Getting Started in C and Assembly Code With the TMS320LF240x DSP

David M. Alter — *DSP Applications – Semiconductor Group*

ABSTRACT

This application report presents basic code for initializing and operating the TMS320LF240x† DSP devices. Two functionally equivalent example programs are presented: one written in assembly language and the other in C language. Detailed discussions of each program are provided that explain numerous compiler and assembler directives, code requirements, and hardware-related requirements. The programs are ready to run on either the TMS320LF2407 Evaluation Module (EVM) or the eZdsp™ LF2407 development kit. However, they are also intended for use as a code template for any TMS320LF240x† (LF240x) or TMS320LF240xA† (LF240xA) DSP target system. The sample code described in this application report can be downloaded from http://www.ti.com/lit/zip/SPRA755.

Contents

List of Figures

† Go to http://www.ti.com for a complete list of TMS320C24x™ devices.

1 Introduction

A hurdle often faced by code developers working with an unfamiliar DSP is knowing how to get started. A good example program that illustrates proper DSP initialization, interrupt handling, and peripheral operation can be invaluable. This application report presents basic code for initializing and operating the TMS320LF240x devices. Although the code herein is designed to run on either the TMS320LF2407 Evaluation Module (EVM) or the eZdsp™ LF2407 development kit, it is also applicable, and intended for use, as a code template for any TMS320LF240x (LF240x) or TMS320LF240xA (LF240xA) DSP target system.

This application report presents two functionally equivalent example programs: one written in assembly language, and the other written in C programming language. A reader who possesses an understanding of one of these two languages can compare the two programs to understand the coding in the other language. The philosophy behind the code examples is to keep them as simple and clear as possible. Therefore, only a few of the numerous peripherals on the LF2407 DSP are exercised. The assumption is that once the basic coding techniques for configuring a peripheral register are illustrated, the reader can easily duplicate the code to configure any other peripheral.

In addition to the example programs, two complete header files are provided that define the addresses of all user-programmable registers on the LF2407 DSP: one for assembly code and the other for C code. Finally, a Code Composer™ v4.1x gel file is provided (along with usage instructions) that conveniently places all LF2407 peripheral registers into a Code Composer menu for easy inclusion into a debugger watch window.

2 Hardware Setup

The example programs in this application note are designed to run out-of-the-box on either the LF2407 EVM or eZdsp™ LF2407. If you do not have one of these boards, all of the benefits of the example programs are still available to you. You simply may not be able to run the programs without making modifications to them for your particular target board.

2.1 LF2407 EVM Setup

The example programs are ready to run on the LF2407 EVM board without any modification. It is assumed that the reader already has the EVM board, an emulator (e.g., XDS510PP-plus from Spectrum Digital Inc. or XDS510) and a debugger (e.g., Code Composer v4.10 or v4.12) installed on a PC and ready to go. It is beyond the scope of this application report to provide direction on tools installation. Please see the installation instructions that came with the tools for assistance with this.

Numerous jumpers exist on the EVM board. The following are jumper setting requirements for proper operation of the example programs:

- JP5: This jumper controls the voltage level of the VCCP pin on the DSP. Normally, leave it in the 1–2 position. If you want to program the FLASH with the example program, set to the 2–3 position during FLASH programming. It is good practice to return it to the 1–2 position after FLASH programming.

Code Composer is a trademark of Texas Instruments.

- JP6: This jumper controls the voltage level of the MP/$\overline{MC}$ pin on the DSP. Normally, set to the 1–2 position to use the external RAM in place of the internal FLASH. Set to the 2–3 position if you want to program the FLASH with the example program, and leave in the 2–3 position to run the program out of the FLASH after it is programmed.
- JP13: Set to the 1–2 position. This uses the clock oscillator on the EVM board.
- JP16: Set to the 1–2 position. This disables bootloading.

All other jumper settings are don't cares for the example programs, since those jumpers affect peripherals that are not used here (e.g., CAN, SPI, ADC, or SCI). Additional information on these jumpers can be found in reference [7].

The LF2407 EVM is assumed to contain a 7.3728 MHz on-board oscillator. The example programs configure the DSP phase-locked loop (PLL) for multiply-by-four mode, which gives a CPU clock of 29.49 MHz (i.e., 30 MHz). The values stated in this report for the PWM carrier frequency, GPIO toggle period, and LED bank update rate have been computed assuming a 30 MHz CPU clock. If a different oscillator value is used, these values will be different. In addition, one must be careful not to exceed the rated clock speed of the DSP in use (30 MHz for current LF240x devices or 40 MHz for current LF240xA devices). Some early LF2407 EVM boards used 15 MHz oscillators that exceed the 30 MHz DSP rating if multiplied by four with the PLL. All you need to do in this situation is edit the example programs so that the PLL is configured for multiply-by-two mode instead. This setting is selected by bits 11–9 in the SCSR1 register. These bits are currently written as 000b. Change these to 001b to get the multiply-by-two PLL (see discussions for program lines 114 and 572 in sections 5 and 6, respectively, of this document). Also, some of the newest EVMs may have been fitted with a 40 MHz LF2407A DSP and use a 10 MHz oscillator. No program change is necessary here, although again timing values for the PWM, GPIO toggle, etc., will differ from those stated in this report (i.e., they will be 33% faster). For more information on the PLL and the SCSR1 register, see reference [2], pages 2–3 to 2–5.

2.2 eZdsp™ LF2407 Setup

The example programs are ready to run on the eZdsp™ LF2407 board (henceforth referred to as eZdsp™) without any modification. It is assumed that the reader already has the eZdsp™ and its debugger software installed and ready to go. It is beyond the scope of this application report to provide direction on tools installation. Please see the installation guide that came with the tools for assistance with this.

Several jumpers exist on the eZdsp™ board. The following are jumper setting requirements for proper operation of the example programs:

- JP3: This jumper controls the voltage level of the VCCP pin on the DSP. Normally, leave it in the 1–2 position. If you want to program the FLASH with the example program, set to the 2–3 position during FLASH programming. It is good practice to return it to the 1–2 position after FLASH programming.
- JP4: This jumper controls the voltage level of the MP/$\overline{MC}$ pin on the DSP. Normally, set to the 1–2 position to use the external RAM in place of the internal FLASH. Set to the 2–3 position if you want to program the FLASH with the example program, and leave in the 2–3 position to run the program out of the FLASH after it is programmed.

All other jumper settings are don't cares for the example programs, since those jumpers affect the ADC peripheral and the ADC is not used by the example programs. Additional information on these jumpers can be found in reference [8].

The eZdsp™ is assumed to contain a 14.7 MHz on-board oscillator. The on-board FPGA then implements a divide-by-two, such that a 7.35 MHz clock signal is ultimately fed into the LF2407 DSP. The example programs configure the DSP phase-locked loop (PLL) for multiply-by-four mode, which gives a CPU clock of 29.4 MHz (i.e., 30 MHz). The values stated in this report for the PWM carrier frequency, GPIO toggle period, and LED bank update rate have been computed assuming a 30 MHz CPU clock. If a different oscillator value is used, these values will be different. In addition, you must be careful not to exceed the rated clock speed of the DSP in use (30 MHz for current LF240x devices, or 40 MHz for current LF240xA devices). For example, if you are using a 30 MHz oscillator instead of the standard 14.7 MHz one, using the multiply-by-four PLL mode would exceed the 30 MHz frequency rating of the LF2407 DSP. All you need to do in this situation is edit the example programs so that the PLL is configured for multiply-by-two mode instead. This setting is selected by bits 11–9 in the SCSR1 register, and these bits are currently written as 000b. Change these to 001b to get the multiply-by-two PLL (see discussions for program lines 114 and 572 in sections 5 and 6, respectively, of this document). For more information on the PLL and the SCSR1 register, see reference [2], pages 2–3 to 2–5.

3 Using the Code Composer GEL File

The file *lf2407.gel* is provided with this application report download. This is a script file containing startup instructions for TMS320C2xx Code Composer v4.1x. Specifically, it has been tested with v4.10 and v4.12 of this debugger. When installed with Code Composer, it causes all LF2407 DSP peripheral registers to be listed on the GEL menu inside Code Composer. When a particular register is selected on this menu (e.g., by clicking on it with your PC mouse), that register is added to the WATCH WINDOW inside Code Composer. This functionality is a useful aid during code debug, and avoids having to fumble through the user's guide to look up register addresses. Note that the peripheral set of the LF2407 DSP is a superset of all other current 240x and 240xA devices. Therefore, this gel file is usable with any 240x or 240xA target.

To use the gel file with Code Composer, place the file in the C:\tic2xx\cc\gel directory on your PC (assuming you used the default installation paths when you installed Code Composer). Then, right-click on the shortcut (or Windows 95/98 Start menu item) that you use to invoke Code Composer. Select Properties, and then click on the Short Cut menu. The Target: box on this menu should show the following (again assuming default installation paths):

C:\tic2xx\cc\bin\cc_app.exe C:\tic2xx\cc\gel\init.gel

To use the lf2407.gel file, simply change the Target: box to read:

C:\tic2xx\cc\bin\cc_app.exe C:\tic2xx\cc\gel\lf2407.gel

4 Overview of the Example Programs

There are two functionally identical programs provided. The first is written in TMS320C2xx DSP assembly language, while the second is written in C programming language. Each example program has been tested on both the LF2407 EVM and the eZdsp™ LF2407. Code generation was performed using the TMS320C2x/C2xx/C5x Code Generation Tools PC v7.00, with C2xx Code Composer Software Package PC v4.12 as the debugger. The C example program was tested with compiler optimization disabled, and also with level 3 (–o3) optimization.

Each example program performs the following tasks on the DSP:

- Configures the two System Control and Status registers
- Disables the Watchdog Timer
- Configures the external memory interface
- Configures the shared device pins
- Configures GP Timer 1 and the compare units to provide 20KHz, 25% duty cycle symmetric PWM on the PWM1 pin. This can be observed by connecting an oscilloscope to the PWM1/IOPA6 signal located on the EVM and eZdsp™ signal headers.
- Configures GP Timer 2 to provide a 250ms interrupt
- Configures the core and event manager interrupt registers, and enables global interrupts
- In the GP Timer 2 interrupt service routine, the following is performed:
  - Context save/restore using a software stack
  - The quad LED bank on the EVM is sequenced (250 ms update rate). This is visible only on the EVM, since the eZdsp™ does not have the quad LED bank. However, no modifications to the program are necessary for operation on the eZdsp™. The LED update instructions produce a benign write to I/O space on the external memory interface of the DSP when using the eZdsp™.
  - The IOPC0 pin is toggled, producing a 2 Hz (500ms) square wave. On the EVM, the IOPC0 pin is connected to the red LED labeled DS1 on the board. On the eZdsp™, the IOPC0 pin is connected to the red LED labeled DS2 on the board. Users can visually see the pin toggle by observing the LED. Alternately, an oscilloscope can be connected to the W/$\overline{R}$/IOPC0 pin on the EVM and eZdsp™ signal headers.

In addition, the example programs illustrate the following concepts:

- How to work with multiple files
- How to work with multiple code sections
- How to write an interrupt service routine using either C or assembly language
- How to insert inline assembly code into a C language program
- How to access the I/O space using either C or assembly language
- How to access peripheral registers using either C or assembly language
- How to use include files
- Complete core interrupt vector tables for C and assembly language programs
- Example LF2407 DSP linker command files for both C and assembly language programs

Finally, note that besides running from external memory, each example program can be programmed into the internal FLASH memory on the LF2407 DSP and run out of the FLASH on either the LF2407 EVM or the eZdsp™ LF2407, without the need for any file modifications.

5 Assembly Language Example Program

The assembly language example program consists of the following files:

- **vectors.asm:** Interrupt vector table
- **example.asm:** Main program
- **f2407.h:** Header file containing peripheral register address definitions
- **example.cmd:** Linker command file
- **example.mak:** C2xx Code Composer v4.1x project file
- **example.out:** Program executable
- **example.map:** Memory map file output by the code generation tools

Only the first five listed files are needed to build the program. The final two files are provided for convenience, and are produced by the code generation tools when the program is built. To build the project using Code Composer, load the project file *example.mak* into Code Composer (click on PROJECT–>OPEN), and then select PROJECT–>BUILD or PROJECT–>REBUILD ALL. See the online help available within Code Composer for additional assistance.

Figure 1 illustrates the code flow for the assembly language example program. Execution begins at the reset vector in the file *vectors.asm* after a hardware reset. This vector branches to the start of the main program in the file *example.asm*. Processor initialization is then performed, after which an endless main loop is entered. The DSP is periodically interrupted out of this endless loop by the timer 2 period interrupt, at which time the interrupt service routine is executed.

Appendix A contains file listings of *vectors.asm*, *example.asm*, and *example.cmd*. Contiguous line numbers have been provided along the left-hand side of the code to facilitate discussion. The assembly instructions used in the code are documented in reference [1], and it is beyond the scope of this application report to provide detailed explanations of the instruction mnemonics and syntax. However, a number of details and useful pieces of information will now be discussed.[1]

Line 1 (*vectors.asm*): The asterisk in column 1 indicates that the entire line is a comment. The asterisk can only be used in column 1. To place a comment in a line beginning at a different column, use a semi-colon, as shown in line 16.

