Freescale Semiconductor
Application Note

Document Number: AN3614
Rev. 0, 05/2008

Designing Code for the MPC5510 Z0 Core

by: Daniel McKenna
MCD Applications Engineering, EKB

1 Introduction

One of the powerful features of the MPC5510 family of microcontrollers is its secondary Z0 core. This core, once out of reset, can function independently of the primary Z1 core and can be used for a range of tasks, such as running gateway functions between communications networks, managing I/O processing, creating virtual peripherals, or simply reducing the load on the primary core by carrying out tasks required by the system.

This application note describes the features of the Z0 core, discusses the aspects of these features that should be considered when creating code for a dual core device, and demonstrates, through the creation of a virtual PWM peripheral, how to set up and produce dual core code.

Contents

1 Introduction 1
2 Requirements 1
3 Acronyms and Abbreviations 2
4 Overview of the MPC5510 Family 2
4.1 Comparison of Z0 and Z1 Cores 3
5 Dual Core Considerations 7
5.1 Semaphores 8
6 Principles of Virtual PWM Module 9
6.1 Mathematics of PWM Signal Generation 9
7 Dual Core Aspects of Code 11
7.1 Compiling the Dual Core Code 12
7.2 Initializing the Z0 Core 12
7.3 Configuring External Interrupts for Z0 12
7.4 Sharing Data Between Cores 13
8 Results 14

2 Requirements

To make use of the software, the following hardware and software components are required:

- MPC5510 microcontroller

- An evaluation board or device platform to run the device
- A hardware interface between the computer and the evaluation board. Tested interfaces:
  — Lauterbach Datentechnik Power Debug Interface
  — P&E Microcomputer Systems USB Multilink
- A compiler for the application code. Projects are included for:
  — Wind River Compiler V5.5.1.0
  — Freescale CodeWarrior for MPC55xx V2.2
- A software tool for debug. Tested software:
  — Lauterbach Datentechnik Trace32 Build 11389
  — P&E Microcomputer Systems ICDPPCNEXUS Debugger V1.13

The CodeWarrior compiler and integrated P&E software debugger are included with the MPC5510 evaluation boards.

3 Acronyms and Abbreviations

CRP	clock, reset and power (control module)
GPIO	general purpose input/output
INTC	interrupt controller (module)
IOP	input/output processor
ISR	interrupt service routine
MCU	microcontroller unit
MMU	memory management unit
PIT	periodic interrupt timer
PWM	pulse width modulation
SIU	system integration unit
VLE	variable length encoding (instruction set)

4 Overview of the MPC5510 Family

The MPC5510 family is a set of 32-bit MCUs aimed at the next generation of automotive body and gateway applications. The devices are built on the proven Power Architecture, and are based on the successful, power-train targeted, MPC5500 family of microcontrollers. These cost-efficient microcontrollers provide excellent performance whilst minimizing power consumption with the introduction of numerous low power modes.

All members of the family come with Power Architecture Book E compliant primary core – the e200z1, with the majority also coming with a secondary IOP – the e200z0. This application note is aimed only at the dual core devices.

4.1 Comparison of Z0 and Z1 Cores

When the part is brought out of reset, only Z1 is active; Z0 remains in reset until it is activated by Z1. Once both cores are running, they are standalone and can operate independently of each other; for example, Z0 will continue to operate if Z1 is disabled. Thus, both cores require their own code, their own ***main*** function in code, and have autonomous stacks.

4.1.1 Instruction Sets

The main difference between the cores is in the instruction sets they execute; Z1 can use the base Power Architecture Book E set and the extended Variable Length Encoded (VLE) set. Z0 is compatible with the VLE instruction set only.

VLE makes use of both 16-bit and 32-bit instructions, giving a greater code density with minimal or no loss of performance. VLE instructions can be recognized in mnemonics by their compulsory prefix.

- All 16-bit VLE instructions are prefixed by “se_”; for example, se_addi
- All 32-bit VLE instructions are prefixed by “e_” ;for example, e_add16i

Table 1 compares the different “add immediate” instructions. Notice the format of the 16-bit VLE instruction: as there are fewer bits available, the source and destination must be the same register, and the distance to the data must be between 1 and 32. If this proves to be unsuitable, the 32-bit VLE instruction can be used instead.

Table 1. Comparison of Add Immediate between Instruction Sets

Type	Mnemonic		Description

Book E	addi	rD, rA, SIMM	rD = destination register
			rA = source register
			SIMM = signed 16-bit data

16-bit VLE	se_addi	RX, OIM5	rX = source and destination register OIM5 = 5 bit immediate offset (0-31)

32-bit VLE	e_add16i	RT, RA, SI	RT = destination register
			RA = source register
			SI = signed 16-bit index

A compiler switch decides if VLE or Book E code is generated from a source file. This can generally be changed on a file by file basis.

