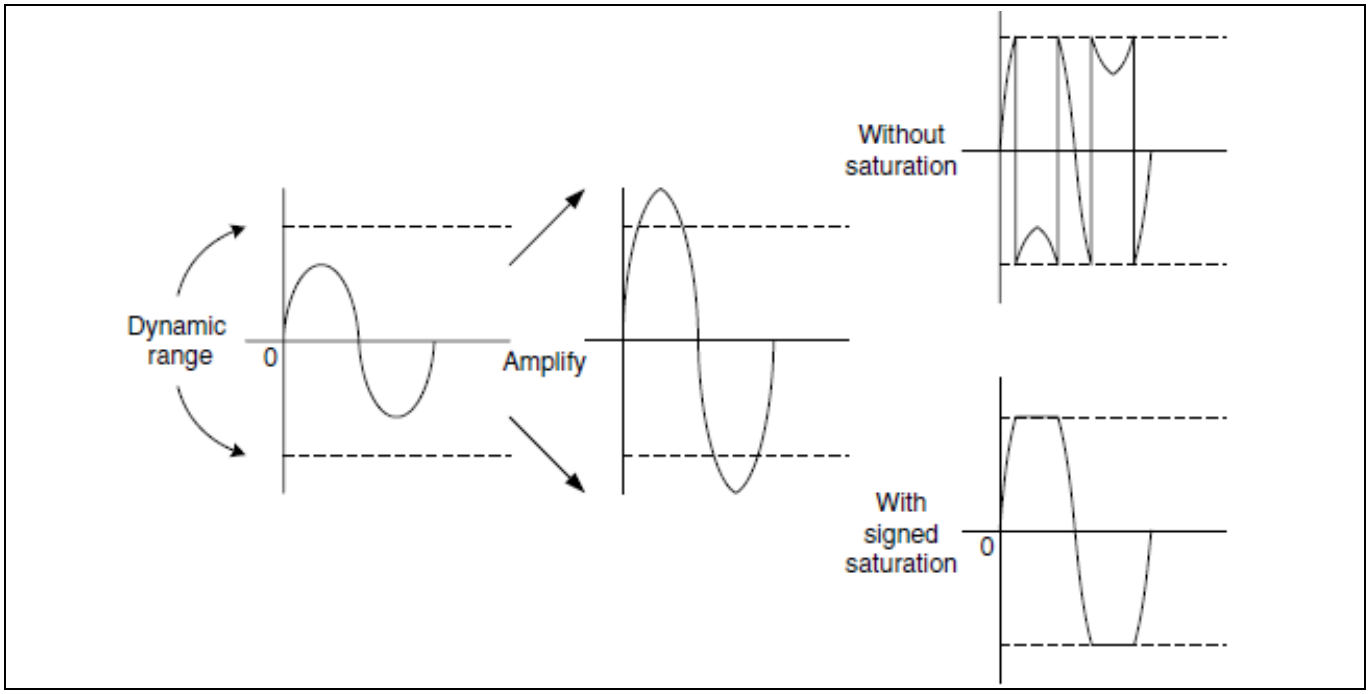

Figure 7 Saturation operation

Saturation can be done in C using multiple comparison and `if-else` statements, but 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

1op 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: These 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*. In ModusToolbox™ software, the functions can be found in the Libs > PSoCxPDL > cmsis> include > *cmsis_<toolchain>.h*.

The saturation intrinsic 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 code

Example	Mixed C and assembler code

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

MDK Example	`void main()`
	`{`
	`  int data_unsat;`
	`  int data_sat;`
	
	`  __asm("ssat data_sat, 8, data_unsat");`
	`  . . .`
	`}`

IAR Example	`void main()`
	`{`
	`  int data_unsat;`
	`  int data_sat;`
	
	`  __asm("SSAT %0, #8, %1");`
	`  : "=r" (data_sat);`
	`  : "r" (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. Therefore, if the value is less than -128, the result is -128; if the value is greater than +127, the result is +127. Thus, 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 I²C port. (The SPI protocol is an exception; for details, see one of the PSoC™ Creator Serial Peripheral Interface (SPI) component datasheets.) With 8-bit CPUs, complex data structures can be transmitted and received byte-by-byte; 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 the memory as 32-bit words; therefore, they work most efficiently when the data is stored in 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, 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 */
}
```

This 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. By default, for Cortex® CPUs, GCC and MDK compilers save structures in the **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. However, a more serious problem can occur in cases such as 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 GC C, MDK, and IAR 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:

Table 6 Syntax

GCC:	MDK:	IAR:

`struct myStruct`
`{`
`. . .`
`} __attribute__ ((packed));`
	`__packed struct myStruct`
	`{`
	`. . .`
	`};`
		`__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 or typedef declarations.

8 Compiler libraries

For GCC, MDK, and IAR 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 7 shows the flash and SRAM memory consumption for the three compilers for PSoC™ 5LP and PSoC™ 4:

Table 7 Memory consumption with a `pow()` function call

	GCC	MDK	IAR
PSoC™ 5LP			
Flash	8939	7696	1498
SRAM	405	196	2241
PSoC™ 4			
Flash	14374	7700	1488
SRAM	364	312	1160

If the call to the `pow()` library function is replaced with the following equivalent code, you use a lot less memory, as shown in Table 8:

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

Table 8 Memory consumption without using a `pow()` function call

	GCC	MDK	IAR
PSoC™ 5LP			
Flash	1582 (-82.3%)	1444 (-81.2%)	982 (-34.4%)
SRAM	301 (-25.7%)	200 (-32.4%)	2241
PSoC™ 4			
Flash	1198 (-91.7%)	1004 (-87.0%)	992 (-33.3%)
SRAM	252 (-30.8%)	216 (-30.8%)	1160

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 4.3, 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.

Compiler libraries

Note: One of the 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 shown in Figure 8. This feature is disabled by default. Enabling it increases flash usage by 10K to 15K bytes.

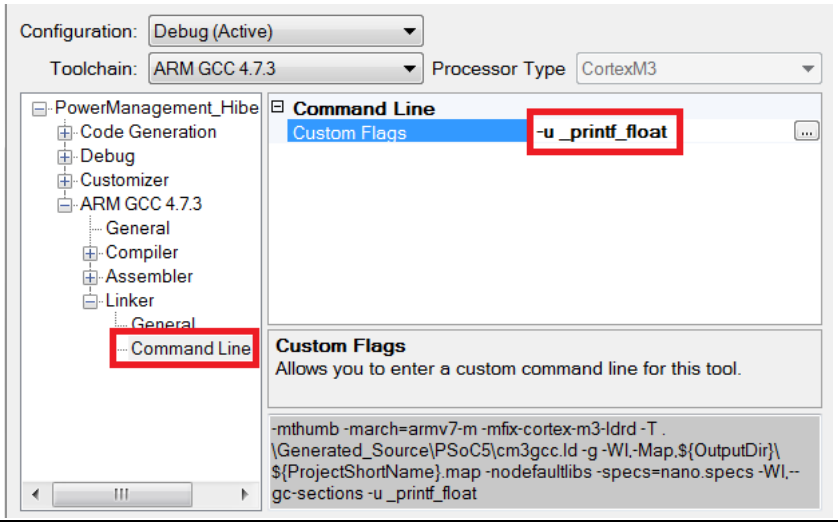

Figure 8 Enable floating-point formatting support in newlib-nano

2. Change the library to the full-featured newlib, as shown in Figure 9. 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.

Compiler libraries

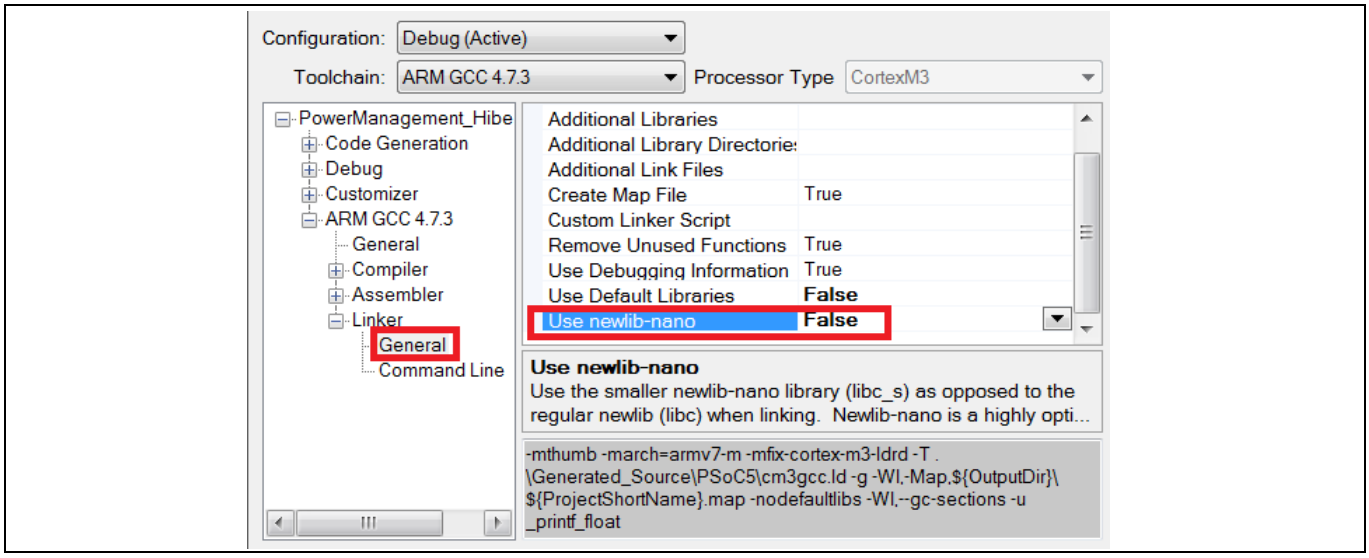

Figure 9 Disabling newlib-nano

MDK also offers a reduced-function library called Arm® C Micro-library. See the C 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. See GCC or C documentation.

In the ModusToolbox™, the default linker script file is in *<Project Directory>\bsps\TARGET_APP_<Kit Name>\TOOLCHAIN_<Toolchain Name>* as shown in Figure 10. You can also specify a custom linker file to be used in the `LINKER_SCRIPT` Makefile variable.

Note: For ModusToolbox™ software 3.1, to use a custom linker file in the application, you need to specify the path of the custom linker script file to LINKER_SCRIPT in the Makefile; otherwise, it will take the default linker script. Also, the default linker scripts can be overwritten by placing the custom link scripts in the template folder of the application. See ModusToolbox™ user guide for details.

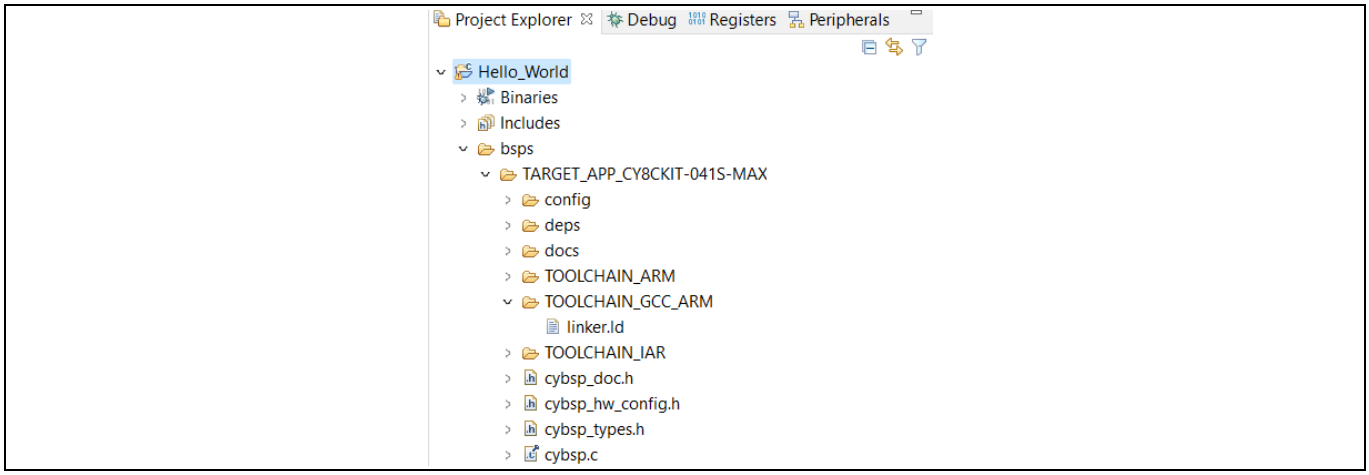

Figure 10 ModusToolbox™ software linker script file

Placing code and variables

In PSoC™ Creator, the default linker script files are in the *Generated Source* folder after the project is built, as shown in Figure 11.

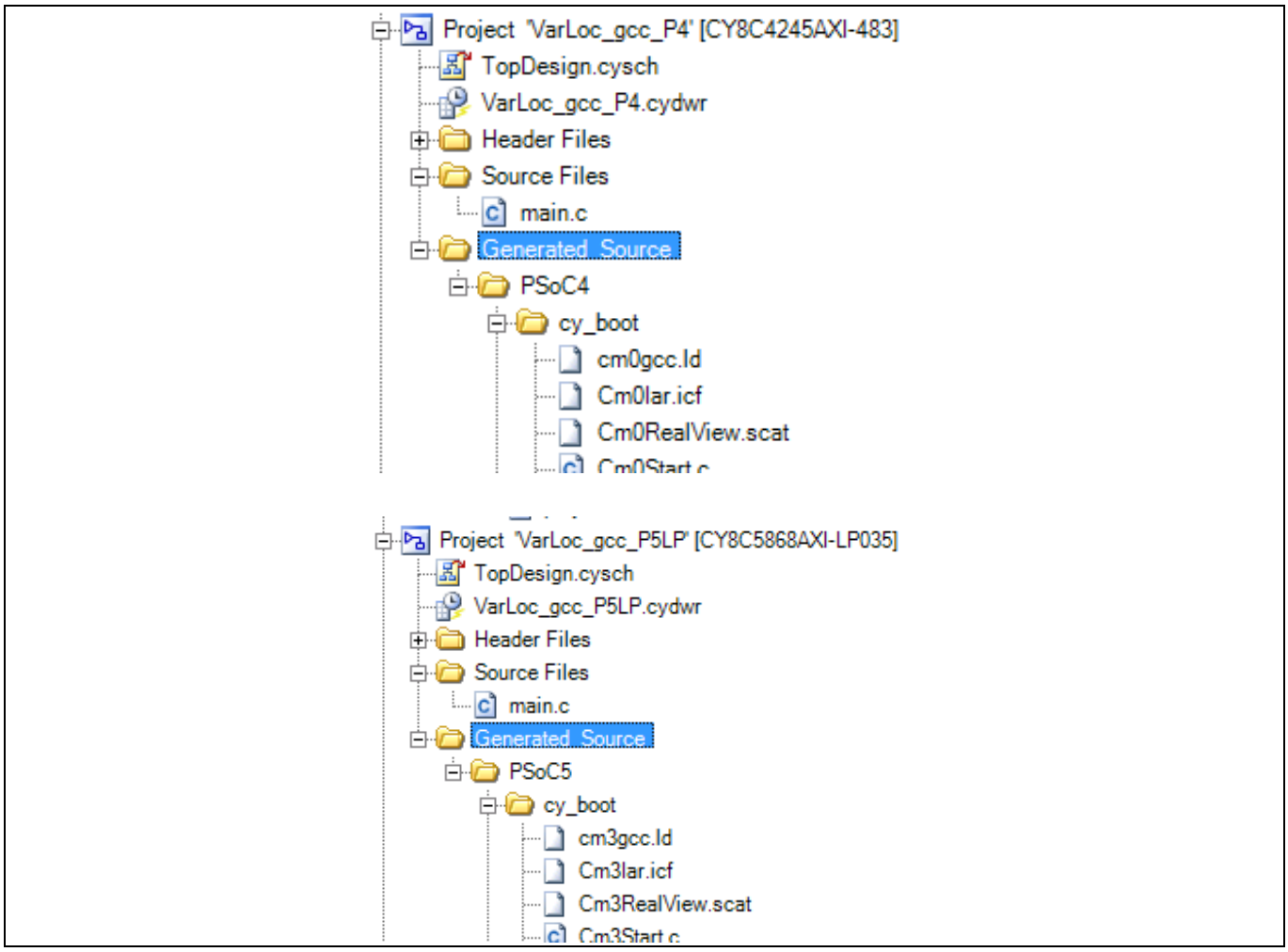

Figure 11 PSoC™ Creator linker script files

For GCC, the linker script file is of type *.ld*, for MDK the linker script file is of type *.scat* or *.sct* (for "scatter"), and for IAR, the linker script file is of type *.icf*.

Note: In PSoC™ Creator, linker script files are automatically generated at project build time; 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: ***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 (**Project** > **Existing Item…**) and save it in the project folder. A custom *.scat* file must be saved in the *PSoCx* folder under *Generated Source*.

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 PSoC™.

For example, In ModusToolbox™ software 3.1, for a PSoC™ 4 with 384 KB flash and 32 KB SRAM:

```
MEMORY
{
  FLASH (rx): ORIGIN = 0x00000000, LENGTH = 0x00060000
  RAM (rwx): ORIGIN = 0x20000000, LENGTH = 0x00008000

}
```

For example, In PSoC™ Creator, 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 `rom` region 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; 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, in PSoC™ Creator, for 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 are shown.) Figure 12 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: { ... } > FLASH
.data: { ... } > RAM AT>FLASH
.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 the 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 other sections are in code SRAM. This can be changed; see Modify the linker script file.

Complete documentation of *.ld* file usage can be found in GCC docmentaiton. For details, see C documentation.

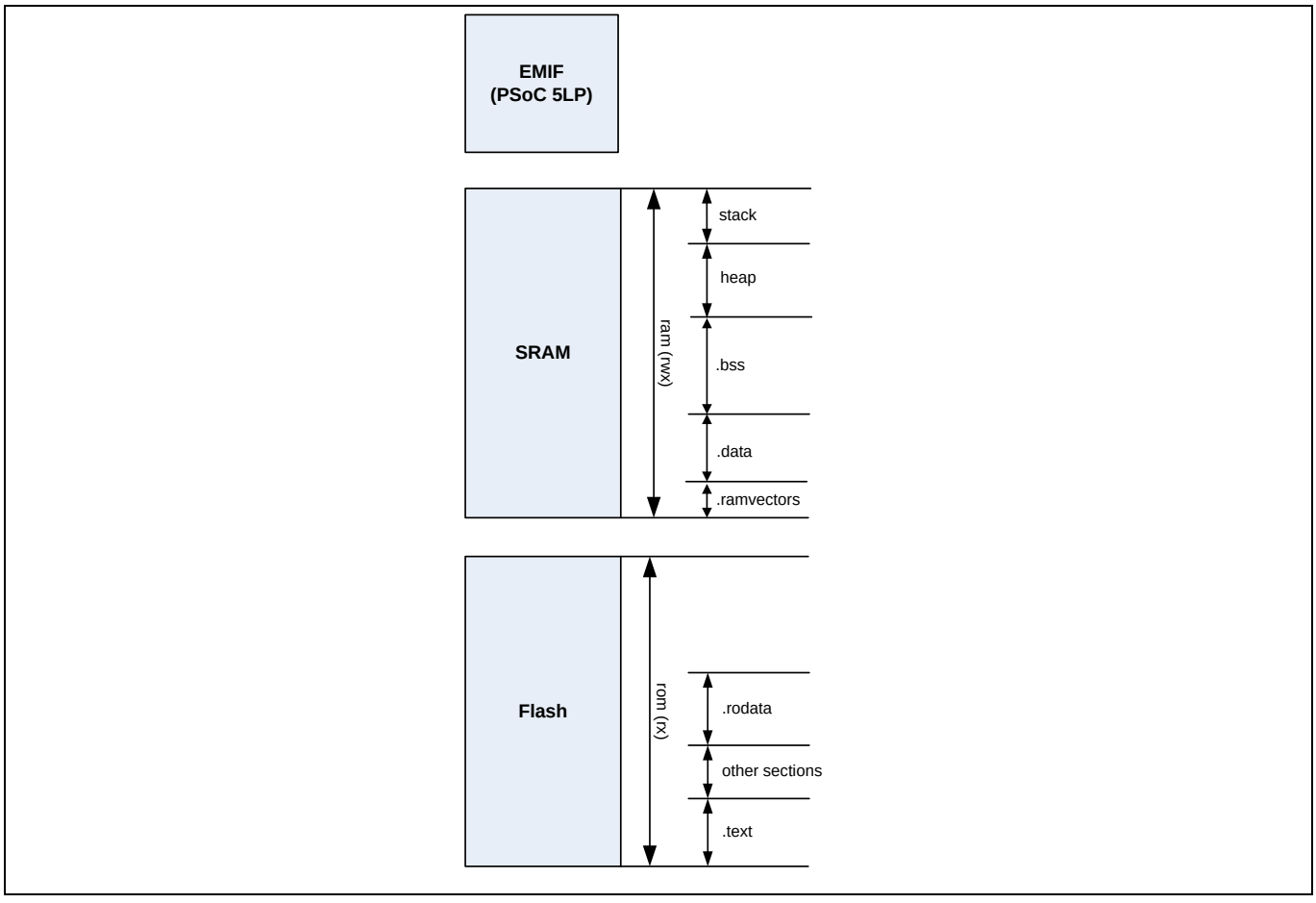

Figure 12 SECTIONS command and PSoC™ memory

9.1.2 Linker script file for MDK

A *.scat* or *.sct* (for “scatter”) file has no commands; instead it defines regions and sections. It has a **load region**. The load region contains several **execution regions**, which in turn contain one or more **section attributes**, for example:

For ModusToolbox™ software **3.1**

```
LR_ROM ...      /* load region */
{
    ER_ROM ...    /* execution region */
    {
        *.oc (RESET, +First)
        *(InRoot$$Sections)
        .ANY (+RO)  /* section attribute */
        .ANY (+XO)
    }

    ER_RAM_VECTORS ...
    {
        * (.bss.RESET_RAM, +FIRST)
    }

    RW_RAM_DATA ...
    {
        * (+RW, +ZI)
    }
    RW_IRAM1 ...
    {
        * (.noinit)
    }
    ARM_LIB_HEAP ...
    {
    }

    ARM_LIB_STACK ...
    {
    }

}
```

For PSoC™ Creator

```
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 are shown.) Figure 13 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 13.*

Note: *For most applications, it can be assumed that the stack and heap are in upper SRAM and other sections are in code SRAM. This can be changed; see Modify the linker script file.*

Complete documentation of *.scat* file usage can be found in MDK documentation. For details, see C documentation.