Line 13 (*vectors.asm*): The .ref directive is used to reference symbols defined externally to a source file. In this case, the symbols “start” and “timer2_isr” are both declared in the file *example.asm*, but are referenced here in *vectors.asm*. The .ref directive is the complement of the .def directive (see Line 69 below). The .ref directive is documented in reference [4], page 4–44.

Line 15 (*vectors.asm*): The .sect directive is used to declare an initialized section. In this case, the interrupt vectors (code) are being placed in the section named “vectors” so that they can be linked to a specific address range. You can find the vectors section listed in the SECTIONS portion of the linker command file *example.cmd*.

1. Cited page and section numbers in references correspond to the specific revision of the documentation listed in the Reference section of this application report. These numbers have been given as a matter of convenience, but may no longer be accurate if a new revision of the documentation is released. The referenced material itself, however, should still be found in the cited document. See the index in the cited document to find the correct page for the referenced subject.

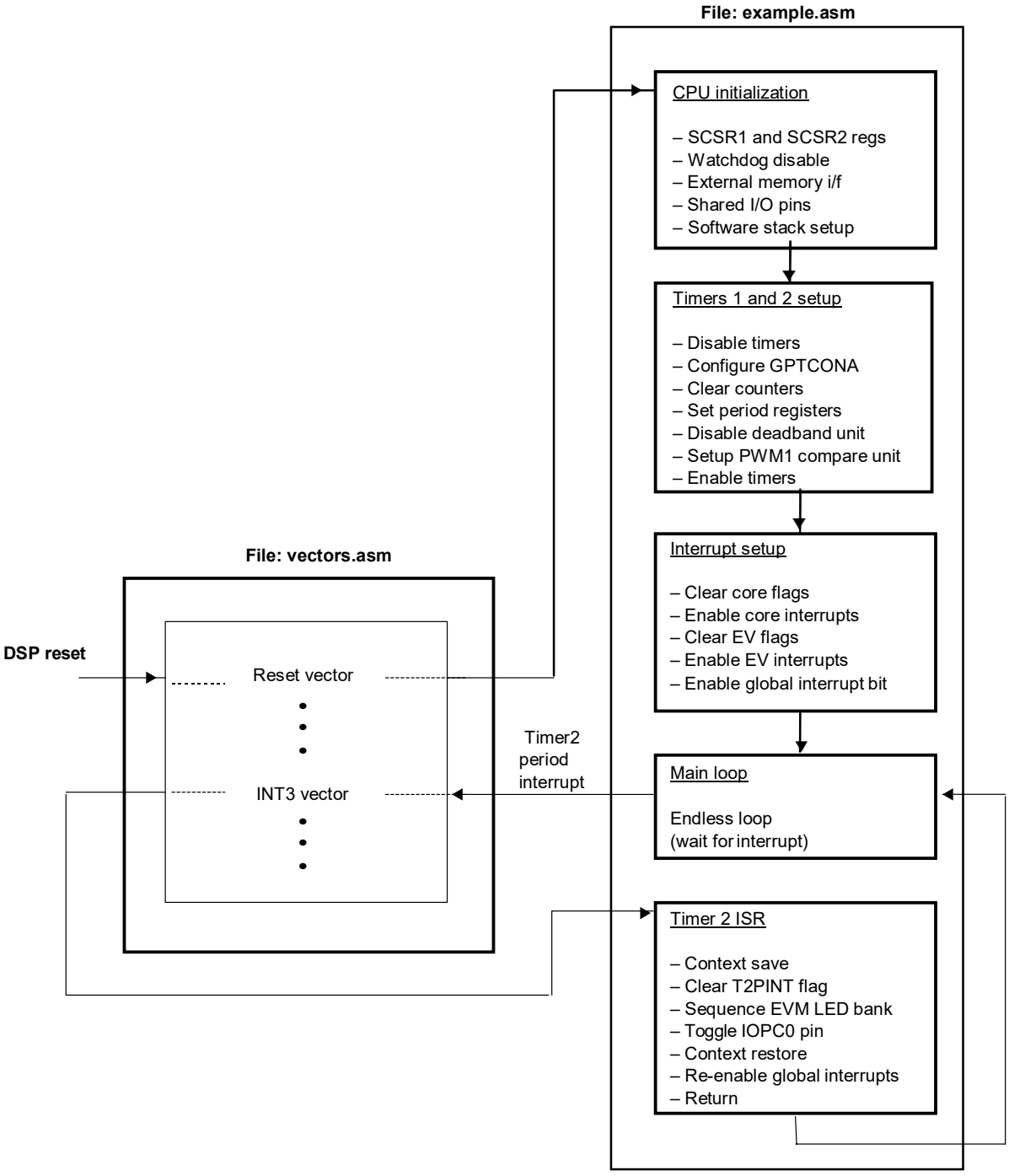

Figure 1. Assembly Language Example Program Code Flow

Lines 16 – 47 (*vectors.asm*): Each interrupt vector is two 16-bit words long and contains a single branch instruction that indicates the address of the interrupt service routine (ISR) associated with it. Line 16 is the reset vector. It branches to the label start located at the beginning of the main program in *example.asm*. Line 19 is the core interrupt 3 vector. This core interrupt is shared between several peripheral interrupts, among them the timer 2 period interrupt. In this example program, the timer 2 period interrupt is the only interrupt enabled on core interrupt 3. Therefore, the interrupt 3 vector can branch directly to the timer 2 interrupt service routine, located at the label timer2_isr in *example.asm*. If more than one peripheral interrupt is enabled on a given core interrupt line, the software needs to differentiate among the peripheral interrupts so that the correct ISR is executed. The procedure for doing this is discussed in reference [2], pages 2–20 to 2–21. The core interrupt vector addresses are documented in reference [1], pages 5–15 to 5–16. The sharing of the core interrupts is documented in reference [5], pages 20 to 23. Note that all remaining interrupt vectors branch to themselves. This is useful during code development to trap erroneous interrupts. In a real application, one would probably want to branch all unused interrupts to an error-handling routine that performs whatever recovery steps are desired.

Line 69 *(example.asm*): The .def directive is used to make symbols declared in a source file visible to code in other files. In this case, the labels “start” and “timer2_isr” are both declared here in the file *example.asm*, but are referenced in *vectors.asm*. The .def directive is the complement of the .ref directive (see Line 13 above). The .def directive is documented in reference [4], page 4–44.

Line 74 (*example.asm*): The .include directive is used to import a file directly into your source code at assembly time. In this case, the address definition file *f2407.h* is being included. The file *f2407.h* contains address definitions for all of the peripheral registers in the LF2407 DSP. The .include directive is documented in reference [4], page 4–29.

Line 76 (*example.asm*): The .set directive is used to define a constant in assembly code. The assembler replaces all occurrences of the constant within the source file with the specified value. The .set directive is documented in reference [4], page 4–67.

Lines 76 – 82 (*example.asm*): The .set directive is being used here to define the addresses in external I/O space of the D/A converter, DIP switch bank, and quad LED bank found on the LF2407 EVM (the eZdsp™ LF2407 does not contain any of these features). Note that only the quad LED bank is exercised by the example program (has no effect on eZdsp™ LF2407). Additional information on these EVM features can be found in reference [7].

Lines 98 – 99 (*example.asm*): The .bss directive is used to allocate space for uninitialized variables in the .bss section. In this case, 1 word (16-bits) is being allocated for the variable temp1, and 1 word is being allocated for the variable LED_index. You can find the .bss section listed in the SECTIONS portion of the linker command file *example.cmd*. The .bss directive is documented in reference [4], page 4–25.

Note that uninitialized variables can also be allocated using the .usect directive, which allows their placement into named sections other than .bss. The .bss directive is nothing but a special case of the .usect directive, and is used mostly for convenience. For example, the following two lines of code are equivalent:

```
            .bss temp1, 1          ;reserve 1 word in .bss section


temp1       .usect ".bss", 1       ;reserve 1 word in .bss section
```

The .usect directive is documented in reference [4], page 4–67. An example of .usect, used in this example program, may be found at Line 252.

Line 106 (*example.asm*): The .text directive is used to signify that the code (or initialized data) which follows is to be placed into the .text section. You can find the .text section listed in the SECTIONS portion of the linker command file, *example.cmd*. The .text directive is documented in reference [4], page 4–75.

Note that code and/or initialized data sections other than .text can be declared using the .sect directive. The .text directive is nothing but a special case of the .sect directive, and is used mostly for convenience. For example, the following two lines of code are equivalent:

```
            .text                  ;code that follows goes in .text section


            .sect ".text"          ;code that follows goes in .text section
```

The .sect directive is documented in reference [4], page 4–66.

Line 107 (*example.asm*): The label "start" is being used to mark the beginning of the main program. The reset vector in *vectors.asm* (see Line 16 above) branches here. Note that all labels must begin in column 1 in an assembly source file. The colon following the label is optional.

Line 112 (*example.asm*): The LDP instruction is used to set the data page for instructions using the direct-addressing mode that follow it. The assembler constant DP_PF1 is defined in the include file *f2407.h*.

Line 114 (*example.asm*): The SPLK instruction is used to load a number into a data space address. In this case, the address corresponds to the SCSR1 register. Since the constant SCSR1 has been defined in the include file *f2407.h* (see Line 74 above), you can simply reference the address by the name of the constant (SCSR1).

The value written to the SCSR1 register contains bit fields to program the PLL for x4 mode. In addition, there are six clock control bits that enable the clocks for the on-chip peripheral modules. Note that a peripheral module does not function if the clock to it is not enabled. In the example here, all clocks have been enabled. In a real application, you might want to disable clocks to any peripherals not being used to reduce power consumption. Finally, the ILLADR bit (illegal address detection bit) is being cleared by writing a 1 to it. Although ILLADR defaults to 0 after a reset, it is good practice to clear it anyway, in case (for example) software were to branch to the reset vector as a means of restarting a program. The SCSR1 register is documented in reference [2], pages 2–3 to 2–5.

Lines 131 – 143 (*example.asm*): The SCSR2 register is being initialized. Boolean operations are being used to set and clear the various bits. This procedure is used instead of a direct register load so it does not disturb the state of the MP/$\overline{\text{MC}}$ bit. This bit reflects the state of the MP/$\overline{\text{MC}}$ pin at reset, and determines whether the FLASH memory is active in program space, or if external memory is mapped to those addresses. Since this example program is designed to run from either FLASH or external memory, the state of the MP/$\overline{\text{MC}}$ pin at power up is unknown. Therefore, the state of the MP/$\overline{\text{MC}}$ pin must be preserved.

Bit 5 of SCSR2 is the WD OVERRIDE bit. This is a clear-only bit. If this bit is cleared (by writing a 1 to it), the watchdog timer cannot be disabled. The example code does not clear this bit, and the watchdog timer is later disabled at Line 150.

The SCSR2 register is documented in reference [2], pages 2–5 to 2–7.

Lines 148 – 157 (*example.asm*): The WDCR register is being configured to disable the watchdog timer. The WDCR register is documented in reference [2], pages 11–9 to 11–10.

Lines 162 – 173 (*example.asm*): The WSGR register is being configured to set the wait states for the external memory interface. On both the LF2407 EVM and eZdsp™ LF2407, zero wait states are needed in both program and data space. For the I/O space, one wait state is needed on the EVM for the D/A converter, whereas this is a don't care on the eZdsp™ LF2407, since there are no devices in the external I/O space. The WSGR register is located in I/O space, so the OUT instruction must be used to write to it. The WSGR register is documented in reference [2], pages 3–18 to 3–19.

Lines 178 – 238 (*example.asm*): The shared pins on the LF2407 are being configured. All shared pins default to the General Purpose Input Output (GPIO) input function after a device reset. The example program is configuring the PWM1/IOPA6 pin for PWM1 function. All other pins are left as GPIO. Note that bits 15–9 of the MCRB register affect operation of the JTAG emulation pins. These bits should always be written as 1. The MCRx registers are documented in reference [2], Chapter 5.

Lines 243 – 246 (*example.asm*): The W/$\overline{R}$/IOPC0 pin is being configured as a GPIO output. The W/$\overline{R}$/IOPC0 pin was previously configured for GPIO function in Line 200 using the MCRB register. The PCDATDIR register now determines GPIO input or output function. The W/$\overline{R}$/IOPC0 pin is toggled by the example program, as discussed in section 4. The PCDATDIR register is documented in reference [2], Chapter 5.

Lines 251 – 254 (*example.asm*): A software stack is being set up. In this example program, the software stack is used only for context saves and restores during interrupt service routines. However, other uses for a software stack include parameter passing to called functions and storage of local function variables. Here, the AR1 register is being used as the stack pointer, which is the same register convention employed by the C compiler.

The .usect directive is used to allocate space for the software stack in the uninitialized section called stack. You can find the stack section allocated in the SECTIONS portion of the linker command file *example.cmd*. The .usect directive is documented in reference [4], page 4–67.

Lines 260 – 261 (*example.asm*): GP Timers 1 and 2 are both disabled before being configured. This is proper programming procedure: always disable a peripheral before configuring it.

Lines 263 – 336 (*example.asm*): The various event manager registers necessary to configure the desired 20KHz PWM output using timer 1 and compare 1, and also the desired 250ms periodic interrupt using timer 2 are being initialized. The event manager is documented in reference [2], Chapter 6.