The Z1 core contains a memory management unit (MMU) that is used to define different sections of memory. One of the key parameters it sets is whether the memory section contains Book E or VLE code. Hence, it is possible for the Z1 core to alternate between Book E and VLE code. This means that there is no requirement to convert existing libraries to VLE code. As the Z0 core is compatible with VLE only, it does not require an MMU.

The MMU is also responsible for selecting the endianness of the memory section. Z1 can use either big endian or little endian; Z0 is big endian only.

4.1.2 Access to Shared Resources

Both cores have access to all memory and peripherals along with the other “bus masters” (FlexRay controller and eDMA). The crossbar switch is used to control access and arbitration. The Z1 core has an additional direct link to the flash memory, allowing it to gain instant access without first going through the crossbar switch. This reduces the access time and allows both cores to access different flash locations simultaneously.

The Z1 core is based on the Harvard architecture, having individual instruction and data buses, whereas the Z0 core is based on the Von Neumann architecture, having a single, combined data/instruction bus. (Variations of the Z0 core do exist on non MPC5510 family devices that are based on the Harvard architecture. This application note focuses on the MPC5510’s Z0.)

Each core has its own set of interrupts. Table 2 shows which interrupts are available on each core:

Table 2. Interrupts Available on Each Core

Core Interrupt Type	IVOR #	Z1	Z0
Critical Input	IVOR0	X	X
Machine Check	IVOR1	X	X
Instruction Storage	IVOR3	X	X
External Input	IVOR4	X	X
Alignment	IVOR5	X	X
Program Interrupt	IVOR6	X	X
Floating Point Unavailable	IVOR7	X	
System Call	IVOR8	X	X
Decrementer	IVOR10	X	
Fixed-Interval Timer	IVOR11	X	
Core Watchdog Timer	IVOR12	X	
Data TLB Error (MMU)	IVOR13	X	
Instruction TLB Error (MMU)	IVOR14	X	
Debug	IVOR15	X	X

External interrupts (IVOR4s) can be routed to Z0, Z1, or both cores, on an individual basis.

IVOR12 corresponds to the watchdog timer, which is present in core Z1 only. However, it is envisaged that the watchdog in the Miscellaneous Control Module (MCM) will be used by most customers, as it has more features. Upon watchdog timeout, this can either generate an external interrupt (an IVOR4) or perform a system reset, resetting both cores.

Both cores run at the same speed, from the same clock source, set up in the System Integration Unit (SIU).

4.1.3 Core Registers

Figure 1 and Figure 2 show the registers in the Z0 core and Z1 core, respectively.

