CHAPTER 10

Zynq-7000 Timers and Watchdogs

10.1 Introduction

With the basic modular structure implemented in the last chapter, we can now start building a more practical embedded program. A critical concept in all processor systems is timing, and this chapter covers the options available in the Zynq-7000. In the related project, the private timer for CPU0 in the Cortex-A9 MPCore will be used to control timing, instead of the crude for-loop method used in earlier programs. In addition, the private watchdog timer for CPU0 will be used to monitor the running program.

To provide more insight into the state of the embedded application, the LEDs on the Zybo-Z7-20 will also be used in a different manner (see also Figure 10.1):

- LED0 is used to show the result of system initialisation — the LED switches on if initialisation is successful, and blinks periodically if initialisation fails.
- In the main code, LED1 is used to show that task 1 is active.
- Similarly, LED2 is used to show that task 2 is active
- LED4 is driven by the WDT code to show that the system is "alive".

As can be seen in the above description, we are also using the term "task" when talking about the application. There are just two tasks in this system, and both are very simple:

- Task 1: LED1 will flash if SW1 is off, otherwise it will remain on.
- Task 2: LED2 will flash if SW2 is off, otherwise it will remain on.

When speaking in terms of tasks we get dangerously close to the terminology used in operating systems, real-time or otherwise. However, the reader should note that the tasks in this text will be very simple, with non-critical timing. The hierarchy introduced in the previous

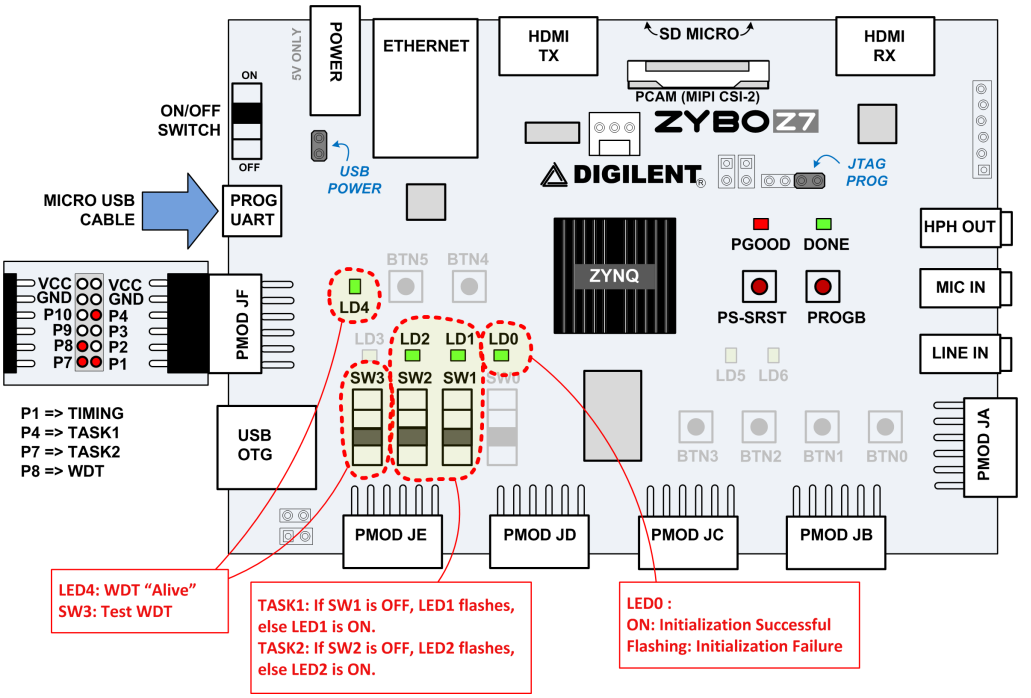

Figure 10.1. Board details for software project 4: LED0 = Initialisation status; LED4 = System "alive"; LED1 = Task 1; LED2 = Task2; SW3 = Watchdog timer test option; PMOD JF = test signal header

chapter is not necessarily intended to be used in large, time-critical projects, and if the developer does use the structure for such projects, the system should be thoroughly tested. Linux or an RTOS may offer a better solution for complex projects, but such decisions are beyond the scope of this book.

The project in this chapter also sees the introduction of a useful test feature, where the GPIO signals on Pmod JF are used to monitor the program flow using a logic analyser. This method would not necessarily be used in large-scale software development — the CoreSight Trace functionality discussed in Section 3.7.1.3 is a much better option - but it is still a very useful (and much cheaper!) method when trying to learn about a system. More details on the proposed test functionality will be given in Section 10.3.4. To start, though, the main timing options in the Zynq-7000 Adaptive SoC will be discussed.

10.2 Main Concepts

Several timing and watchdog features are available in the Zynq-7000 architecture, as shown in Figure 10.2. These are located in the Cortex-A9 MPCore, the Application Processing Unit, and the programmable logic.

The options are summarised by location as follows:

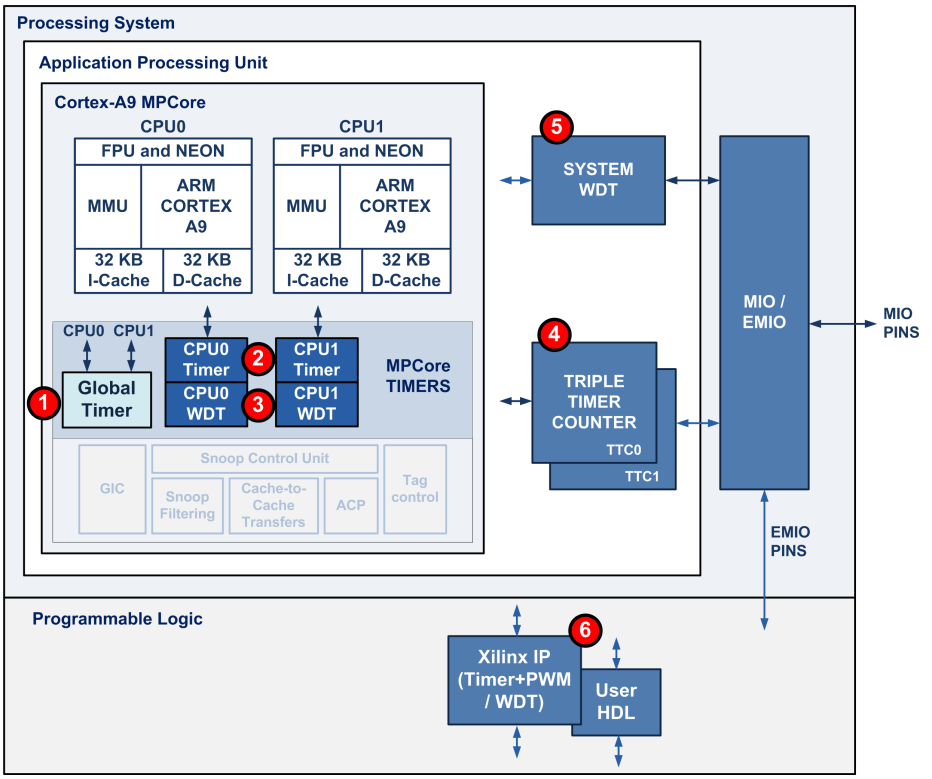

Figure 10.2. Timer and watchdog options in the Zynq-7000

ARM Cortex-A9 MPCore

1. A single 64-bit Global Timer is available which can be accessed by all processors in the MPCore cluster.
2. Each core in the cluster can access its own private 32-bit timer. This option is also known as the SCU Timer.
3. Each core in the cluster can also access its own private 32-bit WDT. This option is also known as the SCU WDT.

Zynq-7000 APU

4. Two 16-bit triple-timer counters (TTC0, TTC1) are available in the APU.
5. A 24-bit system watchdog timer (SWDT) is also available in the APU.

Programmable Logic Timing Options

6. Timing functionality can be added to the programmable logic in the form of Xilinx (or third-party) Timer/PWM/WDT IP blocks, or user-designed HDL modules.

In this chapter, the MPCore options (the global timer, private timer, and private watchdog timer) will receive the most attention, along with a summary of the SWDT. In the related project, the CPU0 private timer will be used to control program timing, and the CPU0 WDT will be used to reset the system in the event of failure. In the next chapter, the application will be improved by using the CPU0 timer in conjunction with the Zynq-7000

interrupt system. The triple timer counter will then be the focus of Chapter 12, where it replaces the CPU0 timer to control program timing.

10.2.1 MPCore Timers

10.2.1.1 MPCore Global Timer

The MPCore global timer is a 64-bit incrementing counter that can be accessed by all cores in the cluster. It includes comparator logic and auto-incrementing functionality, plus an associated interrupt for each core. The auto-incrementing logic ensures that the relevant comparator reference value is automatically updated after an interrupt, removing the burden from the software developer. The comparator can also be used in single-shot mode i.e., the auto-incrementing functionality can be disabled.