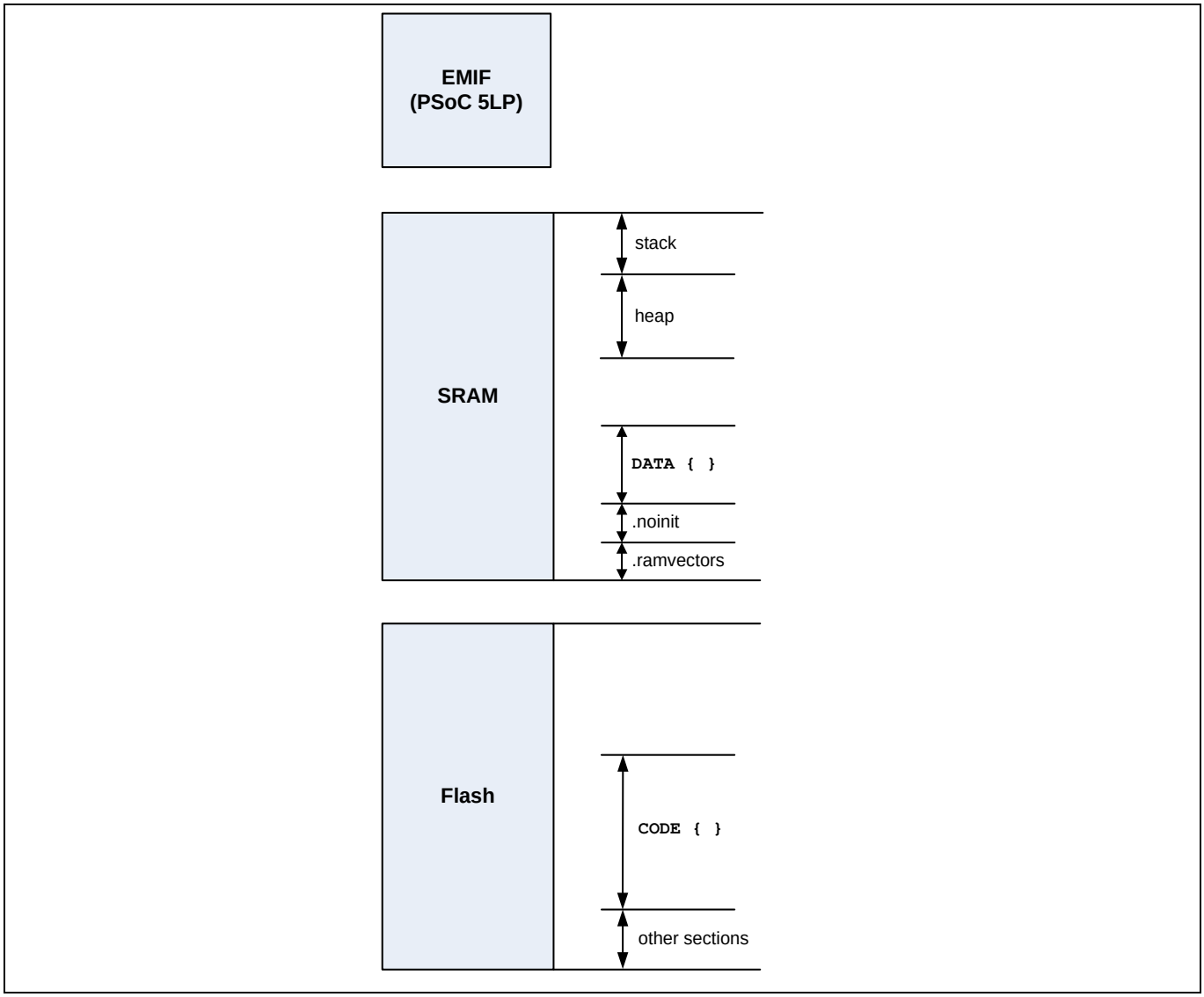

Figure 13 . .*scat* file sections and PSoC™ memory

9.1.3 Linker script file for IAR

The *.icf* file used by IAR is a configuration file that is initialized to a default state for your selected part by IAR's ILINK tool. The default *.icf* file has predefined memory regions presented in the header of the file:

For ModusToolbox™ software 3.1

```
/*-Editor annotation file-*/
/* IcfEditorFile="$TOOLKIT_DIR$\config\ide\IcfEditor\cortex_v1_4.xml" */
/*-Specials-*/
define symbol __ICFEDIT_intvec_start__ = 0x00000000;

/* The symbols below define the location and size of blocks of memory in the target.
 * Use these symbols to specify the memory regions available for allocation.
 */

/* The following symbols control RAM and flash memory allocation.
 * You can change the memory allocation by editing RAM and Flash symbols.
 */
/* RAM */
define symbol __ICFEDIT_region_IRAM1_start__ = 0x20000000;
define symbol __ICFEDIT_region_IRAM1_end__   = 0x20007FFF;
define symbol __ICFEDIT_region_IRAM2_start__ = 0x0;
define symbol __ICFEDIT_region_IRAM2_end__   = 0x0;
define symbol __ICFEDIT_region_IRAM3_start__ = 0x0;
define symbol __ICFEDIT_region_IRAM3_end__   = 0x0;
define symbol __ICFEDIT_region_ERAM1_start__ = 0x0;
define symbol __ICFEDIT_region_ERAM1_end__   = 0x0;
define symbol __ICFEDIT_region_ERAM2_start__ = 0x0;
define symbol __ICFEDIT_region_ERAM2_end__   = 0x0;
define symbol __ICFEDIT_region_ERAM3_start__ = 0x0;
define symbol __ICFEDIT_region_ERAM3_end__   = 0x0;

/* Flash */
define symbol __ICFEDIT_region_IROM1_start__ = 0x00000000;
define symbol __ICFEDIT_region_IROM1_end__   = 0x0005FFFF;
define symbol __ICFEDIT_region_IROM2_start__ = 0x0;
define symbol __ICFEDIT_region_IROM2_end__   = 0x0;
define symbol __ICFEDIT_region_EROM1_end__   = 0x0;
define symbol __ICFEDIT_region_EROM1_start__ = 0x0;
define symbol __ICFEDIT_region_EROM2_start__ = 0x0;
define symbol __ICFEDIT_region_EROM2_end__   = 0x0;
define symbol __ICFEDIT_region_EROM3_start__ = 0x0;
define symbol __ICFEDIT_region_EROM3_end__   = 0x0;

/*-Sizes-*/
if (!isdefinedsymbol(__STACK_SIZE)) {
define symbol __ICFEDIT_size_cstack__ = 0x0400;
} else {
define symbol __ICFEDIT_size_cstack__ = __STACK_SIZE;
}
define symbol __ICFEDIT_size_proc_stack__ = 0x0;

/* Defines the minimum heap size. The actual heap size will be expanded to the end of the
   stack region   */
if (!isdefinedsymbol(__HEAP_SIZE)) {
define symbol __ICFEDIT_size_heap__ = 0x0080;
} else {
define symbol __ICFEDIT_size_heap__ = __HEAP_SIZE;
}

/**** End of ICF editor section. ###ICF###*/
```

Placing code and variables

For PSoC™ Creator

```
/*-Editor annotation file-*/
/* IcfEditorFile="$TOOLKIT_DIR$\config\ide\IcfEditor\cortex_v1_0.xml" */
/*-Specials-*/
define symbol __ICFEDIT_intvec_start__ = 0x00000000;
/*-Memory Regions-*/
define symbol __ICFEDIT_region_ROM_start__ = 0x0;
define symbol __ICFEDIT_region_ROM_end__ = 262144 - 1;
define symbol __ICFEDIT_region_RAM_start__ = 0x20000000 - (65536 / 2);
define symbol __ICFEDIT_region_RAM_end__ = 0x20000000 + (65536 / 2) - 1;
/*-Sizes-*/
define symbol __ICFEDIT_size_cstack__ = 0x0800; define symbol __ICFEDIT_size_heap__ = 0x80;

/**** End of ICF editor section. ###ICF###*/
```

These regions can be reconfigured using the IAR tools. For more information, see IAR documentation.

An *.icf* file contains definitions of memory size, region addresses, initialization, and attributes of the contents of those defined regions. For example:

For ModusToolbox™ software 3.1

```
/* Allocate memory space of the maximum possible addressable size */
define memory mem with size = 4G;

/* Allocate memory regions (ROM and RAM) in the addressable space */
define symbol use_IROM1 = (__ICFEDIT_region_IROM1_start__ != 0x0 ||
__ICFEDIT_region_IROM1_end__ != 0x0);
define symbol use_IRAM1 = (__ICFEDIT_region_IRAM1_start__ != 0x0 ||
__ICFEDIT_region_IRAM1_end__ != 0x0);
define region IROM1_region = mem:[from __ICFEDIT_region_IROM1_start__ to
__ICFEDIT_region_IROM1_end__];
define region IRAM1_region = mem:[from __ICFEDIT_region_IRAM1_start__ to
__ICFEDIT_region_IRAM1_end__];

/* Create a stack and define stack properties */
define block CSTACK     with alignment = 8, size = __ICFEDIT_size_cstack__      { };
define block PROC_STACK with alignment = 8, size = __ICFEDIT_size_proc_stack__ { };

/* Create a heap and define properties */
define block HEAP       with expanding size, alignment = 8, minimum size =
__ICFEDIT_size_heap__ { };
define block HSTACK {block HEAP, block PROC_STACK, last block CSTACK};

/* handle initialization */
initialize by copy { readwrite };
do not initialize  { section <section name> };

/* Place sections in defined regions */
Place at start of <region> { section <section name> };
Place in <region> { <section name> };
```

For PSoC™ Creator

```
/* Allocate memory space of the maximum possible addressable size */
Define memory Mem with size = <size>;

/* Allocate memory regions (ROM and RAM) in the addressable space */
Define region ROM = Mem:[from <start_address> size <size>];
define region RAM = Mem:[from <start_address> size <size>];

/* Create a stack and define stack properties */
define block STACK with size = <size>, alignment = <alignment width> { };
```

Placing code and variables

```
/* Create a heap and define properties */
define block HEAP with size = <size>, alignment = <alignment width> { };

/* handle initialization */
Do not initialize ( section <noinit section name> );
Initialize by copy ( readwrite ); /* initialize read/write sections */

/* Place sections in defined regions */
Place at start of <region> ( section <section name> );

Place in <region> ( <section name> );
```

It is possible to place sections of code or variables into regions by using their access attribute (R/W or R/O). In this case, the syntax is:

```
place in <region> ( <readwrite or readonly>)
```

For example:

```
place in IRAM1_region  { readwrite };
```

Figure 14 shows where these regions and sections are placed in PSoC™ 4 and PSoC™ 5LP memory:

- `.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

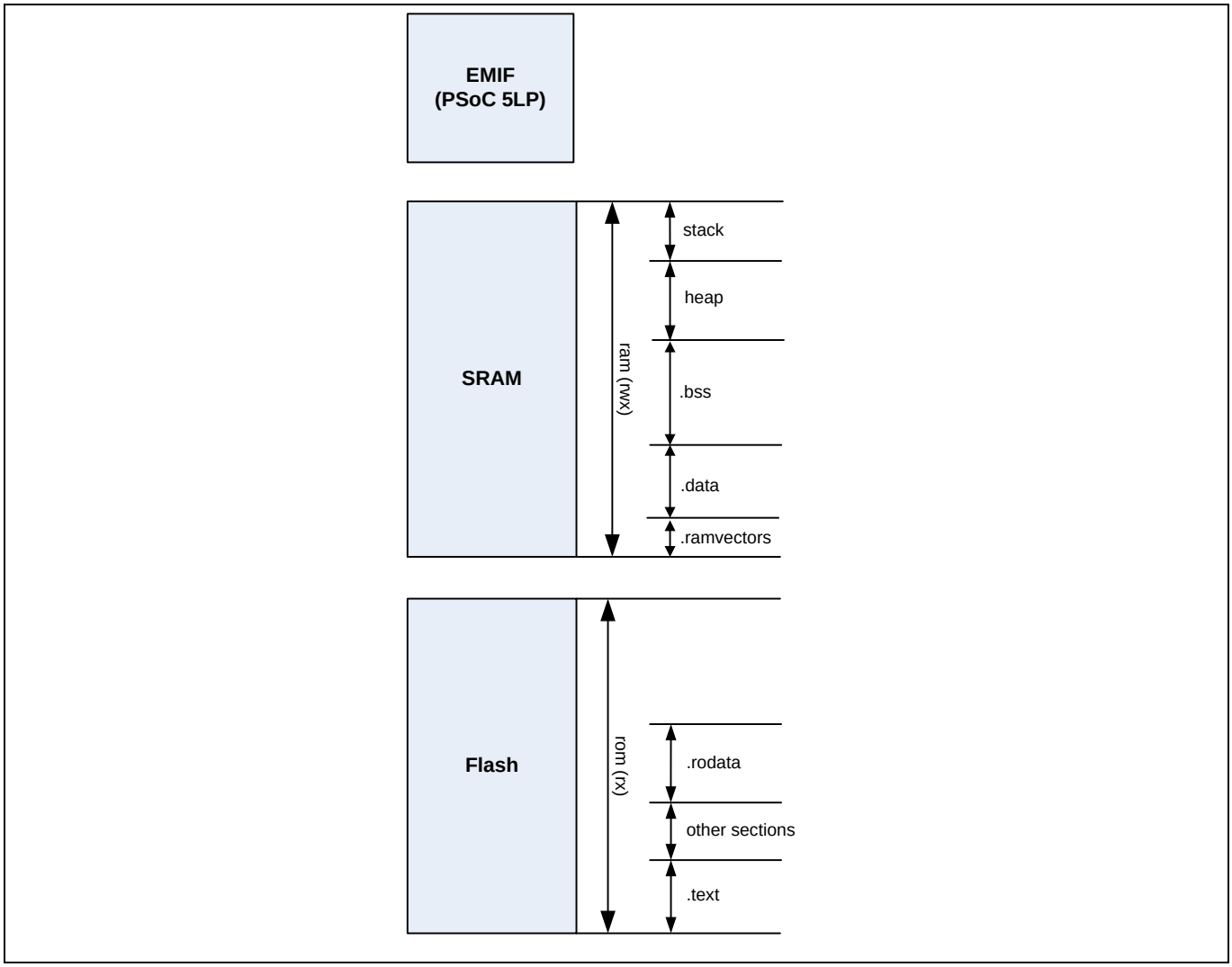

Figure 14 *. .icf* file sections and PSoC™ memory

The following points are valid for all three linker file types:

1. Only global and C static variables are included. C automatic variables are handled differently; see Automatic variables.
2. In PSoC™ Creator, the size of the heap and stack are defined in the project DWR window, **System** tab, or in the project's custom linker file. In ModusToolbox™ software, the size of the heap and stack are defined in the project linker file. 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.

   In a custom linker script file, the stack and heap size can be specified for each compiler. The following examples show configurations for a stack of size 0x0400. The heap size will very according to the RAM (size and origin) and stack size for GCC and MDK compliers. In IAR complier, you can directly set the size for heap.

For ModusToolbox™ software 3.1

GCC *.ld*

```
__STACK_SIZE = 0x00000400;

.heap :
 {
    . = ALIGN(4);
     __HeapBase = .;
     __end__ = .;
     PROVIDE(end = .);
     . = ORIGIN(RAM) + LENGTH(RAM) - __STACK_SIZE;
     __HeapLimit = .;
 } > RAM

.stack :
 {
    . = ORIGIN(RAM) + LENGTH(RAM) - __STACK_SIZE;
    . = ALIGN(4);
    __StackLimit = .;
    . = . + __STACK_SIZE;
    . = ALIGN(4);
    __StackTop = .;
 } > RAM
```

MDK *.sct*

```
#define __STACK_SIZE    0x00000400

ARM_LIB_HEAP  +0 EMPTY ((__RAM_START+__RAM_SIZE)-AlignExpr(ImageLimit(RW_IRAM1),
8)-__STACK_SIZE)
{
}

ARM_LIB_STACK (__RAM_START+__RAM_SIZE) EMPTY -__STACK_SIZE
{
}
```

IAR *.icf*

```
define symbol __STACK_SIZE = 0x0400;
define symbol __HEAP_SIZE   = 0x0080;

/*-Sizes-*/
if (!isdefinedsymbol(__STACK_SIZE)) {
define symbol __ICFEDIT_size_cstack__ = 0x0400;
} else {
define symbol __ICFEDIT_size_cstack__ = __STACK_SIZE;
}
define symbol __ICFEDIT_size_proc_stack__ = 0x0;
/* Defines the minimum heap size. The actual heap size will be expanded to the end
of
   the stack region */
if (!isdefinedsymbol(__HEAP_SIZE)) {
define symbol __ICFEDIT_size_heap__ = 0x0080;
```

Placing code and variables

```
} else {
define symbol __ICFEDIT_size_heap__ = __HEAP_SIZE;
}

define block CSTACK     with alignment = 8, size = __ICFEDIT_size_cstack__     {
};
define block PROC_STACK with alignment = 8, size = __ICFEDIT_size_proc_stack__ {
};
define block HEAP       with expanding size, alignment = 8, minimum size =
__ICFEDIT_size_heap__ { };
define block HSTACK {block HEAP, block PROC_STACK, last block CSTACK};
define block RO     {first section .intvec, readonly};
```

For PSoC™ Creator

GCC *.ld*

```
.heap (NOLOAD) :
{
    . = _end;
    . += 0x80;
    __cy_heap_limit = .;
} >ram

.stack (__cy_stack - 0x0800) (NOLOAD) :
{
    __cy_stack_limit = .;
    . += 0x0800;
} >ram
```

MDK .scat

```
ARM_LIB_HEAP (0x2000000 + (65536 / 2) - 0x80 - 0x0800) EMPTY 0x80
{
}

ARM_LIB_STACK (0x2000000 + (65536 / 2).) EMPTY -0x0800
{
}
```

IAR *.icf*

```
define symbol size_stack__ = 0x0800;
define symbol size_heap__   = 0x80;

define block STACK with alignment = 8, size = size_stack__ {};
define block HEAP with alignment = 8, size = size_heap__ {};

/* Group stack and heap together, with stack last */
define block HSTACK(block HEAP, last block STACK};

/*Place HSTACK block in end of RAM */
"HSTACK"       : place at end of RAM_region { block HSTACK} ;
```

3. Although the heap has a defined size in the PSoC™ Creator 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.

9.1.4 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 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 12) and IAR, and the values in flash are in the `CODE (RO)` section for MDK (Figure 13). They are copied into the variables in the `.data` section for GCC and IAR 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 and IAR) or the `DATA (ZI)` section (MDK).

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

9.1.5 Map file

The GCC and MDK linkers have an option to produce a *.map* file when a project is built. In PSoC™ Creator, you can find the file in the **Results** tab in the Workspace Explorer window, as shown in Figure 15. In ModusToolbox™ software, this file is found in the *<Project Directory>\build\APP_<Kit Name>\<Config>.*

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 navigate to **Project** > **Build Settings** > **Linker** > **Command Line** > **Custom Flags**.

In ModusToolbox™ software, add the `--symbols` flag to the `LDFLAGS` field of the Makefile.

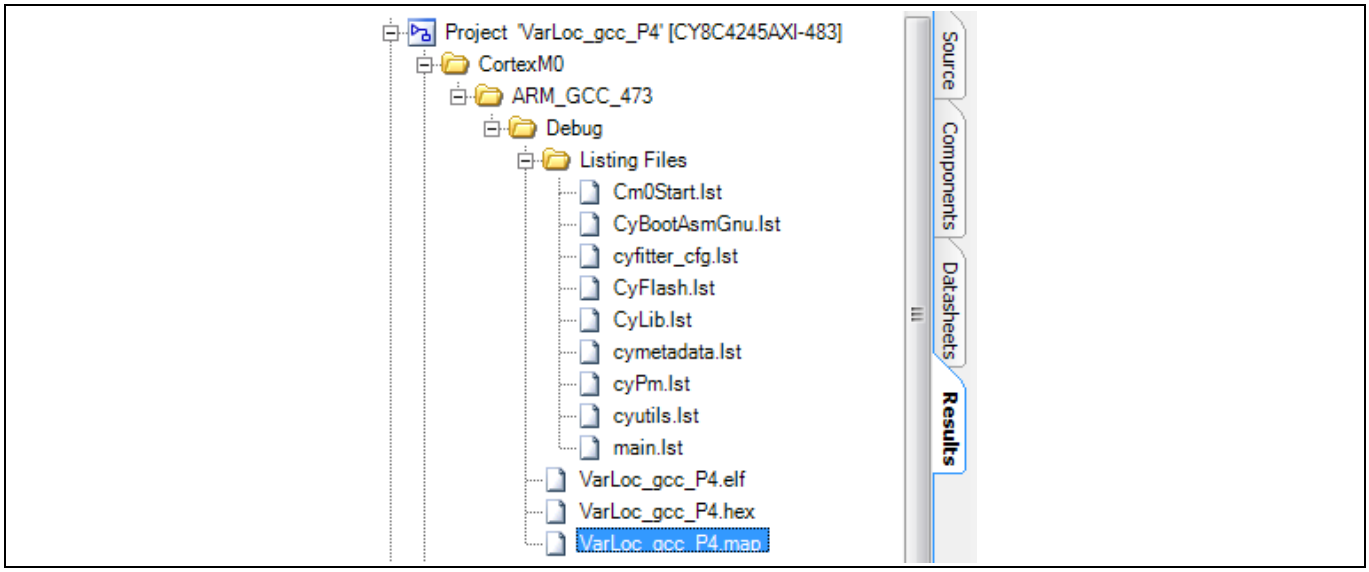

Figure 15 *.map* file in PSoC™ Creator

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

Do the following to place C functions or variables (including arrays and structures) in custom locations:

1. Define 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 project. Copy the generated linker script file to a custom file, and then modify it to add and locate the sections from Step 2.
4. Rebuild the 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 clearly understand 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/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 "attribute"):

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

This keyword can be used with GCC, IAR, and MDK. PSoC™ Creator and ModusToolbox™ software provide a convenient macro `CY_SECTION` to simplify this statement such as the following:

```
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 References for more information. */
CYISR(MyFunction) CY_SECTION(".MY_section");
```

PSoC™ Creator and ModusToolbox™ software also provide a convenient macro `CY_NOINIT`, which places a variable in the `.noinit` section; see Figure 12 and Figure 13:

```
/* 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 files 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, as shown in Table 9.

Table 9 Modify the linker script file

For ModusToolbox™ software 3.1

```
MEMORY
   {
     FLASH (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
   }
```

For PSoC™ Creator

```
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,*

For PSoC creator, change `.stack: { ... } >ram` *to* `.stack: { ... } >upperram.`

For ModusToolbox™ software 3.1, 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, as shown in Table 10.

Table 10 Example

For ModusToolbox™ software 3.1

```
.data:
{
  ...
  *(.MY_section)
  ...
} > RAM AT> FLASH
```

For PSoC™ Creator

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

Placing code and variables

In some cases, a section may be optimized out, for instance, if there are no symbols present in that section. To prevent the removal of a custom section, you can specify that you want to “keep” the section with the keyword `KEEP`. In GCC and IAR, the `KEEP` attribute is applied in the linker file. In MDK, `keep` is a command-line option for the linker, as shown in Table 11.