Lines 348 – 351 (*example.asm*): The core interrupt flag register (IFR) and core interrupt mask register (IMR) are initialized to enable the desired core interrupts. It is good practice to clear the IMR register first, since the IMR is not initialized by a DSP reset, and therefore contains an unknown value after power-up. After doing this, the IFR register is cleared to clear any pending core interrupts, and then the desired interrupts are enabled in the IMR. Note that before enabling any peripheral interrupt, the core interrupt on which it is grouped should first be enabled. The IFR and IMR registers are documented in reference [1], pages 5–17 to 5–20.

Lines 356 – 370 (*example.asm*): The desired event manager interrupts are being enabled. Note that before enabling any peripheral interrupt, the core interrupt on which it is grouped should first be enabled in the IMR register. Also note that the only event manager interrupt used in this program is the timer 2 period interrupt, which is enabled by setting bit 1 in the EVAIMRB register. The event manager interrupt registers are documented in reference [2], Chapter 6.

Line 375 (*example.asm*): The global interrupt mask bit is cleared. Clearing this bit enables all maskable core interrupts that have been enabled via the IMR register. The INTM bit is located in Status Register 0 (ST0), and is documented in reference [1], pages 4–15 to 4–17.

Lines 380 – 382 (*example.asm*): The main loop in the program is nothing but an endless loop containing a single NOP instruction. The DSP loops here until an enabled interrupt occurs, in this case, the timer 2 period interrupt.

Lines 396 – 401 (*example.asm*): A context save to the software stack is being performed. In this example, only ST0, ST1, and the ACC are being saved since these are the only registers used by the ISR. The context save illustrated here can easily be expanded to include other registers as needed.

Note that Line 397 advances the stack pointer (AR1) by one location, which skips a memory word. This is a necessary operation since there can potentially be situations in a program where the stack pointer is not pointing to the next free memory location, but instead is pointing to the last used location on the stack. An example of where this situation can arise is during a context restore, starting at Line 425. If interrupts were being nested here and an interrupt occurred during the context restore, that ISR must skip one memory location with the stack pointer, so as not to overwrite an occupied stack location (note that in this example, the program interrupts have not been nested). Another example is when accessing local variables or passed function parameters which reside on the stack (note that this does not occur in the example program here). To do this, a copy of the stack pointer is generally made using a different ARx register, and the copy then used for addressing. The code to make the stack pointer copy would appear as follows, using AR5 here as the copy register:

```
MAR   *, AR1      ;ARP = stack pointer
SAR   AR1, *      ;Save the stack pointer (AR1) onto the stack.
LAR   AR5, *      ;AR5 now contains a copy of the stack pointer.
                  ; Stack pointer again points to next free location.
```

Upon completion of the above code segment, AR5 can be used to address into the stack and access any local variables or passed function parameters. Notice that between the SAR and LAR instructions, the stack pointer (AR1) is actually pointing to an occupied stack location. If an interrupt were to occur between those two instructions, that ISR would overwrite an occupied stack location unless it had skipped one stack location before putting anything onto the stack.

Lines 404 – 405 (*example.asm*): Peripheral interrupt flags must be manually cleared in the ISR, unlike core interrupt flags in the IFR register which are cleared automatically by the DSP when the interrupt is serviced. In this case, the T2PINT flag is cleared by writing a 1 to its bit in the EVAIFRB register. The EVAIFRB register is documented in reference [2], Chapter 6.

Lines 408 – 415 (*example.asm*): The quad LED bank on the LF2407 EVM is updated. The variable LED_index is first used to update the LED bank, after which it is advanced by 1 by using a left-shift followed by a store back to memory. Since the LED bank exists in the I/O memory space, the OUT instruction is used. After the LED_index has been advanced 4 times (i.e., since there are 4 LEDs), it is reset back to 1.

Lines 418 – 421 (*example.asm*): The W/$\overline{R}$/IOPC0 pin is toggled by an XOR (exclusive OR) of the proper bit (bit 0) in the PCDATDIR register, with a binary 1.

Lines 424 – 429 (*example.asm*): The context restore is performed from the software stack. Note that the restoration order of ST0 and ST1 is important. When using indirect addressing (as was used here), ST0 should be restored first, followed by ST1. If direct addressing is used for the context save/restore, ST1 should be restored first. These sequences for context restore ensure that the ARP and DP bit fields are restored properly in the ST0 and ST1 registers.

Line 430 (*example.asm*): Global interrupts must be manually re-enabled before returning from an ISR by clearing the INTM bit.

Line 431 (*example.asm*): The RET instruction returns from the ISR.

Lines 442 – 455 (*example.cmd*): The MEMORY section of the linker command file should define all available memory on the DSP target system. The basic memory map for the LF2407 DSP is documented in reference [5], pages 16 and 32 to 33. You should tailor external memory definitions to meet those of your particular target board. In this example, a memory map has been defined that works for both the LF2407 EVM and the eZdsp™ LF2407.

The memory definitions are defined separately for program space (linker page 0) and data space (linker page 1). In the program space, the memory region named “VECS” has been specially defined so that the reset and interrupt vectors can be linked to the correct locations, as required by hardware. The interrupt vector addresses are documented in reference [1], pages 5–15 to 5–16. Note that the VECS memory is physically part of either the internal FLASH or external memory, depending on the state of the MP/$\overline{MC}$ pin at reset. The memory region named “FLASH” is defined over a 32Kx16 memory address range that will correspond to two possible physical memories. It will either be the internal FLASH memory (if the MP/$\overline{MC}$ pin was low during DSP reset), or it will map to external memory (if MP/$\overline{MC}$ pin was high during reset). See section 2 of this document for information on jumper settings that control the MP/$\overline{MC}$ pin on the LF2407 EVM and eZdsp™ LF2407 boards. Finally, the memory region named “EXTPROG” defines the external program memory that is available on the LF2407 EVM and eZdsp™ LF2407. Note that on the eZdsp™ LF2407, the external SRAM available at the EXTPROG addresses can also be mapped to the FLASH region addresses. See reference [8] for details.

In the data space, the internal dual-access memory blocks B0, B1, and B2 are defined, as is the 2Kx16 internal single-access RAM block SARAM. The memory region named “EXTDATA” defines the external data memory that is available on the LF2407 EVM and eZdsp™ LF2407.

Lines 457 – 463 (*example.cmd*): The SECTIONS section of the linker command file tells the linker where to locate each section used in the code. Here, only four sections have been used (.text, .bss, vectors, and stack). Notice that the vectors section has been linked to the VECS memory region, which places the interrupt vectors at the correct DSP-specific addresses. The SECTIONS section of the linker command file is documented in reference [4], section 8.7.

Note (*example.cmd*): Numerous linker options exist, all of which can be invoked either on the command line that starts the linker, or by placing the options at the beginning of the linker command file before the MEMORY section. Chapter 8 in reference [4] shows examples that place various options at the beginning of the linker command file (e.g., Example 8–2, page 8–17). However, placing these options in the linker command file is neither necessary nor recommended when using the Code Composer debugger, since Code Composer offers configuration menus that allow you to select desired options for your code project. Code Composer then uses the selected options on the linker command line when it invokes the linker during project building. This is why *example.cmd* only contains the MEMORY and SECTIONS sections, and nothing else. To select linker (and compiler and assembler) options from within Code Composer, click on the PROJECT menu, and then select OPTIONS.

6 C Language Example Program

The C language example program consists of the following files:

- **cvectors.asm:** Interrupt vector table
- **example_c.c:** Main program
- **f2407_c.h:** Header file containing peripheral register address definitions
- **example_c.cmd:** Linker command file
- **example_c.mak:** C2xx Code Composer v4.1x project file
- **example_c.out:** Program executable
- **example.map:** Memory map file output by the code generation tools

Only the first five listed files are needed to build the program. The final two files are provided for convenience, and are produced by the code generation tools when the program is built. To build the project using Code Composer, load the project file *example_c.mak* into Code Composer (click on PROJECT–>OPEN), and then select PROJECT–>BUILD or PROJECT–>REBUILD ALL. See the online help available within Code Composer for additional assistance.

Figure 2 illustrates the code flow for the C language example program. Execution begins at the reset vector in the file *cvectors.asm* after a hardware reset. This vector branches to the c_int0() function in the C language runtime support library *rts2xx.lib*. This library comes with the code generation tools, and must be linked with all C language programs (done by adding it to your Code Composer project). This has already been done in *example_c.mak*. The c_int0() function sets up the C stack, initializes values for all initialized global and static variables, and finally calls main(). The c_int0() function is documented in reference [3], section 6.8.

Once at main(), the processor is initialized, after which an endless main loop is entered. The DSP is periodically interrupted out of this endless loop by the timer 2 period interrupt, at which time the interrupt service routine is executed. ISR context save and restore, global interrupt re-enable, and program return are handled automatically by functions in the runtime support library.

Appendix B contains file listings of *cvectors.asm*, *example_c.c*, and *example_c.cmd*. Contiguous line numbers have been provided along the left-hand side of the code to facilitate discussion. The file *example_c.c* is functionally equivalent to the assembly language program *example.asm*, previously discussed in this document. Readers can compare these two files in order to gain insight into C language implementations of assembly language code segments, and vice versa. A number of details and useful pieces of information will now be discussed.[2]

2. Cited page and section numbers in references correspond to the specific revision of the documentation listed in the Reference section of this application report. These numbers have been given as a matter of convenience, but may no longer be accurate if a new revision of the documentation is released. The referenced material itself, however, should still be found in the cited document. See the index in the cited document to find the correct page for the referenced subject.

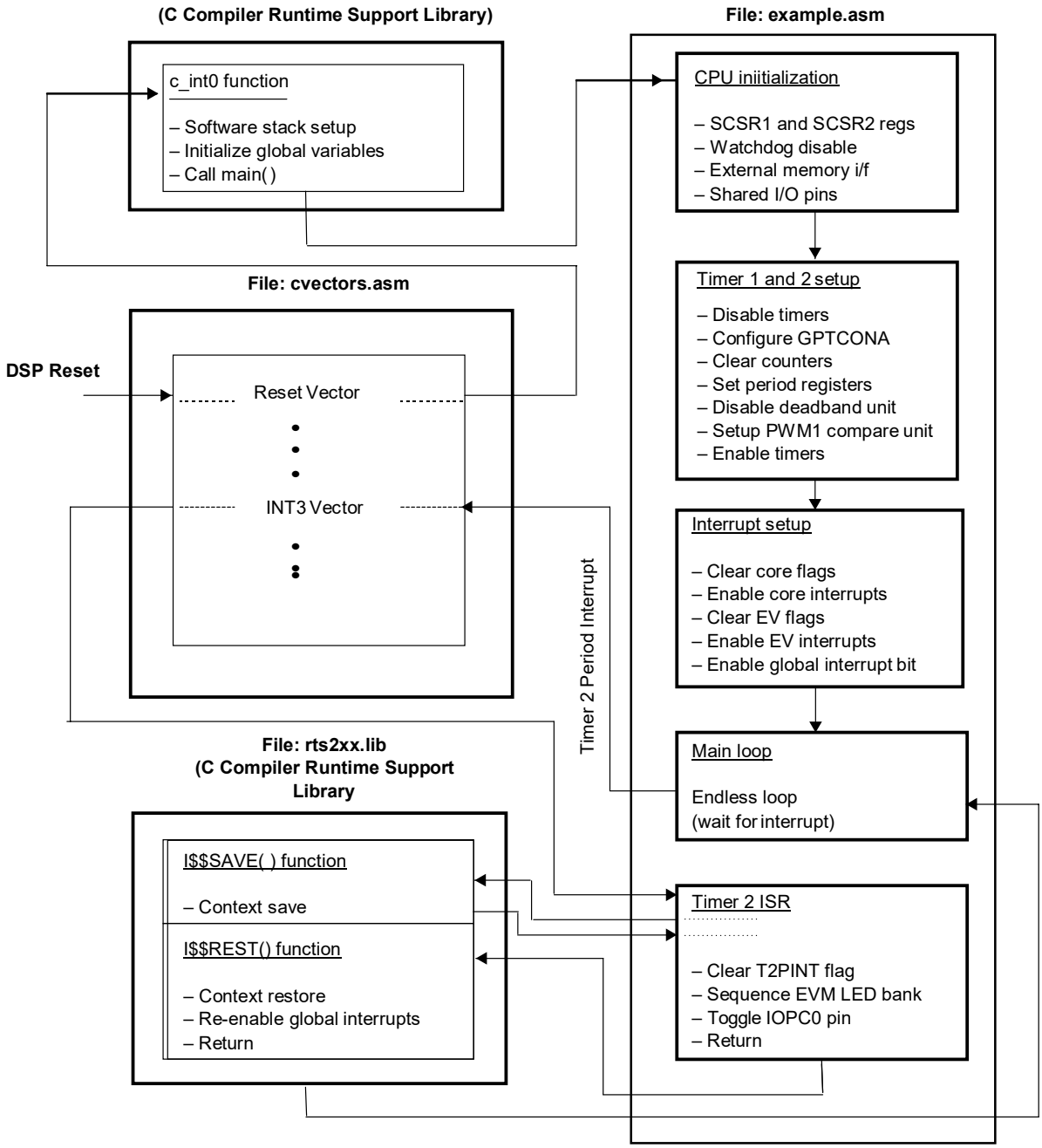

Figure 2. C Language Example Program Code Flow

NOTE: Much of the discussion that follows presents identical points to those made previously in section 5 for the assembly language example program. Information is repeated here to allow independent reading of the discussions of the C and assembly language example programs.