Zynq-7000 Address	Register
0xF800_0200	64-bit Count Register (Lower 32)
0xF800_0204	64-bit Count Register (Upper 32)
0xF800_0208	Counter Control Register
0xF800_020C	Interrupt Status Register [1]
0xF800_0210	64-bit Comparator Register (Lower 32) [2]
0xF800_0214	64-bit Comparator Register (Upper 32) [2]
0xF800_0218	32-bit Auto-Increment Register

[1] Banked register for all Cortex-A9 cores in device.
[2] Each Cortex-A9 core has its own register.

Table 10.1. MPCore Global Timer Counter Registers, showing Zynq-7000 addresses

The register map for the global timer is shown in Table 10.1. In the Zynq-7000 TRM [39] the register settings can be found in Appendix B.24, 'Application Processing Unit (mpcore)', starting at address 0xF800_0200. Note that low-level drivers are not specifically provided for the global timer; instead, some minimal APIs can be found in **`xtime_l.c/h`** in the BSP. (Direct register accesses can also be used.) Further details on the Global Timer can be found in Section 8.3 of the Zynq-7000 TRM [39], and Section 4.3 of the ARM Cortex-A9 MPCore TRM [34].

Some final points about the global timer:

- The interrupt is a rising-edge, private peripheral interrupt (PPI), with ID #27 (these terms will become clearer in the next chapter).
- The global counter does not stop incrementing when the system is halted due to a debug event, unlike the private timer and watchdog (which are suspended).

10.2.1.2 MPCore Private Timers

The private timer for each core has the following features:

- It is a 32-bit decrementing counter that generates an interrupt when it reaches zero.
- It includes an 8-bit prescaler which allows a wider timing range to be used (with lower resolution).
- The available modes are single-shot or auto-reload.

- The starting values are configurable.
- The count is suspended when in software debug mode.
- The interrupt is a rising-edge PPI, with ID #29.

These timers are not implemented in the Cortex-A9 processor itself, but instead are more closely associated with the SCU. Because of this, the private timers are also known as SCU timers, and this term will often be used in the text.

Zynq-7000 Address	Register
0xF800_0600	Timer Load Register
0xF800_0604	Timer Counter Register
0xF800_0608	Timer Control Register
0xF800_060C	Timer Interrupt Status Register

Table 10.2. MPCore Private Timer Registers, showing Zynq-7000 addresses

The register map for the private timer is shown in Table 10.2. In the Zynq-7000 TRM [39] the register settings can be found in Appendix B.24, 'Application Processing Unit (mpcore)', starting at address 0xF800_0600. In contrast to the global timer, drivers are provided by Xilinx for the SCU timer (**`xscutimer.c/h`** and related lower-level files), and they will be used in the project associated with this chapter. Further details on the CPU private timers can be found in Section 8.2 of the Zynq-7000 TRM [39], and Sections 4.1 and 4.2 of the ARM Cortex-A9 MPCore TRM [34].

10.2.2 Watchdog Timers

Embedded systems are often very closed in nature, and human interaction can be difficult (or even impossible) when critical issues occur. Effective techniques are required to ensure that a system can recover without user intervention, and the watchdog timer is one of the more common methods. The basic premise is that the WDT maintains a timed countdown which is periodically refreshed by the application. If, however, the system runs into difficulties, the application might not have the ability to send the "refresh" signal to the WDT, and the timer will expire. The WDT will then initiate a system reset, with the aim of restarting the application or putting the system into a safe state.

There are two WDT options in the Zynq-7000 processing system: the MPCore Private WDT and the APU System WDT. Both are discussed below.

10.2.2.1 MPCore Private Watchdog Timers

A private WDT is available for each core in the MPCore cluster, and the functionality is very similar to the SCU Timers discussed earlier. In fact, the feature list in Section 10.2.1.2 also applies to the watchdog timers, and the WDT can even be configured in timer-only mode. The difference, obviously, is that the WDT can be configured in Watchdog mode, where a system reset is initiated if the count-down reaches zero.

The register map for the private WDT is shown in Table 10.3. In the Zynq-7000 TRM [39] the register settings can be found in Appendix B.24, 'Application Processing Unit (mpcore)', starting at address 0xF800_0620. Drivers are provided by Xilinx for the SCU timer (**`xscuwdt.c/h`**

and related lower-level files), and they will be used in the project associated with this chapter. Further details on the CPU private watchdog timers can be found in Section 8.2 of the Zynq-7000 TRM [39], and Sections 4.1 and 4.2 of the ARM Cortex-A9 MPCore TRM [34].

Zynq-7000 Address	Register
0xF800_0620	Watchdog Load Register
0xF800_0624	Watchdog Counter Register
0xF800_0628	Watchdog Control Register
0xF800_062C	Watchdog Interrupt Status Register
0xF800_0630	Watchdog Reset Status Register
0xF800_0634	Watchdog Disable Register

Table 10.3. MPCore Watchdog Timer Registers, showing Zynq-7000 addresses

The private WDT is an internal watchdog, i.e., it is part of the component it is monitoring. This comes with the risk that the WDT might not function correctly if the core logic fails, and the application will not restart. The System WDT offers some advantages in this regard, and it will be briefly summarised next.

10.2.2.2 Zynq-7000 APU System Watchdog Timer

The System Watchdog Timer (SWDT) is potentially a better solution than the SCU WDT for three main reasons:

1. It is independent of the ARM MPCore, thus removing the limitations of a processor monitoring itself.
2. It can be clocked from an external source, meaning that it is immune (or at least more resistant) to internal clock issues that directly affect an internal WDT.
3. The associated reset signal can be routed to an external device or to the programmable logic, meaning that more options are available for system restart.

The SWDT in the Zynq-7000 includes the following features:

- It contains a 24-bit counter.
- The clock source is selectable from the following options: internal PS clock; internal clock from PL; or external clock via MIO.
- An interrupt can be generated when a timeout occurs.
- Three different reset signals are available (PS, PL, MIO).
- It has a programmable time-out range of 330us to 687.2s (@100MHz).
- A pulse waveform can be used as the time-out signal.

The register map for the private WDT is shown in Table 10.4. In the Zynq-7000 TRM [39] the register settings can be found in Appendix B.31, 'System Watchdog Timer (swdt)', starting

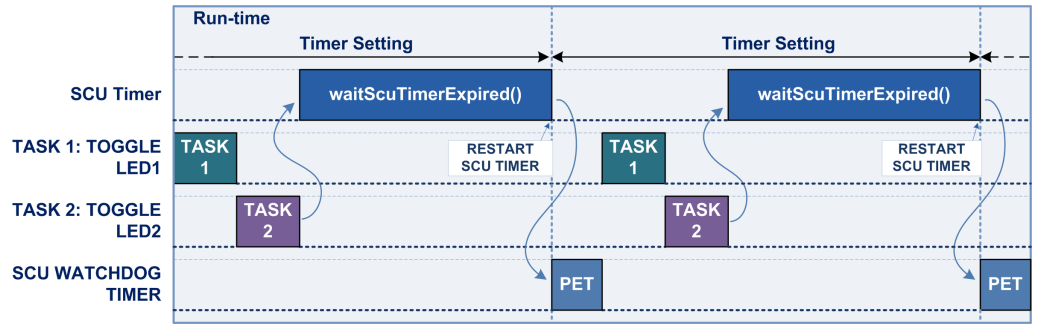

Figure 10.3. Program timing for software project 4

at address 0xF800_0500. The Xilinx drivers for the SWDT are **`xwdtps.c/h`**, and further details on the device can be found in Section 8.4 of the Zynq-7000 TRM [39].

Zynq-7000 Address	Register
0xF800_5000	Watchdog Zero Mode Register
0xF800_5004	Watchdog Counter Control Register
0xF800_5008	Watchdog Restart Key Register
0xF800_500C	Watchdog Status Register

Table 10.4. APU System Watchdog Timer Registers, showing Zynq-7000 addresses

10.2.3 Program Timing

With the discussion of the MPCore timers complete, some related principles will now be used to improve the timing in our application. In the project for the current chapter, the SCU timer (private timer for CPU0) will control program timing, and the SCUWDT (private watchdog for CPU0) will be used as the fail-safe mechanism in the code.

Figure 10.3 shows the expected program flow. The term "task" is now used for any job that the main application carries out — here, task 1 toggles LED1, and task 2 toggles LED2. When the tasks have executed, a function called **`waitScuTimerExpired()`** which waits until the SCU Timer has reached a specified value. waits until the SCU Timer has reached a set value. When the timer expires, the WDT is "serviced" i.e., it is given the signal to restart. (The term "pet" is used in Figure 10.3.) At a broader level, this indicates that there have been no critical failures in the system, and the code returns to task 1, repeating the sequence..

This pattern will become clearer in the code discussion later in this chapter. Before that, the hardware and software details of the project will be outlined.