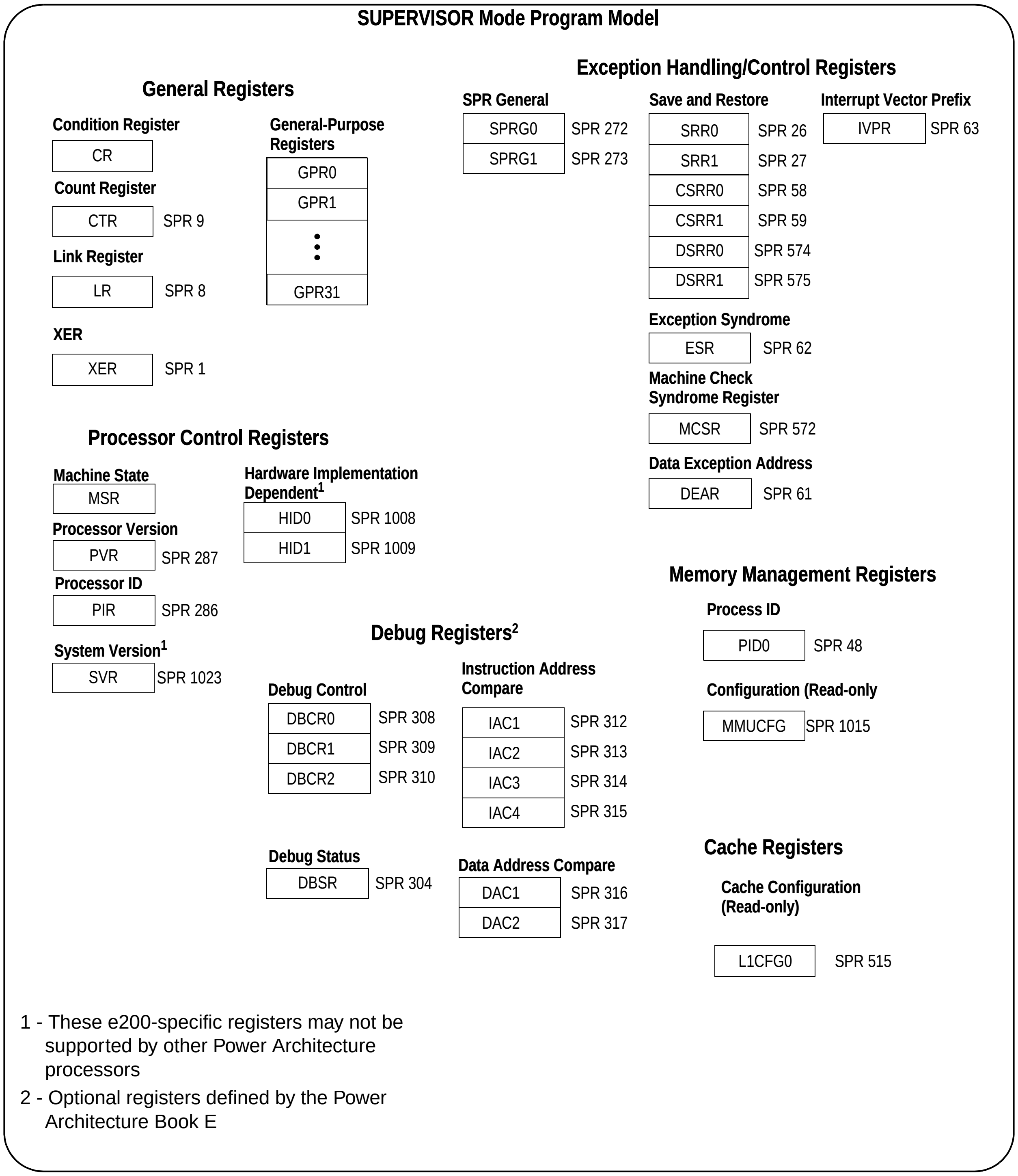

Figure 1. Registers in Z0 Core

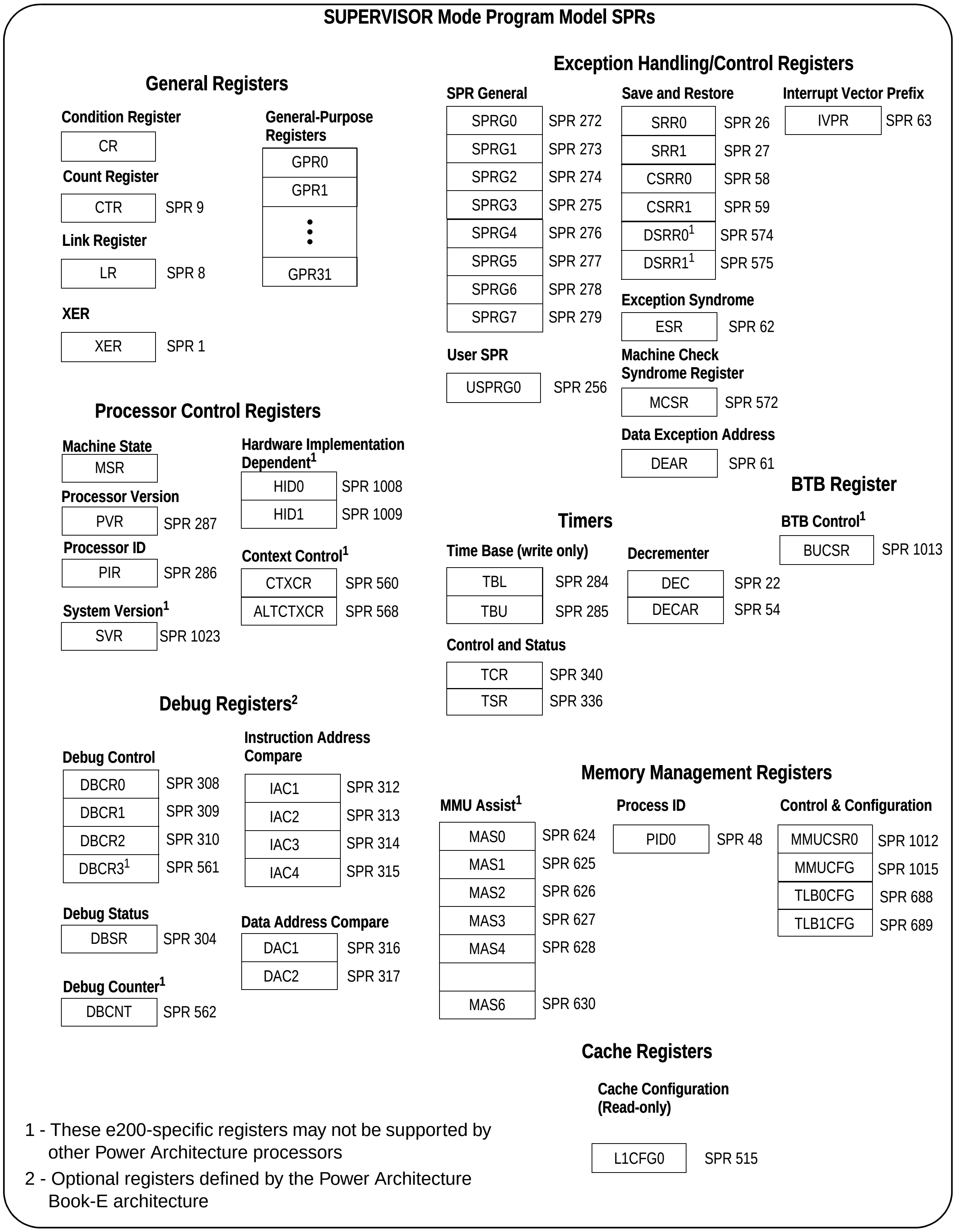

Figure 2. Registers in Z1 Core

- The Z0 core does not have the timing registers present on the Z1 core, including the TBU and TBL time-of-day registers and the decrementer.
- Of the General/User Special Purpose Registers (SPRG0:7/USPRG0) present on the Z1 core, the Z0 core has only SPRG0:1.
- The Z0 core does not have the branch target buffer used in the Z1 core to accelerate the execution of loops.

5 Dual Core Considerations

Writing code for a dual core MCU requires a different approach to that required for a standard single core MCU. This section discusses how to approach the problem, and shows how the MPC5510 family features can help simplify code development.

As both cores are independent, two separate startup files (crt0.s) are required — one for each core. These assign values to global variables, set up both stacks and both small data areas, then set each program counter to the location of the ***main*** function on each core. The Z1 core must also write to all of the SRAM, to initialize error correction syndrome.

Designers have a range of options when splitting code between cores, and a vast amount of literature is available discussing a variety of scenarios. Examples include:

- Using the secondary core solely to deal with interrupts
- Using the secondary core to carry out a standalone function, for example, to act as a gateway between LIN and CAN networks or to act as a virtual module, using software to simulate another module (SCI, DSPI, etc.)
- Putting a computationally intensive function (Manchester decoding, for example) on the secondary core, and having the primary core call it when necessary.
- Using the secondary core to error check the processes being carried out by the primary core.
