TMS320TCI6487/8 Semaphore

User's Guide

Literature Number: SPRUEF6C
December 2006–Revised February 2008

Contents

List of Figures

List of Tables

Read This First

About This Manual

The TMS320TCI6487/8 semaphore module is used to support atomic arbitration among multiple CPUs for shared resources/peripherals. This document describes the usage of the semaphore and some of the CSL calls used to configure/use semaphore module.

Terms and Abbreviations

CPU— Central Processing Unit of the C64x+ Core

DSP— Digital Signal Processor

CSL— Chip Support Library

API— Application Programmer Interface

Trademarks

All trademarks are the property of their respective owners.

User's Guide
SPRUEF6C–December 2006–Revised February 2008

TMS320TCI6487/8 Semaphore

1 Module Overview

1.1 Introduction

In the TMS320TCI6487/8 device, multiple CPUs attempt to access shared resources simultaneously. To avoid resource conflict, the semaphore module is used to access shared resources in mutual exclusion. Unlike the software flag checking mechanism, the semaphore module is atomic, to satisfy the read-modify-write operation successfully.

The semaphore module allows acquisition of a semaphore resource/peripheral through read operations and also by posting a write request.

The number of resources the semaphore can handle is parameterized and, for theTMS320TCI6487/8 device, the following parameters are pre-programmed:

- NUM_SEM: Number of semaphore resources. In the device, the value is 32.
- QUEUE_DEPTH: Number of entries in each request queue. In the device, the value is 2.