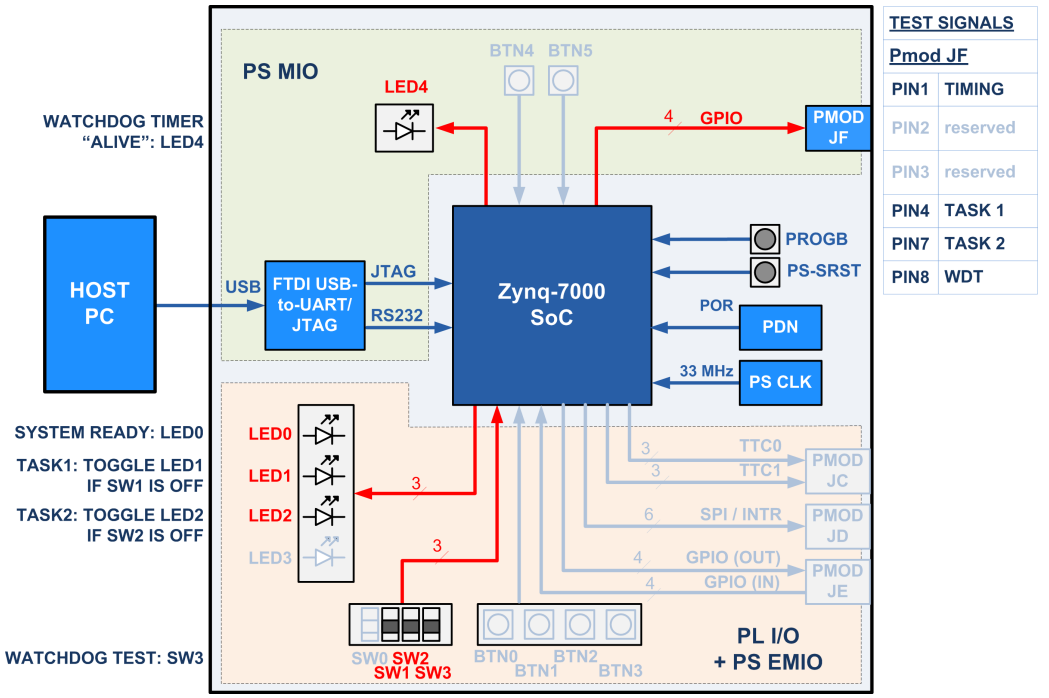

Figure 10.4. Hardware block diagram for software project 4

10.3 Project Details

Table 10.5 summarises the project details. In the guide related to this book, the steps required to run the program can be found in the section called "Software Project 4: SCU Timing".

Requirement	Details
SDK Project	sw_proj4
Project Main File	sw_proj4_main.c
Host SW	Any terminal program e.g. Tera Term, PuTTY, SDK Console.
Extra hardware/electronics	1x Digilent Pmod 12-pin Test Header (Pmod TPH2)
Test Equipment	Oscilloscope or logic analyser e.g. Analog Discovery 2

Table 10.5. Project details for software project 4

10.3.1 System Overview

The proposed system is summarised below (refer also to Figure 10.4):

- As always, initialisation runs first. If initialisation completes successfully, LED0 will be switched on, and the program will continue to the run-time section.

- If initialisation is unsuccessful, LED0 will flash and the program will enter a loop where it might be possible to carry out simple debugging.
- In the run-time section, two simple tasks will execute. LED1 will toggle if SW1 is off (task1), and LED2 will toggle if SW2 is off (task 2).
- When the tasks are complete, the SCU Timer will be used to delay the loop for a fixed amount of time.
- After the timer delay, the WDT will restart and LED4 will toggle, giving a visual indication that the program is running as expected. The run-time section then repeats.
- The WDT phase also includes a test option: if SW3 is on, the WDT will not be serviced. This causes a WDT timeout and subsequent system reset.

10.3.2 Zynq-7000 Block Diagram

In comparison to the project in the previous chapter, the simplified Zynq-7000 block diagram (Figure 10.5) is updated as follows:

1. The private timer and watchdog for CPU0 are used.
2. GPIO (MIO) signals are routed to Pmod JF, where they are used for test visibility.
3. Buttons 4 and 5 are NOT used (MIO50 and MIO51).
4. LED[0-2] and SW[1-3] are used (see Section 10.3.1 for the related functionality).

10.3.3 Software Details

Figure 10.6 shows the updated software block diagram, where the private timer (SCU Timer) and watchdog (SCU WDT) for CPU0 have been added to the project. Table 10.6 summarises the base addresses and drivers in the system.

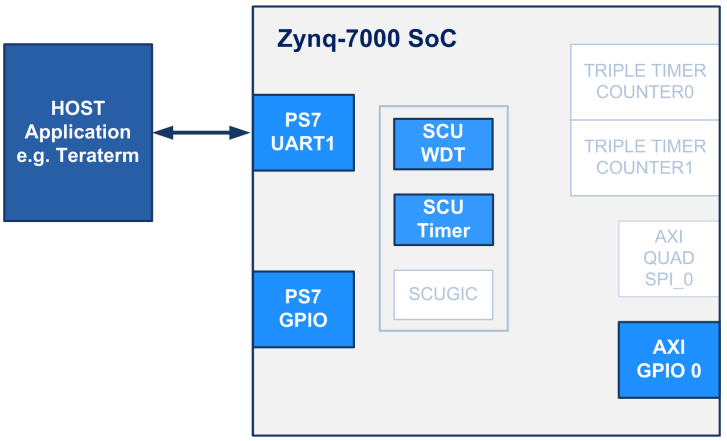

Figure 10.6. Project software block diagram for software project 4

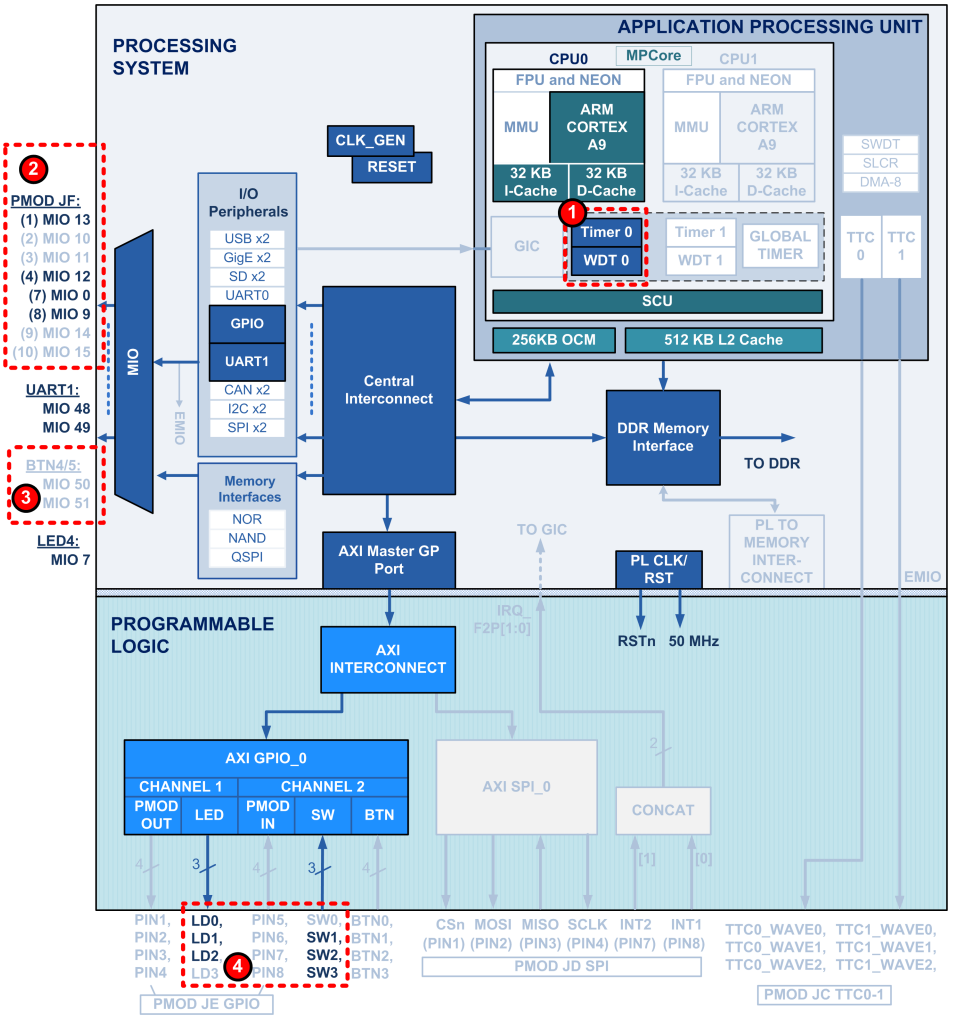

Figure 10.5. Simplified Zynq-7000 block diagram for software project 4

Block	Base Address	Driver [1]
AXI GPIO 0	0x4120_0000	gpio_v4_3
PS7 GPIO	0xE000_A000	gpiops_v3_5
PS7 UART1	0xE000_1000	n/a (stdin/stdout)
SCU Timer	0xF8F0_0600	scutimer_v2_1
SCU Watchdog Timer	0xF8F0_0620	scuwdt_v2_1

[1] Table shows drivers used with SDK 2019.1. For different tools/versions, use the most-up to date drivers.