Table 11 Example

For ModusToolbox™ software 3.1

GCC	IAR	MDK

`.data:`
`{`
`  ...`
`  KEEP(*(.MY_section))`
`  ...`
`} > RAM AT> FLASH`
	`KEEP {  section .MY_section`
	`};`
		`/* In the command line */`
		`--keep=.MY_section`

For PSoC™ Creator

GCC	IAR	MDK

`.data:`
`{`
`  ...`
`  KEEP(*(.MY_section))`
`  ...`
`} > ram AT> rom`
	`KEEP {  section .MY_section`
	`};`
		`/* In the command line */`
		`--keep=.MY_section`

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, as shown in Table 12.

Table 12 Example

For GCC

For ModusToolbox™ software 3.1

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

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

```
RW_RAM_DATA +0
{
  * (.MY_section)
  * (+RW, +ZI)
}
```

For PSoC™ Creator

```
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)
}
```

For IAR

For ModusToolbox™ software 3.1

In IAR, modify the linker script *.icf* file. The procedure for adding a section is simple, just add your execution section as a new line. You can place the section in an absolute address in memory, for example:

```
".My_Section" : place at address mem : <address> { readonly section .My_Section };
```

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

```
/* places "My_Section" in "readwrite" section within RAM */
".My_Section"           : place in RAM_region { readwrite section .My_Section };
```

For PSoC™ Creator

In IAR, modify the linker script *.icf* file. The procedure for adding a section is simple, just add your execution section as a new line. You can place the section in an absolute address in memory, for example:

```
".My_Section" : place at address mem : <address> {<attribute>};
```

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

```
/* places "My_Section" in "readwrite" section within RAM */
".My_Section"            : place in RAM_region { readwrite section readwrite };
```

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 (PSoC™ 5LP only).

1. Define the array to occupy a section that we will call ".bitband":

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

2. Modify the linker script file to place the .bitband section in upper SRAM, starting at address 0x20000000. Table 13 shows how to do this for GCC (*.ld* file) or MDK (*.scat* file):

Table 13 Example modifications of linker script files

	For PSoC™ Creator

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
execution 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
{

}
```

IAR Example *.icf* file

```
/****** Initializations ******/
initialize by copy { readwrite };
do not initialize { section .noinit };
do not initialize { readwrite section .ramvectors };
do not initialize { section .bitband };

/******** Placements *********/
if (CY_APPL_LOADABLE)
{
".cybootloader" : place at start of ROM_region {block
LOADER};
}
"APPL" : place at start of APPL_region {block APPL};

"RAMVEC" : place at start of RAM_region { readwrite
section .ramvectors };
"readwrite" : place in RAM_region { readwrite };
"HSTACK" : place at end of RAM_region { block HSTACK};
".bitband" : place in RAM_region { readwrite section
.bitband };
keep { section .cybootloader,
       section .cyloadermeta,
       section .cyloadablemeta,
       section .cyconfigecc,
       section .cy_checksum_exclude,
       section .cycustnvl,
       section .cywolatch,
       section .cyeeprom,
       section .cyflashprotect,
       section .cymeta,
       section .bitband };
```

3. Build the project, and 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 Linker script file for IAR for details.

See Cortex®-M3 bit band (PSoC™ 5LP only) 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 and IAR, Figure 12) or the `DATA` execution region (MDK, Figure 13). 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 and IAR) 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 these sections are not initialized. Explicit initializations are ignored.
- Functions that are to be located in SRAM must be placed in the `.data` section (GCC and IAR, Figure 12) or the `DATA` execution region (MDK, Figure 13).
- It is more efficient to have all constant data, fixed data tables in arrays for example, located in flash; however, the default is to place them in SRAM. To force the 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 and ModusToolbox™ software bootloaders use the top one or two rows of flash to store information about bootloadable files. For more information, see the Bootloader Component datasheet and DFU middleware for ModusToolbox™.
- 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 the 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 (x86)\Cypress\PSoC Creator\4.3\PSoC Creator\import\gnu\arm\5.4.1\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*.

Complete documentation of .icf file usage can be found in your IAR installation folder, typically: *C:\Program Files (x86)\IAR Systems\Embedded Workbench 8.4\arm\doc\EWARM_DevelopmentGuide.ENU.pdf*

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 External memory interface (EMIF) Component datasheet for details.
- You cannot access the external memory until the EMIF API function `EMIF_Start()` is called. So, you cannot initialize EMIF variables in C startup; you must initialize them after the code reaches `main()` and `EMIF_Start()` is called.
- 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 the 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 Tips and tricks to optimize the PSoC™ 4 device code

10.1 Use simple and fast low-power mode entry functions

To put the CPU into sleep or Deep Sleep power modes, there are four functions:

- `Cy_SysPm_CpuEnterSleep()`
- `Cy_SysPm_CpuEnterDeepSleep()`
- `Cy_SysPm_CpuEnterSleepNoCallbacks()`
- `Cy_SysPm_CpuEnterDeepSleepNoCallbacks()`

The functions `Cy_SysPm_CpuEnterSleep()` and `Cy_SysPm_CpuEnterDeepSleep()` will put the CPU into sleep and Deep Sleep power modes respectively, with the automatic execution of registered callback functions.

While calling the `Cy_SysPm_CpuEnterSleep()` function, prior to entering sleep mode, before transition registered callbacks are executed to allow the peripherals to prepare for CPU sleep. Sleep mode is then entered for the CPU. This is a CPU-centric power mode. It means that the CPU has entered sleep mode and its main clock has been removed. It is identical to CPU active mode from a peripheral point of view. Any enabled interrupt can cause a wakeup from sleep mode. After wakeup from CPU sleep, after transition registered callbacks are executed to return the peripherals to active operation.

While calling the `Cy_SysPm_CpuEnterDeepSleep()` function, prior to entering Deep Sleep mode, before transition registered callbacks are executed to allow the peripherals to prepare for CPU Deep Sleep. Deep Sleep mode is then entered. Any enabled interrupt can cause a wakeup from Deep Sleep mode. After wakeup from the Deep Sleep mode, after transition registered callbacks are executed to return peripherals to active operation.

The functions `Cy_SysPm_CpuEnterSleepNoCallbacks()` and `Cy_SysPm_CpuEnterDeepSleepNoCallbacks()` will put the CPU into sleep and Deep Sleep power modes respectively, without calling the registered callback functions. The application is responsible for preparing a device for entering low-power mode and restoring it to the desired state on wakeup.

Therefore, employing the No Callbacks functions has the advantages of reducing the total power consumption of the device by instantly switching to low-power mode at awakening without executing registered callbacks and saving some flash memory.

Table 14 Flash memory consumption of low-power mode entry functions

Functions	Flash memory consumption on CY8CKIT-041S-MAX kit in bytes					
	GCC_ARM		Arm®		IAR	
	Debug	Release	Debug	Release	Debug	Release
`Cy_SysPm_CpuEnterSleep()`	332	292	324	260	392	416
`Cy_SysPm_CpuEnterDeepSleep()`	360	316	356	284	420	308
`Cy_SysPm_CpuEnterSleepNoCallbacks()`	20	28	24	24	24	24
`Cy_SysPm_CpuEnterDeepSleepNoCallbacks()`	44	52	40	40	40	40

`Cy_SysPm_CpuEnterSleepNoCallbacks()` allows you to save 312 bytes compared to `Cy_SysPm_CpuEnterSleep()in GCC_ARM` Debug mode, as shown in Table 14.

`Cy_SysPm_CpuEnterDeepSleepNoCallbacks()` allows you to save 316 bytes compared to `Cy_SysPm_CpuEnterDeepSleep()` in GCC_ARM Debug mode, as shown in Table 14.

Note: For more details about the functions, see the source file cy_syspm.c, which is part of the PDL library.

10.2 Reusing the resource configuration when multiple peripherals are used in an application

When using multiple peripherals in an application that all require the same resource configuration, the resource configuration can be reused rather than having to be created separately for each peripheral.

For example, when you use the Device Configurator to set the GPIO pins as LED output, it will create separate resource configurations for each LED output pin. In the most common case, the structure of the resource configuration must be the same. As a result, you can reuse the resource configuration for all user LED initializations.

By default, separate resource configurations are created for each LED output pin in the *cycfg_pins.c* file. The file is located in the following path: *<project_directory>/bsps/<target>/config/GeneratedSource*.

Table 15 Current and workaround flows for reusing the configuration

Current flow	`#define CYBSP_LED1_HSIOM HSIOM_SEL_GPIO`
	`#define CYBSP_LED2_HSIOM HSIOM_SEL_GPIO`
	
	`const cy_stc_gpio_pin_config_t CYBSP_LED1_config =`
	`{`
	`.outVal = 1,`
	`.driveMode = CY_GPIO_DM_STRONG_IN_OFF,`
	`.hsiom = CYBSP_LED1_HSIOM,`
	`.intEdge = CY_GPIO_INTR_DISABLE,`
	`.vtrip = CY_GPIO_VTRIP_CMOS,`
	`.slewRate = CY_GPIO_SLEW_FAST,`
	`};`
	
	`const cy_stc_gpio_pin_config_t CYBSP_LED2_config =`
	`{`
	`.outVal = 1,`
	`.driveMode = CY_GPIO_DM_STRONG_IN_OFF,`
	`.hsiom = CYBSP_LED2_HSIOM,`
	`.intEdge = CY_GPIO_INTR_DISABLE,`
	`.vtrip = CY_GPIO_VTRIP_CMOS,`
	`.slewRate = CY_GPIO_SLEW_FAST,`
	`};`
	
	`Cy_GPIO_Pin_Init(CYBSP_LED2_PORT, CYBSP_LED2_PIN,`
	`&CYBSP_LED2_config);`
	
	`Cy_GPIO_Pin_Init(CYBSP_LED1_PORT, CYBSP_LED1_PIN,`
	`&CYBSP_LED1_config);`

Workaround flow	`#define CYBSP_LED_HSIOM HSIOM_SEL_GPIO`
	
	`const cy_stc_gpio_pin_config_t CYBSP_LED_config =`
	`{`
	`.outVal = 1,`
	`.driveMode = CY_GPIO_DM_STRONG_IN_OFF,`
	`.hsiom = CYBSP_LED_HSIOM,`
	`.intEdge = CY_GPIO_INTR_DISABLE,`
	`.vtrip = CY_GPIO_VTRIP_CMOS,`
	`.slewRate = CY_GPIO_SLEW_FAST,`
	`};`
	
	`Cy_GPIO_Pin_Init(CYBSP_LED2_PORT, CYBSP_LED2_PIN, &CYBSP_LED_config);`
	
	`Cy_GPIO_Pin_Init(CYBSP_LED1_PORT, CYBSP_LED1_PIN, &CYBSP_LED_config);`

Table 16 Flash memory saving by using the workaround flow

Flash memory saving on the CY8CKIT-041S-MAX kit in bytes					
GCC_ARM		**Arm®**		**IAR**	
Debug	**Release**	**Debug**	**Release**	**Debug**	**Release**
24	24	24	24	24	24

You can use the workaround flow to save the number of bytes documented in the Table 16 for each peripheral initialization by reusing the existing resource configuration. There are multiple methods to apply this workaround. See the Important note for details.

10.3 Optimization on the clock, peripherals, and pins config reservations

In ModusToolbox™, during BSP initialization, the `cycfg_config_reservations()` function is called by default. But this function must be called only if we are using HAL in an application to reserve the clock, peripherals, and pins configurations.

The HAL-based clock, peripherals, and pin configuration reservation should only be disabled if HAL is not used in an application.

The `cycfg_config_reservations()` function is located in the following path: *<project_directory>/bsps/<target>/config/GeneratedSource/cycfg.c*.

Table 17 Current and workaround flows for optimization on the HAL-based config reservations

Current flow	`void init_cycfg_all(void)`
	`{`
	`cycfg_config_init();`
	`cycfg_config_reservations();`
	`}`

Workaround flow	`void init_cycfg_all(void)`
	`{`
	`cycfg_config_init();`
	`}`

Table 18 Flash memory saving by using the workaround flow

GCC_ARM		Arm®		IAR	
Debug	**Release**	**Debug**	**Release**	**Debug**	**Release**
32	24	24	28	28	20

You can use the workaround flow to save the number of bytes documented in the Table 18. There are multiple methods to apply this workaround. See the section Important note for details.

10.4 Optimization on Device Configurator

After developing your end-level application, you can disable the debug ports. Uncheck the debug check box in the system configuration to disable the debugging option, as follows:

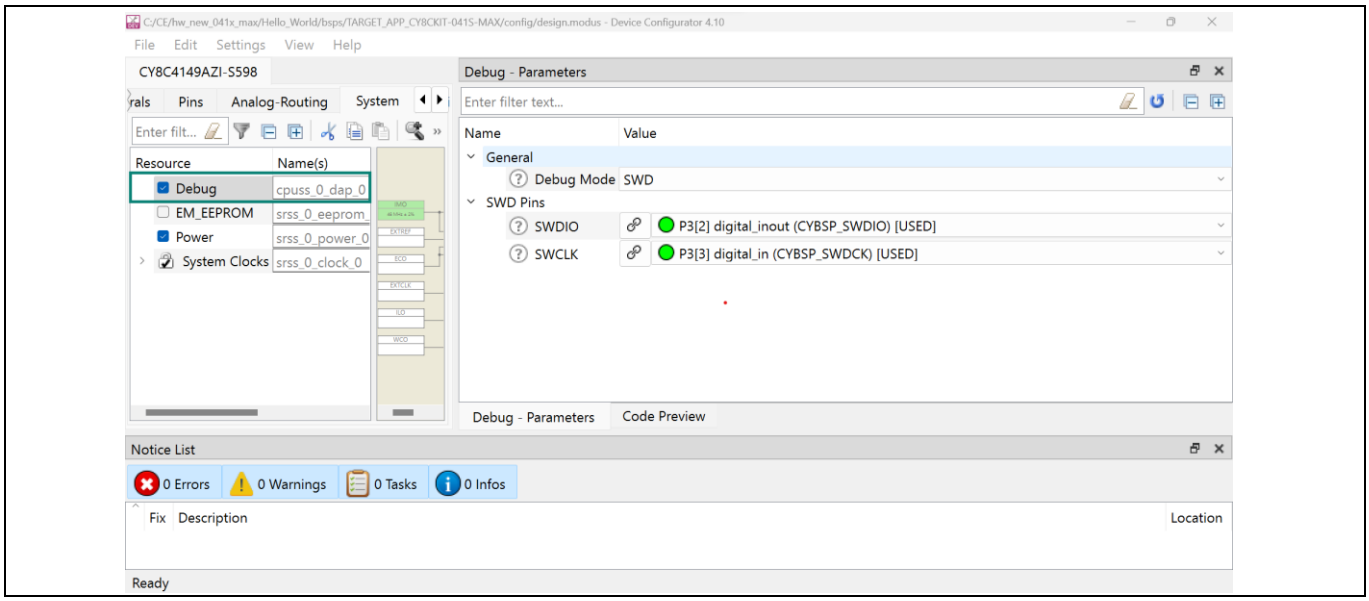

Figure 16 Debug enabled by default

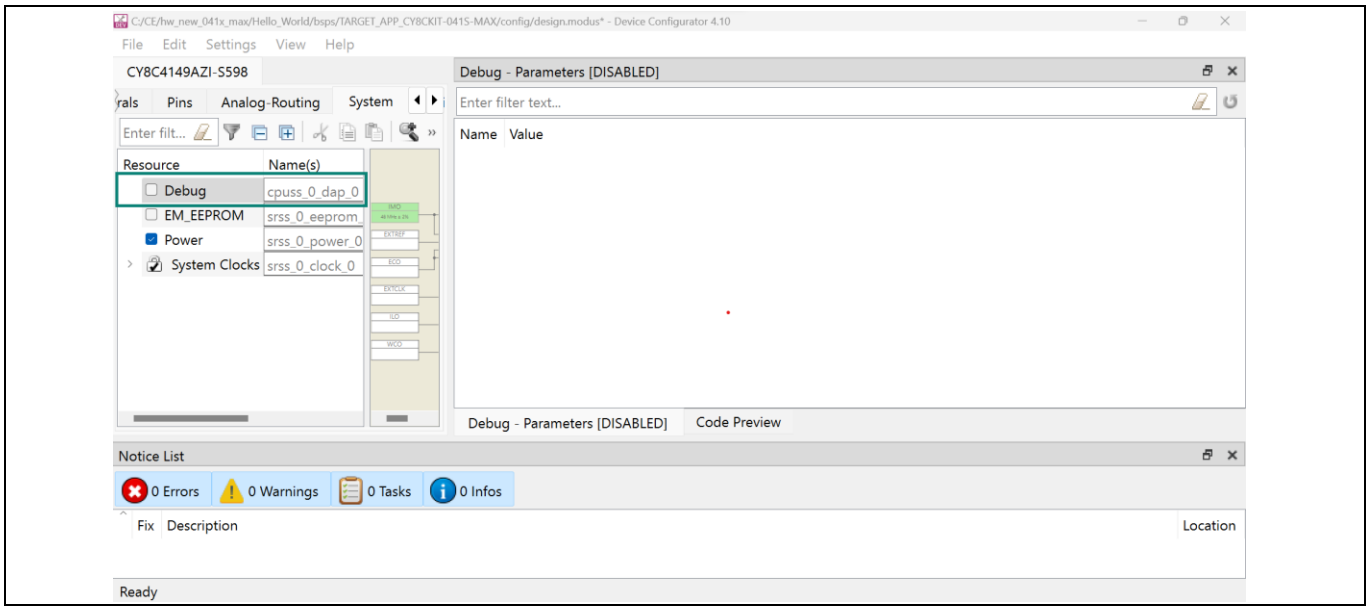

Figure 17 Debug disabled

Table 19 Flash memory saving by disabling the debugging option

Flash memory saving on the CY8CKIT-041S-MAX kit in bytes					
GCC_ARM		Arm®		IAR	
Debug	Release	Debug	Release	Debug	Release
88	80	84	80	84	76

By disabling the Debug option, you can save the number of bytes documented in the Table 19.

Disable the unused pins, peripherals, and DMA in an application if they are enabled by default.

For example: I2C, user LED 2, and CAPSENSE™ related pins are enabled by default on the CE230635 - PSoC™ 4: Hello world code example but those are not used in the application. So, you can disable those configurations.

Uncheck the MSC 0 and MSC 1 check box to disable the CAPSENSE™ personalities as shown in Figure 19.

Uncheck each unused pin's check box in the pins configuration to disable those unused pins, as shown in Figure 21.

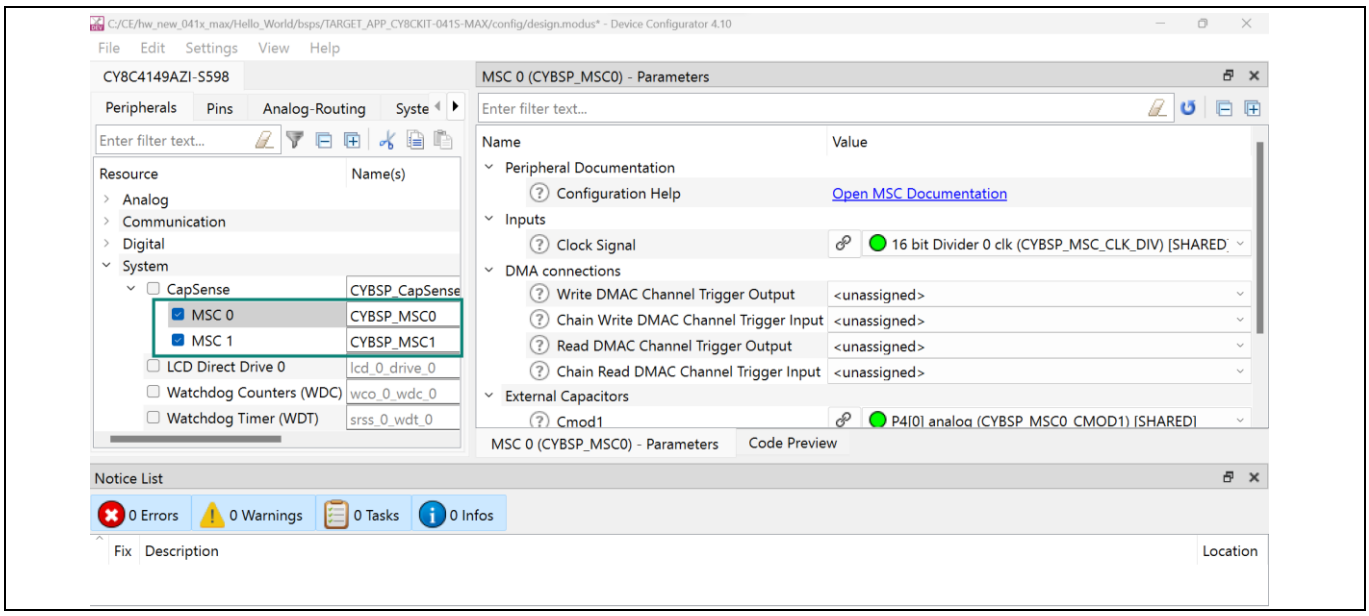

Figure 18 Unused personalities are enabled by default

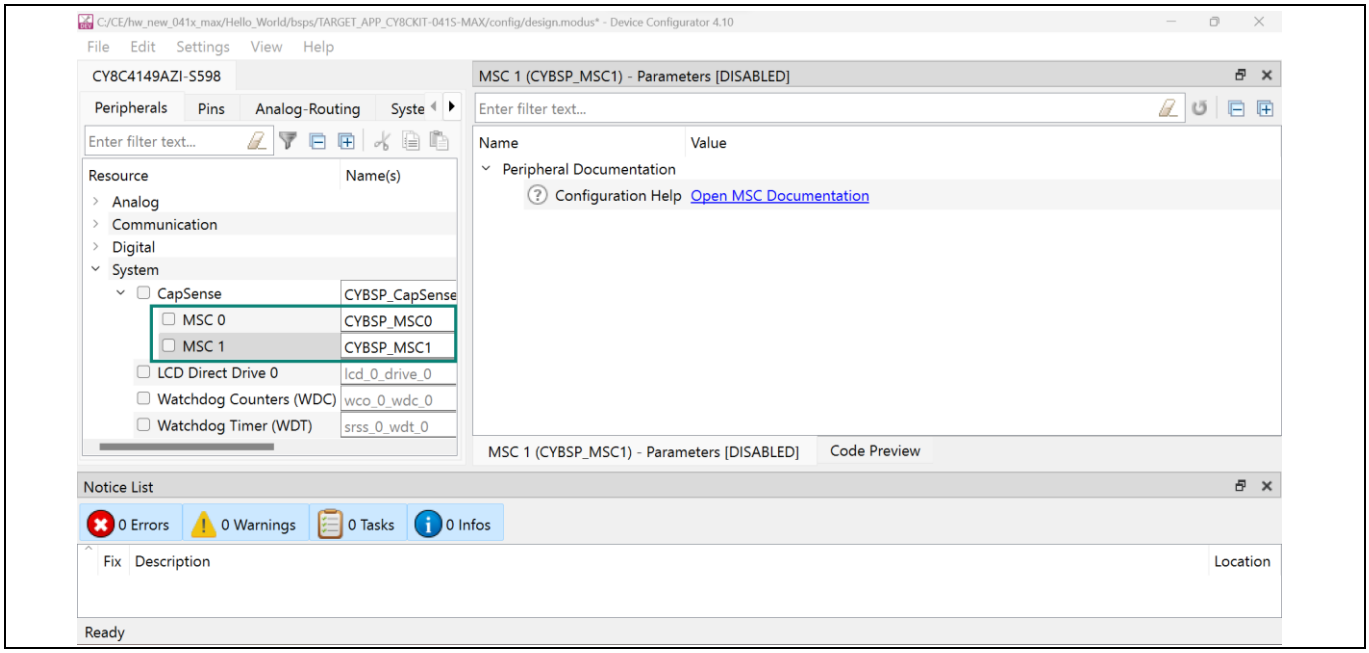

Figure 19 Unused personalities are disabled

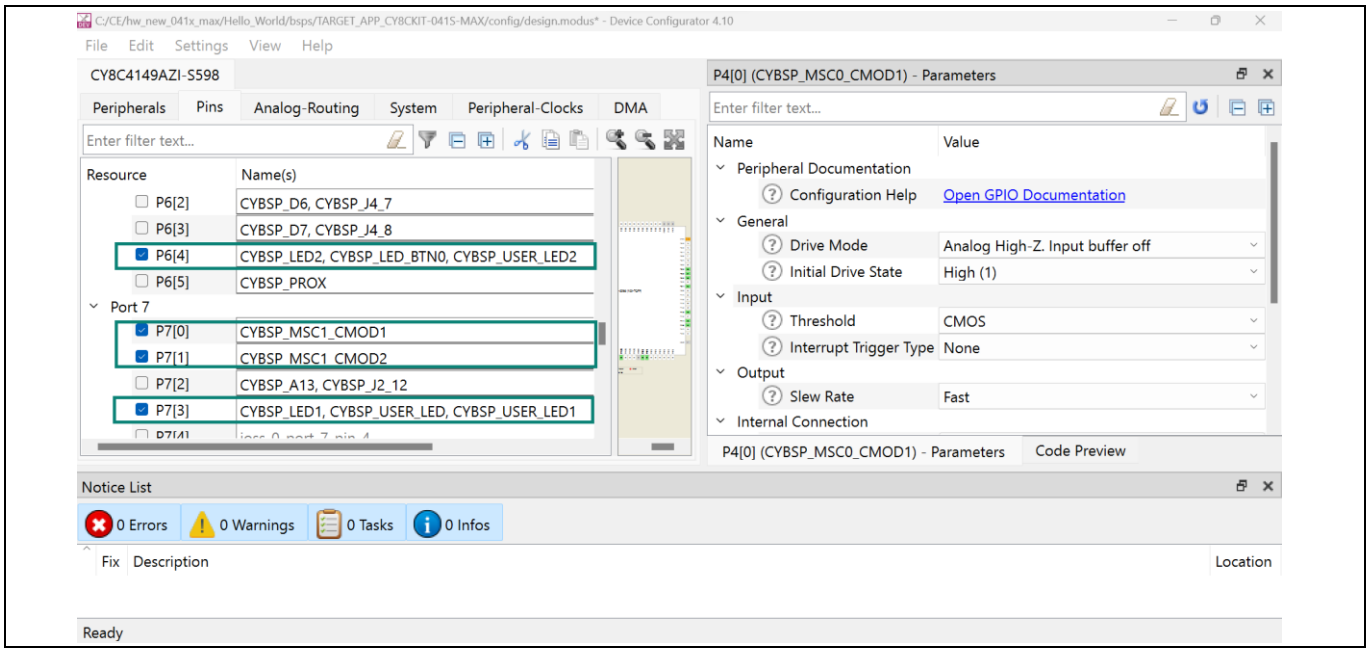

Figure 20 Unused pins are enabled by default

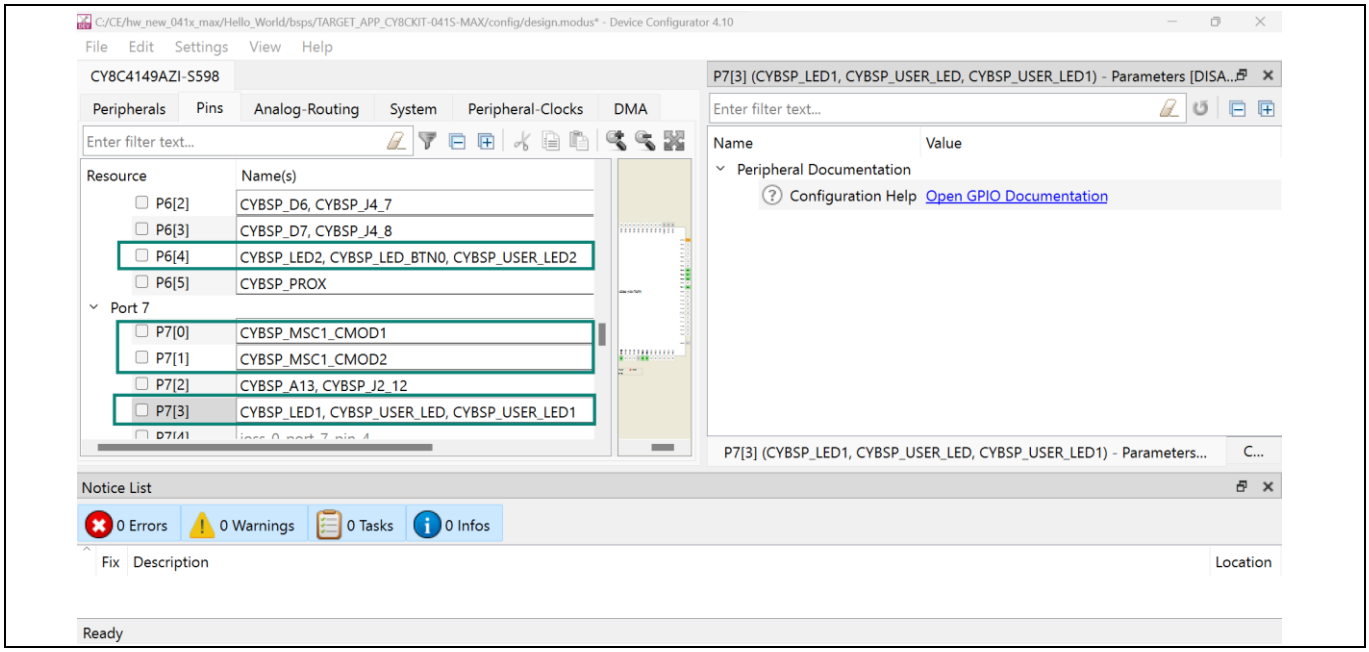

Figure 21 Unused pins are disabled

By disabling unused pins, peripherals, and DMA in an application, a few flash bytes can be saved.

10.5 Optimization on the CAPSENSE™ middleware library

The watchdog timer value is using `uint64_t` in the CAPSENSE™ middleware. However, you can rearrange the watchdog timer value calculation operations to perform 'periClkHz' by 'CY_CAPSENSE_CONVERSION_MEGA' division first to avoid using `uint64_t`.

The calculation operations rearrangement can replace `uint64_t` with `uint32_t` in the *cy_capsense_csd_v2.c* file.

Table 20 Current and workaround flows for optimization on CAPSENSE™ middleware *cy_capsense_csd_v2.c* file

Current flow