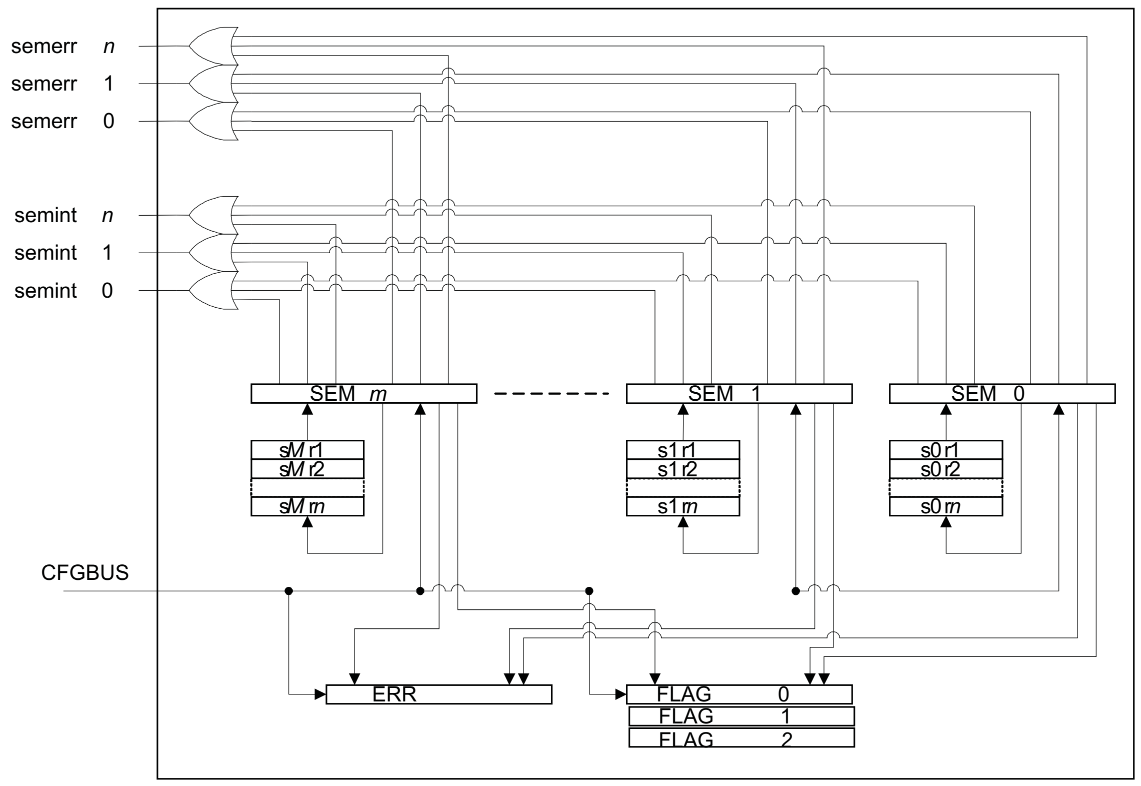
1.2 Semaphore Architecture

The semaphore module has unique interrupts for each of the DSP cores (masters) to identify when that master has acquired the requested resource. Likewise, there are unique error interrupts to each DSP core when a particular master attempts to access the semaphore resource that is already locked by the same/other master. For each semaphore peripheral there are three different registers associated with it to acquire the resource: direct, indirect, or combined mechanism.

Semaphore resources are not directly connected within the module. Through software programming hardware resources can be allocated to any of the semaphore resources. Figure 1 shows the basic building blocks of semaphore module.

In Figure 1, *m* is the number of semaphore resources (i.e., 32) and *n* is the number of the master (i.e., 3). There are three 32-bits registers associated with each semaphore resource: DIRECT, INDIRECT, and QUERY registers. For each DSP core there is a FLAG register to notify the status of all 32 semaphore resources. The ERR register shows the different types of errors and the semaphore peripheral ID and master ID that caused the specific error. Upon receiving the error or interrupt event, the DSP core clears the particular bit field of the FLAG register by programming the FLAG register and writing to the semaphore end-of-interrupt (EOI) register to re-arm or re-enable a particular master's error/interrupt line.

Figure 1. TMS320TCI6487/8 Semaphore Block Diagram



2 Semaphore Operation

Any of the DSP cores can use the shared semaphore resources.

2.1 Semaphore Access Modes

Any semaphore resource can be accessed by three different modes:

- **Direct Mode**

  In direct access mode, a resource is accessed by reading the DIRECTx (x is the semaphore peripheral number) register. Since there is no physical connection between the semaphore resources and the registers, any register can be mapped to any resource by software programming.

  If the resource is free, access is immediately granted to the requested core. The FREE bit of the DIRECTx register is set to 1 signifying that the resource is free and access is granted to the requested core. The OWNER field of the DIRECTx register is set to the requested core ID.

  If the resource is not available, no further action is taken by the semaphore module. The FREE bit of the DIRECTx register is set to 0 signifying that the resource is in use and the requested master should start a fresh request. In any case, there is no interrupt issued and there is no specific bit field set in the FLAGy (y is the granted core ID) register.

- **Indirect Mode**

  In indirect access mode, a resource is accessed by writing to the DIRECTx, INDIRECTx, or QUERYx (x is the semaphore peripheral number) register. Since there is no physical connection between the semaphore resources and the registers, any register can be mapped to any resource by software programming.

  If the resource is free, access is immediately granted to the requested core. The FREE bit of the DIRECTx, INDIRECTx, or QUERYx register is set to 1 signifying that the resource is free and acces is granted to the requested core. The OWNER field of the DIRECTx, INDIRECTx, or QUERYx register is set to the requested core ID. An interrupt is immediately issued to the requested core.

  If the resource is not available, the request is added to the request queue assigned to the semaphore resource. When the resource becomes free, an interrupt is issued and the FREE bit of the DIRECTx, INDIRECTx, or QUERYx register is set to 1 signifying that access is now granted to the requested core. An interrupt is issued in this case. The specific bit field for the semaphore resource of the FLAGy (y is the granted core ID) register is set.

- **Combined Mode**

  In combined access mode, a resource is accessed by reading the INDIRECTx (x is the semaphore peripheral number) register. Since there is no physical connection between the semaphore resources and the registers, any register can be mapped to any resource by software programming.