Table 10.6. Block base addresses and drivers for software project 4

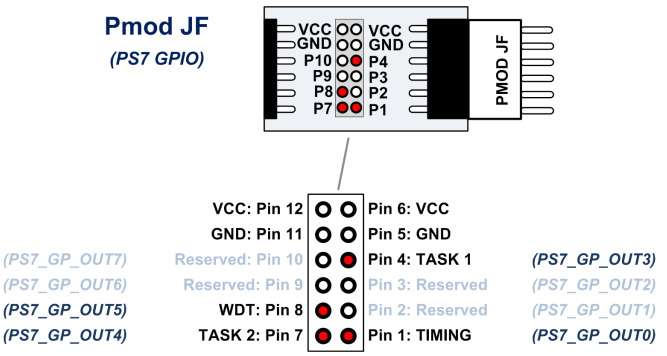

Figure 10.7. Pmod JF test signals for software project 4

10.3.4 Test Connections

A major new feature in this project is the addition of test signals to the code, allowing program execution to be monitored using a suitable logic analyser. (As only four test signals are used in this project, a 4-channel oscilloscope could also be used.) For test visibility, the PS7 GPIO signals that were defined in Section 8.4.2.3.1 will be used. By design, these signals connect directly to Pmod JF on the Zybo platform.

Figure 10.7 shows the test connections for this first project, where the aim is to monitor program timing and task activity. Further information on test connectivity is given in Table 10.7. Note that the code location for each test signal is also listed in the table — this is important as it can be easy to lose track of where this optional test signal code has been added in a large project. Details on how to drive these test signals will be given later in the code discussion.

Test Signal	Code Section	PS7 GPIO	Pmod Pin
Timing	main()	PS7_GP_OUT0	Pin 1
Task 1	main()	PS7_GP_OUT3	Pin 4
Task 2	main()	PS7_GP_OUT4	Pin 7
Watchdog Timer	main()	PS7_GP_OUT5	Pin 8

Table 10.7. Test signal details for software project 4

The test signals can be summarised as follows:

1. Timing: Asserted while waiting for the SCU Timer to expire, and then de-asserted.
2. Task 1: Asserted when "Task1" is executing, then de-asserted.
3. Task 2: Asserted when "Task2" is executing, then de-asserted.
4. WDT: Asserted when the watchdog timer is being serviced, then de-asserted.

At the end of this chapter (Section 10.6), the program will be launched and the test signals viewed on a logic analyser. Before that, the updated code will be discussed in detail, starting with the program hierarchy updates in the next section.

10.4 Program Hierarchy Updates

In the layered hierarchy introduced in the last chapter, the main and system files are located at the top level, and a driver interface layer exists between the top-level and the Xilinx driver layer. The driver interface is effectively a hardware abstraction layer, providing a prescribed range of functions that can be used by other files in the system. In this section we will see how easy it is to scale up the design — if new components are required, we simply add the files to the driver interface layer, and modify the top-level files appropriately.

There are two new components in this project: the SCU timer and the SCU WDT. Scaling up the program can be done using the following simple steps (see also Figure 10.8):

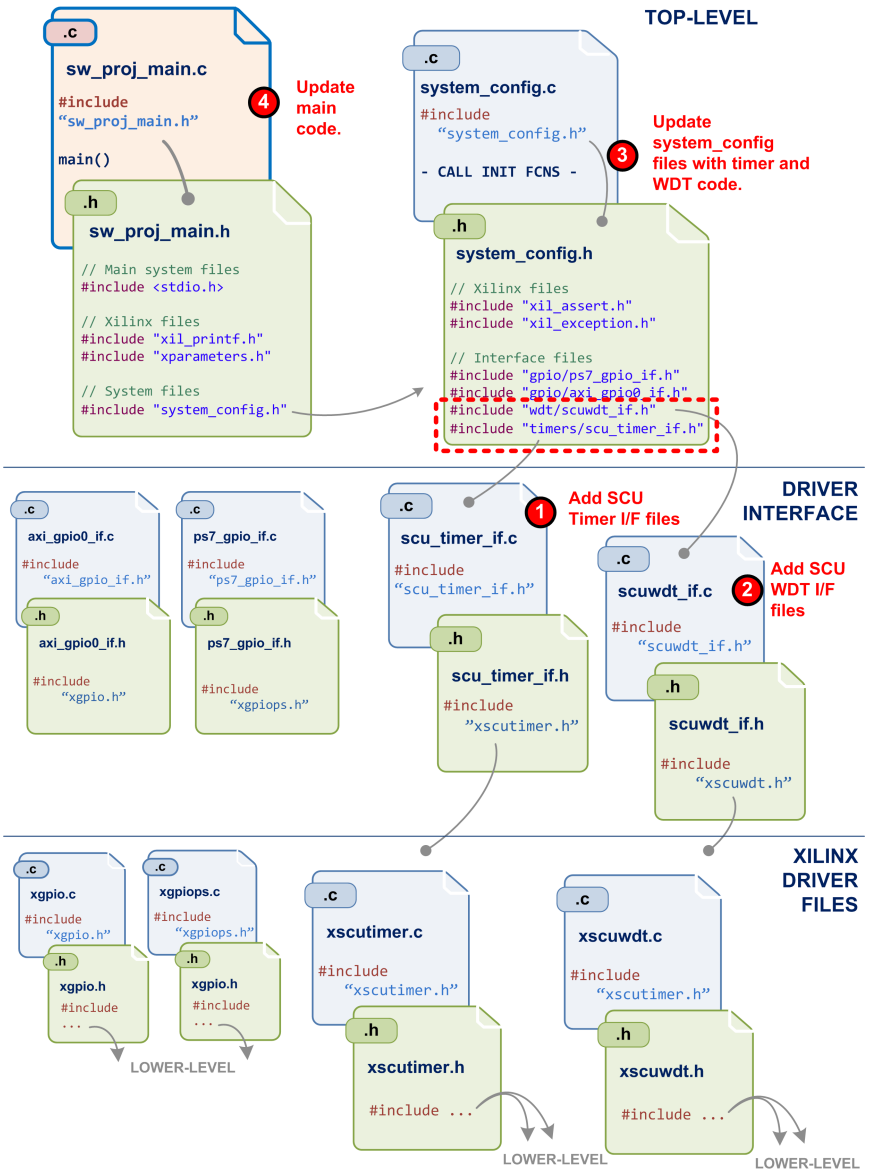

Figure 10.8. Hierarchical structure for software project 4

1. Add SCU Timer interface files to the driver interface layer.
2. Add SCU WDT interface files to the driver interface layer.
3. Update the system configuration files to call the initialisation functions for the SCU timer and SCU WDT.
4. Update **`main()`** with the following functionality:
    a. When the initialisation function returns, LED0 turns on to indicate successful initialisation, and flashes to indicate unsuccessful initialisation.
    b. In the main code, LED1 toggles if SW1 is off (task1); otherwise it remains on. Similarly, LED2 toggles if SW2 is off (task2); otherwise it remains on.
    c. After the tasks run, the WDT timer is serviced.
    d. SW3 is used to test the WDT.

Test signal functionality will also be added to the system, as just discussed in Section 10.3.4, so that the program flow can be monitored using a logic analyser.

10.5 Code Discussion

The code updates in the project will now be discussed, starting at the driver interface layer and moving up to the top-level. As before, only essential code fragments will be given in the discussion, and the project source files should be referred to for complete details.

For quick reference, the source and header files added/modified are listed below:

- SCU Timer interface header file (xscu_timer_if.h): Section 10.5.1.1.1
- SCU Timer interface source file (xscu_timer_if.c): Section 10.5.1.1.2
- SCU Watchdog interface header file (xscuwdt_if.h): Section 10.5.1.2.1
- SCU Watchdog interface source file (xscuwdt_if.c): Section 10.5.1.2.2
- System Configuration header file (system_config.h): Section 10.5.2.1.1
- System Configuration source file (system_config.c): Section 10.5.2.1.2
- Main top-level files (sw_proj4_main.h/.c): Section 10.5.2.2

10.5.1 Updates to the Driver Interface Layer

The SCU timer and the SCU watchdog timer are the two new components in the driver interface layer. The SCU timer will be discussed first in this section, and the WDT will be covered in Section 10.5.1.2.

10.5.1.1 SCU Timer

The SCU timer is used to control the rate at which the application runs, and while the related code will be as self-contained as possible, the following functions must be made available to other files:

1. An initialisation function (used by **`system_config`**).
2. A function to start the timer (used by **`main()`**).
3. A function to wait for the timer to expire (used by **`main()`**).