General (*cvectors.asm*): The interrupt vector table must be written in assembly language. While you could use inline assembly to embed the vector table into a C language file, it is better programming practice to use a separate file, as was done in this example. Multiple source files need simply be added to the Code Composer project. They will then be linked together automatically by the code generation tools during a project build.

Line 464 (*cvectors.asm*): The asterisk in column 1 indicates that the entire line is a comment. The asterisk can only be used in column 1. To place a comment in a line beginning at a different column, use a semi-colon, as shown in line 478.

Line 475 (*cvectors.asm*): The .ref directive is used to reference symbols defined externally to a source file. In this case, the symbol _c_int0 is declared in the runtime support library *rts2xx.lib*, and the label _timer2_isr is declared in the file *example_c.c*. However, both are referenced here in *cvectors.asm*. Note the use of the leading underscore on both labels. All symbols which are defined in C but referenced with assembly code must have a leading underscore appended to the symbol name in the assembly code reference. The .ref directive is documented in reference [4], page 4–44.

Line 477 (*cvectors.asm*): The .sect directive is used to declare an initialized section. In this case, the interrupt vectors (code) are being placed in the section named vectors so that they can be linked to a specific address range. You will find the vectors section listed in the SECTIONS portion of the linker command file *example_c.cmd*.

Lines 478 – 509 (*cvectors.asm*): Each interrupt vector is two 16-bit words long, and contains a single branch instruction that indicates the address of the interrupt service routine (ISR) associated with it. Line 478 is the reset vector, and branches to the label _c_int0 which is the entry point to a routine located in the *rts2xx.lib* library. Line 481 is the core interrupt 3 vector. This core interrupt is shared between several peripheral interrupts, among them the timer 2 period interrupt. In this example program, the timer 2 period interrupt is the only interrupt enabled on core interrupt 3. Therefore, the interrupt 3 vector can branch directly to the timer 2 interrupt service routine, located at the label timer2_isr in *example_c.c*. If more than one peripheral interrupt were enabled on a given core interrupt line, software would need to differentiate among the peripheral interrupts so that the correct ISR could be executed. The procedure for doing this is discussed in reference [2], pages 2–20 to 2–21. The core interrupt vector addresses are documented in reference [1], pages 5–15 to 5–16. The sharing of the core interrupts is documented in reference [5], pages 20 to 23. Note that all remaining interrupt vectors branch to themselves. This is useful during code development to trap erroneous interrupts. In a real application, one would probably want to branch all unused interrupts to an error handling routine that performs whatever recovery steps are desired.

Line 531 (*example_c.c):* The file *f2407_c.h* is being included into the C source file. This file contains address definitions for all of the peripheral registers in the LF2407 DSP. For example,

```
#define WDCR  (volatile unsignedint *)0x7029   /* WD timer control register */
```

defines the data space address for the watchdog timer control register WDCR. The above uses the ANSI C language preprocessor directive #define to associate the text WDCR with an immediate pointer to an unsigned integer at the address 0x7029. This allows one to access the defined register in their C program using a simple pointer construct. For example, to write the value 0x00E8 to the WDCR register, you simply write the following in your C program:

```
*WDCR  = 0x00E8;            /* write 0x00E8 to the WDCR register */
```

The volatile keyword in the address definition tells the compiler that a variable is not under its sole control (e.g., something other than code generated by the compiler, such as hardware, could change its value). It is important to use this keyword with register definitions, especially if the compiler optimizer is enabled. The volatile keyword is documented in reference [3], page 3–10.

Lines 533 – 552 (*example_c.c)*: Address definitions for the I/O space are being made. The I/O space for LF2407 devices consists of a 64Kx16 address block, beginning at address 0x0000. On the LF2407 EVM, the D/A converter, DIP switch bank, and quad-LED bank are located external to the DSP in the I/O space (on the eZdsp™ LF2407, no external devices exist in the I/O space). Note that only the quad LED bank is exercised by the example program (has no effect on eZdsp™ LF2407). The compiler assumes that a variable is located in the data space unless that variable is specifically defined as being in the I/O space.

The compiler provides the “ioport” keyword to declare an I/O space address. The ioport keyword declares an address in the I/O space as being of a particular type, e.g. unsigned int. The address itself is declared using the construct portxxxx, where xxxx is the 64K word address of the port. For example:

```
ioport unsigned int port000C;     /* I/O space address 0x000C is an unsigned int */
```

```
#define      LED    port000C      /* EVM LED bank is at address 0x000C in I/O space */
ioport unsigned int port000C;     /* I/O space address 0x000C is an unsigned int */
```

To make the C code more readable, you can associate a symbol with port000C as follows:

You can now access the declared I/O address using the defined name. For example, to write the value 0x0001 to address 0x000C in I/O space, you would write:

```
      LED = 0x0001;        /* write 0x0001 to EVM LED bank */
```

The ioport keyword is documented in reference [3], page 5–13. For details on the I/O space memory map, see reference [3], or the datasheet for your particular TI DSP device.

Line 568 (*example_c.c)*: This is the function declaration for main(). This function is called by the _c_int0 function, located in the runtime support library *rts2xx.lib*.

Line 572 (*example_c.c)*: The SCSR1 register is being initialized. The value written to the SCSR1 register contains bit fields to program the PLL for x4 mode. In addition, there are six clock control bits that enable the clocks for the on-chip peripheral modules. Note that a peripheral module will not function if the clock to it is not enabled. In the example here, all clocks have been enabled. In a real application, you can disable clocks to any peripherals not being used to reduce power consumption. Finally, the ILLADR bit (illegal address detection bit) is cleared by writing a 1 to it. Although ILLADR defaults to 0 after a reset, it is good practice to clear it anyway in case, for example, software were to branch to the reset vector as a means of restarting a program. The SCSR1 register is documented in reference [2], pages 2–3 to 2–5.

Line 589 (*example_c.c)*: The SCSR2 register is being initialized. Boolean operations are used to set and clear the various bits. This procedure is used instead of a direct register load, so as not to disturb the state of the MP/$\overline{MC}$ bit. This bit reflects the state of the MP/$\overline{MC}$ pin at reset and determines whether the FLASH memory is active in program space, or if external memory is mapped to those addresses. Since this example program is designed to run from either FLASH or external memory, the state of the MP/$\overline{MC}$ pin at power up is unknown. Therefore, the state of the MP/$\overline{MC}$ pin must be preserved.

Bit 5 of SCSR2 is the WD OVERRIDE bit. This is a clear-only bit. If this bit is cleared (by writing a 1 to it), the watchdog timer cannot be disabled. The example code does not clear this bit, and the watchdog timer is later disabled at Line 601.

The SCSR2 register is documented in reference [2], pages 2–5 to 2–7.

Line 601 (*example_c.c)*: The WDCR register is being configured to disable the watchdog timer. The WDCR register is documented in reference [2]. pages 11–9 to 11–10.

Line 612 (*example_c.c)*: The WSGR register is being configured to set the wait states for the external memory interface. On both the LF2407 EVM and eZdsp™ LF2407, zero wait states are needed in both program and data space. For the I/O space, one wait state is needed on the EVM for the D/A converter, whereas this is a don't care on the eZdsp™ LF2407 since there are no devices in the external I/O space. The WSGR register is located in I/O space, and was declared using the ioport keyword in the file *f2407_c.h*. This is why an asterisk (i.e., pointer symbol) is not used in the code in front of WSGR. See lines 533 – 552 above for more information on I/O space accesses in C. The WSGR register is documented in reference [2], pages 3–18 to 3–19.

Lines 622 – 680 (*example_c.c)*: The shared pins on the LF2407 are being configured. All shared pins default to the GPIO (General Purpose Input Output) input function after a device reset. The example program is configuring the PWM1/IOPA6 pin for PWM1 function. All other pins are left as GPIO. Note that bits 15–9 of the MCRB register affect operation of the JTAG emulation pins. These bits must always be written as a 1. The MCRx registers are documented in reference [2], Chapter 5.

Line 683 (*example_c.c)*: The W/$\overline{R}$/IOPC0 pin is being configured as a GPIO output. The W/$\overline{R}$/IOPC0 pin was previously configured for GPIO function in Line 642 using the MCRB register. The PCDATDIR register now determines GPIO input or output function. The W/$\overline{R}$/IOPC0 pin is toggled by the example program, as discussed in section 4. The PCDATDIR register is documented in reference [2], Chapter 5.

Lines 687 – 688 (*example_c.c)*: GP Timers 1 and 2 are both disabled before being configured. This is proper programming procedure: always disable a peripheral before configuring it.

Lines 690 – 766 (*example_c.c)*: The various event manager registers necessary to configure the desired 20KHz PWM output using timer 1 and compare 1, and also the desired 250ms periodic interrupt using timer 2 are being initialized. The event manager is documented in reference [2], Chapter 6.

Lines 772 – 774 (*example_c.c)*: The core interrupt flag register (IFR) and core interrupt mask register (IMR) are initialized to enable the desired core interrupts. It is good practice to clear the IMR register first, since the IMR is not initialized by a DSP reset, and therefore contains an unknown value after power up. After doing this, the IFR register is cleared to clear any pending core interrupts, and then the desired interrupts are enabled in the IMR. Note that before enabling any peripheral interrupt, the core interrupt on which it is grouped should first be enabled. The IFR and IMR registers are documented in reference [1], pages 5–17 to 5–20.

Lines 777 – 789 (*example_c.c)*: The desired event manager interrupts are being enabled. Note that before enabling any peripheral interrupt, the core interrupt on which it is grouped should first be enabled in the IMR register. Also note that the only event manager interrupt in use in this program is the timer 2 period interrupt, which is enabled by setting bit 1 in the EVAIMRB register. The event manager interrupt registers are documented in reference [2], Chapter 6.

Line 792 (*example_c.c)*: Inline assembly is being used to clear the global interrupt mask bit. The INTM bit is not directly accessible from C code, hence the use of inline assembly and the CLRC assembly instruction. Clearing this bit enables all maskable core interrupts that have been enabled via the IMR register. The INTM bit is located in Status Register 0 (ST0), and is documented in reference [1], pages 4–15 to 4–17.

Inline assembly is invoked by using the asm keyword, and placing in quotations the desired assembly language statement:

```
asm(" CLRC INTM");          /* enable global interrupts */
```

With inline assembly, whatever is written between the quotation marks is inserted directly into the compiler-generated assembly code. The functional position of the statement relative to the C code is preserved in the assembly code (i.e., the compiler optimizer will not move instructions around an inline assembly statement). Note that the character immediately following the first quotation mark is placed at column 1 in the assembly file. The space between the quotation mark and the CLRC instruction is therefore important in this example, since only labels and comments can begin in column 1 of an assembly file. The asm keyword is documented in reference [3], page 6–22.

Line 795 (*example_c.c*: The main loop in the program is nothing but an empty endless while() loop. The DSP will loop here until an enabled interrupt occurs, in this case the timer 2 period interrupt.

Line 797 (*example_c.c)*: The closing brace marks the end of main().

Line 800 (*example_c.c)*: This is the function declaration for the interrupt service routine timer2_isr(). Interrupt service routines can neither return a value, nor receive passed values, hence the use of the void data types in the function declaration. The “interrupt” keyword is used to tell the compiler that this function is an ISR. When this keyword is used, the compiler will insert code into the ISR that calls the context save function I$$SAVE upon function entry, and also insert code that branches to the context restore function I$$REST at the end of the ISR. These two functions are located in the runtime support library *rts2xx.lib*. After restoring the context, the I$$REST function re-enables global interrupts (i.e., clears the INTM bit) and then returns to the original interrupted routine. The interrupt keyword is documented in reference [3], page 6–26.

Line 803 (*example_c.c)*: Peripheral interrupt flags must be manually cleared in the ISR, unlike core interrupt flags in the IFR register which are cleared automatically by the DSP when the interrupt is serviced. In this case, the T2PINT flag is cleared by writing a 1 to its bit in the EVAIFRB register. The EVAIFRB register is documented in reference [2], Chapter 6.

Lines 806 – 808 (*example_c.c)*: The quad LED bank on the LF2407 EVM is updated. The variable LED_index is first used to update the LED bank, after which it is advanced by 1 by left-shifting it once. The LED bank exists in the I/O memory space, and the label "LED" was previously defined to access the proper address in I/O space in Lines 551–552. After the LED_index has been advanced 4 times (i.e., since there are 4 LEDs), it is reset back to 1.

Line 811 (*example_c.c)*: The W/$\overline{R}$/IOPC0 pin is toggled by an XOR (exclusive OR) of the proper bit (bit 0) in the PCDATADIR register with a binary 1.

Line 813 (*example_c.c)*: The closing brace marks the end of the ISR.

Lines 825 – 838 (*example_c.cmd*): The MEMORY section of the linker command file defines all available memory on the DSP target system. The basic memory map for the LF2407 DSP is documented in reference [5], page 16 and also pages 32 to 33. You should tailor external memory definitions to meet those of your particular target board. In this example, a memory map has been defined that works for both the LF2407 EVM and the eZdsp™ LF2407.