  If the resource is free, access is immediately granted to the requested core. The FREE bit of the INDIRECTx register is set to 1 signifying that the resource is free and access is granted to the requested core. The OWNER field of the INDIRECTx register is set to the requested core ID. No grant interrupt or flag register settings happen in this case.

  If the resource is not available, the request is added to the request queue assigned to the semaphore resource. When the resource becomes free, an interrupt is issued and the FREE bit of the INDIRECTx register is set to 1 signifying that access is now granted to the requested core. An interrupt is issued in this case. The specific bit field for the semaphore resource of the FLAGy (y is the granted core ID) register is set.

2.2 Interrupt Handling

The first interrupt occurs when the pending queued request is serviced. To ensure that a re-arm of the master's interrupt occurs prior to the next access by the same or other resource, the master should:

- read the FLAGx (x is the particular master id) register
- clear the flag bit by programming the specific bit field for the semaphore resource of the FLAGx register
- write to the EOI register.

2.3 Error Handling

The first error interrupt occurs when there is a semaphore access error. To ensure that a re-arm of the master's error interrupt occurs prior to the next access violation, the master should:

- read the ERR register to determine the error type
- clear the error by programming the ERR_CLEAR register
- write to the EOI register.

2.4 Status Query

Before acquiring any semaphore resource, the requested core checks the availability of the resource by reading the QUERYx (x is the semaphore peripheral number) register. The FREE bit signifies whether a particular resource is free or not; when FREE=0 the resource is not free, when FREE=1 the resource is free. The OWNER field has a master ID that currently holds the resource (when FREE=0) or shows 0x00 (when FREE=1). Before taking any action on a resource, check the resource status.

2.5 Releasing Semaphore Resources

When the master that is granted access is finished with the shared resource, it must free the resource so that another master can access it. Writing 1 to the FREE bit of the DIRECTx, INDIRECTx, or QUERYx register releases the specific resource.

3 Emulation Considerations

During debug, when using the emulator, the CPU(s) may be halted. During emulation halt, the debugger reads to certain semaphore registers is ignored to avoid changing semaphore state. In other words, the debugger read of semaphore registers (DIRECTx, INDIRECTx) does not change the state of the semaphore peripheral. But, during emulation halt, the debugger checks the semaphore peripheral status via the QUERYx register.

4 Semaphore Usage Examples

This section provides some of the API call lists and examples to access semaphore resources.

4.1 CSL API Calls

This section provides the basic usage of the semaphore module using different CSL API calls. Three typical modes of accessing peripherals are illustrated with examples. Example 1 shows some of the API call lists.

Example 1. Sample API Call Lists

```
########CSL_sem_API ##########
#define HW_SEM_RELEASE          1
#define HW_SEM_REQUEST          0
/*******************/
/*   ENUMERATIONS  */
/*******************/
typedef enum {
    CSL_SEM_ID0 = 0,
    CSL_SEM_ID1 = 1,
    CSL_SEM_ID2 = 2,
}CSL_SemOwnerId;

typedef enum {
    CSL_SEM_ERR0 = 0,
    CSL_SEM_ERR1 = 1,
    CSL_SEM_ERR2 = 2,
    CSL_SEM_ERR3 = 3,
    CSL_SEM_ERR4 = 4
}CSL_SemError;

typedef enum {
   CSL_SEM_NOTFREE = 0,        /* Semaphore is not available */
   CSL_SEM_FREE    = 1         /* Semaphore is available     */
}CSL_SemFlag;

typedef enum {
   CSL_SEM_REARM_SEMINT0 = 0,
   CSL_SEM_REARM_SEMINT1 = 1,
   CSL_SEM_REARM_SEMINT2 = 2,
   CSL_SEM_REARM_SEMINT_ALL = 0x10
}CSL_SemEOISet

typedef enum {
   CSL_SEM_QUERY_REVISION,
   CSL_SEM_QUERY_ERROR,
   CSL_SEM_QUERY_FLAGS,
   CSL_SEM_QUERY_STATUS,
   CSL_SEM_QUERY_DIRECT,
   CSL_SEM_QUERY_INDIRECT
}CSL_SemHwStatusQuery;
```