10.5.1.1.1 Header File: xscu_timer_if.h

In the header file, we start by including the low-level Xilinx file **xscutimer.h**.

```
#include "xscutimer.h"
```

Following this, we have the constant definitions.

```
#define SCUTIMER_DEVICE_ID       XPAR_PS7_SCUTIMER_0_DEVICE_ID
#define SCU_TIMER_LOAD_VALUE     (0x00001A0A) // 20us @ 667MHz/2
```

In the above code, we first set the device ID to the associated value in **xparameters.h**. The next parameter, **SCU_TIMER_LOAD_VALUE**, is more interesting, as it controls the program timing. In the Cortex-A9, the SCU Timer runs at half the CPU frequency, and in our hardware design, the CPU frequency is 667MHz (or a period of 1.5ns). This results in a timer period of 3ns (i.e., twice the CPU clock period). A small time interval will be used in the project to allow the test signals to be easily viewed using a logic analyser — by experiment, a setting of 20us is appropriate. To get the **SCU_TIMER_LOAD_VALUE** we simply divide 20us by 3ns to get 6666d, or **0x1A0A** when converted to hex.

After the constant definitions, we just need to add the function prototypes, as shown below. These functions will be described in more detail in the next section.

```
/* Device Initialization */
int xScuTimerInit(void);

/* Interface functions */
void startScuTimer(void);
void waitScuTimerExpired(void);
```

10.5.1.1.2 Source File: xscu_timer_if.c

The SCU Timer interface file follows the layout already outlined for the AXI GPIO interface file in Section 9.3.1.2. To summarise, there are four general steps when writing code for a new driver interface file:

1. Include the associated header file e.g. **#include xscu_timer_if.h**
2. Add static declarations for the device instance and pointer.
3. Write the initialisation function.
4. Add interface functions that can be used by other code.

Steps 1 and 2 are shown in the code fragment below.

```
/* Include Files */
#include "xscu_timer_if.h"

/* Variable Declarations */
static XScuTimer XScuTimerInst;
static XScuTimer *p_XScuTimerInst = &XScuTimerInst;
```

The functions required for steps 3 and 4 will be discussed next, starting with initialisation.

Function: xScuTimerInit()

The initialisation function uses the exact same format as described in Section 8.4.4.2 for the AXI GPIO source code. The required steps are: (1) Device look-up; (2) Driver configuration; (3) Self-test; (4) Project-specific settings. Only the project-specific details for the SCU Timer are listed here, with just two lines of code required:

```
XScuTimer_EnableInterrupt(p_XScuTimerInst);
XScuTimer_LoadTimer(p_XScuTimerInst, SCU_TIMER_LOAD_VALUE);
```

We start by enabling interrupts in the SCU Timer. This might appear confusing, as the main code is based on a polling architecture (i.e., interrupts are not required). However, we can still use the interrupt capabilities of a particular device if it suits our needs, and in this case the best way to check if the SCU timer has expired is to use its interrupt status register. After interrupts are enabled, we then load the timer with the value set earlier in the header file (**SCU_TIMER_LOAD_VALUE**). (Note that the **XScuTimer_*** functions can be found in the Xilinx low-level driver file, **xscutimer.h**.)

Function: startScuTimer()

After the initialisation function is defined, we add a method to start the timer — the low-level Xilinx function, **XScuTimer_Start()**, is used as follows:

```
void startScuTimer(void) {
  XScuTimer_Start(p_XScuTimerInst);
}
```

Function: waitScuTimerExpired()

The second function is more substantial, as it is used to check whether or not the SCU Timer has expired. First, the base address for the SCU Timer must be extracted from the instance pointer, as it is needed later in the function.

```
void waitScuTimerExpired(void)
{
  uint32_t scu_timer_expired;
  /* Extract the base address from the instance pointer */
  uint32_t base_addr = p_XScuTimerInst->Config.BaseAddr;
```

Then the interrupt status register is polled using a **do-while** loop — this terminates when the interrupt **EVENT_FLAG** is set to '1', indicating that the counter has reached zero:

```
  /* 'Spin' in this loop while waiting for the count reach 0 */
  do
  {
    scu_timer_expired = XScuTimer_GetIntrReg(base_addr) &
                                  XSCUTIMER_ISR_EVENT_FLAG_MASK;
  } while (scu_timer_expired == 0);
```

The flag must then be cleared; this is accomplished by writing the event flag mask value back to the interrupt status register:

```
  /* Clear the 'count = 0' bit in the Private Timer Interrupt Register
  * (setting the bit clears it), and call the Xilinx restart function. */
  XScuTimer_SetIntrReg(base_addr, XSCUTIMER_ISR_EVENT_FLAG_MASK);
```

Finally, the timer is restarted.

```
  XScuTimer_RestartTimer(p_XScuTimerInst);
// function end
```

This completes the code for the SCU timer; next up is the SCU watchdog timer.

10.5.1.2 SCU Watchdog Timer

The SCU WDT has the responsibility of monitoring the running application and initiating a system reset if it is not serviced within a set time. The following functions are required:

1. An initialisation function, **`xScuWdtInit()`** (used by **`system_config`**).
2. A function to restart (i.e. service) the timer, **`restartScuWdt()`** (used by **`main()`**).

The source code also includes the ability to display a start-up message in an attached terminal when the program is launched. This uses the reboot status register in the SLCR to show the reason for the current boot (which might be due to a normal POR or a watchdog time-out, for example). The discussion starts, as always, with the header file.

10.5.1.2.1 Header File: xscuwdt_if.h

In the header file, the first action is to include the low-level Xilinx file, **`xscuwdt.h`**.

```
#include "xscuwdt.h"
```

Next, the constants are declared. The two main requirements are the device ID and the WDT load value (the latter being the time-out setting). The WDT is clocked at 111MHz, so the value of **`0x7FFFFFFF`** gives a very large time-out of about 6.5 seconds:

```
/* Constant Definitions */
#define SCUWDT_DEVICE_ID     XPAR_PS7_SCUWDT_0_DEVICE_ID
#define SCUWDT_LOAD_VALUE    (0x7FFFFF) // 6.44s @ 111MHz
```

To use the reboot message functionality mentioned earlier, the system-level control register **`SLCR_REBOOT_STATUS_REG`** and related bit masks are also defined.The reboot status register is at address **`0xF8000258`** in the Zynq-7000 address space. The **`SCUWDT_DEBUG`** parameter is also defined to enable the test functionality — when this is set to **`1`**, the reason

for the current boot will be displayed in the connected terminal when the application is launched.

```
#define SCUWDT_DEBUG  1
#define SLCR_REBOOT_STATUS_REG  0xF8000258U

#define SLCR_RS_SWDT_RST_MASK   0x00010000U
#define SLCR_RS_AWDT0_RST_MASK  0x00020000U
/* Other bit-masks not shown here, see actual file for full details */
```

The function prototypes are declared last in the header file. The first prototype is for the initialisation function, **`xScuWdtInit()`**. Next, we have an interface function, **`restartScuWdt()`**, which is used by higher-level code to service the WDT. Finally, the **`checkRebootStatus()`** function is used for the boot message functionality just discussed above.

```
int xScuWdtInit(void);
void restartScuWdt(void);
void checkRebootStatus(void);
```

10.5.1.2.2 Source File: xscuwdt_if.c

In the WDT source file, we start by including its associated header file **`scuwdt_if.h`**, and then declaring the WDT instance and associated pointer. (By now, it should be seen that these are always the first steps when writing one of our driver interface files.)

```
/* Include Files */
#include "scuwdt_if.h"

/* Variable Declarations */
static XScuWdt XScWdtInst;
static XScuWdt *p_XScuWdtInst = &XScuWdtInst;
```

Function: xScuWdtInit()

We are then straight into the functions section. The initialisation function, **`xScuWdtInit()`** is defined first, following the (by now) familiar driver initialisation sequence. We will just look at two code fragments here. First, the function to check the reboot status is executed:

```
#if SCUWDT_DEBUG
  checkRebootStatus();
#endif
```

(The code for the **`checkRebootStatus()`** function will be shown in more detail later.)

The initialisation code then runs through the device look-up, configuration, and self-test phases. If these three steps are successful, we call the device-specific configuration code, as shown below.

```
// Load the watchdog counter register, and start it.
XScuWdt_LoadWdt(p_XScuWdtInst, SCUWDT_LOAD_VALUE);
XScuWdt_Start(p_XScuWdtInst);
```

We just need to use two Xilinx SCU WDT-defined functions above; first, the WDT is loaded with the desired time-out value (6.44 seconds as defined in the header file), and then the WDT is started. After the initialisation sequence completes, the status is returned to the calling function (that code is not shown here.)

Function: restartScuWdt()

The low-level Xilinx function **`XScuWdt_RestartWdt()`** is used to service the SCU WDT; this is wrapped in another function called **`restartScuWdt()`** so that it can be used by upper level code:

```
void restartScuWdt(void)
{
  XScuWdt_RestartWdt(p_XScuWdtInst);
}
```

In terms of watchdog timer functionality in our project, this is the only method that any other code (specifically **`main()`**) will need to access.

