Freescale Semiconductor
Application Note

Document Number: AN4805
Rev. 2, 01/2014

A Practical Approach to Hardware Semaphores

For MCP56xx and MPC57xx Multi-core Qorivva Devices

by: Mong Sim

1 Introduction

A semaphore is a variable or abstract data type that provides a simple but useful abstraction for controlling access by multiple processes to a common resource in a parallel programming or multi-user environment.

Semaphores are the predominant method in access control in parallel programming and multi-threaded programming environments since their invention in 1965 by the late Edsger Dijkstra, a Dutch computer scientist. Although the semaphore concept was further enhanced from its original principle into many variations, they all serve a similar purpose: to grant access to shared resources without race conditions.

This practical approach to hardware semaphores provides simple semaphore routines that lock, unlock, and reset semaphore gates (flags). These routines are non-blocking and reentrant functions suitable for multi-core application usage. This application note presents each routine with C code, followed by a detailed explanation of its functionality.

Contents

1 Introduction 1
2 Programming Language Prerequisites 2
3 Hardware Semaphore Routines 2
3.1 Get Core Identification, Get_Core_ID 3
3.2 Get Semaphore Gate Status, Get_Gate_Status . . . 4
3.3 Lock Semaphore Gate, Lock_Gate 4
3.4 Unlock Semaphore Gate, Unlock_Gate 5
3.5 Reset Semaphore Gate, Reset_Gate 6
4 Classic Semaphore Examples 7
4.1 Signaling semaphore 7
4.2 Blocking semaphore 9
4.3 Rendezvous semaphore 11
5 Classic Issue: Deadlock 11
6 Summary 12

After the individual semaphore routines are discussed, they are used to build a few classic semaphore examples:

- Simple signaling semaphore
- Blocking semaphore
- Rendezvous semaphore

These application note examples were created for the MPC5643L, but they can also be applied to other MPC56xx and MPC57xx Qorivva multi-core chips with a hardware semaphore module.

The intention of this application note is to provide the necessary understanding for using the hardware semaphore module on MPC56xx and MPC57xx Qorivva multi-core chips. It is not intended as an application note to teach semaphore concepts.

2 Programming Language Prerequisites

In this application note, the software examples are illustrated in PowerPC assembly and C programming language.

Software

- CodeWarrior (freescale.com or codewarrior.com)

Hardware

- XPC56xx motherboard with XPC5643L144QFP mini module (EVB, the target system)
- 12 VDC power supply for the EVB
- Personal computer with XP operating system (the host system)

3 Hardware Semaphore Routines

All the hardware semaphore routines presented in this application note are platform-specific. They only work with MPC56xx and MPC57xx Qorivva multi-core chips with a hardware semaphore module, as shown in Table 1.

Table 1. Hardware semaphore multi-core chips

Chip	Semaphore module	Remarks
MPC5643L (in Decoupled Parallel mode)	SEMA4	First generation hardware semaphore
MPC5643L (in Lock Step mode)	Disabled in Lock Step mode	—
MPC5676R	SEMA4	First generation hardware semaphore
MPC57xxM (Multi-core)	SEMA42	Second generation hardware semaphore

NOTE

The Freescale semaphore module uses the term *semaphore gate*, which is similar to a semaphore flag. A *locked gate* is similar to a set flag and *unlocking* and *resetting a gate* are similar to clearing a flag.

3.1 Get Core Identification, Get_Core_ID

The hardware semaphore module requires that all cores competing for a semaphore gate must present a translated core ID as a valid parameter to lock a gate. This routine reads the core Process ID (PID) and converts it to an equivalent ID the hardware semaphore module will accept.

This routine may be called once in the lifetime of an application; it may also be called within a Lock Gate routine, or a user can simply supply the appropriate core ID to the Lock_Gate routine. In this application note, the Get_Core_ID function is called within the Lock_Gate routine for simplicity and safety. This method also guarantees that the correct core ID is sent to the hardware semaphore for locking a gate.

```
//------------------------------------------------------------------
// Get_Core_ID
//------------------------------------------------------------------
1: uint8_t Get_Core_ID (void)
2: {
3:        register uint32_t temp;
4:
5:        asm (" mfspr temp, 286");
6:
7:        switch (temp)
8:        {
9:                case 0:  return Core0_ID;
10:               case 1:  return Core1_ID;
11:       }
12: }
```

Figure 1. Get_Core_ID routine

Figure 1 lists the C code for the Get_Core_ID routine. This routine reads the core ID and returns the translated core ID to the caller. The description of the code in Figure 1 is as follows:

Line	Description
1	Get_Core_ID routine will return the translated code ID to the caller.
5	A method to call assembly within C code to read special purpose register (SPR) number 286 (PID). This SPR contains the PID. (See your chip's core reference manual for the correct SPR number.)
7–10	Translates the core ID and returns to the caller. (See the SEMA4_GATEn description in the Semaphore chapter of your chip's reference manual for the valid core (processor) ID (PID).) The equation for converting the PID to an ID that the semaphore module will accept is as follows: Translated CoreID = PID + 1 = 0 + 1 = 1 // if PID is equal to 0 = 1 + 1 = 2 // if PID is equal to 1 Constant: Core0_ID = 1// translated core ID for core 0 to be used for semaphore module Core1_ID = 2 // translated core ID for core 1 to be used for semaphore module

The routine in Figure 1 translates IDs for a dual-core system. To modify the above routine to support more cores, simply increase the number of case statements to match the number of cores and return the appropriate translated core IDs.