- Carefully splitting the code between the cores to ease the load on the primary core.

If two cores have access to the same peripheral/memory locations, it is vitally important to avoid race conditions and ensure data coherency. For example, if both cores have to increment a single variable (***count***), ideally the following would happen:

1.	Z0 copies variable ***count*** from memory to register	(***count*** = 0)
2.	Z0 increments ***count*** in register	(***count*** = 1)
3.	Z0 copies ***count*** back to memory	(***count*** = 1)
4.	Z1 copies variable ***count*** from memory to register	(***count*** = 1)
5.	Z1 increments ***count*** in register	(***count*** = 2)
6.	Z1 copies ***count*** back to memory	(***count*** = 2)

So, the value stored in memory would be 2, as expected. However, should both cores access the variable at the same time, the following could happen:

1.	Z0 copies variable ***count*** from memory to register	(***count*** = 0)
2.	Z1 copies variable ***count*** from memory to register	(***count*** = 0)

3. Z0 increments ***count*** in register	(***count*** = 1)
4. Z1 increments ***count*** in register	(***count*** = 1)
5. Z0 copies ***count*** back to memory	(***count*** = 1)
6. Z1 copies ***count*** back to memory	(***count*** = 1)

In this instance, the value would be stored in memory incorrectly as 1. Clearly, this causes errors in the code and must be avoided. To prevent this from happening, the MPC5510 family contains a semaphore mechanism ,which can be used in software to control each core's access to shared resources.

5.1 Semaphores

On the MPC5510 family, semaphores are a set of sixteen hardware enforced gates, each of which can take one of three states:

- Unlocked
- Locked by Z1
- Locked by Z0

Semaphores can be represented by the state machine shown in Figure 3.