Function: checkRebootStatus()

Finally, we have the **`checkRebootStatus()`** function, function, which makes use of the Xilinx low-level memory access functions.

In this case, the 32-bit function **`Xil_In32()`** is used to read the SLCR reboot status register, and the contents are printed to the terminal. The Xilinx low-level memory functions can be very useful when starting to write code, as they allow the beginner to add in novel debug options, like the one here. Along with the 32-bit access function **`Xil_In32()`**, half-word (**`Xil_In16()`**) and byte (**`Xil_In8()`**) options are also available, although the 32-bit option will prove to be the most useful. Write functions are also available (**`Xil_Out8()`**, **`Xil_Out16()`**, **`Xil_Out32()`**), although the user should be more careful when writing to memory as an illegal access can cause the system to crash.

Just a partial code fragment for the **`checkRebootStatus()`** function is shown below; the reader should refer to the full source file to see the complete listing.

```
void checkRebootStatus(void)
{
  uint32_t slcr_reboot_sts = Xil_In32(SLCR_REBOOT_STATUS_REG);

/* PRINT REBOOT STATUS TO CONSOLE... */
/* ... */
}
```

10.5.2 Updates to Top-level Layer

At this point, the new components in the driver interface layer have been discussed, and we move on to the top-level layer. As we already have a good hierarchical structure in place, no new files are needed at the top level; instead, we just need to make some modifications to the existing code. With this in mind, only the new code fragments will be discussed in this section, starting with system configuration.

10.5.2.1 System Configuration

The system configuration files must be updated to call the initialisation functions for the SCU timer and the SCU WDT. We will look at the header file first.

10.5.2.1.1 Header File: system_config.h

At the start of the header file, we need to **`#include`** the header files for both timers.

```
/* !!! New Include Files !!! */
#include "wdt/scuwdt_if.h"
#include "timers/xscu_timer_if.h"
```

Next, several new constants are required, most of which are related to slowing down the toggle rates of the LEDs in the system. As a reminder, LED0 is used in the project to indicate if system initialisation is successful (by remaining on), but it will flash if initialisation is unsuccessful. To set the flash rate for this failure state, the **`INIT_FAIL_LOOP_DELAY`** constant is used. The other three constants are used to slow down the toggle rates of LED1, LED2, and LED4.

```
/* Toggle rate for LED0 if initialization fails */
#define INIT_FAIL_LOOP_DELAY 10000000U

/* Slow down LED toggle rates, when loop rate is very fast */
#define LED1_TOGGLE_COUNT  2500U
#define LED2_TOGGLE_COUNT  5000U
#define LED4_TOGGLE_COUNT  10000U
```

The final modification to the header file is the addition of the **`xscu_wdt`** and **`xscu_timer`** status variables to the initialisation status structure **`init_status_t`**.

```
typedef struct {
    volatile int xscu_wdt; // !!! NEW !!!
    volatile int xgpio0;
    volatile int xgpiops;
    volatile int xscu_timer; // !!! NEW !!!
}init_status_t;
```

The remainder of the header file is exactly as before, and so we next move to the system configuration source file.

10.5.2.1.2 Source File: system_config.c

Just a small number of updates are needed in the source file, and they are all carried out in the **`sys_init()`** function. First, the SCU timer and WDT initialisation functions must be called, and the **`init_status_t`** typedef updated with the results:

```
p_init_status->xscu_wdt = xScuWdtInit(); // SCU WDT
p_init_status->xscu_timer = xScuTimerInit();// SCU TIMER
```

Then, at the end of the **`sys_init()`** function, the code that checks the initialisation status must also be updated:

```
if( (p_init_status->xscu_wdt == XST_SUCCESS)  // SCUWDT
    && (p_init_status->xgpio0 == XST_SUCCESS) // AXI GPIO
    && (p_init_status->xgpiops == XST_SUCCESS)// PS7 GPIO
    && (p_init_status->xscu_timer == XST_SUCCESS) ) // SCU TIMER
{ /* !!! RETURN XST_SUCCESS OR XST_FAILURE !!! */ }
```

These are the only updates needed in the system configuration files, showing that only minimal modifications are required in higher level code once a good hierarchical structure is in place.

10.5.2.2 Top-level files

Finally, we get to the top-level header and source files, **`sw_proj4_main.c/h`**. As it turns out, the header file is identical to the one in the previous project, and it will not be discussed here (see Section 9.3.2.2.1 for a reminder if necessary). The source file, however, is quite different as it must take into account the new timing method and the watchdog timer. LED0 is also being used to show the initialisation result. To help in understanding the new code, Figure 10.9 shows the program timing, and Figure 10.10 shows the program flow.

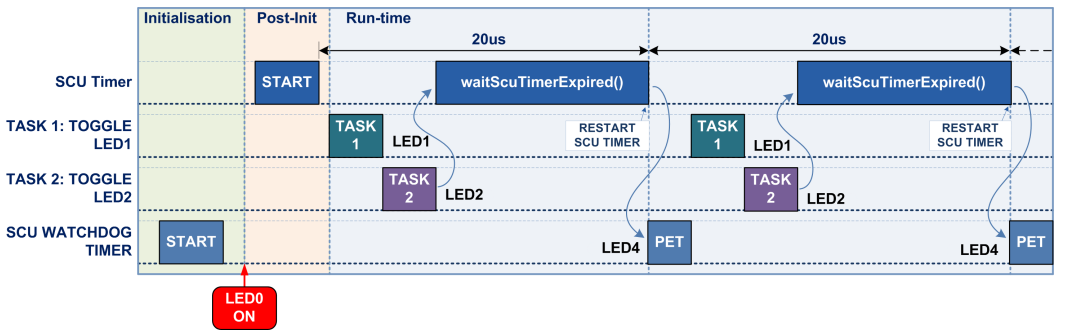

Figure 10.9. Program timing for software project 4, including initialisation

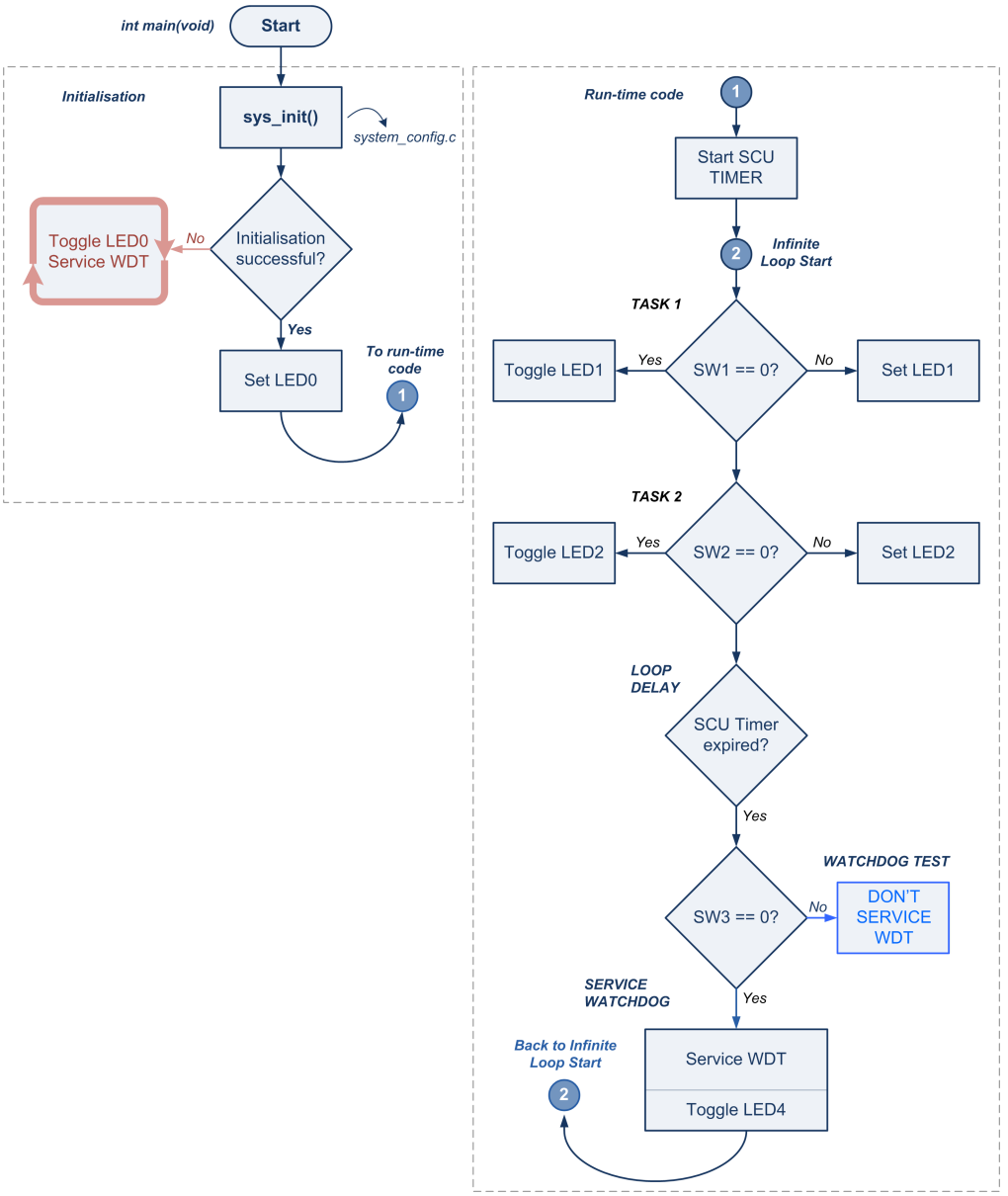

Figure 10.10. Main program flow for software project 4

The program starts by calling the **`sys_init()`** function — if this is successful, LED0 will turn on.

```
int init_status;
init_status = sys_init();

if (init_status == XST_SUCCESS)
{
axiGpOutSet(LED0);
}
```