Example 1. Sample API Call Lists (continued)

```
typedef enum {
   CSL_SEM_CMD_EOI_WRITE,
   CSL_SEM_CMD_FLAG_SET,
   CSL_SEM_CMD_FREE_DIRECT,
   CSL_SEM_CMD_WRITE_POST_DIRECT,
   CSL_SEM_CMD_FREE_INDIRECT,
   CSL_SEM_CMD_WRITE_POST_INDIRECT,
   CSL_SEM_CMD_FREE_QUERY,
   CSL_SEM_CMD_WRITE_POST_QUERY,
   CSL_SEM_CMD_CLEAR_ERR,
   CSL_SEM_CMD_CLEAR_FLAGS
}CSL_SemHwControlCmd;
/*******************/
/* DATA STRUCTURES */
/*******************/
typedef struct _CSL_SemFlagClear_Arg{
   CSL_BitMask32        mask;
   CSL_SemOwnerId       masterId;
}CSL_SemFlagSetClear_Arg;

typedef struct {
   CSL_SemRegsOvly regs;
} CSL_SemBaseAddress;

typedef struct {
   CSL_BitMask16 flags;
} CSL_SemParam;


typedef struct {
   Uint16 contextInfo;
} CSL_SemContext;

typedef struct CSL_SemObj{
   CSL_InstNum     instNum;
   int             semNum;
   CSL_SemRegsOvly regs;
}CSL_SemObj;

typedef volatile CSL_SemObj *CSL_SemHandle;

typedef struct {
   int              semNum;
   CSL_SemOwnerId   semOwner;
   CSL_SemFlag      semFree;
} CSL_SemVal;

typedef struct CSL_SemFaultStatus {
    int           semNum;
    CSL_SemError  errorMask;
    Uint16        faultID;
}CSL_SemFaultStatus;
```

4.2 Accessing Semaphore Resource When Free

A CPU accesses a shared peripheral (semaphore resource) by reading the semaphore direct (DIRECTx) register when the resource is free to use. Checking the semaphore query (QUERYx) register is the way to verify whether a particular resource is free or still in use by other CPU. If the FREE bit of QUERYx (where x is the semaphore peripheral ID) register is 1, the requested CPU directly locks the peripheral by reading the DIRECTx register and the requested peripheral is immediately granted access. After finishing the access, the CPU must release the resource by writing 1 to the FREE bit of the DIRECTx register so that another CPU can access the same resource.

Example 2 shows a semaphore resource (NUM = 4) acquired using the direct-read mode.

Example 2. Sample Code to Access Semaphore in Direct-Read Mode

```
        CSL_SemVal     query;
        CSL_SemHandle  mySemHandle;
        CSL_SemObj     mySemObj;
        CSL_SemContext SemContext;
        CSL_SemParam   mySemParam;

        mySemParam.flags = 4;
        CSL_semInit(&SemContext);

        mySemHandle = CSL_semOpen(&mySemObj, 0,&mySemParam, NULL);

        if(mySemHandle != NULL) {
           CSL_semGetHwStatus(mySemHandle,CSL_SEM_QUERY_STATUS,&query);
        }

        if(query.semFree == CSL_SEM_FREE) {
           CSL_semGetHwStatus(mySemHandle, CSL_SEM_QUERY_DIRECT,&query);
        }
        CSL_semGetHwStatus(mySemHandle,CSL_SEM_QUERY_STATUS,&query);
        if (query.semFree != CSL_SEM_FREE)  {
          // Do peripheral access
        }
        else {
             // return error.
        }

        CSL_semHwControl(mySemHandle, CSL_SEM_CMD_FREE_DIRECT,NULL);
/* END Of Main code */
```