```
uint64_t isBusyWatchdogTimeUs;

isBusyWatchdogTimeUs = (uint64_t)CY_CAPSENSE_CONFIGURED_FREQ_NUM
* ptrWdCxt->maxRawCount;
isBusyWatchdogTimeUs *= (uint64_t)modClkDivider *
CY_CAPSENSE_CONVERSION_MEGA;
isBusyWatchdogTimeUs /= context->ptrCommonConfig->periClkHz;

#if (CY_CAPSENSE_ENABLE == CY_CAPSENSE_MULTI_FREQUENCY_SCAN_EN)
     isBusyWatchdogTimeUs  =
(uint64_t)CY_CAPSENSE_MAX_SUPPORTED_FREQ_NUM * ptrWdCfg->numSns *
ptrWdCfg->ptrWdContext->maxRawCount;
#else
     isBusyWatchdogTimeUs  = (uint64_t)ptrWdCfg->numSns *
ptrWdCfg->ptrWdContext->maxRawCount;
#endif

isBusyWatchdogTimeUs *= (uint64_t)modClkDivider *
CY_CAPSENSE_CONVERSION_MEGA;
isBusyWatchdogTimeUs /= context->ptrCommonConfig->periClkHz;
```

Workaround flow	`uint32_t isBusyWatchdogTimeUs;`
	
	`isBusyWatchdogTimeUs = (uint32_t)CY_CAPSENSE_CONFIGURED_FREQ_NUM * ptrWdCxt->maxRawCount;`
	`isBusyWatchdogTimeUs *= (uint32_t)modClkDivider;`
	`isBusyWatchdogTimeUs /= context->ptrCommonConfig->periClkHz / CY_CAPSENSE_CONVERSION_MEGA;`
	
	`#if (CY_CAPSENSE_ENABLE == CY_CAPSENSE_MULTI_FREQUENCY_SCAN_EN)`
	`  isBusyWatchdogTimeUs = (uint32_t)CY_CAPSENSE_MAX_SUPPORTED_FREQ_NUM * ptrWdCfg->numSns * ptrWdCfg->ptrWdContext->maxRawCount;`
	`#else`
	`  isBusyWatchdogTimeUs = (uint32_t)ptrWdCfg->numSns * ptrWdCfg->ptrWdContext->maxRawCount;`
	`#endif`
	
	`isBusyWatchdogTimeUs *= (uint32_t)modClkDivider;`
	`isBusyWatchdogTimeUs /= context->ptrCommonConfig->periClkHz / CY_CAPSENSE_CONVERSION_MEGA;`

Table 21 Flash memory saving by using the workaround flow

Flash memory saving on the CY8CKIT-041S-MAX kit in bytes					
GCC_ARM		**Arm®**		**IAR**	
Debug	**Release**	**Debug**	**Release**	**Debug**	**Release**
268	52	40	48	60	48

You can use the workaround flow to save the number of bytes documented in Table 21.

The calculation operations rearrangement can replace `uint64_t` with `uint32_t` in the *cy_capsense_csx_v2.c* file.

Table 22 Current and workaround flows for optimization on CAPSENSE™ middleware *cy_capsense_csx_v2.c* file

Current flow	`uint64_t isBusyWatchdogTimeUs;`
	
	`#if (CY_CAPSENSE_ENABLE == CY_CAPSENSE_MULTI_FREQUENCY_SCAN_EN)`
	`  isBusyWatchdogTimeUs = (uint64_t)totalSns * CY_CAPSENSE_MAX_SUPPORTED_FREQ_NUM * ptrWdCfg->ptrWdContext->resolution;`
	`#else`
	`  isBusyWatchdogTimeUs = (uint64_t)totalSns * ptrWdCfg->ptrWdContext->resolution;`
	`#endif`
	
	`isBusyWatchdogTimeUs *= (uint64_t)snsClkDivider * modClkDivider * CY_CAPSENSE_CONVERSION_MEGA;`
	`isBusyWatchdogTimeUs /= context->ptrCommonConfig->periClkHz;`

Workaround flow	`uint32_t isBusyWatchdogTimeUs;`
	
	`#if (CY_CAPSENSE_ENABLE == CY_CAPSENSE_MULTI_FREQUENCY_SCAN_EN)`
	`isBusyWatchdogTimeUs = (uint32_t)totalSns * CY_CAPSENSE_MAX_SUPPORTED_FREQ_NUM * ptrWdCfg->ptrWdContext->resolution;`
	`#else`
	`isBusyWatchdogTimeUs = (uint32_t)totalSns * ptrWdCfg->ptrWdContext->resolution;`
	`#endif`
	
	`isBusyWatchdogTimeUs *= (uint32_t)snsClkDivider * modClkDivider;`
	`isBusyWatchdogTimeUs /= context->ptrCommonConfig->periClkHz / CY_CAPSENSE_CONVERSION_MEGA;`

Table 23 Flash memory saving by using the workaround flow

Flash memory saving on the CY8CKIT-041S-MAX kit in bytes					
GCC_ARM		**Arm®**		**IAR**	
Debug	**Release**	**Debug**	**Release**	**Debug**	**Release**
112	32	32	28	52	36

You can use the workaround flow to save the number of bytes documented in the Table 23.

The calculation operations rearrangement can replace uint64_t with uint32_t in the *cy_capsense_selftest_v2.c* file.

Table 24 Current and workaround flows for optimization on CAPSENSE™ middleware *cy_capsense_selftest_v2.c* file

Current flow	`uint64_t isBusyWatchdogTimeUs;`
	
	`isBusyWatchdogTimeUs = (uint64_t)ptrScanConfig->convNum;`
	`isBusyWatchdogTimeUs *= (uint64_t)snsClkDivider * modClkDivider * CY_CAPSENSE_CONVERSION_MEGA;`
	`isBusyWatchdogTimeUs /= context->ptrCommonConfig->periClkHz;`
	
	`isBusyWatchdogTimeUs = (uint64_t)((uint64_t)0x01 << context->ptrBistContext->eltdCapResolution);`
	`isBusyWatchdogTimeUs *= (uint64_t)modClkDivider * CY_CAPSENSE_CONVERSION_MEGA;`
	`isBusyWatchdogTimeUs /= context->ptrCommonConfig->periClkHz;`

Workaround flow	`Uint32_t isBusyWatchdogTimeUs;`
	
	`isBusyWatchdogTimeUs = (uint32_t)ptrScanConfig->convNum;`
	`isBusyWatchdogTimeUs *= (uint32_t)snsClkDivider * modClkDivider;`
	`isBusyWatchdogTimeUs /= context->ptrCommonConfig->periClkHz / CY_CAPSENSE_CONVERSION_MEGA;`
	
	`isBusyWatchdogTimeUs = (uint32_t)((uint32_t)0x01 << context->ptrBistContext->eltdCapResolution);`
	`isBusyWatchdogTimeUs *= (uint32_t)modClkDivider;`
	`isBusyWatchdogTimeUs /= context->ptrCommonConfig->periClkHz / CY_CAPSENSE_CONVERSION_MEGA;`

Table 25 Flash memory saving by using the workaround flow

Flash memory saving on the CY8CKIT-041S-MAX kit in bytes					
GCC_ARM		**Arm®**		**IAR**	
Debug	**Release**	**Debug**	**Release**	**Debug**	**Release**
128	52	52	72	64	60

You can use the workaround flow to save the number of bytes documented in the Table 25.

10.6 Important note

The flash memory consumption and saving numbers will vary with respect to device family, and it depends up on the compiler optimization level too.

The approaches described in this document uses ModusToolbox™ software 3.1 default compiler optimization level for all three compilers in Debug and Release mode to measure the flash memory consumptions, which is mentioned below.

- In GCC_ARM compiler version v11.3.1, the compiler optimization level is minimal (-Og) for debug mode and the compiler optimization level is size (-Os) for release mode.
- In Arm® compiler version v6.16, the compiler optimization level is minimal (-O1) for debug mode and the compiler optimization level is smaller code size (-Oz) for release mode.
- In IAR compiler version v9.30.1, the compiler optimization level is minimal (-O1) for debug mode and the compiler optimization level is size (-Ohs) for release mode.

If you want to make any changes in the generated files (e.g., *cycfg_pins.c*), you should either copy the configuration to the application-level files (such as *main.c*) and use it instead of generated ones or could do these modifications in the generated files at the final phase of application development and avoid loss of changes done files due to re-generation of the files occurred while saving the *design.modus* file. Note that in case the workaround was applied in the generated files and then *design.modus* file is changed and saved, these workaround should be applied again. See ModusToolbox™ tools package user guide for details.

11 Cortex®-M3 bit band (PSoC™ 5LP only)

As indicated in Address map, Figure 2 and Figure 4, 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 MB 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, to set bit 5 in address 0x20000001, write a ‘1’ to the 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, Cortex®-M3 supports bit band for the peripheral region, in which all PSoC™ 5LP registers are located. However, peripheral region bit band is not supported in PSoC™ 5LP. Writing to the peripheral region’s bit band alias region (0x42000000–0x43FFFFFF) may give unpredictable results in 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 26 shows. When bit band is used, the read-modify-write cycle is done internally by the CPU, thereby saving one instruction.

Table 26 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]`

Cortex®-M3 bit band (PSoC™ 5LP only)

C code	Assembler code

`/* 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);
```

12 DMA addresses

This section assumes that you know how to use the direct memory access (DMA) controller in PSoC™ 5LP or PSoC™ 4. 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) (PSoC™ 5LP), USB, UART, and SPI. There are 24 independent DMA channels on PSoC™ 5LP, and up to 32 DMA channels on PSoC™ 4. 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 shown in Figure 22.

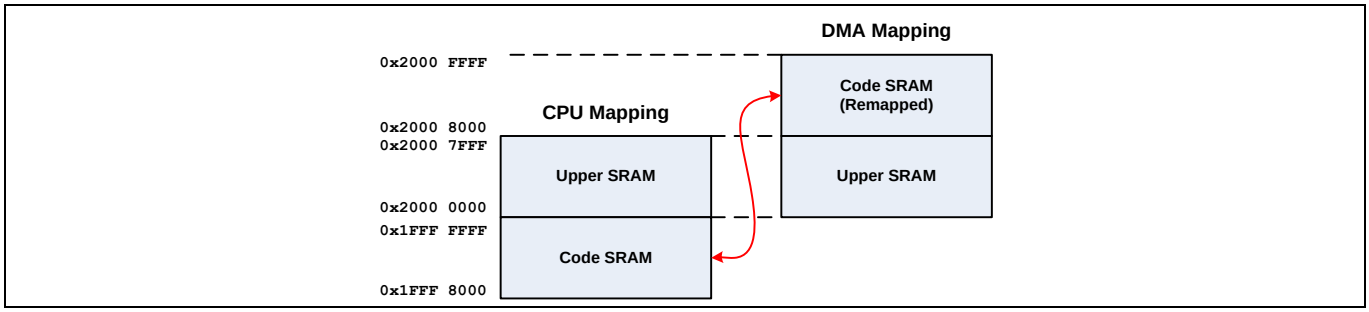

Figure 22 DMA remapping of code SRAM

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 PSoC™ Creator DMA Component API functions that are used to set up a DMA channel handles the remapping. 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.

13 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, MDK, and IAR compilers supported by PSoC™ Creator and ModusToolbox™ software 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: Compiler output details for examples of how to write efficient assembler code.
- Be careful when using standard compiler libraries because 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 PSoC™ 5LP Cortex®-M3; see Cortex®-M3 bit band (PSoC™ 5LP only) and Example.

13.1 Use all of the resources in your PSoC™ device

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; see 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.

Summary

PSoC™ Creator offers a large number of Components to implement various functions in these peripherals. Similarly, ModusToolbox™ provides many resources that allow you to implement functions in the 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.

Infineon 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 References.

14 Appendix: Compiler output details

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

- Compiler output details for GCC Compiler for Cortex®-M3 CPU
- Compiler output details for GCC compiler for Cortex®-M3 CPU
- Compiler output details for GCC compiler for Cortex®-M0/M0+ CPU
- Compiler output details for GCC compiler for Cortex®-M0/M0+ CPU
- Compiler output details for MDK compiler for Cortex®-M3 CPU
- Compiler output details for MDK compiler for Cortex®-M0/M0+ CPU
- Compiler output details for IAR compiler for Cortex®-M3 CPU
- Compiler output details for IAR compiler for Cortex®-M3 CPU
- Compiler output details for IAR compiler for Cortex®-M0/M0+ CPUe
- Compiler output details for IAR compiler for Cortex®-M0/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.

14.1 Assembler examples, GCC for Cortex®-M3

14.1.1 GCC for Cortex®-M3, none, size, and speed optimization

Table 27 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 and result for details on register usage and stack usage in compiler functions.

Table 27 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`

`// For loop:` `void ForLoop(uint8 i)`	`; function prolog` `; i is saved on the stack`	`; function prolog` `push {r4, lr}`	`; function prolog` `push {r3, lr}`

Appendix: Compiler output details

C code	GCC, Cortex®-M3, no optimization	GCC, Cortex®-M3, size optimization	GCC, Cortex®-M3, speed optimization

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

// 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

Appendix: Compiler output details

C code	GCC, Cortex®-M3, no optimization	GCC, Cortex®-M3, size optimization	GCC, Cortex®-M3, 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)`
	`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`

`// 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:`
		`; no prolog`
		`; check value of i`
		`cmp  r0, #1`
		
		`; "ite" stands for if-then-`
		`; else instruction`
		`; "ne" condition checks`
		`; if the previous`
			`; same as for size optimization`

Appendix: Compiler output details

C code	GCC, Cortex®-M3, no optimization	GCC, Cortex®-M3, size optimization	GCC, Cortex®-M3, speed optimization

	mov r3, #100
	
	.L18:
	mov r0, r3
	bl LCD_PrintInt8
	; epilog not shown
		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	

// 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
		; 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

Appendix: Compiler output details

C code	GCC, Cortex®-M3, no optimization	GCC, Cortex®-M3, size optimization	GCC, Cortex®-M3, speed optimization
	`mov r0, r3 ; return value` `; epilog not shown`		

`// 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`

`// 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;`
	`; 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]`
		`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`

Appendix: Compiler output details

C code	GCC, Cortex®-M3, no optimization	GCC, Cortex®-M3, size optimization	GCC, Cortex®-M3, speed optimization

`  myfoo_p.memberd = 20;`
`}`
	`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]`		

`// 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`

14.1.2 GCC for Cortex® -M3, Debug, Minimal, and High Optimization

Table 28 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 and result for details on register usage and stack usage in compiler functions.