The memory definitions are defined separately for program space (linker page 0) and data space (linker page 1). In the program space, the memory region named VECS has been specially defined so that the reset and interrupt vectors can be linked to the correct locations, as required by hardware. The interrupt vector addresses are documented in reference [1], pages 5–15 to 5–16. Note that the VECS memory is physically part of either the internal FLASH or external memory, depending on the state of the MP/$\overline{MC}$ pin at reset. The memory region named FLASH is defined over a 32Kx16 memory address range that will correspond to two possible physical memories. It will either be the internal FLASH memory if the MP/$\overline{MC}$ pin was low during DSP reset, or it will map to external memory if MP/$\overline{MC}$ pin was high during reset. See section 2 of this document for information on jumper settings that control the MP/$\overline{MC}$ pin on the LF2407 EVM and the eZdsp™ LF2407 boards. Finally, the memory region named "EXTPROG" defines the external program memory that is available on the LF2407 EVM and eZdsp™ LF2407. Note that on the eZdsp™ LF2407, the external SRAM available at the EXTPROG addresses may also be mapped to the FLASH region addresses. See reference [8] for details.

In the data space, the internal dual-access memory blocks B0, B1, and B2 are defined, as is the 2Kx16 internal single-access RAM block SARAM. The memory region named "EXTDATA" defines the external data memory that is available on the LF2407 EVM and eZdsp™ LF2407.

Lines 840 – 853 (*example_c.cmd*): The SECTIONS section of the linker command file tells the linker where to locate each section used in the code. The C compiler uses seven specific sections (Lines 843 – 849), although it does not necessarily always allocate each section (e.g., .switch section will only be allocated if you have used switch statements in your C code). The function of each of these sections is documented in reference [3], pages 6–3 to 6–4. In addition to these sections, one user defined section called vectors (defined in the file *cvectors.asm*) is also used by this program, and therefore must be specified in the linker command file. The vectors section has been linked to the VECS memory region, which places the interrupt vectors at the correct DSP specific addresses. The SECTIONS section of the linker command file is documented in reference [4], section 8.7.

Note (*example_c.cmd*): Numerous linker options exist, all of which can be invoked either on the command line that starts the linker, or by placing the options at the beginning of the linker command file before the MEMORY section. Chapter 8 in reference [4] shows examples that place various options at the beginning of the linker command file (e.g., Example 8–2, page 8–17). However, placing these options in the linker command file is neither necessary nor recommended when using the Code Composer debugger, since Code Composer offers configuration menus that allow you to select desired options for your code project. Code Composer then uses the selected options on the linker command line when it invokes the linker during project building. This is why *example_c.cmd* only contains the MEMORY and SECTIONS sections, and nothing else. To select linker (and compiler and assembler) options from within Code Composer, click on the PROJECT menu, and then select OPTIONS.

7 Conclusion

Two functionally identical example programs, one written in assembly language, and the other in the C programming language, have been presented and discussed. These programs illustrate basic code for initializing and operating the TMS320LF240x DSP. The programs are ready to run on either the TMS320LF2407 Evaluation Module (EVM) or the eZdsp™ LF2407 development kit, but are also intended for use as code templates for any LF240x or LF240xA DSP target system.

8 References

1. *TMS320F/C24x DSP Controllers CPU and Instruction Set Reference Guide* (SPRU160C).
2. *TMS320LF/LC240x DSP Controllers System and Peripherals Reference Guide* (SPRU357A).
3. *TMS320C2x/C2xx/C5x Optimizing C Compiler User's Guide* (SPRU024E).
4. *Fixed-Point DSP Assembly Language Tools User's Guide* (SPRU018D).
5. *TMS320LF2407, TMS320LF2406, TMS320LF2402 DSP Controllers datasheet* (SPRS094F).
6. *TMS320LF2407 DSP Controller Silicon Errata* (SPRZ158F).
7. *TMS320LF2407 Evaluation Module Technical Reference* (from Spectrum Digital, Inc.)
8. *eZdsp™ Technical Reference* (from Spectrum Digital, Inc.)