If initialisation fails, however, the program enters an infinite loop where LED0 will flash.

```
else { // INIT FAILED!!!
  while(1) { // Stay in this loop
  axiGpOutToggle(LED0);
  restartScuWdt();

  uint32_t delay = 0U;
  for (delay = 0; delay < INIT_FAIL_LOOP_DELAY; delay++)
    {}
  }
```

The watchdog timer will also be serviced in the above loop, allowing the user to carry out some simple debug. For example, it may still be possible to interrogate memory-mapped registers via the command line tool, XSCT.

Next, we start the SCU Timer using the dedicated interface function:

```
startScuTimer();
```

Then we are straight into the main program loop, with the code for task 1 coming first. This section also demonstrates how the PS7 GPIO outputs are used as test signals here, for example, **PS_GP_OUT3** is used to show that task 1 is active. This signal is set at the start of the task (shown below), and cleared when the task is complete (shown later).

```
for(;;) { // Infinite loop
// TASK 1
psGpOutSet(PS_GP_OUT3); /// SET TEST SIGNAL
```

The main task 1 code now runs. First, the state of SW1 is read — if it is '0', LED1 will toggle every time the **led1_count** variable reaches the **LED1_TOGGLE_COUNT** value:

```
sw1_state = axiGpInRead(SW1); // Read SW1 state
if (sw1_state == 0U)
{  // Toggle LED1
  led1_count++;
  if (led1_count == LED1_TOGGLE_COUNT)
  {
    axiGpOutToggle(LED1);
    led1_count = 0U;
  }
}
```

Alternatively, if SW1 is set, then LED1 will remain permanently on:

```
else // SW1 is on
{
  axiGpOutSet(LED1); // LED1 on permanently
}
```

Finally, the test signal **PS_GP_OUT3** is cleared to indicate that task 1 is complete:

```
psGpOutClear(PS_GP_OUT3); /// CLEAR TEST SIGNAL
```

The program continues on to task 2, which is very similar to task 1, and thus will not be listed here. The only difference is that SW2 and LED2 are used in the task, and **PS_GP_OUT4** is used as the test signal to show that the task is active.

With the second task complete, the program now waits until the SCU timer has expired, using the SCU Timer interface function **waitScuTimerExpired()**. **PS_GP_OUT0** is used as the test signal to show that this part of the program is now active.

```
psGpOutSet(PS_GP_OUT0); /// SET TEST SIGNAL
waitScuTimerExpired();
psGpOutClear(PS_GP_OUT0); /// CLEAR TEST SIGNAL
```

When the SCU timer reaches zero, the **waitScuTimerExpired()** function returns and the program moves to the watchdog section. First, the test signal **PS_GP_OUT5** is used to show that the code has reached this phase, and then the state of SW3 is read. As a reminder, this switch is used as a test mechanism to force the WDT to time out: if SW3 is set, then the watchdog will not be serviced, and the Zynq-7000 processor will reset. We will see the effect of this in Section 10.6.2.

```
psGpOutSet(PS_GP_OUT5);  /// SET TEST SIGNAL
sw3_state = axiGpInRead(SW3);
```

In normal operation, however, this test option is not used, and instead the watchdog will be serviced using the function **restartScuWdt()**. LED4 will also toggle to show that the system is "alive".

```
if (sw3_state == 0U)
{
  led4_count++;
  if (led4_count == LED4_TOGGLE_COUNT)
  {
    psGpOutToggle(LED4);
    led4_count = 0U;
  }
  restartScuWdt();
}
```

The test signal **PS_GP_OUT5** is then cleared, and the program returns to the start of the run-time loop, where the sequence repeats.

```
psGpOutClear(PS_GP_OUT5);/// CLEAR TEST SIGNAL
/* => BACK TO START */
```

At this point, all the relevant code has been discussed, and so the final step is to test the program.

10.6 Running the Program

10.6.1 Normal Board Power-up

In the guide related to this book, the steps required to run the current program can be found in the section called "Software Project 4: SCU Timing". The reader should also refer back to Figure 10.1 and Section 10.3 of this chapter for further details on the project set-up.

The Zybo-Z7-20 platform must be connected to a host PC via the micro-USB connector, J12, and a terminal program should be running on the PC. To ensure that the program text is displayed in the terminal, the following **DEBUG** parameters need to be set in the related files:

```
/* sw_proj4_main.h */
#define MAIN_DEBUG    1
/* system_config.h */
#define SYS_CONFIG_DEBUG   1
/* scuwdt_if.h */
#define SCUWDT_DEBUG  1
```

When the program is launched successfully, the terminal will display the output as shown in Figure 10.11. The SLCR Reboot Status Register should indicate that the most recent boot event was due to a normal power-on reset (POR). (This assumes that no other reboot event occurred since powering up the board.) Note that in the programs in this book, the POR bit in the SLCR register should always read as '1'. If any of the other reboot bits are set because of a different reboot event, they can only be cleared by power-cycling the board (i.e. using the board On/Off switch, SW4); in that case, the *POR* bit will still be set to '1'.

The terminal should also indicate that the drivers in the application were initialized successfully — for example, Figure 10.11 shows that the SCU WDT and SCU Timer have been added to the system. On the platform itself, LED0 should be on (indicating successful system initialisation) and LED4 should be toggling (indicating that the program is running and the WDT is being serviced). Also, LED1 and LED2 should be toggling, indicating that **`task1()`** and **`task2()`** are executing.

10.6.2 Test the Watchdog Timer

In this project, we added the option to test the watchdog timer by setting SW3 on the platform to 'on'. When we do this, the watchdog timer will not be serviced, and the system will reset after about 6.5 seconds. If this happens, only the *PGOOD* LED on the platform will remain illuminated.

To recover from this situation, the following steps should be followed:

1. DO NOT power off the board. Instead, start by setting SW3 back to 'off'.
2. Reprogram the FPGA in SDK.

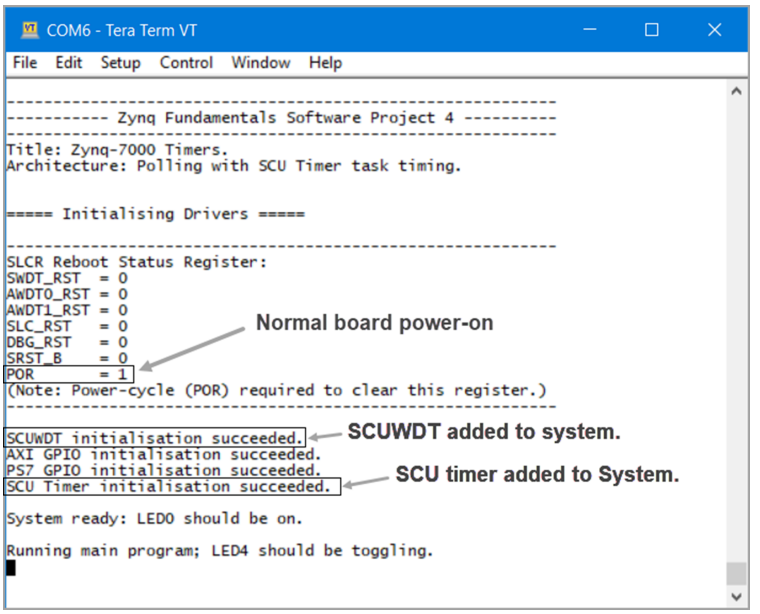

Figure 10.11. Terminal output for software project 4 after normal board power-up.

3. On the terminal application on the host PC, clear the display and/or empty the buffer. (In Tera Term, select *Edit->Clear Screen*, or *Edit->Clear Buffer.*)
4. Right-click the program in SDK and select *Run As-> Launch on Hardware (System Debugger)*.

Figure 10.12 shows the terminal output when the program is relaunched. As expected, the *AWDT0_RST* flag is set, indicating that the private WDT for CPU0 timed out.