Table 28 Compiler output details for GCC compiler for Cortex®-M3 CPU

C code	GCC, Cortex®-M3 debug optimization	GCC, Cortex®-M3, minimal optimization	GCC, Cortex®-M3, high optimization

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

`// Calling a function with`	`; R0 = first argument` `; conditional flags`	`; same as for Debug optimization`	`; same as for Debug optimization`

Appendix: Compiler output details

C code	GCC, Cortex®-M3 debug optimization	GCC, Cortex®-M3, minimal optimization	GCC, Cortex®-M3, high optimization

`// one argument`
`LCD_PrintInt8(128);`
	`are NOT`
	`; updated by mov`
	`r0, #128`
	`bl LCD_PrintInt8`		

`// 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 Debug optimization`	`; same as for Debug optimization`

`// For loop:`
`void ForLoop(uint8 i)`
`{`
`for(i = 0; i < 10; i++)`
`{`
`LCD_PrintInt8(i);`
`}`
`}`
	`; function prolog`
	`push {r4, lr}`
	
	`; i = 0`
	`mov r4, #0`
	`b .L2`
	
	`; do the function call with`
	`; i as the argument in R0`
	`.L3:`
	`mov r0, r4`
	`bl LCD_PrintInt8`
	`adds r4, r4, #1`
	`uxtb r4, r4 ; sign extend`
	
	`; check i 10, by comparing`
	`; it with 9`
	`.L2:`
	`cmp r4, #9`
	`bls .L3`
	
	`pop {r4, pc};`
	`return`
		`; function prolog`
		`push {r4, lr}`
		
		`; R4 = i`
		`movs r4, #0`
		
		`.L2:`
		`; do the function call with`
		`; i as the argument in R0`
		`uxtb r0, r4 ; sign extend`
		`bl LCD_PrintInt8`
		`adds r4, r4, #1 ; i++`
		
		`; check i not equal to 10`
		`cmp r4, #10`
		`bne .L2`
		
		`; function epilog`
		`pop {r4, pc} ; return`
			`; function prolog`
			`push {r4, lr}`
			
			`; R4 = i`
			`movs r4, #0`
			
			`.L2:`
			`; do the function call with`
			`; i as the argument in R0`
			`uxtb r0, r4 ; sign extend`
			`adds r4, r4, #1 ; i++`
			`bl LCD_PrintInt8`
			
			`; check i not equal to 10`
			`cmp r4, #10`
			`bne .L2`
			
			`; function epilog`
			`pop {r4, pc} ; return`

`// 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 r4, #0`
	`b .L6`
	
	`.L7:`
	`; LCD_PrintInt8(i)`
	`mov r0, r4`
	`bl LCD_PrintInt8`
	`adds r4, r4, #1 ; i++`
	`bl LCD_PrintInt8`
	
	`.L6`
	`; i++`
	`cmp r4, #9`
	`bls .L7`
	
	`; epilog not shown`
		`; prolog not shown`
		`; i = 0`
		`movs r4, #0`
		
		`.L6:`
		`uxtb r0, r4`
		`bl LCD_PrintInt8`
		`adds r4, r4, #1 ; i++`
		
		`; check i not equal to 10`
		`cmp r4, #10`
		`bne .L6`
		
		`; epilog not shown`
			`; prolog not shown`
			
			`movs r4, #0`
			`.L7:`
			`uxtb r0, r4`
			`adds r4, r4, #1`
			`bl LCD_PrintInt8`
			
			`cmp r4, #10`
			`bne .L7`
			
			`; epilog not shown`

`// Conditional statement`
`void Conditional(uint8 i, uint8 j)`
`{`
`if(j == 1)`
	`; prolog not shown`
	`; if(j == 1)`
	`cmp r1, #1`
	`bne .L10`
	
	`; LCD_PrintInt8(i)`
		`; no prolog`
		`; if(j == 1)`
		`cmp r1, #1`
		`itt ne`
		`addne r0, r0, #1`
			`; no prolog`
			`; if(j == 1)`
			`cmp r1, #1`
			`itt ne`
			`addne r0, r0, #1`

Appendix: Compiler output details

C code	GCC, Cortex®-M3 debug optimization	GCC, Cortex®-M3, minimal optimization	GCC, Cortex®-M3, high optimization

{
LCD_PrintInt8(i);
}
else
{
LCD_PrintInt8(i + 1);
}
}
	bl LCD_PrintInt8
	pop {r3, pc}.L10:
	adds r0, r0, #1
	uxtb r0, r0
	; LCD_PrintInt8(i)
	bl LCD_PrintInt8
	pop {r3, pc}
	; epilog not shown
		uxtbne r0, r0
		; LCD_PrintInt8(i)
		bl LCD_PrintInt8
			uxtbne r0, r0
			; LCD_PrintInt8(i)
			b LCD_PrintInt8

// 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)
	cbz r0, .L15
	cmp r0, #1
	beq .L16
	b .L18
	
	.L15: ; case 0
	mov r0, #1
	bl LCD_PrintInt8
	b .L7 ; break
	
	.L16: ; case 1
	movs r0, #2
	bl LCD_PrintInt8
	; break
	
	.L18: ; default
	movs r0, #0
	bl LCD_PrintInt8
	pop {r3, pc};
	break
	
	; epilog not shown
		; no prolog
		; switch(j)
		cbz r0, .L15
		cmp r0, #1
		beq .L16
		
		; default
		b .L18
		
		.L15: ; case 0
		movs r0, #1
		bl LCD_PrintInt8
		
		.L16: ; case 1
		movs r0, #2
		bl LCD_PrintInt8
		
		.L18
		movs r0, #0
		bl LCD_PrintInt8
			; no prolog
			; switch(j)
			cbz r0, .L15
			cmp r0, #1
			bne .L19
			
			; case 1
			movs r0, #2
			b LCD_PrintInt8
			
			; default
			.L19:
			movs r0, #0
			b LCD_PrintInt8
			
			; case 0
			.L15:
			movs r0, #1 ;
			b LCD_PrintInt8

// Ternary operator
void Ternary(uint8 i)
{
LCD_PrintInt8(
(i == 1) ? 80 : 100);
}
	; prolog not shown
	; check value of i
	cmp r0, #1
	bne .L22
	movs r0, #80
	b .L21
	
	.L22:
	movs r0, #100
	
	.L21:
	bl LCD_PrintInt8
	; epilog not shown
		; no prolog
		; check value of i
		cmp r0, #1
		
		; “ite” stands for if-then-
		; else instruction
		; “eq” condition checks
		; if the previous compare
		; instruction has set the
		; “equal to” flag
		ite eq
		
		; mov if the result of the
		; previous “ite” instruction is
		; “equal”
		moveq r0, #80
		
		; mov if the result of the
		; previous “ite” instruction is
			; same as for minimal optimization

Appendix: Compiler output details

C code	GCC, Cortex®-M3 debug optimization	GCC, Cortex®-M3, minimal optimization	GCC, Cortex®-M3, high optimization

		`; "not equal"`
		`movne r0, #100`
		
		`; no epilog`
		`bl LCD_PrintInt8`	

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

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

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

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

`// Modulo operator`
`int DoMod(int x, int y)`
`{`
`  return x % y;`
`}`
	`; 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 Debug optimization`	`; same as for Debug optimization`

`// Pointer`
`void Pointer(uint8 x, uint8 *ptr)`
`{`
`  *ptr = *ptr + x;`
`  ptr++;`
`  LCD_PrintInt8(*ptr);`
`}`
	`; *ptr = *ptr + x`
	`ldrb r3, [r1]; R1 = ptr`
	`add r0, r0, r3; R0 = x`
	`; ptr++`
	`; LCD_PrintInt8(*ptr)`
	`ldrb r0, [r1, #1]`
	`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]`
		`b LCD_PrintInt8`
			`; same as for minimal optimization`

`// Function pointer`
`void FuncPtr(uint8 x, void *fptr(uint8))`
`{`
`  (*fptr)(x);`
`}`
	`; (*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 debug optimization`	`; same as for debug optimization`

Appendix: Compiler output details

C code	GCC, Cortex®-M3 debug optimization	GCC, Cortex®-M3, minimal optimization	GCC, Cortex®-M3, high 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;`
`}`
	`ldr  r3, .L34`
	`movs  r2, #5`
	`strb  r2, [r3];`
	`memberb = 10`
	`movs  r2, #10`
	`strb  r2, [r3, #1]`
	`; memberc = 15`
	`movs  r2, #0`
	`movs  r1, #15`
	`strb  r1, [r3, #2]`
	`strb  r2, [r3, #3]`
	`strb  r2, [r3, #4]`
	`strb  r2, [r3, #5]`
	`; memberd = 20`
	`movs  r1, #20`
	`strb  r1, [r3, #6]`
	`strb  r2, [r3, #7]`
	`bx  lr`
	
	`.L34:`
	`.word  myfoo_p`
		`; same as for Debug optimization`	`ldr  r3, .L32`
			`movs  r5, #5`
			`; memberb = 10`
			`movs  r4, #10`
			`strb  r5, [r3]`
			`; memberc = 15`
			`movs  r2, #0`
			`movs  r0, #15`
			`strb  r4, [r3, #1]`
			`; memberd = 20`
			`movs  r1, #20`
			`strb  r0, [r3, #2]`
			`strb  r1, [r3, #6]`
			`strb  r2, [r3, #3]`
			`strb  r2, [r3, #4]`
			`strb  r2, [r3, #5]`
			
			`strb  r2, [r3, #7]`
			`bx  lr`
			
			`.L32:`
			`.word  myfoo_p`

`// 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, .L37`
	`movs  r2, #5`
	`strb  r2, [r3]`
	`; memberb = 10`
	`movs  r2, #10`
	`strb  r2, [r3, #1]`
	`; memberc = 15`
	`movs  r2, #15`
	`str  r2, [r3, #4];`
	`memberd = 20`
	`movs  r2, #20`
	`strh  r2, [r3, #8]`
	`bx  lr`
	
	`.L37:`
	`.word  myfoo`
		`; same as for Debug optimization`	`; membera = 5`
			`ldr  r3, .L36`
			`push  {r4}`
			`movs  r4, #5`
			`; memberb = 10`
			`movs  r0, #10`
			`strb  r4, [r3]`
			`; memberc = 15`
			`movs  r1, #15`
			`strb  r0, [r3, #1];`
			`memberd = 20`
			`movs  r2, #20`
			`str  r1, [r3, #4]`
			`strh  r2, [r3, #8]`
			`pop  {r4}`
			`bx  lr`
			
			`.L36:`
			`.word  myfoo`

14.2 Assembler examples, GCC for Cortex®-M0/M0+

14.2.1 GCC for Cortex®-M0/M0+, none, size, and speed optimizations

Table 29 shows, for the GCC compiler for the Cortex-M0/M0+, examples of compiler output for different optimization options. The examples were extracted from the *.lst* files generated by the compiler.

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

Table 29 Compiler output details for GCC compiler for Cortex®-M0/M0+ CPU

C code	GCC, Cortex®-M0/M0+, no optimization	GCC, Cortex®-M0/M0+, size optimization	GCC, Cortex®-M0/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`
	`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`

`// 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`
		`; 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`

Appendix: Compiler output details

C code	GCC, Cortex®-M0/M0+, no optimization	GCC, Cortex®-M0/M0+, size optimization	GCC, Cortex®-M0/M0+, speed optimization

	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}		

// 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

// 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]
		; 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

Appendix: Compiler output details

C code	GCC, Cortex®-M0/M0+, no optimization	GCC, Cortex®-M0/M0+, size optimization	GCC, Cortex®-M0/M0+, speed optimization

	add r3, r3, #1
	uxtb r3, r3
	mov r0, r3
	bl LCD_PrintInt8
	
	.L7:
	; epilog not shown		

// 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

// 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

Appendix: Compiler output details

C code	GCC, Cortex®-M0/M0+, no optimization	GCC, Cortex®-M0/M0+, size optimization	GCC, Cortex®-M0/M0+, speed 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`

`// 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`
		`; (*fptr)(x)`
		`; in a blx instruction, the LS`
		`; bit of the register must be`
		`; 1 to keep the CPU in Thumb`
		`; mode, or an`
			`; same as for size optimization`

Appendix: Compiler output details

C code	GCC, Cortex®-M0/M0+, no optimization	GCC, Cortex®-M0/M0+, size optimization	GCC, Cortex®-M0/M0+, speed optimization
	register ; must be 1 to keep the CPU ; in Thumb mode, or an ; exception occurs blx r3	exception occurs blx r1	

// 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

// unpacked structures
struct FOO
{
uint8 membera;
uint8 memberb;
uint32 memberc;
uint16 memberd;
};
 	
extern struct FOO
myfoo;
 	
void
PackedStruct(void)
	; 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]
		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
			; same as for size optimization

Appendix: Compiler output details

C code	GCC, Cortex®-M0/M0+, no optimization	GCC, Cortex®-M0/M0+, size optimization	GCC, Cortex®-M0/M0+, speed optimization

{
  myfoo.membera = 5;
  myfoo.memberb = 10;
  myfoo.memberc = 15;
  myfoo.memberd = 20;
}
	mov r2, #20
	strh r2, [r3, #8]
	
	.L35:
	.word myfoo
		.L33:
		.word myfoo	

14.2.2 GCC for Cortex®-M0/M0+, debug, minimal, and high optimization

Table 30 shows, for the GCC compiler for the Cortex®-M0/M0+, examples of compiler output for different optimization options. The examples were extracted from the *.lst* files generated by the compiler.

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

Table 30 Compiler output details for GCC compiler for Cortex®-M0/M0+ CPU

C code	GCC, Cortex®-M0/M0+, debug optimization	GCC, Cortex®-M0/M0+, minimal optimization	GCC, Cortex®-M0/M0+, high optimization

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

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

// For loop:
void ForLoop(uint8 i)
{
  for(i = 0; i < 10; i++)
  {
  LCD_PrintInt8(i);
  }
}
	; function prolog
	push {r4, lr}
	
	; i = 0
	movs r4, #0
	b .L2
	
	; do the function call with
	; i as the argument in R0
	.L3:
	movs r0, r4
	bl LCD_PrintInt8
	adds r4, r4, #1 ; i++
	uxtb r4, r4
	
	; check i 10, by comparing
	; it with 9
	.L2:
	cmp r4, #9
	bls .L3
	
	; function epilog
	pop {r4, pc}
		; function prolog
		push {r4, lr}
		
		; R4 = i
		movs r4, #0
		
		.L2:
		; do the function call with
		; i as the argument in R0
		movs r0, r4 ; sign extend
		bl LCD_PrintInt8
		adds r4, r4, #1 ; i++
		uxtb r4, r4
		
		; check i not equal to 10
		cmp r4, #10
		bne .L2
		
		pop {r4, pc} ; return
			; function prolog
			push {r4, lr}
			
			; R4 = i
			movs r4, #0
			
			.L2:
			; do the function call with
			; i as the argument in R0
			movs 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
			
			pop {r4, pc} ; return

Appendix: Compiler output details

C code	GCC, Cortex®-M0/M0+, debug optimization	GCC, Cortex®-M0/M0+, minimal optimization	GCC, Cortex®-M0/M0+, high 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`
	`movs  r4, #0`
	`b  .L5`
	
	`.L6:`
	`; LCD_PrintInt8(i)`
	`movs  r0, r4`
	`bl  LCD_PrintInt8`
	`adds  r4, r4, #1`
	`uxtb  r4, r4`
	
	`.L5:`
	`; while(i  10)`
	`cmp  r4, #9`
	`bls  .L6`
	
	`; epilog not shown`
		`; prolog not shown`
		`; i = 0`
		`movs  r4, #0`
		
		`.L5:`
		`movs  r0, r4`
		`; LCD_PrintInt8(i)`
		`bl  LCD_PrintInt8`
		`adds  r4, r4, #1 ; i++`
		`uxtb  r4, r4`
		
		`; check i not equal to 10`
		`cmp  r4, #10`
		`bne  .L5`
		
		`; epilog not shown`
			`; prolog not shown`
			`; i = 0`
			`movs  r4, #0`
			
			`.L6:`
			`movs  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`

`// Conditional statement`
`void Conditional(uint8 i, uint8 j)`
`{`
`  if(j == 1)`
`  {`
`  LCD_PrintInt8(i);`
`  }`
`  else`
`  {`
`  LCD_PrintInt8(i + 1);`
`  }`
`}`
	`; prolog`
	`push  {r4, lr}`
	`; if(j == 1)`
	`cmp  r1, #1`
	`bne  .L8`
	
	`; LCD_PrintInt8(i)`
	`bl  LCD_PrintInt8`
	`b  .L7`
	
	`.L8:`
	`; LCD_PrintInt8(i + 1)`
	`adds  r0, r0, #1`
	`uxtb  r0, r0`
	`bl  LCD_PrintInt8`
	
	`.L7:`
	`pop  {r4, pc}`
	`; epilog not shown`
		`; same as for debug optimization`	`; prolog`
			`push  {r4, lr}`
			`; if(j == 1)`
			`cmp  r1, #1`
			`beq  .L11`
			
			`.L11:`
			`; LCD_PrintInt8(i + 1)`
			`adds  r0, r0, #1`
			`uxtb  r0, r0`
			`bl  LCD_PrintInt8`
			
			`pop  {r4, pc}`
			`; epilog not shown`

`// 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)`
	`cmp  r0, #0`
	`beq  .L12`
	`cmp  r0, #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`
		`; ; prolog not shown`
		`; switch(j)`
		`cmp  r0, #0`
		`beq  .L12`
		`cmp  r0, #1`
		`beq  .L13`
		`b  .L15`
		
		`.L12  ; case 0`
		`mov  r0, #1`
		`bl  LCD_PrintInt8`
		`b  .L10  ; break`
		
		`.L13:  ; case 1`
		`movs  r0, #2`
		`bl  LCD_PrintInt8`
		`b  .L10  ; break`
		
		`.L15:  ; default`
		`movs  r0, #0`
		`bl  LCD_PrintInt8`
		`movs  r8, r8 ; break - nop`
			`; ; prolog not shown`
			`; switch(j)`
			`cmp  r0, #0`
			`beq  .L14`
			`cmp  r0, #1`
			`beq  .L15`
			`movs  r0, #0 ; default`
			`bl  LCD_PrintInt8`
			
			`.L14  ; case 0`
			`mov  r0, #1`
			`bl  LCD_PrintInt8`
			`b  .L10  ; break`
			
			`.L15:  ; case 1`
			`movs  r0, #2`
			`bl  LCD_PrintInt8`
			`b  .L12  ; break`
			
			`.L12:`
			`; epilog not shown`

Appendix: Compiler output details

C code	GCC, Cortex®-M0/M0+, debug optimization	GCC, Cortex®-M0/M0+, minimal optimization	GCC, Cortex®-M0/M0+, high optimization
	.L10: ; epilog not shown	.L10: ; epilog not shown	

// Ternary operator
void Ternary(uint8 i)
{
  LCD_PrintInt8(
  (i == 1) ? 80 :
100);
}
	; prolog not shown
	; check value of i
	cmp r0, #1
	bne .L18
	adds r0, r0, #79
	mov r3, #80
	b .L17
	
	.L18:
	mov r0, #100
	
	.L17:
	bl LCD_PrintInt8
	; epilog not shown
		; prolog not shown
		cmp r0, #1
		beq .L18
		movs r0, #100
		b .L17
		
		.L18:
		movs r0, #80
		
		.L17:
		bl LCD_PrintInt8
		; epilog not shown
			; prolog not shown
			cmp r0, #1
			beq .L21
			movs r0, #100
			
			.L21:
			movs r0, #80
			b .L20
			
			.L20:
			bl LCD_PrintInt8
			; epilog not shown

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

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

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

// Division
int DoDiv(int x, int
y)
{
  return x / y;
}
	push {r4, lr}
	; call into c helper
	function
	bl __aeabi_idiv
	; return with result
	pop {r4, pc}
		; same as for debug
		optimization
			; same as for debug
			optimization

// Modulo operator
int DoMod(int x, int
y)
{
  return x % y;
}
	push {r4, lr}
	; call into c helper
	function
	bl __aeabi_idivmod
	; return with result
	movs r0, r1
	pop {r4, pc}
		; same as for debug
		optimization
			; same as for debug
			optimization

// Pointer
void Pointer(uint8 x,
uint8 *ptr)
{
  *ptr = *ptr + x;
  ptr++;
  LCD_PrintInt8(*ptr);
}
	; *ptr = *ptr + x
	ldrb r3, [r1] ; R1
	= ptr
	adds r0, r3, r0 ; R0
	= x
	strb r0, [r1]
	
	; ptr++
	; LCD_PrintInt8(*ptr)
	ldrb r0, [r1, #1]
	bl LCD_PrintInt8
		; same as for debug
		optimization
			; same as for debug
			optimization

// Function pointer void FuncPtr(uint8 x,	; (*fptr)(x) ; in a blx	; same as for debug optimization	; same as for debug optimization

Appendix: Compiler output details

C code	GCC, Cortex®-M0/M0+, debug optimization	GCC, Cortex®-M0/M0+, minimal optimization	GCC, Cortex®-M0/M0+, high optimization

void
*fptr(uint8))
{
  (*fptr)(x);
}
	instruction, the LS
	; bit of the register
	must be
	; 1 to keep the CPU
	in Thumb
	; mode, or an
	exception occurs
	blx r1		

// 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
		; membera = 5
		ldr r3, .L27
		movs r2, #5
		strb r2, [r3]
		; memberb = 10
		adds r2, r2, #5
		strb r2, [r3, #1]
		; memberc = 15
		movs r1, #15
		strb r1, [r3, #2]
		movs r1, #0
		strb r1, [r3, #3]
		strb r1, [r3, #4]
		strb r1, [r3, #5]
		; memberd = 20
		movs r2, #20
		strb r2, [r3, #6]
		movs r2, #0
		strb r2, [r3, #7]
		bx lr
		
		.L27:
		.word myfoo_p
			; membera = 5
			movs r2, #5
			ldr r3, .L30
			; memberd = 20
			movs r1, #20
			; membera = 5
			strb r2, [r3]
			; memberb = 10
			adds r2, r2, #5
			strb r2, [r3, #1]
			; memberd = 20
			adds r2, r2, #5
			strb r2, [r3, #2]
			movs r2, #0
			strb r1, [r3, #6]
			strb r2, [r3, #3]
			
			strb r2, [r3, #4]
			strb r2, [r3, #5]
			strb r2, [r3, #7]
			bx lr
			
			.L30:
			.word myfoo_p

// unpacked structures
struct FOO
{
  uint8 membera;
  uint8 memberb;
  uint32 memberc;
  uint16 memberd;
};
 	
extern struct FOO
myfoo;
 	
void
	; 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
		; membera = 5
		ldr r3, .L30
		movs r2, #5
		strb r2, [r3]
		; memberb = 10
		adds r2, r2, #5
		strb r2, [r3, #1]
		; memberc = 15
		adds r2, r2, #5
		str r2, [r3, #4]
		; memberd = 20
		adds r2, r2, #5
			; same as for minimal
			optimization

Appendix: Compiler output details

C code	GCC, Cortex®-M0/M0+, debug optimization	GCC, Cortex®-M0/M0+, minimal optimization	GCC, Cortex®-M0/M0+, high optimization

`PackedStruct(void)`
`{`
`  myfoo.membera =  5;`
`  myfoo.memberb = 10;`
`  myfoo.memberc = 15;`
`  myfoo.memberd = 20;`
`}`
	`ldr  r3, [pc, .L35]`
	`mov  r2, #20`
	`strh  r2, [r3, #8]`
	
	`.L35:`
	`.word  myfoo`
		`strh  r2, [r3, #8]`
		`bx  lr`
		
		`.L30:`
		`.word  myfoo`	

14.3 Assembler examples, MDK for Cortex®-M3

Note: *Table 31 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 and result for details on register usage and stack usage in compiler functions.