Appendix A Assembly Language Example Program Listing

Line numbers listed along the left-hand side of the file listings are numerically contiguous across all files in Appendixes A and B.

```
 1    ************************************************************************
 2    * Filename: vectors.asm                                                *
 3    *                                                                      *
 4    * Author: David M. Alter, Texas Instruments Inc.                       *
 5    *                                                                      *
 6    * Last Modified: 03/14/01                                              *
 7    *                                                                      *
 8    * Description: Interrupt vector table for ’240x DSP core               *
 9    * for use with assembly language programs.                             *
10    *                                                                      *
11    ************************************************************************
12
13            .ref start, timer2_isr
14
15            .sect    ”vectors”
16    rset:   B       start           ;00h reset
17    int1:   B       int1            ;02h INT1
18    int2:   B       int2            ;04h INT2
19    int3:   B       timer2_isr      ;06h INT3
20    int4:   B       int4            ;08h INT4
21    int5:   B       int5            ;0Ah INT5
22    int6:   B       int6            ;0Ch INT6
23    int7:   B       int7            ;0Eh reserved
24    int8:   B       int8            ;10h INT8  (software)
25    int9:   B       int9            ;12h INT9  (software)
26    int10:  B       int10           ;14h INT10 (software)
27    int11:  B       int11           ;16h INT11 (software)
28    int12:  B       int12           ;18h INT12 (software)
29    int13:  B       int13           ;1Ah INT13 (software)
30    int14:  B       int14           ;1Ch INT14 (software)
31    int15:  B       int15           ;1Eh INT15 (software)
32    int16:  B       int16           ;20h INT16 (software)
33    int17:  B       int17           ;22h TRAP
34    int18:  B       int18           ;24h NMI
35    int19:  B       int19           ;26h reserved
36    int20:  B       int20           ;28h INT20 (software)
37    int21:  B       int21           ;2Ah INT21 (software)
38    int22:  B       int22           ;2Ch INT22 (software)
39    int23:  B       int23           ;2Eh INT23 (software)
40    int24:  B       int24           ;30h INT24 (software)
41    int25:  B       int25           ;32h INT25 (software)
42    int26:  B       int26           ;34h INT26 (software)
43    int27:  B       int27           ;36h INT27 (software)
44    int28:  B       int28           ;38h INT28 (software)
45    int29:  B       int29           ;3Ah INT29 (software)
46    int30:  B       int30           ;3Ch INT30 (software)
47    int31:  B       int31           ;3Eh INT31 (software)
```

```
48  ;*********************************************************************
49  ; Filename: example.asm                                              *
50  ;                                                                    *
51  ; Author: David M. Alter, Texas Instruments Inc.                     *
52  ;                                                                    *
53  ; Last Modified: 03/14/01                                           *
54  ;                                                                    *
55  ; Description: This program illustrates basic initialization and    *
56  ; operation of the LF2407 DSP.  The following peripherals are       *
57  ; exercised:                                                        *
58  ;  1) Timer 2 is configured to generate a 250ms period interrupt.   *
59  ;  2) The LED bank on the LF2407 EVM is sequenced in the Timer2 ISR. *
60  ;  3) The IOPC0 pin is toggled in the Timer2 ISR.                    *
61  ;  4) Timer 1 is configured to drive 20KHz 25% duty cycle symmetric  *
62  ;     PWM on the PWM1 pin.                                          *
63  ;                                                                    *
64  ;*********************************************************************
65
66  ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
67  ;Global symbol declarations
68  ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
69                  .def start, timer2_isr
70
71  ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
72  ;Address definitions
73  ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
74                  .include f2407.h
75
76  DAC0            .set    0000h           ;EVM DAC register 0 (I/O space)
77  DAC1            .set    0001h           ;EVM DAC register 1 (I/O space)
78  DAC2            .set    0002h           ;EVM DAC register 2 (I/O space)
79  DAC3            .set    0003h           ;EVM DAC register 3 (I/O space)
80  DACUD           .set    0004h           ;EVM DAC update register (I/O space)
81  DIPSWCH         .set    0008h           ;EVM DIP switch (I/O space)
82  LED             .set    000Ch           ;EVM LED bank (I/O space)
83
84  ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
85  ;Constant definitions
86  ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
87  timer2_per      .set    58594           ;250ms timer2 period with a 1/128
88                                          ;timer prescaler and 30MHz CPUCLK
89
90  pwm_half_per    .set    750             ;period/2, 20KHz symmetric PWM with
91                                          ;a 30MHz CPUCLK
92
93  pwm_duty        .set    563             ;25% PWM duty cycle
94
95  ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
96  ;Uninitialized global variable definitions
97  ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
98                  .bss    temp1,1         ;general purpose variable
99                  .bss    LED_index,1     ;LED index
100
101
```

```
102    ********************************************************************
103    *                     M A I N   R O U T I N E                      *
104    ********************************************************************
105
106             .text
107  start:
108
109  ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
110  ;Configure the System Control and Status Registers
111  ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
112             LDP     #DP_PF1         ;set data page
113
114             SPLK #0000000011111101b, SCSR1
115    *                ||||||||||||||||
116    *                FEDCBA9876543210
117    * bit 15         0:     reserved
118    * bit 14         0:     CLKOUT = CPUCLK
119    * bit 13-12      00:    IDLE1 selected for low-power mode
120    * bit 11-9       000:   PLL x4 mode
121    * bit 8          0:     reserved
122    * bit 7          1:     1 = enable ADC module clock
123    * bit 6          1:     1 = enable SCI module clock
124    * bit 5          1:     1 = enable SPI module clock
125    * bit 4          1:     1 = enable CAN module clock
126    * bit 3          1:     1 = enable EVB module clock
127    * bit 2          1:     1 = enable EVA module clock
128    * bit 1          0:     reserved
129    * bit 0          1:     clear the ILLADR bit
130
131             LACC     SCSR2                  ;ACC = SCSR2 register
132             OR       #0000000000001011b     ;OR in bits to be set
133             AND      #0000000000001111b     ;AND out bits to be cleared
134    *                 ||||||||||||||||
135    *                 FEDCBA9876543210
136    * bit 15-6       0's:   reserved
137    * bit 5          0:     do NOT clear the WD OVERRIDE bit
138    * bit 4          0:     XMIF_HI-Z, 0=normal mode, 1=Hi-Z'd
139    * bit 3          1:     disable the boot ROM, enable the FLASH
140    * bit 2      no change  MP/MC* bit reflects the state of the MP/MC* pin
141    * bit 1-0       11:     11 = SARAM mapped to prog and data (default)
142
143             SACL     SCSR2                   ;store to SCSR2 register
144
145    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
146    ;Disable the watchdog timer
147    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
148    ;        LDP     #DP_PF1                 ;set data page
149    ;
150              SPLK    #0000000011101000b, WDCR
151    *                 ||||||||||||||||
152    *                 FEDCBA9876543210
153    * bits 15-8     0's      reserved
154    * bit 7         1:       clear WD flag
155    * bit 6         1:       disable the dog
156    * bit 5-3      101:      must be written as 101
157    * bit 2-0      000:      WDCLK divider = 1
158
```

```
159     ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
160     ;Setup external memory interface for LF2407 EVM
161     ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
162             LDP     #temp1                  ;set data page
163
164             SPLK    #0000000001000000b, temp1
165     *                |||||||||||||||
166     *                FEDCBA9876543210
167     * bit 15-11      0's:    reserved
168     * bit 10-9       00:     bus visibility off
169     * bit 8-6        001:    1 wait-state for I/O space
170     * bit 5-3        000:    0 wait-state for data space
171     * bit 2-0        000:    0 wait state for program space
172
173             OUT     temp1, WSGR
174
175     ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
176     ;Setup shared I/O pins
177     ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
178             LDP     #DP_PF2                 ;set data page
179
180             SPLK    #0000000001000000b,MCRA ;group A pins
181     *                |||||||||||||||
182     *                FEDCBA9876543210
183     * bit 15         0:      0=IOPB7,     1=TCLKINA
184     * bit 14         0:      0=IOPB6,     1=TDIRA
185     * bit 13         0:      0=IOPB5,     1=T2PWM/T2CMP
186     * bit 12         0:      0=IOPB4,     1=T1PWM/T1CMP
187     * bit 11         0:      0=IOPB3,     1=PWM6
188     * bit 10         0:      0=IOPB2,     1=PWM5
189     * bit 9          0:      0=IOPB1,     1=PWM4
190     * bit 8          0:      0=IOPB0,     1=PWM3
191     * bit 7          0:      0=IOPA7,     1=PWM2
192     * bit 6          1:      0=IOPA6,     1=PWM1
193     * bit 5          0:      0=IOPA5,     1=CAP3
194     * bit 4          0:      0=IOPA4,     1=CAP2/QEP2
195     * bit 3          0:      0=IOPA3,     1=CAP1/QEP1
196     * bit 2          0:      0=IOPA2,     1=XINT1
197     * bit 1          0:      0=IOPA1,     1=SCIRXD
198     * bit 0          0:      0=IOPA0,     1=SCITXD
199
200             SPLK    #1111111000000000b,MCRB ;group B pins
201     *                |||||||||||||||
202     *                FEDCBA9876543210
203     * bit 15         1:      0=reserved,  1=TMS2 (always write as 1)
204     * bit 14         1:      0=reserved,  1=TMS  (always write as 1)
205     * bit 13         1:      0=reserved,  1=TD0  (always write as 1)
206     * bit 12         1:      0=reserved,  1=TDI  (always write as 1)
207     * bit 11         1:      0=reserved,  1=TCK  (always write as 1)
208     * bit 10         1:      0=reserved,  1=EMU1 (always write as 1)
209     * bit 9          1:      0=reserved,  1=EMU0 (always write as 1)
210     * bit 8          0:      0=IOPD0,     1=XINT2/ADCSOC
211     * bit 7          0:      0=IOPC7,     1=CANRX
212     * bit 6          0:      0=IOPC6,     1=CANTX
213     * bit 5          0:      0=IOPC5,     1=SPISTE
214     * bit 4          0:      0=IOPC4,     1=SPICLK
215     * bit 3          0:      0=IOPC3,     1=SPISOMI
216     * bit 2          0:      0=IOPC2,     1=SPISIMO
217     * bit 1          0:      0=IOPC1,     1=BIO*
218     * bit 0          0:      0=IOPC0,     1=W/R*
219
```

```
220             SPLK    #0000000000000000b,MCRC ;group C pins
221     *                ||||||||||||||||
222     *                FEDCBA9876543210
223     * bit 15        0:      reserved
224     * bit 14        0:      0=IOPF6,     1=IOPF6
225     * bit 13        0:      0=IOPF5,     1=TCLKINB
226     * bit 12        0:      0=IOPF4,     1=TDIRB
227     * bit 11        0:      0=IOPF3,     1=T4PWM/T4CMP
228     * bit 10        0:      0=IOPF2,     1=T3PWM/T3CMP
229     * bit 9         0:      0=IOPF1,     1=CAP6
230     * bit 8         0:      0=IOPF0,     1=CAP5/QEP4
231     * bit 7         0:      0=IOPE7,     1=CAP4/QEP3
232     * bit 6         0:      0=IOPE6,     1=PWM12
233     * bit 5         0:      0=IOPE5,     1=PWM11
234     * bit 4         0:      0=IOPE4,     1=PWM10
235     * bit 3         0:      0=IOPE3,     1=PWM9
236     * bit 2         0:      0=IOPE2,     1=PWM8
237     * bit 1         0:      0=IOPE1,     1=PWM7
238     * bit 0         0:      0=IOPE0,     1=CLKOUT
239
240     ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
241     ;Configure IOPC0 pin as an output
242     ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
243             LDP     #DP_PF2                 ;set data page
244             LACC    #0100h                  ;ACC = 0100h
245             OR      PCDATDIR                ;OR in PCDATDIR register
246             SACL    PCDATDIR                ;store result to PCDATDIR
247
248     ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
249     ;Setup the software stack
250     ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
251     stk_len         .set    100             ;stack length
252     stk             .usect  "stack",stk_len  ;reserve space for stack
253
254             LAR     AR1, #stk               ;AR1 is the stack pointer
255
256     ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
257     ;Setup timers 1 and 2, and the PWM configuration
258     ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
259             LDP     #DP_EVA                 ;set data page
260             SPLK    #0000h, T1CON           ;disable timer 1
261             SPLK    #0000h, T2CON           ;disable timer 2
262
263             SPLK    #0000000000000000b, GPTCONA
264     *                ||||||||||||||||
265     *                FEDCBA9876543210
266     * bit 15        0:      reserved
267     * bit 14        0:      T2STAT, read-only
268     * bit 13        0:      T1STAT, read-only
269     * bit 12-11     00:     reserved
270     * bit 10-9      00:     T2TOADC, 00 = no timer2 event starts ADC
271     * bit 8-7       00:     T1TOADC, 00 = no timer1 event starts ADC
272     * bit 6         0:      TCOMPOE, 0 = Hi-z all timer compare outputs
273     * bit 5-4       00:     reserved
274     * bit 3-2       00:     T2PIN, 00 = forced low
275     * bit 1-0       00:     T1PIN, 00 = forced low
276
```

```
277     ;Timer 1: Configure to clock the PWM on PWM1 pin.
278     ;Symmetric PWM, 20KHz carrier frequency, 25% duty cycle
279             SPLK    #0000h, T1CNT           ;clear timer counter
280             SPLK    #pwm_half_per, T1PR     ;set timer period
281             SPLK    #0000h, DBTCONA         ;deadband units off
282             SPLK    #pwm_duty, CMPR1        ;set PWM duty cycle
283
284             SPLK    #0000000000000010b, ACTRA
285     *                ||||||||||||||||
286     *                FEDCBA9876543210
287     * bit 15        0:      space vector dir is CCW (don't care)
288     * bit 14-12     000:    basic space vector is 000 (dont' care)
289     * bit 11-10     00:     PWM6/IOPB3 pin forced low
290     * bit 9-8       00:     PWM5/IOPB2 pin forced low
291     * bit 7-6       00:     PWM4/IOPB1 pin forced low
292     * bit 5-4       00:     PWM3/IOPB0 pin forced low
293     * bit 3-2       00:     PWM2/IOPA7 pin forced low
294     * bit 1-0       10:     PWM1/IOPA6 pin active high
295
296             SPLK    #1000001000000000b, COMCONA
297     *                ||||||||||||||||
298     *                FEDCBA9876543210
299     * bit 15        1:      1 = enable compare operation
300     * bit 14-13     00:     00 = reload CMPRx regs on timer 1 underflow
301     * bit 12        0:      0 = space vector disabled
302     * bit 11-10     00:     00 = reload ACTR on timer 1 underflow
303     * bit 9         1:      1 = enable PWM pins
304     * bit 8-0       0's:    reserved
305
306             SPLK    #0000100001000000b, T1CON
307     *                ||||||||||||||||
308     *                FEDCBA9876543210
309     * bit 15-14     00:     stop immediately on emulator suspend
310     * bit 13        0:      reserved
311     * bit 12-11     01:     01 = continous-up/down count mode
312     * bit 10-8      000:    000 = x/1 prescaler
313     * bit 7         0:      reserved in T1CON
314     * bit 6         1:      TENABLE, 1 = enable timer
315     * bit 5-4       00:     00 = CPUCLK is clock source
316     * bit 3-2       00:     00 = reload compare reg on underflow
317     * bit 1         0:      0 = disable timer compare
318     * bit 0         0:      reserved in T1CON
319
320     ;Timer 2: configure to generate a 250ms periodic interrupt
321             SPLK    #0000h, T2CNT           ;clear timer counter
322             SPLK    #timer2_per, T2PR       ;set timer period
323
324             SPLK    #1101011101000000b, T2CON
325     *                ||||||||||||||||
326     *                FEDCBA9876543210
327     * bit 15-14     11:     stop immediately on emulator suspend
328     * bit 13        0:      reserved
329     * bit 12-11     10:     10 = continous-up count mode
330     * bit 10-8      111:    111 = x/128 prescaler
331     * bit 7         0:      T2SWT1, 0 = use own TENABLE bit
332     * bit 6         1:      TENABLE, 1 = enable timer
333     * bit 5-4       00:     00 = CPUCLK is clock source
334     * bit 3-2       00:     00 = reload compare reg on underflow
335     * bit 1         0:      0 = disable timer compare
336     * bit 0         0:      SELT1PR, 0 = use own period register
337
```

```
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;Other setup
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   ;LED index initialization
            LDP     #LED_index              ;set data page
            SPLK    #1h, LED_index          ;initialize the LED index

;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;Setup the core interrupts
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
            LDP     #0h                     ;set data page
            SPLK    #0h,IMR                 ;clear the IMR register
            SPLK    #111111b,IFR            ;clear any pending core interrupts
            SPLK    #000100b,IMR            ;enable desired core interrupts

;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;Setup the event manager interrupts
   ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
            LDP     #DP_EVA                 ;set data page
            SPLK    #0FFFFh, EVAIFRA        ;clear all EVA group A interrupts
            SPLK    #0FFFFh, EVAIFRB        ;clear all EVA group B interrupts
            SPLK    #0FFFFh, EVAIFRC        ;clear all EVA group C interrupts
            SPLK    #00000h, EVAIMRA        ;enabled desired EVA group A interrupts
            SPLK    #00001h, EVAIMRB        ;enabled desired EVA group B interrupts
            SPLK    #00000h, EVAIMRC        ;enabled desired EVA group C interrupts

            LDP     #DP_EVB                 ;set data page
            SPLK    #0FFFFh, EVBIFRA        ;clear all EVB group A interrupts
            SPLK    #0FFFFh, EVBIFRB        ;clear all EVB group B interrupts
            SPLK    #0FFFFh, EVBIFRC        ;clear all EVB group C interrupts
            SPLK    #00000h, EVBIMRA        ;enabled desired EVB group A interrupts
            SPLK    #00000h, EVBIMRB        ;enabled desired EVB group B interrupts
            SPLK    #00000h, EVBIMRC        ;enabled desired EVB group C interrupts

   ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;Enable global interrupts
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
            CLRC    INTM                    ;enable global interrupts

;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;Main loop
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   loop:
            NOP
            B       loop                    ;branch to loop


   ************************************************************************
   *  I N T E R R U P T  S E R V I C E  R O U T I N E S                   *
   ************************************************************************

;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;GP Timer 2 period interrupt (core interrupt INT3)
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

timer2_isr:

;Context save to the software stack
            MAR     *,AR1                   ;ARP=stack pointer
```

```
397             MAR     *+                      ;skip one stack location (required)
398             SST     #1, *+                  ;save ST1
399             SST     #0, *+                  ;save ST0
400             SACH    *+                      ;save ACCH
401             SACL    *+                      ;save ACCL
402
403     ;Clear the T2PINT interrupt flag
404             LDP     #DP_EVA                 ;set data page
405             SPLK    #00001h, EVAIFRB        ;clear T2PINT flag
406
407     ;Sequence the LED bank on the LF2407 EVM
408             LDP     #LED_index              ;set data page
409             OUT     LED_index, LED          ;light the LED
410             LACC    LED_index,1             ;load LED index with left shift of 1
411             SACL    LED_index               ;store updated index
412             SUB     #0010h                  ;subtract the mask
413             BCND    done, LT                ;branch if index not ready for reset
414             SPLK    #1h, LED_index          ;reset LED index to 1
415     done:
416
417     ;Toggle the IOPC0 pin
418             LDP     #DP_PF2                 ;set data page
419             LACC    #0001h                  ;ACC = 0001h
420             XOR     PCDATDIR                ;XOR the IOPC0 bit to toggle the pin
421             SACL    PCDATDIR                ;store result to PCDATDIR
422
423     ;context restore from the software stack
424             MAR     *, AR1                  ;ARP = AR1
425             MAR     *-                      ;SP points to last entry
426             LACL    *-                      ;restore ACCL
427             ADD     *-,16                   ;restore ACCH
428             LST     #0, *-                  ;restore ST0
429             LST     #1, *-                  ;restore ST1, unskip one stack location
430             CLRC    INTM                    ;re-enable interrupts
431             RET                             ;return from the interrupt
```

```
/*****************************************************************
* Filename: example.cmd                                          *
*                                                                *
* Author: David M. Alter, Texas Instruments Inc.                 *
*                                                                *
* Last Modified: 03/14/01                                        *
*                                                                *
* Description: Assembly code linker command file for LF2407 DSP. *
*****************************************************************/

MEMORY
{
  PAGE 0:    /* Program Memory */
    VECS:           org=00000h,   len=00040h   /* internal FLASH */
    FLASH:          org=00044h,   len=07FBCh   /* internal FLASH */
    EXTPROG:        org=08800h,   len=07800h   /* external SRAM  */

  PAGE 1:    /* Data Memory */
    B2:             org=00060h,   len=00020h   /* internal DARAM */
    B0:             org=00200h,   len=00100h   /* internal DARAM */
    B1:             org=00300h,   len=00100h   /* internal DARAM */
    SARAM:          org=00800h,   len=00800h   /* internal SARAM */
    EXTDATA:        org=08000h,   len=08000h   /* external SRAM  */
}

SECTIONS
{
       .text:  >    FLASH         PAGE 0
        .bss:  >    B1            PAGE 1
    vectors:  >    VECS          PAGE 0
      stack:  >    SARAM         PAGE 1
}
```