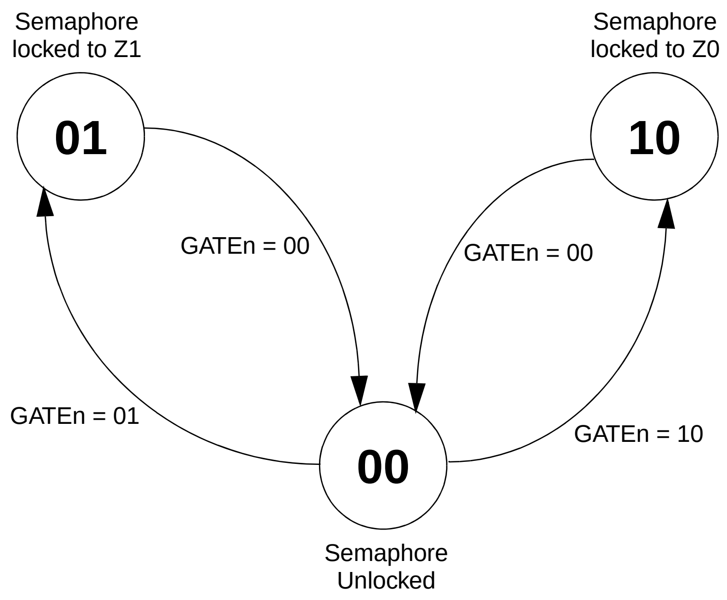

Figure 3. Semaphore State Machine

Semaphores are software enforced, and the designer can assign one to a shared resource. When a core requires access to the resource, it should attempt to lock the appropriate gate. If the lock is successful, the core then has sole access to the resource; if not, the resource is currently in use by the other core. If the semaphore was locked to the other core, the designer can either enter a loop, continuously trying to lock the gate until it is successful, or make use of the notification interrupt, to maximize performance. If enabled, the notification interrupt is automatically generated when a gate becomes unlocked, allowing the core to continue executing other code rather than continually polling the semaphore.

The correct use of semaphores ensures data coherency in memory and ensures that peripherals work as expected.

The generation of dual core code is demonstrated in the following sections by setting up a virtual PWM module on the Z0 core and using the Z1core to alter the parameters of individual waveforms.

6 Principles of Virtual PWM Module

On MPC5510 devices, PWM is usually achieved using the easily configurable eMIOS module. However, it is sometimes desirable to expand upon the number of available PWM outputs. The Z0 core is an ideal means of creating additional unique PWM signals with no loading on the primary core (Z1).

The virtual PWM module generates signals by using the Z0 core in conjunction with the Periodic Interrupt Timer (PIT) module and the System Integration Unit (SIU).

The PIT module is configured to trigger a periodic interrupt at a pre-defined period. This interrupt is routed to the Z0 core via the Interrupt Controller, which then executes an interrupt service routine (ISR) stored in memory. The ISR configures the output on utilized I/O pins, based on the user-supplied parameters of each PWM signal. Thus, the resolution of the PWM signal depends on the PIT timeout period; that is, a shorter timeout period leads to more interrupts over the period of the wave — leading to a higher resolution.

An overview of the setup can be seen in Figure 4.

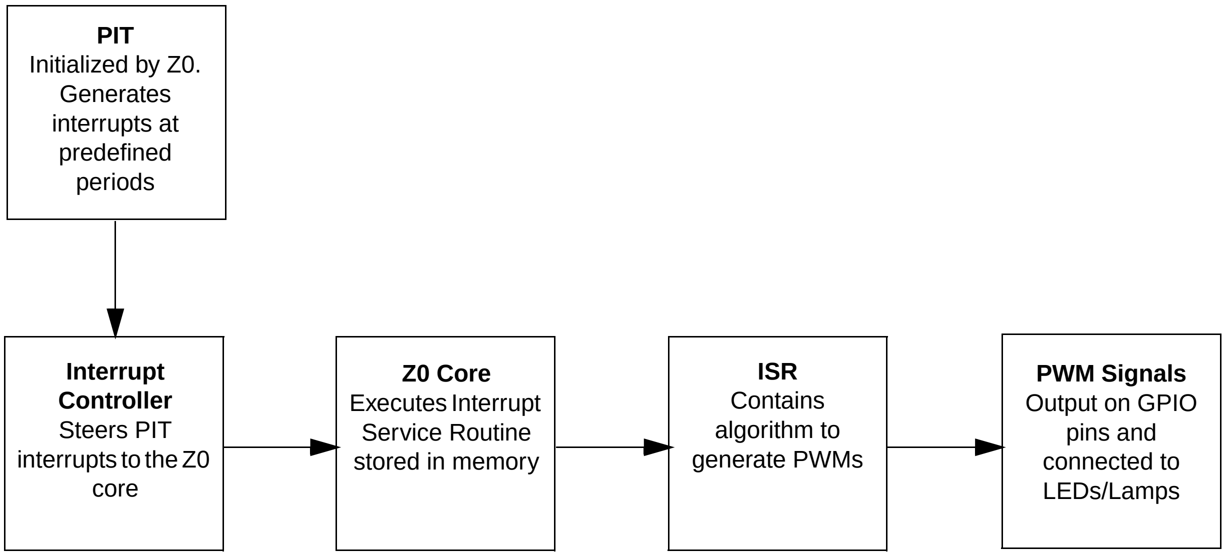
PIT
Initialized by Z0. Generates interrupts at predefined periods
Interrupt Controller
Steers PIT interrupts to the Z0 core
Z0 Core
Executes Interrupt Service Routine stored in memory
ISR
Contains algorithm to generate PWMs
PWM Signals
Output on GPIO pins and connected to LEDs/Lamps