Table 31 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`

`// 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`
		`; 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`

Appendix: Compiler output details

C code	MDK, Cortex®-M3, no optimization	MDK, Cortex®-M3, size optimization	MDK, Cortex®-M3, speed optimization
	blt.n <ForLoop+0x8> pop {r4, pc} ; return		

// 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

// 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	; prolog push {r4, lr}	; prolog not shown ; switch(j)	; same as for size optimization

Appendix: Compiler output details

C code	MDK, Cortex®-M3, no optimization	MDK, Cortex®-M3, size optimization	MDK, Cortex®-M3, speed optimization

void SwitchCase(uint8 j)
{
  switch(j)
  {
  case 0:
  LCD_PrintInt8(1);
  break;
 	
  case 1:
  LCD_PrintInt8(2);
  break;
 	
  default:
  LCD_PrintInt8(0);
  break;
  }
}
	; 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
		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	

// 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	; no prolog mov r2, r0	; no prolog subs r0, r0, r1	; same as for size optimization

Appendix: Compiler output details

C code	MDK, Cortex®-M3, no optimization	MDK, Cortex®-M3, size optimization	MDK, Cortex®-M3, speed optimization

`int DoSub(int x, int y)`
`{`
`  return x - y;`
`}`
	`subs  r0, r2, r1`
	`bx    lr ; return with result`
		`bx    lr ; return with result`	

`// 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`

`// 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`
		`; (*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`
			`; same as for size optimization`

Appendix: Compiler output details

C code	MDK, Cortex®-M3, no optimization	MDK, Cortex®-M3, size optimization	MDK, Cortex®-M3, speed optimization
	; in Thumb mode, or an ; exception occurs blx r4 pop {r4, r5, r6, pc} ; return	function bx r1	

// 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

// 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

14.4 Assembler examples, MDK for Cortex®-M0/M0+

Table 32 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 and result for details on register usage and stack usage in compiler functions.

Appendix: Compiler output details

Table 32 Compiler output details for MDK compiler for Cortex®-M0/M0+ CPU

C code	MDK, Cortex®-M0/M0+, no optimization	MDK, Cortex®-M0/M0+, size optimization	MDK, Cortex®-M0/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`

`// 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`
		`; 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 <`
			`; same as for size optimization`

Appendix: Compiler output details

C code	MDK, Cortex®-M0/M0+, no optimization	MDK, Cortex®-M0/M0+, size optimization	MDK, Cortex®-M0/M0+, speed optimization
	<WhileLoop+0x10>: cmp r4, #a ; i < 10 blt.n <WhileLoop+0x6> pop {r4, pc}; return	10 bcc.n <WhileLoop+0x4> pop {r4, pc}; return	

// 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>
		; 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>:
			; same as for size optimization

Appendix: Compiler output details

C code	MDK, Cortex®-M0/M0+, no optimization	MDK, Cortex®-M0/M0+, size optimization	MDK, Cortex®-M0/M0+, speed optimization
	<SwitchCase+0x1e>: movs r0, #0 ; default bl LCD_PrintInt8 nop <SwitchCase+0x26>: nop pop {r4, pc} ; return	bl LCD_PrintInt8 pop {r4, pc} ; return	

// 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

Appendix: Compiler output details

C code	MDK, Cortex®-M0/M0+, no optimization	MDK, Cortex®-M0/M0+, size optimization	MDK, Cortex®-M0/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`

`// Packed structures`
`struct FOO_P`
`{`
`  uint8  membera;`
`  uint8  memberb;`
`  uint32 memberc;`
`  uint16 memberd;`
`} __attribute__ ((packed));`
 	
`extern struct FOO_P`
	`; 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`
		`; 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`
			`; same as for size optimization`

C code	MDK, Cortex®-M0/M0+, no optimization	MDK, Cortex®-M0/M0+, size optimization	MDK, Cortex®-M0/M0+, speed optimization

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

// 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

14.5 Assembler examples, IAR for Cortex®-M3

14.5.1 IAR for Cortex®-M3, none, size, and speed optimization

Table 33 shows, for the IAR 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 and result for details on register usage and stack usage in compiler functions.

Table 33 Compiler output details for IAR compiler for Cortex®-M3 CPU

C code	IAR, Cortex®-M3 no optimization	IAR, Cortex®-M3, size optimization	IAR, 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
	MOVS R0,#+128
	BL
	LCD_PrintInt8
		; same as for no optimization	; same as for size optimization

// Calling a function with	; R0 = first argument ; R1 = second	; same as for no optimization	; same as for size optimization

Appendix: Compiler output details

C code	IAR, Cortex®-M3 no optimization	IAR, Cortex®-M3, size optimization	IAR, Cortex®-M3, speed optimization

`// two arguments`
`LCD_Position(0, 2);`
	`argument`
	`MOVS R1,#+2`
	`MOVS R0,#+0`
	`BL LCD_Position`		

`// For loop:`
`void ForLoop(uint8 i)`
`{`
`  for(i = 0; i < 10; i++)`
`  {`
`  LCD_PrintInt8(i);`
`  }`
`}`
	`; function prolog`
	`; i is saved on the stack`
	`PUSH {R4,LR}`
	`; R4 = i`
	`MOVS R4,#+0`
	`; do the function call with`
	`; i as the argument in R0`
	`??ForLoop_0:`
	`MOVS R0,R4`
	`UXTB R0,R0`
	`; check i not equal to 10`
	`CMP R0,#+10`
	`BGE.N ??ForLoop_1`
	
	`MOVS R0,R4`
	`UXTB R0,R0`
	`BL`
	`LCD_PrintInt8`
	
	`ADDS R4,R4,#+1 ; i++`
	`B.N ??ForLoop_0`
	
	`; function epilog`
	`??ForLoop_1:`
	`POP {R4,PC};`
	`return`
		`; function prolog`
		`PUSH {R4,LR}`
		
		`B.N ?Subroutine0`
		`?Subroutine0: (+1)`
		`MOVS R4,#+0`
		`?Subroutine0_0: (+1)`
		`UXTB R0,R4`
		`BL`
		`LCD_PrintInt8`
		`ADDS R4,R4,#+1 ; i++`
		`UXTB R0,R4`
		`CMP R0,#+10 ; i = 10?`
		`BLT.N ??Subroutine0_0`
		`POP {R4,PC} ; return`
			`; function prolog`
			`PUSH {R4,LR}`
			`MOVS R4,#+0`
			
			`UXTB R0,R4`
			`BL`
			`LCD_PrintInt8`
			
			`ADDS R4,R4,#+1 ; i++`
			`UXTB R0,R4`
			`CMP R0,#+10 ; i = 10?`
			`BLT.N ??ForLoop_0`
			
			`POP {R4,PC} ; return`

`// 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`
	`MOVS R4,#+0`
	
	`??WhileLoop_0:`
	`MOVS R0,R4`
	`UXTB R0,R0`
	`; check i not equal to 10`
	`CMP R0,#+10`
	`BGE.N ??WhileLoop_1`
	
	`LCD_PrintInt8:`
	`MOVS R0,R4`
	`UXTB R0,R0`
	`; LCD_PrintInt8(i)`
	`BL`
	`LCD_PrintInt8`
	
	`ADDS R4,R4,#+1 ; i++`
	`B.N ??WhileLoop_0`
	
	`??WhileLoop_1:`
	`POP {R4,PC}`
		`; prolog not shown`
		`; optimizes to match the for ; loop function`
		`REQUIRE ?Subroutine0`
		`;;  Fall through to label ?Subroutine0`
		
		`; epilog not shown`
			`; function prolog`
			`PUSH {R4,LR}`
			`MOVS R4,#+0`
			
			`UXTB R0,R4`
			`BL`
			`LCD_PrintInt8`
			
			`ADDS R4,R4,#+1 ; i++`
			`UXTB R0,R4`
			`CMP R0,#+10 ; i = 10?`
			`BLT.N ??WhileLoop_0`
			
			`POP {R4,PC} ; return`

Appendix: Compiler output details

C code	IAR, Cortex®-M3 no optimization	IAR, Cortex®-M3, size optimization	IAR, Cortex®-M3, 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)`
	`Conditional:`
	`PUSH {R3-R5,LR}`
	`MOVS R4,R0`
	`MOVS R5,R1`
	
	`MOVS R0,R5`
	`UXTB R0,R0`
	`CMP R0,#+1`
	`BNE.N ??Conditional_0`
	
	`; LCD_PrintInt8(i);`
	`MOVS R0,R4`
	`UXTB R0,R0`
	`BL LCD_PrintInt8`
	`B.N ??Conditional_1`
	
	`; LCD_PrintInt8(i + 1);`
	`??Conditional_0:`
	`ADDS R0,R4,#+1`
	`UXTB R0,R0`
	`BL LCD_PrintInt8`
	
	`??Conditional_1:`
	`POP {R0,R4,R5,PC}`
		`; no prolog`
		`; if(j == 1)`
		`CMP R1,#+1`
		`; if-then-then instruction,`
		`; if not equal, then add 1 to ; i and call LCd_PrintInt8`
		`; with i`
		`ITT NE`
		`ADDNE R0,R0,#+1`
		`UXTBNE R0,R0`
		
		`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 not shown`
	`; switch(j)`
	`UXTB R0,R0`
	`CMP R0,#+0`
	`BEQ.N ??SwitchCase_0`
	`CMP R0,#+1`
	`BEQ.N ??SwitchCase_1`
	`B.N ??SwitchCase_2`
	
	`; case 0`
	`??SwitchCase_0:`
	`MOVS R0,#+1`
	`BL LCD_PrintInt8`
	`B.N ??SwitchCase_3`
	
	`; case 1`
	`??SwitchCase_1:`
	`MOVS R0,#+2`
	`BL LCD_PrintInt8`
	`B.N ??SwitchCase_3`
	
	`; default`
	`??SwitchCase_2:`
	`MOVS R0,#+0`
		`; no prolog`
		`; switch(j)`
		`CBZ.N R0,??SwitchCase_0`
		`CMP R0,#+1`
		`BEQ.N ??SwitchCase_1`
		`B.N ??SwitchCase_2`
		
		`; case 0`
		`MOVS R0,#+1`
		`B.N ??SwitchCase_3`
		
		`; case 1`
		`MOVS R0,#+2`
		`B.N ??SwitchCase_3`
		
		`; case 2`
		`MOVS R0,#+0`
		`; default`
		`B.W LCD_PrintInt8`
			`; same as for size optimization`

Appendix: Compiler output details

C code	IAR, Cortex®-M3 no optimization	IAR, Cortex®-M3, size optimization	IAR, Cortex®-M3, speed optimization
	BL LCD_PrintInt8 ; no epilog		

// Ternary operator
void Ternary(uint8 i)
{
  LCD_PrintInt8(
  (i == 1) ? 80 :
100);
}
	; prolog not shown
	; check value of i
	CMP R0,#+1
	; branch if not equal
	BNE.N ??Ternary_0
	MOVS R0,#+80
	B.N ??Ternary_1
	
	??Ternary_0:
	MOVS R0,#+100
	??Ternary_1:
	UXTB R0,R0
	BL
	LCD_PrintInt8
	; no epilog
		; 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.W
		LCD_PrintInt8
			; same as for size
			optimization

// Addition operation
int DoAdd(int x, int
y)
{
  return x + y;
}
	; no prolog
	ADDS R0,R1,R0
	BX LR 3 ;
	return value
		; same as for no
		optimization
			; same as for size
			optimization

// Subtraction
operation
int DoSub(int x, int
y)
{
  return x - y;
}
	; no prolog
	SUBS R0,R0,R1
	BX LR ; return
	value
		; same as for no
		optimization
			; same as for size
			optimization

// Multiplication
int DoMul(int x, int
y)
{
  return x * y;
}
	; no prolog
	MULS R0,R0,R1
	BX LR ;
	return value
		; same as for no
		optimization
			; same as for size
			optimization

// Division
int DoDiv(int x, int
y)
{
  return x / y;
}
	; no prolog
	SDIV R0,R0,R1
	BX LR ;
	return value
		; same as for no
		optimization
			; same as for size
			optimization

Appendix: Compiler output details

C code	IAR, Cortex®-M3 no optimization	IAR, Cortex®-M3, size optimization	IAR, Cortex®-M3, speed optimization

`// Modulo operator`
`int DoMod(int x, int y)`
`{`
`  return x % y;`
`}`
	`; no prolog`
	`SDIV R2,R0,R1`
	`; multiply and subtract instruction`
	`; implements remainder =`
	`; dividend - (quotient * divisor)`
	`MLS R0,R1,R2,R0`
	`BX LR ; return value`
		`; same as for no optimization`	`; same as for size optimization`

`// Pointer`
`void Pointer(uint8 x, uint8 *ptr)`
`{`
`  *ptr = *ptr + x;`
`  ptr++;`
`  LCD_PrintInt8(*ptr);`
`}`
	`; *ptr = *ptr + x`
	`LDRB R0,[R1, #+0]`
	`ADDS R0,R4,R0`
	`STRB R0,[R1, #+0]`
	
	`; ptr++`
	`; LCD_PrintInt8(*ptr)`
	`ADDS R5,R1,#+1`
	`LDRB R0,[R5, #+0]`
	`BL LCD_PrintInt8`
		`; *ptr = *ptr + x`
		`LDRB R2, [R1, #+0] ; R1 = ptr`
		`ADDS R0, R0, R2 ; R0 = x`
		`STRB R0, [R1, #+0]`
		
		`; ptr++`
		`; LCD_PrintInt8(*ptr)`
		`LDRB R0, [R1, #+1]`
		`B.W LCD_PrintInt8`
			`; same as for size optimization`

`// Function pointer`
`void FuncPtr(uint8 x, void *fptr(uint8))`
`{`
`  (*fptr)(x);`
`}`
	`; (*fptr)(x)`
	`PUSH {R3-R5,LR}`
	`MOVS R4,R0`
	`MOVS R5,R1`
	
	`MOVS R0,R4`
	`UXTB R0,R0`
	`; 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 R5`
		`; (*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`
		`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 =`
	`; myfoo.membera = 5`
	`LDR.N R0,??DataTable2`
	`MOVS R1,#+5`
	`STRB R1,[R0, #+0]`
	`; myfoo_p.memberb = 10;`
	`MOVS R1,#+10`
	`STRB R1,[R0, #+1]`
	`; myfoo_p.memberc = 15;`
	`MOVS R1,#+15`
	`STR R1,[R0, #+2]`
	`; myfoo_p.memberd = 20;`
	`MOVS R1,#+20`
	`STRH R1,[R0, #+6]`
	
	`BX LR`
		`; myfoo.membera = 5`
		`LDR.N R0,??DataTable2`
		`MOVS R1,#+5`
		`STRB R1,[R0, #+0]`
		`; myfoo_p.memberb = 10;`
		`MOVS R2,#+10`
		`; myfoo_p.memberc = 15;`
		`MOVS R3,#+15`
		`; myfoo_p.memberd = 20;`
		`MOVS R1,#+20`
		`STRB R2,[R0, #+1]`
		`STR R3,[R0, #+2]`
		`STRH R1,[R0, #+6]`
		
		`BX LR`
			`; same as for size optimization`

Appendix: Compiler output details

C code	IAR, Cortex®-M3 no optimization	IAR, Cortex®-M3, size optimization	IAR, Cortex®-M3, speed optimization
20; }			

// 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;
}
	; myfoo.membera = 5
	LDR.N
	R0,??DataTable2_1
	MOVS R1,#+5
	STRB R1,[R0, #+0]
	; myfoo.memberb = 10;
	MOVS R1,#+10
	STRB R1,[R0, #+1]
	; myfoo.memberc = 15;
	MOVS R1,#+15
	STR R1,[R0, #+4]
	; myfoo.memberd = 20;
	MOVS R1,#+20
	STRH R1,[R0, #+8]
	
	BX LR
		; myfoo.membera = 5
		LDR.N
		R0,??DataTable2_1
		MOVS R1,#+5
		STRB R1,[R0, #+0]
		; myfoo.memberb = 10;
		MOVS R2,#+10
		; myfoo.memberc = 15;
		MOVS R3,#+15
		; myfoo.memberd = 20;
		MOVS R1,#+20
		STRB R2,[R0, #+1]
		STR R3,[R0, #+4]
		STRH R1,[R0, #+8]
		
		BX LR
			; same as for size optimization

14.5.2 IAR for Cortex®-M3, low, medium, and balanced optimization

Table 34 shows, for the IAR 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 and result for details on register usage and stack usage in compiler functions.

Table 34 Compiler output details for IAR compiler for Cortex®-M3 CPU

C code	IAR, Cortex®-M3 low optimization	IAR, Cortex®-M3, medium optimization	IAR, Cortex®-M3, balanced 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
	MOVS R0,#+128
	BL
	LCD_PrintInt8
		; same as for no optimization	; same as for size 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 low optimization	; same as for size 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 {R4,LR}
	0; i++)
	MOVS R4,#+0
	B.N ??ForLoop_0
		; same as for low optimization	; function prolog
			PUSH {R4,LR}
			
			B.N
			?Subroutine0
			
			?Subroutine0: (+1)
			MOVS R4,#+0

Appendix: Compiler output details

C code	IAR, Cortex®-M3 low optimization	IAR, Cortex®-M3, medium optimization	IAR, Cortex®-M3, balanced optimization

}
 	
	??ForLoop_1:
	MOVS R0,R4
	UXTB R0,R0
	BL
	LCD_PrintInt8
	
	ADDS R4,R4,#+1
	??ForLoop_0
	MOVS R0,R4
	UXTB R0,R0
	CMP R0,#+10
	BLT.N ??ForLoop_1
	
	; function epilog
	POP {R4,PC};
	return
			?Subroutine0_0: (+1)
			UXTB R0,R4
			BL
			LCD_PrintInt8
			ADDS R4,R4,#+1 ; i++
			UXTB R0,R4
			CMP R0,#+10 ; i = 10?
			BLT.N
			??Subroutine0_0
			POP {R4,PC} ; return

// While loop
// i is type automatic, see
// Accessing Automatic Variables
// for details
uint8 i = 0;
 	
while (i < 10)
{
LCD_PrintInt8(i);
i++;
}
	; function prolog
	; i is saved on the stack PUSH
	{R4,LR}
	
	MOVS R4,#+0
	B.N
	??WhileLoop_0
	
	??WhileLoop_1:
	MOVS R0,R4
	UXTB R0,R0
	BL
	LCD_PrintInt8
	
	ADDS R4,R4,#+1
	
	??WhileLoop_0:
	MOVS R0,R4
	UXTB R0,R0
	CMP R0,#+10
	BLT.N
	??WhileLoop_1
	; function epilog
	POP {R4,PC};
	return
		; same as for low optimization	; prolog not shown
			; optimizes to match the for ; loop function
			REQUIRE ?Subroutine0
			;; Fall through to label ?Subroutine0
			
			; epilog not shown

// Conditional statement
void Conditional(uint8 i, uint8 j)
{
if(j == 1)
{
LCD_PrintInt8(i);
}
else
{
LCD_PrintInt8(i + 1);
}
}
	PUSH {R7,LR}
	; if(j == 1)
	UXTB R1,R1
	CMP R1,#+1
	BNE.N
	??Conditional_0
	; LCD_PrintInt8(i);
	UXTB R0,R0
	BL
	LCD_PrintInt8
	B.N
	??Conditional_1
	; LCD_PrintInt8(i + 1);
	Conditional_0:
	ADDS R0,R0,#+1
	UXTB R0,R0
	BL
	LCD_PrintInt8
	
	??Conditional_1:
		; no prolog
		; if(j == 1)
		CMP R1,#+1
		BNE.N
		??Conditional_0
		
		; LCD_PrintInt8(i);
		B.W
		LCD_PrintInt8
		; LCD_PrintInt8(i + 1);
		??Conditional_0:
		ADDS R0,R0,#+1
		UXTB R0,R0
		B.W
		LCD_PrintInt8
			; no prolog
			; if(j == 1)
			CMP R1,#+1
			; if-then-then instruction,
			; if not equal, then add 1 to ; i and call LCd_PrintInt8
			; with i
			ITT NE
			ADDNE R0,R0,#+1
			UXTBNE R0,R0
			
			B.W
			LCD_PrintInt8

Appendix: Compiler output details

C code	IAR, Cortex®-M3 low optimization	IAR, Cortex®-M3, medium optimization	IAR, Cortex®-M3, balanced optimization
	POP {R0,PC}; return		

// 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)
	UXTB R0,R0
	CMP R0,#+0
	BEQ
	??SwitchCase_0
	CMP R0,#+1
	BEQ
	??SwitchCase_1
	B.N
	??SwitchCase_2
	; case 0
	??SwitchCase_0:
	MOVS R0,#+1
	BL
	LCD_PrintInt8
	B.N
	??SwitchCase_3
	
	; case 1
	??SwitchCase_1:
	MOVS R0,#+2
	BL
	LCD_PrintInt8
	B.N
	??SwitchCase_3
	
	; default
	??SwitchCase_2:
	MOVS R0,#+0
	BL
	LCD_PrintInt8
	; epilog not shown
		; no prolog
		; switch(j)
		CMP R0,#+0
		BEQ.N
		??SwitchCase_0
		CMP R0,#+1
		BEQ.N
		??SwitchCase_1
		B.N
		??SwitchCase_2
		
		case 0:
		??SwitchCase_0: (+1)
		MOVS R0,#+1
		B.W
		LCD_PrintInt8
		
		; case 1:
		??SwitchCase_1:
		MOVS R0,#+2
		B.W
		LCD_PrintInt8
		
		; default:
		??SwitchCase_2:
		MOVS R0,#+0
		B.W
		LCD_PrintInt8
			; no prolog
			; switch(j)
			CBZ.N
			R0,??SwitchCase_0
			CMP R0,#+1
			BEQ.N
			??SwitchCase_1
			B.N
			??SwitchCase_2
			
			; case 0
			MOVS R0,#+1
			B.N
			??SwitchCase_3
			
			; case 1
			MOVS R0,#+2
			B.N
			??SwitchCase_3
			
			; case 2
			MOVS R0,#+0
			; default
			B.W
			LCD_PrintInt8

// Ternary operator
void Ternary(uint8 i)
{
  LCD_PrintInt8(
  (i == 1) ? 80 :
100);
}
	; prolog not shown
	; check value of i
	CMP R0,#+1
	; branch if not equal
	BNE.N ??Ternary_0
	MOVS R0,#+80
	B.N ??Ternary_1
	
	??Ternary_0:
	MOVS R0,#+100
	??Ternary_1:
	UXTB R0,R0
	BL
	LCD_PrintInt8
	; no epilog
		; prolog not shown
		; check value of i
		CMP R0,#+1
		; branch if not equal
		BNE.N ??Ternary_0
		MOVS R0,#+80
		B.N ??Ternary_1
		
		??Ternary_0:
		MOVS R0,#+100
		??Ternary_1:
		BL
		LCD_PrintInt8
		; no epilog
			; prolog not shown
			; check value of i
			CMP R0,#+1
			; “ite” stands for if-then-
			; else instruction
			; “eq” condition checks
			; if the previous compare
			; instruction has set the
			; “equal to” flag
			ITE EQ
			MOVEQ R0,#+80
			MOVNE R0,#+100
			B.W
			LCD_PrintInt8

// Addition operation
int DoAdd(int x, int y)
{
  return x + y;
}
	; no prolog
	ADDS R0,R1,R0
	BX LR ; return value
		; same as for low optimization	; same as for low optimization

// Subtraction operation int DoSub(int x, int y)	; no prolog SUBS R0,R0,R1 BX LR ; return value	; same as for low optimization	; same as for low optimization

Appendix: Compiler output details

C code	IAR, Cortex®-M3 low optimization	IAR, Cortex®-M3, medium optimization	IAR, Cortex®-M3, balanced optimization