10.6.3 Test the System Reset Button Functionality

Next, we can see what happens when the system reset button, *PS-SRST* (BTN7), is pressed. It turns out that the behaviour is somewhat similar to the WDT reset, in that the code will stop executing on the platform, and only the *PGOOD* LED will remain illuminated. The system can be restarted as follows:

1. DO NOT power off the board. Instead, reprogram the FPGA in SDK.
2. On the terminal application on the host PC, clear the display and/or empty the buffer. (In Tera Term, select *Edit->Clear Screen*, or *Edit->Clear Buffer.*)
3. Right-click the program in SDK and select *Run As-> Launch on Hardware (System Debugger)*.

Figure 10.13) shows the result of relaunching the application using these steps — the *SRST_B* flag is now set, along with *AWDT0_RST* and *POR*. As discussed earlier, we expect that the *POR* flag will always be set, but now we see that the system also 'remembers' that there was a WDT reset event during recent operation. To clear the *AWDT0_RST* and *SRST_B* flags,

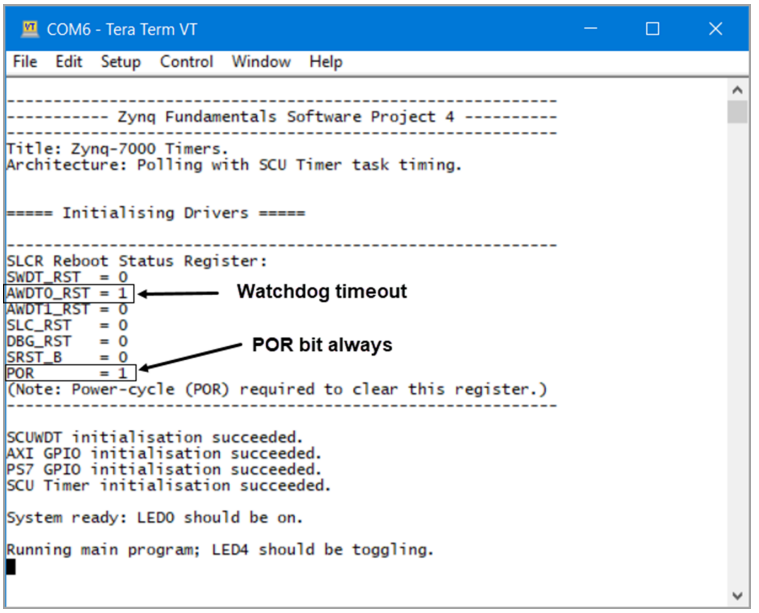

Figure 10.12. Terminal output after system reset due to CPU0 WDT time-out.

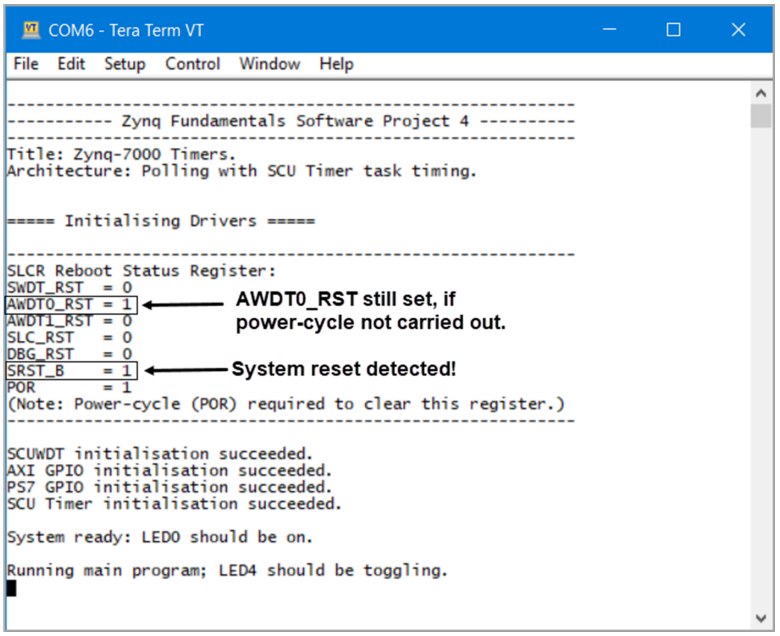

Figure 10.13. Terminal output for software project 4 after PS-SRST (BTN7) system reset.

the board power must be cycled (SW4), the FPGA reprogrammed, and the application relaunched. When this is done, the system should power up as discussed at the start of this section (see Figure 10.11 for a reminder).

10.6.4 Viewing Test Signals on a Logic Analyser

Finally, if a logic analyser is available and connected to Pmod JF, the user will be able to see the signal activity as shown in Figure 10.14. In this case, the Digilent Analog Discovery 2 [112] is used in conjunction with the Digilent Waveforms GUI [113]. The test signal connections have already been detailed in Section 10.3.4, and they are summarised again in Table 10.8 (along with the Analog Discovery pins):

Test Signal	C Code	Pmod Pin	LA Pin
TIMING	PS7_GP_OUT0	Pin 1	DIO 0
TASK1	PS7_GP_OUT3	Pin 4	DIO 3
TASK2	PS7_GP_OUT4	Pin 7	DIO 4
AWDT	PS7_GP_OUT5	Pin 8	DIO 5

Table 10.8. Test signal details for software project 4

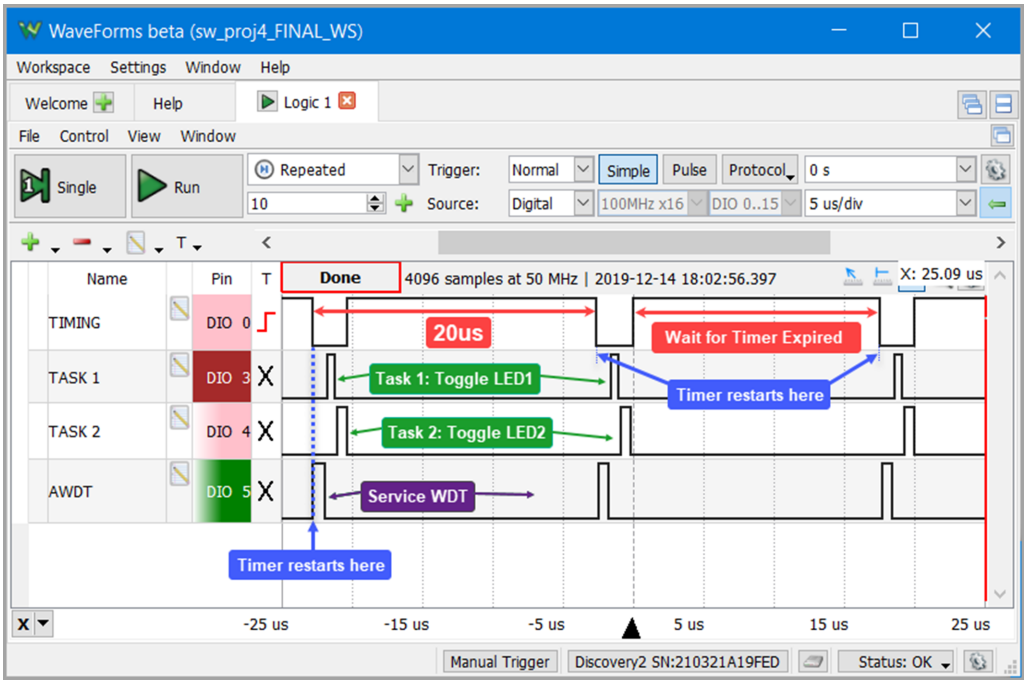

Figure 10.14. Test signals for software project 4. The sample rate is 50 MHz, and the time-base is 5us/div. The LA is set to trigger on the "TIMING" signal, DIO 0.

In Figure 10.14, the Waveforms GUI displays the program timing that was outlined earlier (Section 10.2.3). If we take the SCU timer restart as a reference point, the sequence can be tracked as follows:

1. The SCU timer restarts.

2. The watchdog timer is serviced, ensuring that it doesn't time out.
3. Task 1 executes (LED 1 is toggled).
4. Task 2 executes (LED 2 is toggled).
5. The program now waits for the timer to expire, and then the sequence repeats.

The logic analyser waveforms reveal an important point: the timer value should be set high enough such that the tasks can run to completion before timer expiry occurs. If the combined duration of each task exceeds the expiry time, then the timing is no longer dependent on the SCU Timer. The program might still work, but the timing now depends on how long it takes the tasks to complete, and the timer serves no purpose.

Moving On

In the next chapter, the SCU Timer will be used in conjugation with the Zynq-7000 interrupt system to provide an improved method for program timing.

10.7 References

[111] AXI Timebase Watchdog Timer v3.0 - LogiCORE IP Product Guide, Xilinx PG128 October 4, 2017.

[39] Zynq-7000 SoC Technical Reference Manual, Xilinx UG585 (v1.12.2).

[34] Cortex-A9 MPCore Technical Reference Manual (r3p0), ARM DDI0407G.

[112] Digilent Analog Discovery 2, https://store.digilentinc.com/

[113] Digilent Waveforms Software, https://store.digilentinc.com/