Figure 4. Overview of Virtual PWM Implementation

6.1 Mathematics of PWM Signal Generation

6.1.1 Periodic Interrupt Timer

To create a signal to drive an LED at 160 Hz (period = 6.25 ms) with a resolution of 0.5% (that is, 200 interrupts per period), an interrupt is required every 6.25 ms/200 = 31.25 µs.

The PIT interrupt timer should be initialized to 31.25 µs. For example, with a system clock of 64 MHz (period = 15.625 ns), the timer reset value must be set to 31.25 µs/15.625 ns = 2000.

6.1.2 Signal Parameters

Each PIT channel can be used to generate a large number of PWM signals. Each of these signals must have the following parameters declared:

Period	The number of counts (or PIT interrupts) contained in the period of the wave.
Duty	The count value at which the signal changes from low to high.
Counter	Initially holds the starting value for the count; from then on it holds the current count value. The starting value for each waveform can be varied, to prevent poor EMC behavior resulting from many pins toggling at once.
Pad	The GPIO port on which the signal is generated.

6.1.3 PIT Interrupt Service Routine

The ISR code is run every time a PIT interrupt occurs. It compares the count value to the period and duty values, toggling the relevant pin, if necessary. It then decrements the count value.

A flow chart of the PWM algorithm is shown in Figure 5.