3.2 Get Semaphore Gate Status, Get_Gate_Status

This routine reads the status of a gate state machine and returns the status to the caller. Users can determine from the gate status which core locked this gate or whether the gate is available for all cores. An unlocked gate will return a status of zero while a locked gate will return a number associated with the core that locked the gate.

```
//-------------------------------------------------------------------
// Get_Gate_Status
//-------------------------------------------------------------------
1: uint8_t Get_Gate_Status (uint8_t gate_no)
2: {
3:         return SEMA4.GATE [gate_no].R;
4: }
```

Figure 2. Get_Gate_Status routine

Figure 2 lists the C code for Get_Gate_Status routine. This routine reads the status of a gate state machine and returns the status to the caller. The description of the code in Figure 2 is as follows:

Line	Description
1	Get_Gate_Status reads the gate requested by the caller and returns the gate status to the caller.
3	Reads the status of the gate state machine and returns the status to the caller.

3.3 Lock Semaphore Gate, Lock_Gate

See the Semaphore chapter of your chip's reference manual chapter for the number of gates supported by the device you are using. The MPC5643L, in Decoupled Parallel mode (DPM), has 16 gates.

Each gate can be locked by any core and can only be unlocked by the core that locked it. The caller to this routine must supply a valid gate number to lock a gate; invalid gate numbers are ignored. The caller must also parse the returned status to determine if the operation was successful. A successful operation returns a status equal to the translated core ID associated with the core that locked it.

```
//-------------------------------------------------------------------
// Lock_Gate
//-------------------------------------------------------------------
1: uint8_t Lock_Gate (uint8_t gate_no)
2: {
3:         uint8_t Core_ID;
4:
5:         Core_ID = Get_Core_ID ();
6:
7:         SEMA4.GATE [gate_no].R = Core_ID;
8:
9:         return Get_Gate_Status (gate_no);
10: }
```

Figure 3. Lock_Gate routine

Figure 3 lists the C code for the Lock_Gate routine. This routine tries to lock a gate and returns the gate status to its caller. The description of the code in Figure 3 is as follows:

Line	Description
1	Lock_Gate locks a gate requested by the caller and returns the status of that gate.
3	Refers to Get_Core_ID
7	Tries to lock the requested gate with its translated core ID
9	Refers to Get_Gate_Status

3.4 Unlock Semaphore Gate, Unlock_Gate

A semaphore gate can only be unlocked by the core that locked it. However, a locked gate or locked gates can also be released from a locked state using a reset operation provided by the hardware semaphore module. Resetting a gate is discussed later in Section 3.5, "Reset Semaphore Gate, Reset_Gate." The caller to the Unlock_Gate routine must also parse the returned status to determine whether the unlock operation was successful.

```
//-------------------------------------------------------------------
// Unlock_Gate
//-------------------------------------------------------------------
1: uint8_t Unlock_Gate (uint8_t gate_no)
2: {
3:    SEMA4.GATE [gate_no].R = UNLOCK;
4:
5:    return Get_Gate_Status (gate_no);
6: }
```

Figure 4. Unlock_Gate routine

Figure 4 lists the C code for the Unlock_Gate routine. This routine tries to unlock a gate and returns the gate status to its caller. The description of the code in Figure 4 is as follows:

Line	Description
1	Unlock_Gate unlocks a gate requested by the caller and returns the status of that gate.
3	Tries to unlock the gate (Gate can only be unlocked by the core that locked it) Constant: UNLOCK = 0
5	Refers to Get_Gate_Status