Appendix B C Language Example Program Listing

Line numbers listed along the left-hand side of the file listings are numerically contiguous across all files in Appendixes A and B.

```
464     ;=====================================================================
465     ; Filename: cvectors.asm                                             =
466     ;                                                                   =
467     ; Author: David M. Alter, Texas Instruments Inc.                    =
468     ;                                                                   =
469     ; Last Modified: 03/14/01                                           =
470     ;                                                                   =
471     ; Description: Interrupt vector table for ’240x DSP core            =
472     ; for use with C language programs.                                 =
473     ;                                                                   =
474     ;=====================================================================
475             .ref _c_int0, _timer2_isr
476
477             .sect    ”vectors”
478     rset:   B       _c_int0         ;00h reset
479     int1:   B       int1            ;02h INT1
480     int2:   B       int2            ;04h INT2
481     int3:   B       _timer2_isr     ;06h INT3
482     int4:   B       int4            ;08h INT4
483     int5:   B       int5            ;0Ah INT5
484     int6:   B       int6            ;0Ch INT6
485     int7:   B       int7            ;0Eh reserved
486     int8:   B       int8            ;10h INT8  (software)
487     int9:   B       int9            ;12h INT9  (software)
488     int10:  B       int10           ;14h INT10 (software)
489     int11:  B       int11           ;16h INT11 (software)
490     int12:  B       int12           ;18h INT12 (software)
491     int13:  B       int13           ;1Ah INT13 (software)
492     int14:  B       int14           ;1Ch INT14 (software)
493     int15:  B       int15           ;1Eh INT15 (software)
494     int16:  B       int16           ;20h INT16 (software)
495     int17:  B       int17           ;22h TRAP
496     int18:  B       int18           ;24h NMI
497     int19:  B       int19           ;26h reserved
498     int20:  B       int20           ;28h INT20 (software)
499     int21:  B       int21           ;2Ah INT21 (software)
500     int22:  B       int22           ;2Ch INT22 (software)
501     int23:  B       int23           ;2Eh INT23 (software)
502     int24:  B       int24           ;30h INT24 (software)
503     int25:  B       int25           ;32h INT25 (software)
504     int26:  B       int26           ;34h INT26 (software)
505     int27:  B       int27           ;36h INT27 (software)
506     int28:  B       int28           ;38h INT28 (software)
507     int29:  B       int29           ;3Ah INT29 (software)
508     int30:  B       int30           ;3Ch INT30 (software)
509     int31:  B       int31           ;3Eh INT31 (software)
```

```
/*************************************************************************
* Filename: example_c.c                                                  *
*                                                                        *
* Author: David M. Alter, Texas Instruments Inc.                         *
*                                                                        *
* Last Modified: 03/14/01                                                *
*                                                                        *
* Description: This program illustrates basic initialization and         *
* operation of the LF2407 DSP.  The following peripherals are            *
* exercised:                                                             *
*  1) Timer 2 is configured to generate a 250ms period interrupt.        *
*  2) The quad LED bank on the LF2407 EVM is sequenced in the            *
*     Timer2 ISR.                                                        *
*  3) The IOPC0 pin is toggled in the Timer2 ISR.                        *
*  4) Timer 1 is configured to drive 20KHz 25% duty cycle symmetric      *
*     PWM on the PWM1 pin.                                               *
*                                                                        *
*************************************************************************/


/*** Address Definitions ***/
#include       "f2407_c.h"

#define         DAC0    port0000        /* EVM DAC register 0 (I/O space) */
ioport unsigned port0000;               /* '24xx compiler specific keyword */

#define         DAC1    port0001        /* EVM DAC register 1 (I/O space) */
ioport unsigned port0001;               /* '24xx compiler specific keyword */

#define         DAC2    port0002        /* EVM DAC register 2 (I/O space) */
ioport unsigned port0002;               /* '24xx compiler specific keyword */

#define         DAC3    port0003        /* EVM DAC register 3 (I/O space) */
ioport unsigned port0003;               /* '24xx compiler specific keyword */

#define         DACUD   port0004        /* EVM DAC update register (I/O space) */
ioport unsigned port0004;               /* '24xx compiler specific keyword */

#define         DIPSWCH port0008        /* EVM DIP switch (I/O space) */
ioport unsigned port0008;               /* '24xx compiler specific keyword */

#define         LED     port000C        /* EVM LED bank (I/O space) */
ioport unsigned port000C;               /* '24xx compiler specific keyword */


/*** Constant Definitions ***/
#define         timer2_per    58594     /* 250ms timer2 period with a 1/128
                                           timer prescaler and 30MHz CPUCLK */

#define         pwm_half_per  750       /* period/2, 20KHz symmetric PWM with
                                           a 30MHz CPUCLK */

 #define        pwm_duty      563       /* 25% PWM duty cycle */


 /*** Global Variable Definitions ***/
 unsigned int    LED_index;             /* LED_index */
```

```
567     /****************************** MAIN ROUTINE *************************/
568     void main(void)
569    {
570
571    /*** Configure the System Control and Status registers ***/
572        *SCSR1 = 0x00FD;
573    /*
574     bit 15        0:      reserved
575     bit 14        0:      CLKOUT = CPUCLK
576     bit 13-12     00:     IDLE1 selected for low-power mode
577     bit 11-9      000:    PLL x4 mode
578     bit 8         0:      reserved
579     bit 7         1:      1 = enable ADC module clock
580     bit 6         1:      1 = enable SCI module clock
581     bit 5         1:      1 = enable SPI module clock
582     bit 4         1:      1 = enable CAN module clock
583     bit 3         1:      1 = enable EVB module clock
584     bit 2         1:      1 = enable EVA module clock
585     bit 1         0:      reserved
586     bit 0         1:      clear the ILLADR bit
587    */
588
589        *SCSR2 = (*SCSR2 | 0x000B) & 0x000F;
590    /*
591     bit 15-6      0's:    reserved
592     bit 5         0:      do NOT clear the WD OVERRIDE bit
593     bit 4         0:      XMIF_HI-Z, 0=normal mode, 1=Hi-Z'd
594     bit 3         1:      disable the boot ROM, enable the FLASH
595     bit 2     no change   MP/MC* bit reflects state of MP/MC* pin
596    bit 1-0       11:      11 = SARAM mapped to prog and data
597    */
598
599
600    /*** Disable the watchdog timer ***/
601        *WDCR  = 0x00E8;
602    /*
603     bits 15-8     0's:     reserved
604     bit 7         1:       clear WD flag
605     bit 6         1:       disable the dog
606     bit 5-3       101:     must be written as 101
607     bit 2-0       000:     WDCLK divider = 1
608    */
609
610
611    /*** Setup external memory interface for LF2407 EVM ***/
612        WSGR = 0x0040;
613    /*
614     bit 15-11     0's:    reserved
615     bit 10-9      00:     bus visibility off
616     bit 8-6       001:    1 wait-state for I/O space
617     bit 5-3       000:    0 wait-state for data space
618     bit 2-0       000:    0 wait state for program space
619    */
620
621    /*** Setup shared I/O pins ***/
622        *MCRA = 0x0040;                    /* group A pins */
623    /*
624     bit 15        0:      0=IOPB7,     1=TCLKINA
625     bit 14        0:      0=IOPB6,     1=TDIRA
```

```
626     bit 13        0:      0=IOPB5,     1=T2PWM/T2CMP
627     bit 12        0:      0=IOPB4,     1=T1PWM/T1CMP
628     bit 11        0:      0=IOPB3,     1=PWM6
629     bit 10        0:      0=IOPB2,     1=PWM5
630     bit 9         0:      0=IOPB1,     1=PWM4
631     bit 8         0:      0=IOPB0,     1=PWM3
632     bit 7         0:      0=IOPA7,     1=PWM2
633     bit 6         1:      0=IOPA6,     1=PWM1
634     bit 5         0:      0=IOPA5,     1=CAP3
635     bit 4         0:      0=IOPA4,     1=CAP2/QEP2
636     bit 3         0:      0=IOPA3,     1=CAP1/QEP1
637     bit 2         0:      0=IOPA2,     1=XINT1
638     bit 1         0:      0=IOPA1,     1=SCIRXD
639     bit 0         0:      0=IOPA0,     1=SCITXD
640    */
641
642        *MCRB = 0xFE00;                    /* group B pins */
643    /*
644     bit 15        1:      0=reserved,  1=TMS2 (always write as 1)
645     bit 14        1:      0=reserved,  1=TMS  (always write as 1)
646     bit 13        1:      0=reserved,  1=TD0  (always write as 1)
647     bit 12        1:      0=reserved,  1=TDI  (always write as 1)
648     bit 11        1:      0=reserved,  1=TCK  (always write as 1)
649     bit 10        1:      0=reserved,  1=EMU1 (always write as 1)
650     bit 9         1:      0=reserved,  1=EMU0 (always write as 1)
651     bit 8         0:      0=IOPD0,     1=XINT2/ADCSOC
652     bit 7         0:      0=IOPC7,     1=CANRX
653     bit 6         0:      0=IOPC6,     1=CANTX
654     bit 5         0:      0=IOPC5,     1=SPISTE
655     bit 4         0:      0=IOPC4,     1=SPICLK
656     bit 3         0:      0=IOPC3,     1=SPISOMI
657     bit 2         0:      0=IOPC2,     1=SPISIMO
658     bit 1         0:      0=IOPC1,     1=BIO*
659     bit 0         0:      0=IOPC0,     1=W/R*
660    */
661
662        *MCRC = 0x0000;                    /* group C pins */
663    /*
664     bit 15        0:      reserved
665     bit 14        0:      0=IOPF6,     1=IOPF6
666     bit 13        0:      0=IOPF5,     1=TCLKINB
667     bit 12        0:      0=IOPF4,     1=TDIRB
668     bit 11        0:      0=IOPF3,     1=T4PWM/T4CMP
669     bit 10        0:      0=IOPF2,     1=T3PWM/T3CMP
670     bit 9         0:      0=IOPF1,     1=CAP6
671     bit 8         0:      0=IOPF0,     1=CAP5/QEP4
672     bit 7         0:      0=IOPE7,     1=CAP4/QEP3
673     bit 6         0:      0=IOPE6,     1=PWM12
674     bit 5         0:      0=IOPE5,     1=PWM11
675     bit 4         0:      0=IOPE4,     1=PWM10
676     bit 3         0:      0=IOPE3,     1=PWM9
677     bit 2         0:      0=IOPE2,     1=PWM8
678     bit 1         0:      0=IOPE1,     1=PWM7
679     bit 0         0:      0=IOPE0,     1=CLKOUT
680    */
681
682    /*** Configure IOPC0 pin as an output ***/
683        *PCDATDIR = *PCDATDIR | 0x0100;
684
685
```

```
686     /*** Setup timers 1 and 2, and the PWM configuration ***/
687         *T1CON = 0x0000;                    /* disable timer 1 */
688         *T2CON = 0x0000;                    /* disable timer 2 */
689
690         *GPTCONA = 0x0000;                  /* configure GPTCONA */
691     /*
692      bit 15        0:      reserved
693      bit 14        0:      T2STAT, read-only
694      bit 13        0:      T1STAT, read-only
695      bit 12-11     00:     reserved
696      bit 10-9      00:     T2TOADC, 00 = no timer2 event starts ADC
697      bit 8-7       00:     T1TOADC, 00 = no timer1 event starts ADC
698      bit 6         0:      TCOMPOE, 0 = Hi-z all timer compare outputs
699      bit 5-4       00:     reserved
700      bit 3-2       00:     T2PIN, 00 = forced low
701      bit 1-0       00:     T1PIN, 00 = forced low
702     */
703
704
705     /* Timer 1: configure to clock the PWM on PWM1 pin */
706     /* Symmetric PWM, 20KHz carrier frequency, 25% duty cycle */
707         *T1CNT = 0x0000;                    /* clear timer counter */
708         *T1PR = pwm_half_per;               /* set timer period */
709         *DBTCONA = 0x0000;                  /* deadband units off */
710         *CMPR1 = pwm_duty;                  /* set PWM1 duty cycle */
711
712         *ACTRA = 0x0002;                    /* PWM1 pin set active high */
713     /*
714      bit 15        0:      space vector dir is CCW (don't care)
715      bit 14-12     000:    basic space vector is 000 (dont' care)
716      bit 11-10     00:     PWM6/IOPB3 pin forced low
717      bit 9-8       00:     PWM5/IOPB2 pin forced low
718      bit 7-6       00:     PWM4/IOPB1 pin forced low
719      bit 5-4       00:     PWM3/IOPB0 pin forced low
720      bit 3-2       00:     PWM2/IOPA7 pin forced low
721      bit 1-0       10:     PWM1/IOPA6 pin active high
722     */
723
724          *COMCONA = 0x8200;                 /* configure COMCON register */
725     /*
726      bit 15        1:      1 = enable compare operation
727      bit 14-13     00:     00 = reload CMPRx regs on timer 1 underflow
728      bit 12        0:      0 = space vector disabled
729      bit 11-10     00:     00 = reload ACTR on timer 1 underflow
730      bit 9         1:      1 = enable PWM pins
731     bit 8-0       0's:    reserved
732     */
733
734
735          *T1CON = 0x0840;                   /* configure T1CON register */
736     /*
737      bit 15-14     00:     stop immediately on emulator suspend
738      bit 13        0:      reserved
739      bit 12-11     01:     01 = continous-up/down count mode
740      bit 10-8      000:    000 = x/1 prescaler
741      bit 7         0:      reserved in T1CON
742      bit 6         1:      TENABLE, 1 = enable timer
743      bit 5-4       00:     00 = CPUCLK is clock source
744      bit 3-2       00:     00 = reload compare reg on underflow
```

```
745     bit 1         0:      0 = disable timer compare
746     bit 0         0:      reserved in T1CON
747    */
748
749
750    /* Timer 2: configure to generate a 250ms periodic interrupt */
751        *T2CNT = 0x0000;                 /* clear timer counter */
752        *T2PR = timer2_per;              /* set timer period */
753
740        *T2CON = 0xD740;                 /* configure T2CON register */
755    /*
756     bit 15-14     11:     stop immediately on emulator suspend
757     bit 13        0:      reserved
758     bit 12-11     10:     10 = continous-up count mode
759     bit 10-8      111:    111 = x/128 prescaler
760     bit 7         0:      T2SWT1, 0 = use own TENABLE bit
761     bit 6         1:      TENABLE, 1 = enable timer
762     bit 5-4       00:     00 = CPUCLK is clock source
763     bit 3-2       00:     00 = reload compare reg on underflow
764     bit 1         0:      0 = disable timer compare
765     bit 0         0:      SELT1PR, 0 = use own period register
766    */
767
768    /*** Other setup ***/
769        LED_index = 0x0001;              /* initialize the LED index */
770
771    /*** Setup the core interrupts ***/
772        *IMR = 0x0000;                   /* clear the IMR register */
773        *IFR = 0x003F;                   /* clear any pending core interrupts */
774        *IMR = 0x0004;                   /* enable desired core interrupts */
775
776    /*** Setup the event manager interrupts ***/
777        *EVAIFRA = 0xFFFF;               /* clear all EVA group A interrupts */
778        *EVAIFRB = 0xFFFF;               /* clear all EVA group B interrupts */
779        *EVAIFRC = 0xFFFF;               /* clear all EVA group C interrupts */
780        *EVAIMRA = 0x0000;               /* enable desired EVA group A interrupts */
781        *EVAIMRB = 0x0001;               /* enable desired EVA group B interrupts */
782        *EVAIMRC = 0x0000;               /* enable desired EVA group C interrupts */
783
784        *EVBIFRA = 0xFFFF;               /* clear all EVB group A interrupts */
785        *EVBIFRB = 0xFFFF;               /* clear all EVB group B interrupts */
786         *EVBIFRC = 0xFFFF;              /* clear all EVB group C interrupts */
787         *EVBIMRA = 0x0000;              /* enable desired EVB group A interrupts */
788         *EVBIMRB = 0x0000;              /* enable desired EVB group B interrupts */
789         *EVBIMRC = 0x0000;              /* enable desired EVB group C interrupts */
790
791    /*** Enable global interrupts ***/
792        asm(" CLRC INTM");               /* enable global interrupts */
793
794    /*** Proceed with main routine ***/
795        while(1);                        /* endless loop, wait for interrupt */
796
797    }                                    /* end of main() */
798
```

```
799     /********************** INTERRUPT SERVICE ROUTINES *********************/
800     interrupt void timer2_isr(void)
801     {
802
803          *EVAIFRB = *EVAIFRB & 0x0001;     /* clear T2PINT flag */
804
805     /*** Sequence the LED bank on the LF2407 EVM ***/
806          LED = LED_index;                  /* light the LEDs */
807          LED_index = LED_index << 1;       /* left shift LED index */
808          if(LED_index == 0x0010) LED_index = 0x0001;   /* reset LED index */
809
810     /*** Toggle the IOPC0 pin ***/
811          *PCDATDIR = *PCDATDIR ^ 0x0001;   /* XOR the IOPC0 bit to toggle the pin */
812
813     }
```

```
/*****************************************************************
* Filename: example_c.cmd                                        *
*                                                                *
* Author: David M. Alter, Texas Instruments Inc.                 *
*                                                                *
* Last Modified: 03/14/01                                        *
*                                                                *
* Description: C code linker command file for LF2407 DSP.        *
*****************************************************************/


MEMORY
{
  PAGE 0:    /* Program Memory */
    VECS:           org=00000h,   len=00040h   /* internal FLASH */
    FLASH:          org=00044h,   len=07FBCh   /* internal FLASH */
    EXTPROG:        org=08800h,   len=07800h   /* external SRAM  */

  PAGE 1:    /* Data Memory */
    B2:             org=00060h,   len=00020h   /* internal DARAM */
    B0:             org=00200h,   len=00100h   /* internal DARAM */
    B1:             org=00300h,   len=00100h   /* internal DARAM */
    SARAM:          org=00800h,   len=00800h   /* internal SARAM */
    EXTDATA:        org=08000h,   len=08000h   /* external SRAM  */
}

SECTIONS
{
/* Sections generated by the C-compiler */
        .text:  >   FLASH      PAGE 0   /* initialized */
       .cinit:  >   FLASH      PAGE 0   /* initialized */
       .const:  >   B1         PAGE 1   /* initialized */
      .switch:  >   FLASH      PAGE 0   /* initialized */
         .bss:  >   B1         PAGE 1   /* uninitialized */
       .stack:  >   SARAM      PAGE 1   /* uninitialized */
      .sysmem:  >   B1         PAGE 1   /* uninitialized */

/* Sections declared by the user */
     vectors:  >   VECS       PAGE 0   /* initialized */
}
```