{
  return x - y;
}			

// Multiplication
int DoMul(int x, int y)
{
  return x * y;
}
	; no prolog
	MULS R0,R1,R0
	BX LR ;
	return value
		; same as for low optimization	; same as for low optimization

// Division
int DoDiv(int x, int y)
{
  return x / y;
}
	; no prolog
	SDIV R0,R0,R1
	BX LR ;
	return value
		; same as for low optimization	; same as for low optimization

// Modulo operator
int DoMod(int x, int y)
{
  return x % y;
}
	; no prolog
	SDIV R2,R0,R1
	; multiply and subtract instruction
	; implements remainder =
	; dividend - (quotient * divisor)
	MLS R0,R1,R2,R0
	BX LR ;
	return value
		; same as for low optimization	; same as for low optimization

// Pointer
void Pointer(uint8 x, uint8 *ptr)
{
  *ptr = *ptr + x;
  ptr++;
  LCD_PrintInt8(*ptr);
}
	; *ptr = *ptr + x
	LDRB R2,[R1, #+0]
	ADDS R0,R0,R2
	STRB R0,[R1, #+0]
	
	; ptr++
	; LCD_PrintInt8(*ptr)
	ADDS R0,R1,#+1
	LDRB R0,[R0, #+0]
	BL LCD_PrintInt8
		; *ptr = *ptr + x
		LDRB R2, [R1, #+0]
		; R1 = ptr
		ADDS R0, R0, R2
		; R0 = x
		STRB R0, [R1, #+0]
		
		; ptr++
		; LCD_PrintInt8(*ptr)
		LDRB R0, [R1, #+1]
		B.W LCD_PrintInt8
			; same as for medium optimization

// Function pointer
void FuncPtr(uint8 x, void *fptr(uint8))
{
  (*fptr)(x);
}
	; (*fptr)(x)
	PUSH {R7,LR}
	UXTB R0,R0
	BLX R1
	POP {R0,PC}
		; (*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
		BX R1
			; same as for medium 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.membera = 5
	LDR.N R0,??DataTable2
	MOVS R1,#+5
	STRB R1,[R0, #+0]
	; myfoo_p.memberb = 10;
	MOVS R1,#+10
	STRB R1,[R0, #+1]
	; myfoo_p.memberc = 15;
	MOVS R1,#+15
	STR R1,[R0, #+2]
	; myfoo_p.memberd = 20;
	MOVS R1,#+20
		; same as for low optimization	; myfoo.membera = 5
			LDR.N R0,??DataTable2
			MOVS R1,#+5
			STRB R1,[R0, #+0]
			; myfoo_p.memberb = 10;
			MOVS R2,#+10
			; myfoo_p.memberc = 15;
			MOVS R3,#+15
			; myfoo_p.memberd = 20;
			MOVS R1,#+20
			STRB R2,[R0, #+1]
			STR R3,[R0, #+2]

Appendix: Compiler output details

C code	IAR, Cortex®-M3 low optimization	IAR, Cortex®-M3, medium optimization	IAR, Cortex®-M3, balanced optimization

{
  myfoo_p.membera = 5;
  myfoo_p.memberb = 10;
  myfoo_p.memberc = 15;
  myfoo_p.memberd = 20;
}
	STRH R1,[R0, #+6]
	
	BX LR
			STRH R1,[R0, #+6]
			
			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;
}
	; myfoo.membera = 5
	LDR.N R0,??DataTable2_1
	MOVS R1,#+5
	STRB R1,[R0, #+0]
	; myfoo.memberb = 10;
	MOVS R1,#+10
	STRB R1,[R0, #+1]
	; myfoo.memberc = 15;
	MOVS R1,#+15
	STR R1,[R0, #+4]
	; myfoo.memberd = 20;
	MOVS R1,#+20
	STRH R1,[R0, #+8]
	
	BX LR
		; same as for low optimization	; myfoo.membera = 5
			LDR.N R0,??DataTable2_1
			MOVS R1,#+5
			STRB R1,[R0, #+0]
			; myfoo.memberb = 10;
			MOVS R2,#+10
			; myfoo.memberc = 15;
			MOVS R3,#+15
			; myfoo.memberd = 20;
			MOVS R1,#+20
			STRB R2,[R0, #+1]
			STR R3,[R0, #+4]
			STRH R1,[R0, #+8]
			
			BX LR

14.6 Assembler examples, IAR for Cortex®-M0/M0+

Table 35 shows, for the IAR compiler for the Cortex®-M0/M0+, examples of compiler output for different optimization options. The examples were extracted from the *.lst* files generated by the compiler.

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

Table 35 Compiler output details for IAR compiler for Cortex®-M0/M0+ CPU

C Code	IAR, Cortex®-M0/M0+ no optimization	IAR, Cortex®-M0/M0+, size optimization	IAR, Cortex®-M0/N0+, 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`
	`MOVS R0,#+128`
	`BL LCD_PrintInt8`
		`; same as for no optimization`	`; same as for size 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 size 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 {R4,LR}`
	`; R4 = i`
	`MOVS R4,#+0`
	`; do the function call with`
	`; i as the argument in R0`
	`??ForLoop_0:`
	`MOVS R0,R4`
	`UXTB R0,R0`
	`; check i not equal to 10`
	`CMP R0,#+10`
	`BGE ??ForLoop_1`
	
	`MOVS R0,R4`
	`UXTB R0,R0`
	`BL LCD_PrintInt8`
	
	`ADDS R4,R4,#+1 ; i++`
	`B ??ForLoop_0`
	
	`; function epilog`
	`??ForLoop_1:`
	`POP {R4,PC};`
	`return`
		`; function prolog`
		`PUSH {R4,LR}`
		
		`B.N ?Subroutine0`
		
		`?Subroutine0:`
		`MOVS R4,#+0`
		
		`?Subroutine0_0:`
		`UXTB R0,R4`
		`BL LCD_PrintInt8`
		`ADDS R4,R4,#+1 ; i++`
		`UXTB R0,R4`
		`CMP R0,#+10 ; i = 10?`
		`BLT ??Subroutine0_0`
		`POP {R4,PC} ; return`
			`; function prolog`
			`PUSH {R4,LR}`
			`MOVS R4,#+0`
			
			`UXTB R0,R4`
			`BL LCD_PrintInt8`
			
			`ADDS R4,R4,#+1 ; i++`
			`UXTB R0,R4`
			`CMP R0,#+10 ; i = 10?`
			`BLT ??ForLoop_0`
			
			`POP {R4,PC} ; return`

`// While loop`
`// i is type automatic, see`
`// Accessing Automatic Variables`
`// for details`
	`; prolog not shown`
	`; i = 0`
	`MOVS R4,#+0`
	
	`??WhileLoop_0:`
	`MOVS R0,R4`
		`; optimizes to match the for ; loop function`
		`REQUIRE ?Subroutine0`
			`; function prolog`
			`PUSH {R4,LR}`
			`MOVS R4,#+0`
			
			`??WhileLoop_0:`
			`UXTB R0,R4`

Appendix: Compiler output details

C Code	IAR, Cortex®-M0/M0+ no optimization	IAR, Cortex®-M0/M0+, size optimization	IAR, Cortex®-M0/N0+, speed optimization

uint8 i = 0;
 	
while (i < 10)
{
  LCD_PrintInt8(i);
  i++;
}
	UXTB R0,R0
	; check i not equal to 10
	CMP R0,#+10
	BGE
	??WhileLoop_1
	
	LCD_PrintInt8:
	MOVS R0,R4
	UXTB R0,R0
	; LCD_PrintInt8(i)
	BL
	LCD_PrintInt8
	
	ADDS R4,R4,#+1 ; i++
	B
	??WhileLoop_0
	
	??WhileLoop_1:
	POP {R4,PC}
		;; Fall through to label ?Subroutine0	BL
			LCD_PrintInt8
			
			ADDS R4,R4,#+1 ; i++
			UXTB R0,R4
			CMP R0,#+10 ; i = 10?
			BLT
			??WhileLoop_0
			
			POP {R4,PC} ; return

// 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)
	Conditional:
	PUSH {R3-R5,LR}
	MOVS R5,R0
	MOVS R4,R1
	
	MOVS R0,R4
	UXTB R0,R0
	CMP R0,#+1
	BNE
	??Conditional_0
	
	; LCD_PrintInt8(i);
	MOVS R0,R4
	UXTB R0,R0
	BL
	LCD_PrintInt8
	B
	??Conditional_1
	
	; LCD_PrintInt8(i + 1);
	??Conditional_0:
	ADDS R0,R4,#+1
	UXTB R0,R0
	BL
	LCD_PrintInt8
	
	??Conditional_1:
	POP {R0,R4,R5,PC}
		; prolog not shown
		; if(j == 1)
		CMP R1,#+1
		BEQ
		??Conditional_0
		ADDS R0,R0,#+1
		UXTB R0,R0
		
		??Conditional_0:
		BL
		LCD_PrintInt8
			; same as for size optimization

// Switch case statements
void SwitchCase(uint8 j)
{
  switch(j)
  {
  case 0:
  LCD_PrintInt8(1);
  break;
	; prolog not shown
	; switch(j)
	UXTB R0,R0
	CMP R0,#+0
	BEQ
	??SwitchCase_0
	CMP R0,#+1
	BEQ
	??SwitchCase_1
	B
	??SwitchCase_2
		; prolog not shown
		; switch(j)
		CMP R0,#+0
		BEQ
		??SwitchCase_0
		CMP R0,#+1
		BEQ
		??SwitchCase_1
		B
		??SwitchCase_2
			; same as for size optimization

Appendix: Compiler output details

C Code	IAR, Cortex®-M0/M0+ no optimization	IAR, Cortex®-M0/M0+, size optimization	IAR, Cortex®-M0/N0+, speed optimization

`    case 1:`
`    LCD_PrintInt8(2);`
`    break;`
 	
`    default:`
`    LCD_PrintInt8(0);`
`    break;`
`  }`
`}`
	
	`; case 0`
	`??SwitchCase_0:`
	`MOVS R0,#+1`
	`BL`
	`LCD_PrintInt8`
	`B`
	`??SwitchCase_3`
	
	`; case 1`
	`??SwitchCase_1:`
	`MOVS R0,#+2`
	`BL`
	`LCD_PrintInt8`
	`B`
	`??SwitchCase_3`
	
	`; default`
	`??SwitchCase_2:`
	`MOVS R0,#+0`
	`BL`
	`LCD_PrintInt8`
	`; no epilog`
		`; case 0`
		`??SwitchCase_0:`
		`MOVS R0,#+1`
		`B`
		`??SwitchCase_3`
		
		`; case 1`
		`??SwitchCase_1:`
		`MOVS R0,#+2`
		`B`
		`??SwitchCase_3`
		
		`; case 2`
		`??SwitchCase_2:`
		`MOVS R0,#+0`
		`; default`
		`??SwitchCase_3:`
		`BL LCD_PrintInt8`
		`; epilog not shown`	

`// Ternary operator`
`void Ternary(uint8 i)`
`{`
`  LCD_PrintInt8(`
`    (i == 1) ? 80 :`
`100);`
`}`
	`; prolog not shown`
	`; check value of i`
	`CMP R0,#+1`
	`; branch if not equal`
	`BNE ??Ternary_0`
	`MOVS R0,#+80`
	`B ??Ternary_1`
	
	`??Ternary_0:`
	`MOVS R0,#+100`
	`??Ternary_1:`
	`UXTB R0,R0`
	`BL`
	`LCD_PrintInt8`
	`; epilog not shown`
		`; prolog not shown`
		`; check value of i`
		`CMP R0, #1`
		`; branch if not equal`
		`BNE ??Ternary_0`
		`MOVS R0,#+80`
		`B ??Ternary_1`
		
		`??Ternary_0:`
		`MOVS R0,#+100`
		
		`??Ternary_1:`
		`BL`
		`LCD_PrintInt8`
			`; same as for size optimization`

`// Addition operation`
`int DoAdd(int x, int y)`
`{`
`  return x + y;`
`}`
	`; no prolog`
	`ADDS R0,R0,R1`
	`BX LR ; return value`
		`; same as for no optimization`	`; same as for size optimization`

`// Subtraction operation`
`int DoSub(int x, int y)`
`{`
`  return x - y;`
`}`
	`; no prolog`
	`SUBS R0,R0,R1`
	`BX LR ; return value`
		`; same as for no optimization`	`; same as for size optimization`

`// Multiplication`
`int DoMul(int x, int y)`
`{`
`  return x * y;`
`}`
	`; no prolog`
	`MULS R0,R1,R0`
	`BX LR ; return value`
		`; same as for no optimization`	`; same as for size optimization`

`// Division`
`int DoDiv(int x, int y)`
`{`
	`; prolog not shown`
	`; Uses clib helper div`
	`; function`
	`BL __aeabi_idiv`
		`; same as for no optimization`	`; same as for size optimization`

Appendix: Compiler output details

C Code	IAR, Cortex®-M0/M0+ no optimization	IAR, Cortex®-M0/M0+, size optimization	IAR, Cortex®-M0/N0+, speed optimization
`  return x / y;` `}`	`POP {PC} ;` `return value`		

`// Modulo operator`
`int DoMod(int x, int y)`
`{`
`  return x % y;`
`}`
	`; prolog not shown`
	`; Uses clib helper function`
	`BL`
	`__aeabi_idivmod`
	`MOVS R0,R1`
	`POP {PC} ;`
	`return value`
		`; same as for no optimization`	`; same as for size optimization`

`// Pointer`
`void Pointer(uint8 x, uint8 *ptr)`
`{`
`  *ptr = *ptr + x;`
`  ptr++;`
`  LCD_PrintInt8(*ptr);`
`}`
	`; *ptr = *ptr + x`
	`LDRB R0,[R1, #+0]`
	`ADDS R0,R0,R4`
	`STRB R0,[R1, #+0]`
	
	`; ptr++`
	`; LCD_PrintInt8(*ptr)`
	`ADDS R5,R1,#+1`
	`LDRB R0,[R5, #+0]`
	`BL`
	`LCD_PrintInt8`
		`; *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`
			`; same as for size optimization`

`// Function pointer`
`void FuncPtr(uint8 x, void *fptr(uint8))`
`{`
`  (*fptr)(x);`
`}`
	`; (*fptr)(x)`
	`PUSH {R3-R5,LR}`
	`MOVS R5,R0`
	`MOVS R4,R1`
	
	`MOVS R0,R5`
	`UXTB R0,R0`
	`; 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`
		`; (*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`

`// 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 =`
	`; myfoo.membera = 5`
	`LDR R4,??DataTable2`
	`MOVS R0,#+5`
	`STRB R0,[R4, #+0]`
	`; myfoo_p.memberb = 10;`
	`MOVS R0,#+10`
	`STRB R0,[R4, #+1]`
	`; myfoo_p.memberc = 15;`
	`ADDS R1,R4,#+2`
	`MOVS R0,#+15`
	`BL`
	`__aeabi_uwrite4`
	`; myfoo_p.memberd = 20;`
	`MOVS R0,#+20`
	`STRB R0,[R4, #+6]`
	`LSRS R0,R0,#+8`
	`STRB R0,[R4, #+7]`
	
	`POP {R4,PC} ;`
	`return value`
		`; myfoo.membera = 5`
		`LDR R4,??DataTable2`
		`MOVS R0,#+5`
		`STRB R0,[R4, #+0]`
		`; myfoo_p.memberb = 10;`
		`MOVS R0,#+10`
		`STRB R0,[R4, #+1]`
		`; myfoo_p.memberc = 15;`
		`ADDS R1,R4,#+2`
		`MOVS R0,#+15`
		`BL`
		`__aeabi_uwrite4`
		`; myfoo_p.memberd = 20;`
		`MOVS R0,#+20`
		`STRB R0,[R4, #+6]`
		`MOVS R0,#+0`
		`STRB R0,[R4, #+7]`
		
		`POP {R4,PC} ;`
		`return value`
			`; same as for size optimization`

Appendix: Compiler output details

C Code	IAR, Cortex®-M0/M0+ no optimization	IAR, Cortex®-M0/M0+, size optimization	IAR, Cortex®-M0/N0+, speed optimization
20; }			

// 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;
}
	; myfoo.membera = 5
	LDR
	R0,??DataTable2_1
	MOVS R1,#+5
	STRB R1,[R0, #+0]
	; myfoo.memberb = 10;
	MOVS R1,#+10
	STRB R1,[R0, #+1]
	; myfoo.memberc = 15;
	MOVS R1,#+15
	STR R1,[R0, #+4]
	; myfoo.memberd = 20;
	MOVS R1,#+20
	STRH R1,[R0, #+8]
	
	BX LR
		; same as for no optimization	; same as for size optimization

14.6.1 IAR for Cortex®-M0/M0+, none, size, and speed optimization

Table 36 shows, for the IAR compiler for the Cortex®-M0/M0+, examples of compiler output for different optimization options. The examples were extracted from the *.lst* files generated by the compiler.

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

Table 36 Compiler output details for IAR compiler for Cortex®-M0/M0+ CPU

C code	IAR, Cortex®-M0/M0+ no optimization	IAR, Cortex®-M0/M0+, size optimization	IAR, Cortex®-M0/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`
	`MOVS R0,#+128`
	`BL LCD_PrintInt8`
		`; same as for no optimization`	`; same as for size 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 size 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 {R4,LR}`
	`; R4 = i`
	`MOVS R4,#+0`
	`; do the function call with`
	`; i as the argument in R0`
	`??ForLoop_0:`
	`MOVS R0,R4`
	`UXTB R0,R0`
	`; check i not equal to 10`
	`CMP R0,#+10`
	`BGE ??ForLoop_1`
	
	`MOVS R0,R4`
	`UXTB R0,R0`
	`BL LCD_PrintInt8`
	
	`ADDS R4,R4,#+1 ; i++`
	`B ??ForLoop_0`
	
	`; function epilog`
	`??ForLoop_1:`
	`POP {R4,PC};`
	`return`
		`; function prolog`
		`PUSH {R4,LR}`
		
		`B.N ?Subroutine0`
		
		`?Subroutine0:`
		`MOVS R4,#+0`
		
		`?Subroutine0_0:`
		`UXTB R0,R4`
		`BL LCD_PrintInt8`
		`ADDS R4,R4,#+1 ; i++`
		`UXTB R0,R4`
		`CMP R0,#+10 ; i = 10?`
		`BLT ??Subroutine0_0`
		`POP {R4,PC} ; return`
			`; function prolog`
			`PUSH {R4,LR}`
			`MOVS R4,#+0`
			
			`UXTB R0,R4`
			`BL LCD_PrintInt8`
			
			`ADDS R4,R4,#+1 ; i++`
			`UXTB R0,R4`
			`CMP R0,#+10 ; i = 10?`
			`BLT ??ForLoop_0`
			
			`POP {R4,PC} ; return`

`// While loop`
`// i is type automatic, see`
`// Accessing Automatic Variables`
`// for details`
	`; prolog not shown`
	`; i = 0`
	`MOVS R4,#+0`
	
	`??WhileLoop_0:`
	`MOVS R0,R4`
		`; optimizes to match the for ; loop function`
		`REQUIRE ?Subroutine0`
			`; function prolog`
			`PUSH {R4,LR}`
			`MOVS R4,#+0`
			
			`??WhileLoop_0:`
			`UXTB R0,R4`

Appendix: Compiler output details

C code	IAR, Cortex®-M0/M0+ no optimization	IAR, Cortex®-M0/M0+, size optimization	IAR, Cortex®-M0/M0+, speed optimization

uint8 i = 0;
 	
while (i < 10)
{
  LCD_PrintInt8(i);
  i++;
}
	UXTB R0,R0
	; check i not equal to 10
	CMP R0,#+10
	BGE
	??WhileLoop_1
	
	LCD_PrintInt8:
	MOVS R0,R4
	UXTB R0,R0
	; LCD_PrintInt8(i)
	BL
	LCD_PrintInt8
	
	ADDS R4,R4,#+1 ; i++
	B
	??WhileLoop_0
	
	??WhileLoop_1:
	POP {R4,PC}
		;; Fall through to label ?Subroutine0	BL
			LCD_PrintInt8
			
			ADDS R4,R4,#+1 ; i++
			UXTB R0,R4
			CMP R0,#+10 ; i = 10?
			BLT
			??WhileLoop_0
			
			POP {R4,PC} ; return

// 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)
	Conditional:
	PUSH {R3-R5,LR}
	MOVS R5,R0
	MOVS R4,R1
	
	MOVS R0,R4
	UXTB R0,R0
	CMP R0,#+1
	BNE
	??Conditional_0
	
	; LCD_PrintInt8(i);
	MOVS R0,R4
	UXTB R0,R0
	BL
	LCD_PrintInt8
	B
	??Conditional_1
	
	; LCD_PrintInt8(i + 1);
	??Conditional_0:
	ADDS R0,R4,#+1
	UXTB R0,R0
	BL
	LCD_PrintInt8
	
	??Conditional_1:
	POP {R0,R4,R5,PC}
		; prolog not shown
		; if(j == 1)
		CMP R1,#+1
		BEQ
		??Conditional_0
		ADDS R0,R0,#+1
		UXTB R0,R0
		
		??Conditional_0:
		BL
		LCD_PrintInt8
			; same as for size optimization

// Switch case statements
void SwitchCase(uint8 j)
{
  switch(j)
  {
  case 0:
  LCD_PrintInt8(1);
  break;
	; prolog not shown
	; switch(j)
	UXTB R0,R0
	CMP R0,#+0
	BEQ
	??SwitchCase_0
	CMP R0,#+1
	BEQ
	??SwitchCase_1
	B
	??SwitchCase_2
		; prolog not shown
		; switch(j)
		CMP R0,#+0
		BEQ
		??SwitchCase_0
		CMP R0,#+1
		BEQ
		??SwitchCase_1
		B
		??SwitchCase_2
			; same as for size optimization

Appendix: Compiler output details

C code	IAR, Cortex®-M0/M0+ no optimization	IAR, Cortex®-M0/M0+, size optimization	IAR, Cortex®-M0/M0+, speed optimization

`    case 1:`
`    LCD_PrintInt8(2);`
`    break;`
 	
`    default:`
`    LCD_PrintInt8(0);`
`    break;`
`  }`
`}`
	
	`; case 0`
	`??SwitchCase_0:`
	`MOVS R0,#+1`
	`BL`
	`LCD_PrintInt8`
	`B`
	`??SwitchCase_3`
	
	`; case 1`
	`??SwitchCase_1:`
	`MOVS R0,#+2`
	`BL`
	`LCD_PrintInt8`
	`B`
	`??SwitchCase_3`
	
	`; default`
	`??SwitchCase_2:`
	`MOVS R0,#+0`
	`BL`
	`LCD_PrintInt8`
	`; no epilog`
		`; case 0`
		`??SwitchCase_0:`
		`MOVS R0,#+1`
		`B`
		`??SwitchCase_3`
		
		`; case 1`
		`??SwitchCase_1:`
		`MOVS R0,#+2`
		`B`
		`??SwitchCase_3`
		
		`; case 2`
		`??SwitchCase_2:`
		`MOVS R0,#+0`
		`; default`
		`??SwitchCase_3:`
		`BL LCD_PrintInt8`
		`; epilog not shown`	