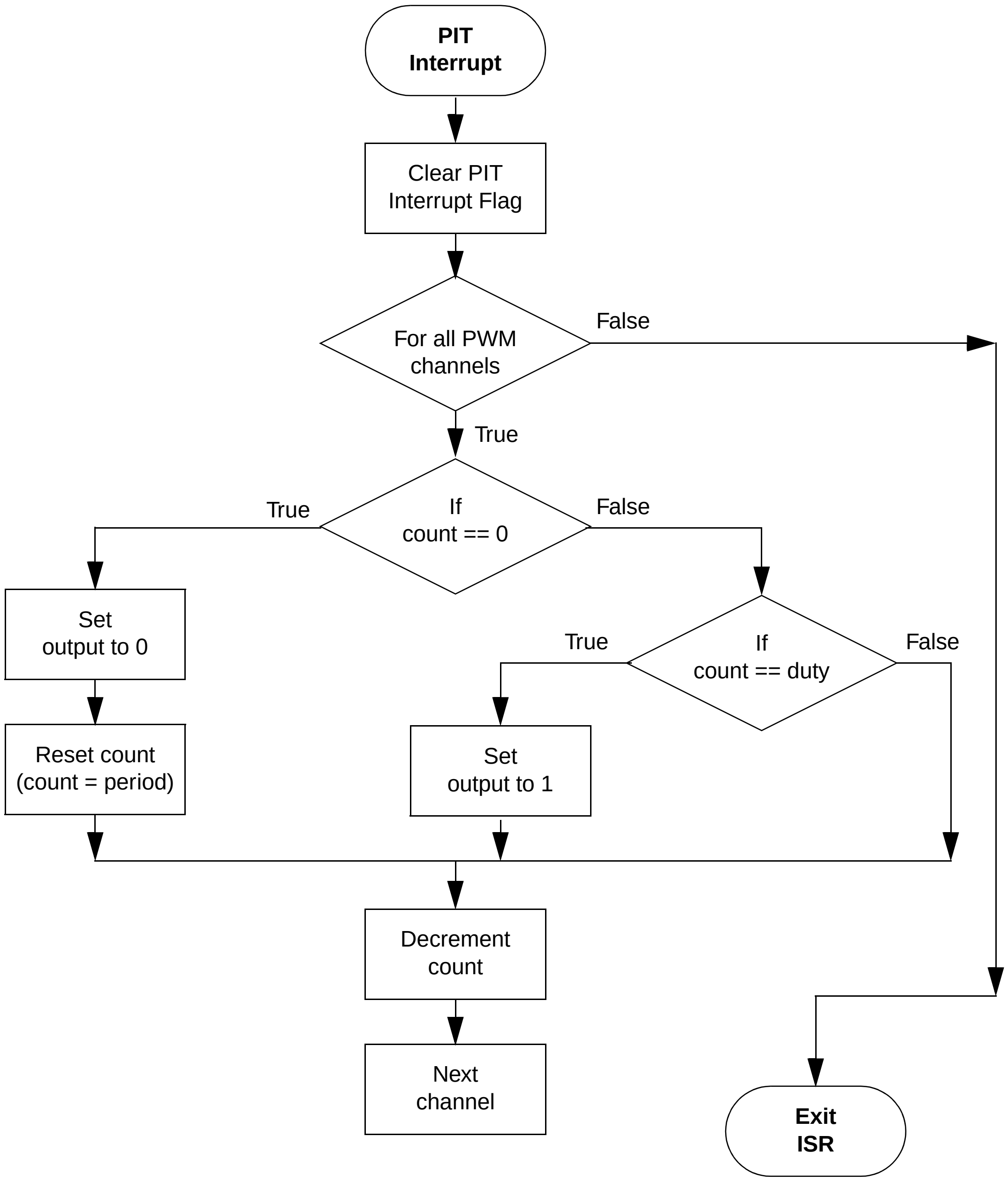

Figure 5. Flowchart of PWM Algorithm

7 Dual Core Aspects of Code

This section considers the dual core aspects of the code, discussing its compilation, its initialization on the Z0 core, the configuration of interrupts, and how the semaphores are set up.

7.1 Compiling the Dual Core Code

When compiling the code for both cores a choice has to be made whether to have individual output files for each core, or a combined output file that contains the code for both cores. In some cases, the linker dictates which method must be used; in other cases, the decision is left to the programmer. A brief overview of the advantages of each approach follows:.

One file:

- Can be easier to manage boundaries and flash programming as only one linker file is required
- Common code and variables can be shared easily between the two cores, in the same way as they would be shared across multiple *.C files.

Two Files:

- The code on each core can be a completely separate entity. Thus, it is easy to split the cores between different programming groups or even companies.
- Having separate linker files for each core can be easier to manage than a single large linker file containing two stacks and small data areas.

For our example, the code compiled using WindRiver has two separate output files, whereas the Codewarrior compiled code has a single output file. Both compilers use the VLE instruction set for both cores.

7.2 Initializing the Z0 Core

When the device powers up from reset, only the Z1 core is activated. The Z1 core sets up the clock as required and then brings the Z0 core out of reset, by writing the Z0 code start location to the ***Z0VEC*** register in the Clock, Reset and Power Control (CRP) Module. When the Z0 core is brought out of reset, the value of the ***Z0VEC*** register is loaded to the program counter, and execution begins immediately.

```
CRP.Z0VEC.R=0x4000A000;    /* Bring Z0 out of reset
                           and set Program Counter
                           to 0x4000A000 */
```

Figure 6. Code to Initialize Z0 Core

When both cores are running, it is possible to disable either core (but not both) by setting the reset bit in ***ZnVEC.*** This is an extremely useful power-saving feature, as the Z0 core need be brought out of reset only for as long as required**.** When only one core is running, any attempt to set its own reset bit is blocked.

7.3 Configuring External Interrupts for Z0

This section discusses the interrupt possibilities on the Z0 core. Full details of how to set up interrupts on the MPC5510 family, and an explanation of software and hardware based interrupt vectors, are available in the *5510 Cookbook*.

The word “external” denotes that the interrupt occurred external to the core; that is, caused by a peripheral or triggered by software. As discussed in the PWM sections above, the code requires a PIT interrupt that can be serviced by the Z0 core. As is the case with interrupts on the Z1, the Z0 requires its own prolog and epilog code, ISR vector table, and IVOR table.

The code sets up the IVPR base address as it would on Z1, loading the value to the IVPR core register.

The Interrupt Controller (INTC) module is then set up as in the Z1 case, using the registers for Z0 (those ending in PRC1), rather than those ending in PRC0. For our example, Z0 is configured for software interrupts as shown in Figure 7.

```
/* Initialize INTC for software vector mode */
INTC.MCR.B.HVEN_PRC1 = 0;

/* Use the default vector table entry offsets of 4
bytes */
INTC.MCR.B.VTES_PRC1 = 0;

/* Set INTC ISR vector table base addr. */
INTC.IACKR_PRC1.R =
(uint32_t)&IntcIsrVectorTable[0];
```

Figure 7. Code To initialize interrupts on Z0

The final step is to set the enable interrupts bit in the Machine State Register (MSR) in the core, thereby completing the configuration of the Z0 core to receive interrupts. Individual interrupts can then be directed to Z0, Z1, or both cores by using the Priority Select Register (PSR) in the INTC module.