4.3 Accessing Semaphore Resource When Not Free

In certain situations when a shared resource is in use by another master, the requesting master/CPU can still post a request indirectly so that when the resource becomes available the requested master/CPU gets the access. Writing to the semaphore indirect (INDIRECTx) register when the resource is not free is the way to post a request in the request queue for the particular semaphore peripheral. Checking the semaphore query (QUERYx) register is the way to verify whether a particular resource is free or still in use by other CPU. If the FREE bit of the QUERYx (where x is the semaphore peripheral ID) register is 0 (indicating resource is in use), then the requested CPU can post a request by writing to the INDIRECTx register. When the resource becomes free, the master is notified by the interrupt and the requested peripheral is granted access. After finishing the access the CPU must release the resource by writing 1 to the FREE bit of the INDIRECTx register so that another CPU can access the same resource.

Example 3 shows a semaphore resource (NUM = 22) acquired using indirect-write mode.

Example 3. Sample Code to Access Semaphore in Indirect-Write Mode

```
 CSL_SemVal     query;
 CSL_SemHandle  mySemHandle;
 CSL_SemObj     mySemObj;
 CSL_SemContext SemContext;
 CSL_SemParam    mySemParam;

 mySemParam.flags = 22;
 CSL_semInit(&SemContext);

 mySemHandle = CSL_semOpen(&mySemObj, 0,&mySemParam, NULL);

 if(mySemHandle != NULL) {
    CSL_semGetHwStatus(mySemHandle,CSL_SEM_QUERY_STATUS,&query);
 }

 if(query.semFree == CSL_SEM_FREE) {
   // Direct access
   CSL_semGetHwStatus(mySemHandle,CSL_SEM_QUERY_DIRECT,&query);
 }

 CSL_semGetHwStatus(mySemHandle,CSL_SEM_QUERY_STATUS,&query);

 if (query.semFree != CSL_SEM_FREE)  {
   // Indirect access
    CSL_semHwControl(mySemHandle, CSL_SEM_CMD_WRITE_POST_INDIRECT,NULL);
    // Release the previously locked semaphore
   CSL_semHwControl(mySemHandle, CSL_SEM_CMD_FREE_DIRECT,NULL);
 }
// Get the semaphore status.Still lock. This time for indirect pending access
 CSL_semGetHwStatus(mySemHandle,CSL_SEM_QUERY_STATUS,&query);
if (query.semFree != CSL_SEM_FREE) {
   // do semaphore access
 }
 else {
    // Return error
 }

 CSL_semHwControl(mySemHandle, CSL_SEM_CMD_FREE_INDIRECT,NULL);
 /* END Of Main code */
```

4.4 Accessing Semaphore Resource in Combined Mode

A semaphore resource can be accessed in combination of direct-read mode and indirect-write mode. When the resource is free, access is immediately granted and when the resource is not free, it posts a request in the request queue.

Example 4 shows a semaphore resource (NUM = 0) acquired using combined-access mode.

Example 4. Sample Code to Access Semaphore in Combined-Access Mode

```
        CSL_SemVal     query;
        CSL_SemHandle  mySemHandle;
        CSL_SemObj     mySemObj;
        CSL_SemContext SemContext;
        CSL_SemParam   mySemParam;

        mySemParam.flags = 0;
        CSL_semInit(&SemContext);

        mySemHandle = CSL_semOpen(&mySemObj, 0,&mySemParam, NULL);

        if(mySemHandle != NULL) {
           CSL_semGetHwStatus(mySemHandle,CSL_SEM_QUERY_STATUS,&query);
        }

        if(query.semFree == CSL_SEM_FREE) {
           CSL_semGetHwStatus(mySemHandle, CSL_SEM_QUERY_INDIRECT,&query);
        }
        CSL_semGetHwStatus(mySemHandle,CSL_SEM_QUERY_STATUS,&query);
        if (query.semFree != CSL_SEM_FREE)  {
          // Do peripheral access
        }
        else {
     // return error.
        }

        CSL_semHwControl(mySemHandle, CSL_SEM_CMD_FREE_QUERY,NULL);
/* END Of Main code */
```

5 Semaphore Registers

This section provides the semaphore memory map and descriptions of the peripheral registers.