Appendix C Addendum to Getting Started in C and Assembly Code With the TMS320LF240x DSP

C.1 Introduction

The C-code example has been modified to include the real-time monitor (RTM) feature of the C24x debugger tools, specifically C2xx Code Composer v4.1x, and also the newer TMS320C2000™ Code Composer Studio™ v2.xx (henceforth, these tools are collectively referred to as Code Composer Studio with no version designation)[3]. The RTM is useful for monitoring and changing code variables in Code Composer Studio while the DSP is running. The RTM runs on the DSP as a background task off the lowest priority CPU interrupt. It allows itself to be interrupted by any other enabled interrupt, and therefore provides minimal intrusion to the execution of the main application code.

Code Composer Studio provides a tutorial on the RTM, and it is not intended for this addendum to provide detailed information on it. The objective here is to provide working code that uses the RTM, and also some step-by-step instructions on how to activate the RTM in Code Composer Studio. These are typically the most difficult issues when learning a new tool feature such as the RTM. If you would like to access the RTM tutorial in Code Composer v4.1x, click Help→Tutorial→Code Composer Tutorial, and then select “Real-Time Emulation” from the contents list on the help screen. If using Code Composer Studio v2.xx, click Help→Contents, and then select “Real-Time Emulation” from the contents list on the help screen.

C.2 RTM Example Program

The RTM example program consists of the following files:

- **cvecs_rtm.asm:** Interrupt vector table
- **example_rtm.c:** Main program
- **example_rtm.cmd:** Linker command file
- **example_rtm.mak:** C2xx Code Composer v4.1x project file
- **example_rtm.pjt:** C2000 Code Composer Studio v2.xx project file

In addition, the following two files are needed from the Code Composer Studio installation:

- **c200mnrt.i:** Real-time monitor include file
- **c200mnrt.asm:** Real-time monitor assembly source code

If you are using Code Composer v4.1x, these files can be found in the \tic2xx\c2000\monitor directory. If using Code Composer Studio v2.xx, these files will be in the \ti\c2400\monitor directory. Both of the given paths are assuming that you used the default installation path when installing the tools.

Finally, the following file is used from the C Language Example Program that came with the original application report:

- **f2407_c.h:** Header file containing peripheral register address definitions

3. If you are unsure of what tools version you are using, start Code Composer Studio and click HELP→ABOUT. A window will pop up that will tell you your software version.

To create the RTM example, the following modifications were made to the C language example program files that came with the original application report:

1. The file *cvecs_rtm.asm* was created from *cvectors.asm*. Modification was made to include the needed RTM interrupt vectors. The RTM uses hardware interrupt vector 7, and also software interrupt vector 19. The interrupt routines are actually embedded directly into the interrupt vector table, using up several of the locations normally used by software interrupts. The direct embedding allows for maximum response speed of the RTM, and minimal intrusion into your main application code. If the software interrupts used by the RTM are needed for user code, you can avoid embedding the RTM interrupt routines into the vector table by changing the value for the assembly time constant, MON_VECTOR, in the file *c200mnrt.i* from the present value “MON_VECTOR_MACRO” to “MON_VECTOR_BRANCH.”

   Note that the C24x code generation tools (specifically, the linker) require assembly source filenames to be, at most, 10 characters in length, or an error is produced. This is why the filename was chosen as *cvecs_rtm.asm*, and not *cvectors_rtm.asm*.

2. The file *example_rtm.c* was created from *example_c.c*. The modifications made were:
   a. The RTM interrupt (core INT7) was enabled in the IMR register.
   b. The MON_RT_CNFG() function was called to initialize the RTM. This was done using an inline assembly CALL instruction since MON_RT_CNFG was not defined in the RTM source code with a leading underscore, and hence cannot be called using C code.
   c. The constant REALTIME was defined using a #define statement to allow the RTM to easily be turned on or off prior to compiling the project (set REALTIME to 0 to turn off, then re-compile). Turning off the REALTIME monitor is necessary when you are finished debugging and want to make a normal embedded software build. All code required for the RTM is conditionally compiled based on the value of REALTIME, so setting REALTIME to 0 effectively removes all traces of the RTM from your code.
   d. A simple software counter from 0 to 9 was added to timer2_isr() for RTM display purposes in this application report only (this modification was not required to operate the RTM). The global variable name is *count*. This variable will be used later to illustrate the RTM capabilities with the watch window and with graphs.
   e. Two assembly language NOP instructions were added inline to the endless while(1) loop in main() for RTM test and illustration purposes in this application report only (this modification was not required to operate the RTM). These NOP’s will be used later to illustrate the background single-stepping capability of the RTM.

3. The file *example_rtm.cmd* was created from *example_c.cmd*. Modification was made to link in the required sections for the RTM. Note that the section *mon_pge0* must be linked to the first 0x80 words of data space (i.e., direct addressing page 0).

C.3 Building and Running the RTM Example Program

The RTM example program can be run directly on the LF2407 EVM or eZdsp™ LF2407 board without modification. Here’s what to do to get the real-time monitor program working:

1. Place all the required files in a single directory. This includes the files *c200mnrt.i*, *c200mnrt.asm*, and f2407_c.h, which should be be copied from their existing locations.
2. Start Code Composer Studio.

3. Click PROJECT→OPEN to open the project (this is *example_rtm.mak* for Code Composer v4.1x, or *example_rtm.pjt* for Code Composer Studio v2.xx). Any errors are probably due to a file path problem. This is easy to fix. Simply remove the problematic file from the project, and then add the file back in.
4. Build the project by clicking PROJECT→BUILD.
5. If you do not have automatic loading after build enabled, manually load *example_rtm.out* into the DSP using FILE→LOAD PROGRAM.
6. Reset the DSP. (in Code Composer v4.1x, click DEBUG→ RESET DSP, in Code Composer Studio v2.xx click DEBUG→RESET CPU).
7. Click on DEBUG→RUN (or hit F5). This will run the DSP up to and including the RTM init function.
8. Click DEBUG→HALT (or hit Shift-F5). The program should halt inside the RTM init function (just after the label MON_WAIT).
9. Click DEBUG→REAL TIME MODE, which will check-mark this option. This activates the RTM in Code Composer Studio. It was necessary to perform steps 6–8 prior to doing this step, as you need to run the RTM init function before activating the RTM.
10. Click DEBUG→RUN (or hit F5). The DSP will now be running main() with the RTM active.
11. You can now open whatever display windows you wish. A simple test is to open a watch window, and add the variable *count*.
12. You can also open a graph of the variable count. Click VIEW→GRAPH→TIME/FREQUENCY. Configure the following graph properties:

Start Address	&count
Acquisition Buffer Size	1
Display Data Size	100
DSP Data Type	16-bit unsigned integer
Sampling Rate (Hz)	10

All other graph properties can be left at their default values. Note the ampersand in front of count in the starting address, which means address of count. The Acquisition Buffer Size is set to 1, since we are updating the graph with only one point per RTM interrupt. The Display Data Size is set to whatever you would like the x-axis length of the graph to be, for example, 100 data points. The DSP Data Type is set to the data type of count. Finally, the Sampling Rate is set to 10, since we will be setting the RTM update rate to 100 ms in the next step.

13. You now need to tell Code Composer Studio what windows you want updated with the RTM. There are two ways to do this:
    a. Global continuous refresh – all windows are updated. Click VIEW→REAL-TIME REFRESH OPTIONS, and check the box for global refresh. The minimum update interval is 100 ms, although you can make it slower if you want (since the RTM has limited throughput bandwidth).

b. Selective refresh – assuming global refresh is unchecked, right-click the window or graph you want to update, and select CONTINUOUS REFRESH. The refresh interval is still specified on the VIEW→REALTIME REFRESH OPTIONS menu.

   At this point, the watch window and the graph should be updating in real time while the DSP is executing the code. The graph will look like a simple ramp wave, as you are graphing an incrementing counter from 0 to 9.

14. To illustrate the background single-stepping capability of the RTM, click DEBUG→HALT. You will notice that your graph and watch window are still updating, which means that the timer2 interrupt is still being serviced. Now click DEBUG→STEP (or hit the F8 key) a few times. You will see that you are single-stepping through the inline NOP instructions (which are in the software background) while your timer2 interrupt is still being serviced in the foreground. This is convenient for debugging background tasks in your code while your time-critical interrupts are still being serviced and keeping your closed-loop control system running.
15. To stop the interrupts from being serviced as well (i.e., to completely halt the DSP), click DEBUG→REAL TIME MODE (to uncheck the Real Time Mode).
16. In addition to viewing variables in real time, you can also use the RTM to change memory (variables) without halting the DSP. This is great for tuning controllers or changing set points, etc. All you need to do is view the variable in a watch window (or directly in a memory window), and change it like you normally would in Code Composer Studio. As an example, let's change the rate at which the timer2 interrupt occurs. Add the expression "*0x7407,u" to the watch window. 0x7407 is the address of the T2PR (Timer 2 Period) register, and the comma u indicates to display this as an unsigned quantity. The code had set this register to 58594 (which corresponds to a 250-ms interrupt with a 30-MHz CPUCLK, or a 188-ms interrupt with a 40-MHz CPUCLK). Double-click this entry in the watch window and change the value to 30000. You will see the LED bank and the red LED DS1 (or DS2 on eZdsp™ LF2407 board) blink at roughly double the rate that they were previously blinking at. This confirms that you have successfully changed the period of the timer 2 interrupt.

IMPORTANT NOTICE AND DISCLAIMER

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

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

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

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