	0	1	2	3	4	5	6	7
R	PRC_SEL		0	0	PRIO			
W								
Reset	0	0	0	0	0	0	0	0

PRC_SEL defines the interrupt destination.
- 00: Interrupt request sent to processor 0 (e200z1)
- 01: Interrupt request sent to both processors
- 10: Reserved
- 11: Interrupt request sent to processor 1 (e200z0)

PRIO determines the interrupt priority.
- 0 is the lowest priority (effectively disabled); 15 is the highest.

Figure 8. Interrupt Priority Select Register

7.4 Sharing Data Between Cores

To demonstrate how to maintain data coherency whilst sharing variables between cores, an extra feature has been added to the code to give the Z1 core the ability to periodically alter the parameters of the PWM signals.

The parameters for each PWM waveform are stored in a structure that is shared between both cores. The Z1 core accesses this structure intermittently and adjusts the duty cycle of the first signal. The Z0 core makes use of the structure only during the PIT ISR. To avoid race conditions, a semaphore must be used to designate which core has control of the structure of PWMs.

If the interrupt occurs and Z0 cannot obtain the semaphore, there is a small chance that the accuracy of the signal might deteriorate temporarily; however, as the parameters of the signal are being altered at this point and the signal will change anyway, this can be ignored.

The Z1 core requires control of the structure only while it is making alterations to the values.

Figure 9 shows a flow chart of how semaphores can be locked.

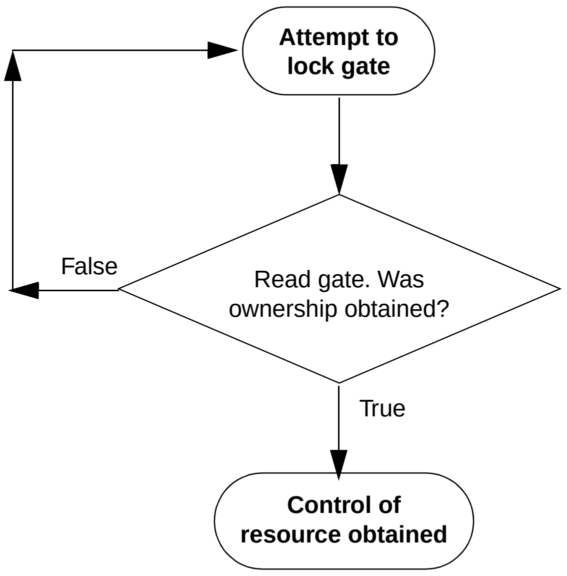
Attempt to lock gate
Read gate. Was ownership obtained?
False
True
Control of resource obtained

Figure 9. Flowchart of Lock Semaphore Process

A semaphore should be released by the core to which it is locked; however, a reset mechanism is available to allow a semaphore to be released by either core, should the need arise.

8 Results

Figure 10 shows an oscilloscope capture of the PWM outputs with four different unique PWMs being generated.

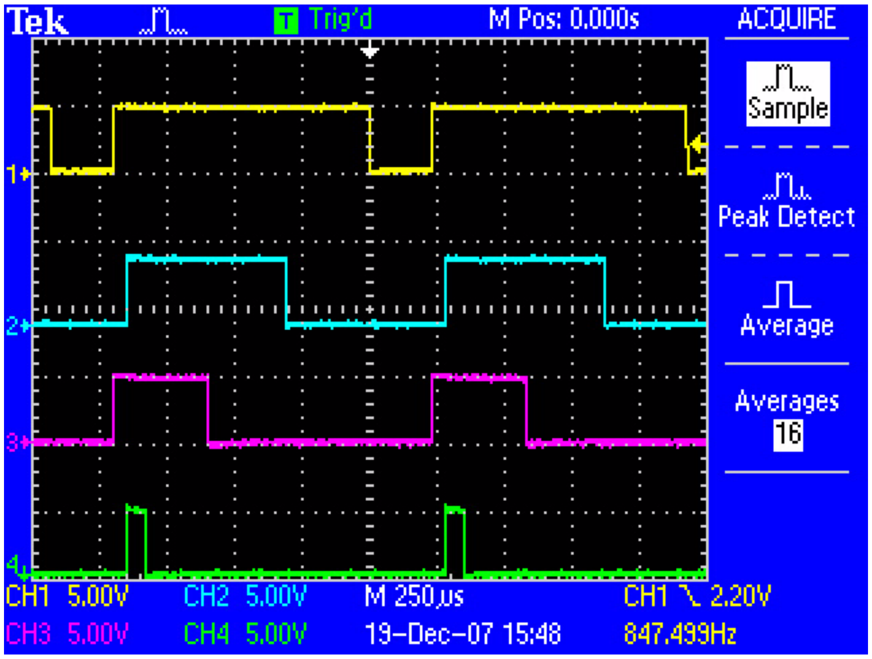

Figure 10. Scope Capture of Four PWM Outputs

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Number: AN3614
Rev. 0
05/2008