5.1 Register Memory Map

Table 1. Register Memory Map

Offset	Register	Description	See
0x000	PID	Peripheral Revision ID Register	Section 5.2.1
0x00C	EOI	EOI Register	Section 5.2.2
0x100 - 0x17C	DIRECT0-31	Direct Registers 0-31	Section 5.2.3
0x200 - 0x27C	INDIRECT0-31	Indirect Registers 0-31	Section 5.2.4
0x300 - 0x37C	QUERY0-31	Query Registers 0-31	Section 5.2.5
0x400	FLAG0	Flag0 Register (for C64x+ Core0)	Section 5.2.6
0x404	FLAG1	Flag1 Register (for C64x+ Core1)	Section 5.2.6
0x408	FLAG2	Flag2 Register (for C64x+ Core2)	Section 5.2.6
0x480	FLAG_SET0	Flag Set0 Register (for C64x+ Core0)	Section 5.2.7
0x484	FLAG_SET1	Flag Set1 Register (for C64x+ Core1)	Section 5.2.7
0x488	FLAG_SET2	Flag Set2 Register (for C64x+ Core2)	Section 5.2.7
0x500	ERR	Error Register	Section 5.2.8
0x504	ERROR_CLR	Error Clear Register	Section 5.2.9

5.2 Register Descriptions

The following are some sample semaphore peripheral registers.

5.2.1 Peripheral ID Register (PID)

The semaphore peripheral ID (PID) register is shown in Figure 2 and described in Table 2.

Figure 2. Peripheral ID Register (PID)

31-30	29-28	27-16
SCHEME	Reserved	FUNC
R-0x1	R-0x	R-0x802

15-11	10-8	7-6	5-0
RTL	MAJOR	CUSTOM	MINOR
R-0x0	R-0x1	R-0x0	R-0x0

LEGEND: R/W = Read/Write; R = Read only; -*n* = value after reset

Table 2. Peripheral ID Register (PID) Field Descriptions

Bit	Field	Value	Description
31-30	SCHEME	01b	Used to distinguish which ID scheme is used.
29-28	RSVD	00	Reserved. Read returns 0. Write has no effect.
27-16	FUNC	0x802	Specifies module family
15-11	RTL	0000b	RTL Version
10-8	MAJOR	01b	Major Revision
7-6	CUSTOM	00b	Special/Custom Revision
5-0	MINOR	000000b	Minor Revision

5.2.2 End-of-Interrupt Register (EOI)

The semaphore end-of-interrupt (EOI) register is used for re-arming the error/interrupt line after serving the existing error/interrupt. The EOI register is shown in Figure 3 and described in Table 3.

Figure 3. End-of-Interrupt Register (EOI)

31-16
Reserved
W

15-8	7-0
Reserved	INTERRUPT/ERROR SELECT
W	W

LEGEND: W = Write only; -*n* = value after reset

Table 3. End-of-Interrupt Register (EOI) Field Descriptions[(1)]

Bit	Field	Value	Description
31-8	Reserved		Reserved, Write has no effect
7-0	INTERRUPT/ERROR SELECT		Selection of particular Error/Interrupt line to re-arm.
		0x00	Re-enable semint0 (For C64x+ Core0)
		0x01	Re-enable semint1 (For C64x+ Core1)
		0x02	Re-enable semint2 (For C64x+ Core2)
		0x10	Re-enable All Error interrupt (For all C64x+ Cores)

(1) Reading of the EOI register will result in a memory-read exception being generated.

5.2.3 Direct Register (DIRECTx)

The semaphore direct (DIRECTx) register acquires the semaphore resource in direct-read mode as well as indirect-write mode. If the resource is free, the FREE field of the DIRECTx (x is the semaphore peripheral ID being requested) signifies whether the resource is granted or not. The DIRECTx register is shown in Figure 4 and described in Table 4.