`// Ternary operator`
`void Ternary(uint8 i)`
`{`
`  LCD_PrintInt8(`
`    (i == 1) ? 80 :`
`100);`
`}`
	`; prolog not shown`
	`; check value of i`
	`CMP R0,#+1`
	`; branch if not equal`
	`BNE ??Ternary_0`
	`MOVS R0,#+80`
	`B ??Ternary_1`
	
	`??Ternary_0:`
	`MOVS R0,#+100`
	`??Ternary_1:`
	`UXTB R0,R0`
	`BL`
	`LCD_PrintInt8`
	`; epilog not shown`
		`; prolog not shown`
		`; check value of i`
		`CMP R0, #1`
		`; branch if not equal`
		`BNE ??Ternary_0`
		`MOVS R0,#+80`
		`B ??Ternary_1`
		
		`??Ternary_0:`
		`MOVS R0,#+100`
		
		`??Ternary_1:`
		`BL`
		`LCD_PrintInt8`
			`; same as for size optimization`

`// Addition operation`
`int DoAdd(int x, int y)`
`{`
`  return x + y;`
`}`
	`; no prolog`
	`ADDS R0,R0,R1`
	`BX LR ; return value`
		`; same as for no optimization`	`; same as for size optimization`

`// Subtraction operation`
`int DoSub(int x, int y)`
`{`
`  return x - y;`
`}`
	`; no prolog`
	`SUBS R0,R0,R1`
	`BX LR ; return value`
		`; same as for no optimization`	`; same as for size optimization`

`// Multiplication`
`int DoMul(int x, int y)`
`{`
`  return x * y;`
`}`
	`; no prolog`
	`MULS R0,R1,R0`
	`BX LR ; return value`
		`; same as for no optimization`	`; same as for size optimization`

`// Division`
`int DoDiv(int x, int y)`
`{`
	`; prolog not shown`
	`; Uses clib helper div`
	`; function`
	`BL __aeabi_idiv`
		`; same as for no optimization`	`; same as for size optimization`

Appendix: Compiler output details

C code	IAR, Cortex®-M0/M0+ no optimization	IAR, Cortex®-M0/M0+, size optimization	IAR, Cortex®-M0/M0+, speed optimization
`  return x / y;` `}`	`POP {PC} ;` `return value`		

`// Modulo operator`
`int DoMod(int x, int y)`
`{`
`  return x % y;`
`}`
	`; prolog not shown`
	`; Uses clib helper function`
	`BL`
	`__aeabi_idivmod`
	`MOVS R0,R1`
	`POP {PC} ;`
	`return value`
		`; same as for no optimization`	`; same as for size optimization`

`// Pointer`
`void Pointer(uint8 x, uint8 *ptr)`
`{`
`  *ptr = *ptr + x;`
`  ptr++;`
`  LCD_PrintInt8(*ptr);`
`}`
	`; *ptr = *ptr + x`
	`LDRB R0,[R1, #+0]`
	`ADDS R0,R0,R4`
	`STRB R0,[R1, #+0]`
	
	`; ptr++`
	`; LCD_PrintInt8(*ptr)`
	`ADDS R5,R1,#+1`
	`LDRB R0,[R5, #+0]`
	`BL`
	`LCD_PrintInt8`
		`; *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`
			`; same as for size optimization`

`// Function pointer`
`void FuncPtr(uint8 x, void *fptr(uint8))`
`{`
`  (*fptr)(x);`
`}`
	`; (*fptr)(x)`
	`PUSH {R3-R5,LR}`
	`MOVS R5,R0`
	`MOVS R4,R1`
	
	`MOVS R0,R5`
	`UXTB R0,R0`
	`; 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`
		`; (*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`

`// 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 =`
	`; myfoo.membera = 5`
	`LDR R4,??DataTable2`
	`MOVS R0,#+5`
	`STRB R0,[R4, #+0]`
	`; myfoo_p.memberb = 10;`
	`MOVS R0,#+10`
	`STRB R0,[R4, #+1]`
	`; myfoo_p.memberc = 15;`
	`ADDS R1,R4,#+2`
	`MOVS R0,#+15`
	`BL`
	`__aeabi_uwrite4`
	`; myfoo_p.memberd = 20;`
	`MOVS R0,#+20`
	`STRB R0,[R4, #+6]`
	`LSRS R0,R0,#+8`
	`STRB R0,[R4, #+7]`
	
	`POP {R4,PC} ;`
	`return value`
		`; myfoo.membera = 5`
		`LDR R4,??DataTable2`
		`MOVS R0,#+5`
		`STRB R0,[R4, #+0]`
		`; myfoo_p.memberb = 10;`
		`MOVS R0,#+10`
		`STRB R0,[R4, #+1]`
		`; myfoo_p.memberc = 15;`
		`ADDS R1,R4,#+2`
		`MOVS R0,#+15`
		`BL`
		`__aeabi_uwrite4`
		`; myfoo_p.memberd = 20;`
		`MOVS R0,#+20`
		`STRB R0,[R4, #+6]`
		`MOVS R0,#+0`
		`STRB R0,[R4, #+7]`
		
		`POP {R4,PC} ;`
		`return value`
			`; same as for size optimization`

Appendix: Compiler output details

C code	IAR, Cortex®-M0/M0+ no optimization	IAR, Cortex®-M0/M0+, size optimization	IAR, Cortex®-M0/M0+, speed optimization
20; }			

// 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;
}
	; myfoo.membera = 5
	LDR
	R0,??DataTable2_1
	MOVS R1,#+5
	STRB R1,[R0, #+0]
	; myfoo.memberb = 10;
	MOVS R1,#+10
	STRB R1,[R0, #+1]
	; myfoo.memberc = 15;
	MOVS R1,#+15
	STR R1,[R0, #+4]
	; myfoo.memberd = 20;
	MOVS R1,#+20
	STRH R1,[R0, #+8]
	
	BX LR
		; same as for no optimization	; same as for size optimization

14.6.2 IAR for Cortex®-M0/M0+, low, medium, and balanced optimization

Table 37 shows, for the IAR compiler for the Cortex®-M0/M0+, examples of compiler output for different optimization options. The examples were extracted from the *.lst* files generated by the compiler.

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

Table 37 Compiler output details for IAR compiler for Cortex®-M3 CPU

C code	IAR, Cortex®-M0/M0+ low optimization	IAR, Cortex®-M0/M0+, medium optimization	IAR, Cortex®-M0/M0+, balanced optimization

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

`// Calling a function with`
`// one argument`
`LCD_PrintInt8(128);`
	`; R0 = first argument`
	`; conditional flags are NOT`
	`; updated by mov`
	`MOVS R0,#+128`
	`BL LCD_PrintInt8`
		`; same as for low`
		`; optimization`
			`; same as for low`
			`; 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 low`
		`; optimization`
			`; same as for low`
			`; 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 {R4,LR}`
	`; R4 = i`
	`MOVS R4,#+0`
	`B ??ForLoop_0`
	
	`??ForLoop_1:`
	`; do the function call with`
	`; i as the argument in R0`
	`MOVS R0,R4`
	`UXTB R0,R0`
	`BL LCD_PrintInt8`
	
	`ADDS R4,R4,#+1 ; i++`
	`B ??ForLoop_0`
	
	`??ForLoop_0:`
	`MOVS R0,R4`
	`UXTB R0,R0`
	`; check i not equal to 10`
	`CMP R0,#+10`
	`BLT ??ForLoop_1`
	
	`; function epilog`
	`POP {R4,PC};`
	`return`
		`; function prolog`
		`MOVS R4,#+0`
		`B ??ForLoop_0`
		
		`LCD_PrintInt8(i);`
		`??ForLoop_1: (+1)`
		`MOVS R0,R4`
		`UXTB R0,R0`
		`BL LCD_PrintInt8`
		
		`ADDS R4,R4,#+1`
		`??ForLoop_0:`
		`MOVS R0,R4`
		`UXTB R0,R0`
		`CMP R0,#+10`
		`BLT ??ForLoop_1`
		
		`; function epilog`
		`POP {R4,PC};`
		`return`
			`; function prolog`
			`PUSH {R4,LR}`
			
			`B.N ?Subroutine0`
			
			`?Subroutine0:`
			`MOVS R4,#+0`
			
			`?Subroutine0_0:`
			`UXTB R0,R4`
			`BL LCD_PrintInt8`
			`ADDS R4,R4,#+1 ; i++`
			`UXTB R0,R4`
			`CMP R0,#+10 ; i = 10?`
			`BLT ??Subroutine0_0`
			`POP {R4,PC} ; return`

`// While loop`
`// i is type automatic, see`
`// Accessing Automatic`
	`; prolog not shown`
	`; i = 0`
	`??WhileLoop_1:`
	`MOVS R0,R4`
		`; prolog`
		`WhileLoop:`
		`PUSH {R4,LR}`
		`MOVS R4,#+0`
			`; optimizes to match the for ; loop function`
			`REQUIRE ?Subroutine0`

Appendix: Compiler output details

C code	IAR, Cortex®-M0/M0+ low optimization	IAR, Cortex®-M0/M0+, medium optimization	IAR, Cortex®-M0/M0+, balanced optimization

Variables
// for details
uint8 i = 0;
 	
while (i < 10)
{
  LCD_PrintInt8(i);
  i++;
}
	UXTB R0,R0
	BL
	LCD_PrintInt8
	
	ADDS R4,R4,#+1
	
	??WhileLoop_0:
	MOVS R0,R4
	UXTB R0,R0
	; check i not equal to 10
	CMP R0,#+10
	BLT
	??WhileLoop_1
	
	; function epilog
	POP {R4,PC};
	return
		B
		??WhileLoop_0
		
		LCD_PrintInt8(i);
		??WhileLoop_1:
		MOVS R0,R4
		UXTB R0,R0
		BL
		LCD_PrintInt8
		ADDS R4,R4,#+1
		
		??WhileLoop_0:
		MOVS R0,R4
		UXTB R0,R0
		CMP R0,#+10
		BLT
		??WhileLoop_1
		
		; function epilog
		POP {R4,PC};
		return
			;; Fall through to label ?Subroutine0

// Conditional statement
void Conditional(uint8 i, uint8 j)
{
  if(j == 1)
  {
  LCD_PrintInt8(i);
  }
  else
  {
  LCD_PrintInt8(i + 1);
  }
}
	PUSH {R7,LR}
	; if(j == 1)
	UXTB R1,R1
	CMP R1,#+1
	BNE
	??Conditional_0
	; LCD_PrintInt8(i);
	UXTB R0,R0
	BL
	LCD_PrintInt8
	B
	??Conditional_1
	; LCD_PrintInt8(i + 1);
	Conditional_0:
	ADDS R0,R0,#+1
	UXTB R0,R0
	BL
	LCD_PrintInt8
	
	??Conditional_1:
	POP {R0,PC};
	return
		Conditional:
		PUSH {R7,LR}
		CMP R1,#+1
		BNE
		??Conditional_0
		
		LCD_PrintInt8(i);
		BL
		LCD_PrintInt8
		POP {R0,PC}
		
		LCD_PrintInt8(i + 1);
		??Conditional_0:
		ADDS R0,R0,#+1
		UXTB R0,R0
		BL
		LCD_PrintInt8
		
		POP {R0,PC} ;
		return
			; prolog not shown
			; if(j == 1)
			CMP R1,#+1
			BEQ
			??Conditional_0
			ADDS R0,R0,#+1
			UXTB R0,R0
			
			??Conditional_0:
			BL
			LCD_PrintInt8

// 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);
	; prolog not shown
	; switch(j)
	UXTB R0,R0
	CMP R0,#+0
	BEQ
	??SwitchCase_0
	CMP R0,#+1
	BEQ
	??SwitchCase_1
	B
	??SwitchCase_2
	; case 0
	??SwitchCase_0:
	MOVS R0,#+1
	BL
	LCD_PrintInt8
		SwitchCase:
		PUSH {R7,LR}
		CMP R0,#+0
		BEQ
		??SwitchCase_0
		CMP R0,#+1
		BEQ
		??SwitchCase_1
		B
		??SwitchCase_2
		
		case 0:
		LCD_PrintInt8(1);
		??SwitchCase_0:
		MOVS R0,#+1
		BL
		LCD_PrintInt8
		POP {R0,PC}
			; prolog not shown
			; switch(j)
			CMP R0,#+0
			BEQ
			??SwitchCase_0
			CMP R0,#+1
			BEQ
			??SwitchCase_1
			B
			??SwitchCase_2
			
			; case 0
			??SwitchCase_0:
			MOVS R0,#+1
			B
			??SwitchCase_3
			
			; case 1

Appendix: Compiler output details

C code	IAR, Cortex®-M0/M0+ low optimization	IAR, Cortex®-M0/M0+, medium optimization	IAR, Cortex®-M0/M0+, balanced optimization

break;
}
}
	B
	??SwitchCase_3
	; case 1
	??SwitchCase_1:
	MOVS R0,#+2
	BL
	LCD_PrintInt8
	B
	??SwitchCase_3
	
	; default
	??SwitchCase_2:
	MOVS R0,#+0
	BL
	LCD_PrintInt8
	; epilog not shown
		
		case 1:
		LCD_PrintInt8(2);
		??SwitchCase_1:
		MOVS R0,#+2
		BL
		LCD_PrintInt8
		POP {R0,PC}
		
		default:
		LCD_PrintInt8(0);
		??SwitchCase_2:
		MOVS R0,#+0
		BL
		LCD_PrintInt8
		
		POP {R0,PC};
		return
			??SwitchCase_1:
			MOVS R0,#+2
			B
			??SwitchCase_3
			
			; case 2
			??SwitchCase_2:
			MOVS R0,#+0
			; default
			??SwitchCase_3:
			BL LCD_PrintInt8
			; epilog not shown

// Ternary operator
void Ternary(uint8 i)
{
LCD_PrintInt8(
(i == 1) ? 80 :
100);
}
	; prolog not shown
	; check value of i
	CMP R0,#+1
	; branch if not equal
	BNE ??Ternary_0
	MOVS R0,#+80
	B ??Ternary_1
	
	??Ternary_0:
	MOVS R0,#+100
	??Ternary_1:
	UXTB R0,R0
	BL
	LCD_PrintInt8
	; no epilog
		; same as for low
		; optimization
			; prolog not shown
			; check value of i
			CMP R0, #1
			; branch if not equal
			BNE ??Ternary_0
			MOVS R0,#+80
			B ??Ternary_1
			
			??Ternary_0:
			MOVS R0,#+100
			
			??Ternary_1:
			BL
			LCD_PrintInt8

// Addition operation
int DoAdd(int x, int
y)
{
return x + y;
}
	; no prolog
	ADDS R0,R1,R0
	BX LR ; return
	value
		; same as for low
		; optimization
			; same as for low
			; optimization

// Subtraction
operation
int DoSub(int x, int
y)
{
return x - y;
}
	; no prolog
	SUBS R0,R0,R1
	BX LR ; return
	value
		; same as for low
		; optimization
			; same as for low
			; optimization

// Multiplication
int DoMul(int x, int
y)
{
return x * y;
}
	; no prolog
	MULS R0,R0,R1
	BX LR ;
	return value
		; same as for low
		; optimization
			; same as for low
			; optimization

// Division
int DoDiv(int x, int
y)
{
return x / y;
}
	; no prolog
	; call into c helper
	; function
	BL __aeabi_idiv
		; same as for low
		; optimization
			; same as for low
			; optimization

// Modulo operator int DoMod(int x, int y)	; no prolog ; call into c helper ; function	; same as for low ; optimization	; same as for low ; optimization

Appendix: Compiler output details

C code	IAR, Cortex®-M0/M0+ low optimization	IAR, Cortex®-M0/M0+, medium optimization	IAR, Cortex®-M0/M0+, balanced optimization

{
  return x % y;
}
	BL
	__aeabi_idivmod
	MOVS R0,R1		

// Pointer
void Pointer(uint8 x,
uint8 *ptr)
{
  *ptr = *ptr + x;
  ptr++;
  LCD_PrintInt8(*ptr);
}
	; *ptr = *ptr + x
	LDRB R2,[R1, #+0]
	ADDS R0,R2,R0
	STRB R0,[R1, #+0]
	
	; ptr++
	; LCD_PrintInt8(*ptr)
	ADDS R0,R1,#+1
	LDRB R0,[R0, #+0]
	BL
	LCD_PrintInt8
		; *ptr = *ptr + x
		LDRB R2,[R1, #+0]
		ADDS R0,R2,R0
		STRB R0,[R1, #+0]
		
		; ptr++
		; LCD_PrintInt8(*ptr)
		LDRB R0,[R1, #+1]
		BL
		LCD_PrintInt8
			; same as for medium optimization

// Function pointer
void FuncPtr(uint8 x,
void
*fptr(uint8))
{
  (*fptr)(x);
}
	; (*fptr)(x)
	 UXTB R0,R0
	; 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
		; (*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 medium 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;
}
	; myfoo.membera = 5
	LDR
	R4,??DataTable2
	MOVS R0,#+5
	STRB R0,[R4, #+0]
	; myfoo_p.memberb =
	10;
	MOVS R0,#+10
	STRB R0,[R4, #+1]
	; myfoo_p.memberc =
	15;
	ADDS R1,R4,#+2
	MOVS R0,#+15
	BL
	__aeabi_uwrite4;
	myfoo_p.memberd = 20;
	MOVS R0,#+20
	STRB R0,[R4, #+6]
	LSRS R0,R0,#+8
	STRB R0,[R4, #+7]
	
	POP {R4,PC}
	;return
		; same as for low
		; optimization
			; myfoo.membera = 5
			LDR
			R4,??DataTable2
			MOVS R0,#+5
			STRB R0,[R4, #+0]
			; myfoo_p.memberb =
			10;
			MOVS R0,#+10
			STRB R0,[R4, #+1]
			; myfoo_p.memberc =
			15;
			ADDS R1,R4,#+2
			MOVS R0,#+15
			BL
			__aeabi_uwrite4;
			myfoo_p.memberd = 20;
			MOVS R0,#+20
			STRB R0,[R4, #+6]
			MOVS R0,#+0
			STRB R0,[R4, #+7]
			
			POP {R4,PC}
			;return

// unpacked structures
struct FOO
{
  uint8 membera;
  uint8 memberb;
  uint32 memberc;
  uint16 memberd;
};
 	
extern struct FOO
myfoo;
	; myfoo.membera = 5
	LDR
	R0,??DataTable2_1
	MOVS R1,#+5
	STRB R1,[R0,
	#+0]; myfoo.memberb =
	10;
	MOVS R1,#+10
	STRB R1,[R0,
	#+1]; myfoo.memberc =
	15;
	MOVS R1,#+15
		; same as for low
		; optimization
			; same as for low optimization

Appendix: Compiler output details

C code	IAR, Cortex®-M0/M0+ low optimization	IAR, Cortex®-M0/M0+, medium optimization	IAR, Cortex®-M0/M0+, balanced optimization

`void PackedStruct(void) { myfoo.membera = 5; myfoo.memberb = 10; myfoo.memberc = 15; myfoo.memberd = 20; }`	`STR R1,[R0, #+4] ; myfoo.memberd = 20; MOVS R1,#+20 STRH R1,[R0, #+8]`
	
	`BX LR ;return`		

14.7 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, MDK, and IAR 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));
```

Appendix: Compiler output details

```
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() */
```

Appendix: Compiler output details

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++;
    }
}
```

Appendix: Compiler output details

```
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);
}
```

Appendix: Compiler output details

```
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;
```

Appendix: Compiler output details

```
    myfoo_p.memberd = 20;
}

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

References

[1] Application notes

- AN77759 – Getting Started with PSoC™ 5LP
- AN79953 – Getting Started with PSoC™ 4 MCU
- 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

[2] C documentation

- GCC documentation can be found in your PSoC™ Creator installation folder.
  Compiler documentation:
  C:\Program Files\Cypress\PSoC Creator\4.3\PSoC Creator\import\gnu_cs\arm\5.4.1\share\doc\gcc-arm-none-eabi\pdf\gcc\gcc.pdf
  Linker script file documentation:
  C:\Program Files\Cypress\PSoC Creator\4.3\PSoC Creator\import\gnu_cs\arm\5.4.1\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*.
- For IAR, the documentation can be found in your IAR installation folder, typically: *C:\Program Files (x86)\IAR Systems\Embedded Workbench 8.4\arm\doc*. Compiler and linker script information can be found in *EWARM_DevelopmentGuide.ENU.pdf*.

[3] Arm® Cortex® documentation

Arm® provides on their website a wealth of information about Cortex®-M3 and Cortex®-M0/M0+ CPUs:

- Cortex®-M0 Instruction Set
- Cortex®-M0+ Instruction Set
- Cortex®-M3 Instruction Set
- Cortex® Microcontroller Software Interface Standard (CMSIS) library
- Arm® related books

Revision history

Document revision	Date	Description of changes
**	2014-02-07	New application note.
*A	2015-10-29	Clarified that PSoC™ 5LP cannot execute code from an 8-bit EMIF memory. Updated Related Documents: Updated Application Notes: Added “AN90799, PSoC™ 4 Interrupts” in the list.
*B	2017-02-06	Corrected a missing comma in a code snippet, that may cause a compile error. Updated to new template. Completing Sunset Review.
*C	2017-04-18	Updated Cypress Logo and Copyright.
*D	2017-05-25	Added PSoC™ Analog Coprocessor in Associated Part Family in page 1. Added PSoC™ Analog Coprocessor related information in all instances across the document. Added PSoC™ Analog Coprocessor/PSoC™ 4000S/4100S related information in “Register Set”.
*E	2018-08-14	Removed PSoC™ Analog Coprocessor in Associated Part Family in page 1. Added PSoC™ 4100PS related information in all instances across the document. Updated to new template.
*F	2020-09-09	Added ModusToolbox™ software 2.1 in Software Version in page 1. Added ModusToolbox™ related information in all instances across the document. Updated Compiler Output Details: Added “Assembler Examples, IAR for Cortex-M3”. Added “Assembler Examples, IAR for Cortex-M0/M0+”.
*G	2023-10-11	Updated ModusToolbox™ software 2.1 related information to ModusToolbox™ software 3.1 in all instances across the document. Added tips and tricks to optimize the PSoC 4 device code. Updated Modify the Linker Script File: Migrated to Infineon template.
*H	2024-02-21	Fixed the broken links.

Trademarks
All referenced product or service names and trademarks are the property of their respective owners.

Edition 2024-02-21

Published by

Infineon Technologies AG

81726 Munich, Germany

Do you have a question about this document?

Email: erratum@infineon.com

Document reference

001-89610 Rev. *H

Important notice

The information contained in this application note is given as a hint for the implementation of the product only and shall in no event be regarded as a description or warranty of a certain functionality, condition or quality of the product. Before implementation of the product, the recipient of this application note must verify any function and other technical information given herein in the real application. Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind (including without limitation warranties of non-infringement of intellectual property rights of any third party) with respect to any and all information given in this application note.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer's technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

Warnings

Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies' products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.