3.5 Reset Semaphore Gate, Reset_Gate

The Reset semaphore gate feature provides a flexible way for any core to reset a gate, or all gates, to a ready state (unlocked state). Specifically, Reset_Gate will release a locked gate, or gates, from a locked state to a ready state. The semaphore module uses a secure reset mechanism that requires a dual-write access pattern to reset a gate. This mechanism requires that consecutive writes with predefined data are issued from the same core. These consecutive writes are illustrated in Figure 5, which displays the Reset_Gate software routine. This software routine resets one or all semaphore gates, depending on the value of the gate number provided to the Reset_Gate routine. To reset a single gate, the gate number should be provided to the routine. To reset all gates, a value larger than 63 should be provided to the Reset_Gate routine.

```
//-------------------------------------------------------------------
// Reset_Gate
//-------------------------------------------------------------------
1: uint8_t Reset_Gate(uint8_tgate_no)
2: {
3:     uint8_t cnt = 0;
4:     SEMA4.RSTGT.R = (0xE2 << 8);
5:
6:     while (SEMA4.RSTGT.B.RSTGSM != 0x01)
7:     {
8:        if (cnt++>10) return 1
9:     }
10:
11:    SEMA4.RSTGT.R = (0x1D << 8) | gate_no;
12:
13:    return 0
14: }
```

Figure 5. Reset_Gate routine

The following table provides a description of the Reset_Gate software routine.

Line	Description
1	Reset_Gate will reset a gate or all gates requested by the caller • Reset one gate: a valid gate number • Reset all gates: a gate number greater than 63
4	Writes the first reset data pattern (0xE2<<8)
6	Waits for the state machine to prompt for the next data pattern
8	If it takes more than 10 whole loops, the routine will time out. (The state machine should not take more than two clock cycles.)
11	Writes the second reset data pattern (0xID << 8) and resets the gate(s) as specified by “gate_no“

4 Classic Semaphore Examples

Section 1, “Introduction,” mentions that semaphore routines are non-blocking. This seems like a contradiction because a semaphore should block to function. However, a non-blocking semaphore routine typically refers to a semaphore routine that allows an application to try to acquire a semaphore gate (lock gate) without entering into a spinlock (waiting in a loop while repeatedly checking if the lock is available). If the semaphore gate cannot be locked, then it will return fail but not block; it will not go into a spinlock trying to lock that gate. If the gate can be locked, the semaphore will lock it and return true (semaphore gate is locked).

4.1 Signaling semaphore

Perhaps the simplest use for a semaphore is signaling, which means that one task sends a signal to another task to indicate that something has happened. Signaling provides a mechanism to guarantee that a section of code of a task runs before a section of code for another task.

```
//-------------------------------------------------------------------
// Task 0 - Writer
//-------------------------------------------------------------------
1: void Task0 (void)
2: {
3:        uint8_t status = CORE0_LOCK;
4:
5:        //wait for gate0 unlock
6:        while (status != UNLOCK)
7:        {
8:                  status = Get_Gate_Status (GATE_0);
9:        }
10:
11:       sprintf (buf, "%s\n", "Core 0 send this message");
12:
13:       while (status != CORE0_LOCK)
14:       {
15:
16:                 status = Lock_Gate (GATE_0);
17:       }
18:  }
```

Figure 6. Signaling semaphore, writer function

```
//-------------------------------------------------------------------
// Task 1 - Reader
//-------------------------------------------------------------------
1: void Task1 (void)
2: {
3:        uint8_t status = CORE0_LOCK;
4:
5:        //wait for gate0 lock
6:        while (status != CORE0_LOCK)
7:        {
8:                  status = Get_Gate_Status (GATE_0);
9:        }
10:
11:       printf ("%s", buf);
12:
13:       while (status != UNLOCK)
14:       {
15:                 Reset_Gate (GATE_0);
16:                 status = Get_Gate_Status (GATE_0);
17:       }
18: }
```

Figure 7. Signaling semaphore, reader function

In this example, task 0 and task 1 will run in core 0 and core 1, respectively. Both tasks have shared access to a resource. Task 0 is the writer and task 1 is the reader. The reader must wait for the writer to finish writing the information to the shared resource and signal the reader before the reader can read the shared resource information and display it.

Figure 6 and Figure 7 list the C code for the writer and reader, task 0 and task 1, respectively.

This example illustrates simple writer and reader tasks using a signaling semaphore to resolve a *serialization* problem in a multi-core system. The writer and the reader work as follows:

Line	Writer (task 0)	Reader (task 1)
3	Constant: CORE0_LOCK = 1	Constant: CORE0_LOCK = 1
6–9	Waits for semaphore flag clear (gate 0 unlock) before proceeding to write shared resource Constant: Gate_0 = 0	Waits for semaphore flag to set (Waits for task 0 to signal, lock gate 0) before proceed to read shared resource. Constant: Gate_0 = 0
11	Writes shared resource	—
13–17	Set semaphore flag (Lock Gate 0); Signals task 1 that information is ready for read.	—
11	—	(Gate 0 is locked) Task 1 reads shared resource and print the line of text.
13–17	—	Task 1 clears semaphore flag using Reset_Gate and verifies if gate 0 is unlocked.

Note that Reset_Gate is used to release GATE_0 in this example instead of Unlock_Gate to unlock gate 0. This is because core 0 has locked gate 0 and has no idea when core 1 will read the information written by core 0. Moreover, core 1 cannot unlock the gate locked by core 0. The workaround is to use Reset_Gate to release gate 0 after core 1 has consumed the information written by core 0.