Figure 4. Direct Register (DIRECTx)

31	16
Reserved	
R-0x0000	

15	8	7	1	0
OWNER		Reserved		FREE
R-0x00		R-0x00		R/W-0x1

LEGEND: R/W = Read/Write; R = Read only; -*n* = value after reset

Table 4. Direct Register (DIRECTx) Field Descriptions

Bit	Field	Value	Description
31-16	Reserved	0x0000	Reserved. Read returns 0. Write has no effect.
15-8	OWNER	0x00	When FREE = 0: Semaphore resource owner ID When FREE = 1: The value returns 0x00
7-1	Reserved	0x00	Reserved. Read returns 0. Write has no effect.
0	FREE		Read Operation:
		0	Semaphore is not granted.
		1	Semaphore is granted to the Master. FREE is cleared by the hardware at the end of the access.
			Write Operation:
		0	Request is posted in the queue (indirect mode).
		1	Semaphore is freed.

5.2.4 Indirect Register (INDIRECTx)

The semaphore indirect (INDIRECTx) register acquires the semaphore resource in indirect-write mode. If the resource is free, the FREE bit of the INDIRECTx (x is the semaphore peripheral ID being requested) signifies whether the resource is granted or not. In addition, an interrupt is sent to the requested master. The INDIRECT register is shown in Figure 5 and described in Table 5.

Figure 5. Indirect Register (INDIRECTx)

31–16
Reserved
R-0x0000

15–8	7–1	0
OWNER	Reserved	FREE
R-0x00	R-0x00	R/W-0x1

LEGEND: R/W = Read/Write; R = Read only; -*n* = value after reset

Table 5. Indirect Register (INDIRECTx) Field Descriptions

Bit	Field	Value	Description
31-16	Reserved	0x0000	Reserved. Read returns 0. Write has no effect.
15-8	OWNER	0x00	When FREE = 0: Semaphore resource owner ID When FREE = 1: The value returns 0x00
7-1	Reserved	0x00	Reserved. Read returns 0. Write has no effect.
0	FREE		Read Operation:
		0	Semaphore is not granted.
		1	Semaphore is granted to the Master. FREE is cleared by the hardware at the end of the access.
			Write Operation:
		0	Request is posted in the queue (indirect mode).
		1	Semaphore is freed.

5.2.5 Query Register (QUERYx)

Each semaphore query register (QUERYx) checks the current semaphore resource status and also can be used for indirect-write mode to access the particular resource. Reading the QUERYx (x is the semaphore peripheral ID being accessed) register does not affect the status of the particular peripheral. The QUERY register is shown in Figure 6 and described in Table 6.

Figure 6. Query Register (QUERYx)

31 – 16
Reserved
R-0x0000

15 – 8	7 – 1	0
OWNER	Reserved	FREE
R-0x00	R-0x00	R/W-0x1

LEGEND: R/W = Read/Write; R = Read only; -*n* = value after reset

Table 6. Query Register (QUERYx) Field Descriptions

Bit	Field	Value	Description
31-16	Reserved	0x0000	Reserved. Read returns 0. Write has no effect.
15-8	OWNER	0x00	When FREE = 0: Semaphore resource owner ID When FREE = 1: The value returns 0x00
7-1	Reserved	0x00	Reserved. Read returns 0. Write has no effect.
0	FREE		Read Operation:
		0	Semaphore is not granted
		1	Semaphore is not granted to the Master
			Write Operation:
		0	Request is posted in the queue (indirect mode)
		1	Semaphore is freed

5.2.6 Flag Register (FLAGx)

Each semaphore flag (FLAGx, where x is the Master ID) register checks whether the particular master/core is currently holding the semaphore resource or not. There is one register for each core and the particular bit field signifies the holding status of the particular semaphore peripheral by the requested master. Once access to the particular semaphore resource is complete, the corresponding flag bit is cleared by writing a 1 to the particular bit of the FLAG register. The FLAGx register is shown in Figure 7 and described in Table 7.