4.2 Blocking semaphore

This example uses a symmetric solution to resolve a coherency problem between two tasks; that is, both tasks call the same routine. Both cores competing to lock gate 0 will call the printer function to gain access to the line printer device to print a line of text. The first core to lock gate 0 will proceed to print its line of text and then unlock gate 0 before other core can lock gate 0.

This method ensures that the printer function is able to print a complete message from a task before beginning to print another message from a different task. This example avoids a potential data coherency problem between these two tasks.

```
//-------------------------------------------------------------------
// Printer
//-------------------------------------------------------------------
1: void Printer(uint8_t core_lock, char *msg)
2: {
3:         uint8_t status = UNLOCK;
4:
5:         // try to lock gate 0
6:         while (status != core_lock)
7:         {
8:                 status = Lock_Gate (GATE_0);
9:         }
10:
11:        Lpt1 (msg);
12:
13:        while (status == core_lock)
14:        {
15:                status = Unlock_Gate (GATE_0);
16:        }
17: }
```

Figure 8. Binary semaphore, printer function

Figure 8 lists the C code for the printer routine. This example illustrates a simple blocking semaphore (binary semaphore) between two tasks using symmetric solution. The printer routine works as follows:

Line	Core 0	Core 1
	Core 0 call printer function --- Printer(CORE0_LOCK,msg0); Constant: CORE0_LOCK = 1 UNLOCK = 0	Core 1 call printer function --- Printer(CORE1_LOCK,msg1); Constant: CORE1_LOCK = 2 UNLOCK = 0
6–9	Core 0 tries to lock gate 0 Constant: GATE_0 = 0	Core 1 tries to lock gate 0 Constant: GATE_0 = 0
	Assume core 0 locks gate 0	—
11	Core 0 print msg0	—
12–16	Core 0 unlock gate 0	—
	—	Assume core 1 now locks gate 0
11	—	Core 1 print msg1
12–16	—	Core 1 unlock gate 0

4.3 Rendezvous semaphore

Rendezvous requires that each task arriving at a rendezvous point must signal its presence and wait for all tasks to arrive at the rendezvous point before executing the next stage of code. This example attempts to solve a synchronization problem using a rendezvous semaphore.

In this example, two tasks running in core 0 and core 1 are synchronized at the rendezvous point. When either of these two tasks arrives at the rendezvous point, it must signal (lock a gate) to inform the other task of its arrival. When both tasks have arrived at the rendezvous point and signaled, then both tasks can start running the next stage of their code. The two tasks are then synchronized.

Figure 9 lists the C code for the rendezvous example. This example illustrates a simple rendezvous between two tasks to achieve synchronization. The code for this example is arranged side-by-side in the table to show the relationship between the two tasks.

Task 0	Task 1

//Task 0 arrives at rendezvous point: signals it is
//present.
// Constant:
// GATE_0 = 0
// GATE_1 = 1
// UNLOCK = 1
 	
Lock_Gate (GATE_0);
	//Task 1 arrives at rendezvous point: signals it is
	//present.
	// Constant:
	// GATE_0 = 0
	// GATE_1 = 1
	// UNLOCK = 1
	
	Lock_Gate (GATE_1);

//Check if all task 1 arrived?
status = UNLOCK;
while (status==UNLOCK)
{
  Status = Get_Gate_Status (GATE_1);
}
	//Check if all task 0 arrived?
	status = UNLOCK;
	while (status==UNLOCK)
	{
	  Status = Get_Gate_Status (GATE_0);
	}

All tasks arrived at rendezvous point and signaled their present	
Task 0 calls A_Function ()	Task 1 calls B_Function ()

Figure 9. Rendezvous semaphore, synchronization

5 Classic Issue: Deadlock

The example in Section 4.3, “Rendezvous semaphore,” can be rewritten, by changing the order of the code, to create a deadlock between task 0 and task1 at the rendezvous point.

In this scenario, when task 0 and task 1 arrive at the rendezvous point, neither signals its presence because it is waiting for the other task to signal first. That is, task 0 waits for task 1 to signal arrival while task 1 also waits for task 0 to signal arrival. In this case, there is a deadlock situation. This is a classic application problem using semaphore concepts. There are other classic and non-classic issues related to the usage of semaphore concepts. Please refer to semaphore literature for more details and resolutions.

6 Summary

The source code of these routines, Get_Core_ID, Get_Gate_Status, Lock_Gate, Unlock_Gate and Reset_Gate, provide examples of how the semaphore module can be accessed and configured. Users can use these routines in their applications without any modification. However, for those who wish to write their own hardware semaphore routines, these examples provide a head start.

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number: AN4805
Rev. 2
01/2014