Figure 7. Flag Register (FLAGx)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
F31	F30	F29	F28	F27	F26	F25	F24	F23	F22	F21	F20	F19	F18	F17	F16

R-0x0000

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
F15	F14	F13	F12	F11	F10	F9	F8	F7	F6	F5	F4	F3	F2	F1	F0

R-0x0000

LEGEND: R/W = Read/Write; R = Read only; -*n* = value after reset

Table 7. Flag Register (FLAGx) Field Descriptions

Bit	Field	Value	Description
31-0	Fy		Semaphore Flag Value
		0x0	Semaphore resource y is not owned by the master x
		0x1	Semaphore resource y is owned by the master x

5.2.7 Flag Set Register (FLAG_SETx)

Each semaphore flag set (FLAG_SETx, where x is the Master ID) register sets the flag bit of the particular semaphore peripheral. This register is implemented to check, by software programming, whether any resource can be accessed by the any of the masters. There is one register for each core and writing to the particular bit field sets the semaphore flag registers (FLAGx) corresponding bit-field value. The FLAG_SETx register is shown in Figure 8 and described in Table 8.

Figure 8. Flag Set Register (FLAG_SETx)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
F31	F30	F29	F28	F27	F26	F25	F24	F23	F22	F21	F20	F19	F18	F17	F16

W-0x0000

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
F15	F14	F13	F12	F11	F10	F9	F8	F7	F6	F5	F4	F3	F2	F1	F0

W-0x0000

LEGEND: R/W = Read/Write; R = Read only; -*n* = value after reset

Table 8. Flag Set Register (FLAG_SETx) Field Descriptions

Bit	Field	Value	Description
31-0	Fy		Semaphore Flag Set
		0x0	Do nothing
		0x1	Semaphore flag y is cleared

5.2.8 Error Register (ERR)

The semaphore error (ERR) register updates any kind of error that occurs while acquiring a semaphore resource by any core. By reading the ERR register the particular master/core should clear the particular error and program the EOI register so that a re-arm occurs for the next semaphore access by the same master. The ERR register is shown in Figure 9 and described in Table 9.

Figure 9. Error Register (ERR)

31 — 16
Reserved
R-0x0000

15 — 8	7 — 3	2 — 0
FAULTID	SEM_NUM	ERR
R-0x00	R-0x0	R-0x0

LEGEND: R/W = Read/Write; R = Read only; -*n* = value after reset

Table 9. Error Register (ERR) Field Descriptions

Bit	Field	Value	Description
31-16	Reserved	0x0000	Reserved. Read returns 0. Write has no effect.
15-8	FAULTID	0x00	Master ID number that caused the error
7-3	SEM_NUM	0x00	Semaphore peripheral ID number (0 to 31)
2-0	ERR		Semaphore error code.
		000	No semaphore access error has occurred.
		001	Master ID FAULTID attempted to free semaphore NUM when it was already free.
		010	Master ID FAULTID attempted to free semaphore NUM while not currently owned by FAULTID.
		011	Master ID FAULTID attempted to acquire semaphore NUM while it was already owned by FAULTID.
		100	Master ID FAULTID attempted to acquire semaphore NUM while FAULTID already had a request pending.

5.2.9 Error Clear Register (ERROR_CLR)

The semaphore error clear (ERROR_CLR) register clears the existing error code. The master should reprogram the EOI register after clearing the error so that a re-arm occurs for the next error event. The ERROR_CLR register is shown in Figure 10 and described in Table 10.

Figure 10. Error Clear Register (ERROR_CLR)

31 – 16
Reserved
R-0x0000

15 – 8	7 – 3	2 – 0
FAULTID	SEM_NUM	ERR
R-0x00	R-0x0	R-0x0

LEGEND: R/W = Read/Write; R = Read only; -*n* = value after reset

Table 10. Error Register (ERROR_CLR) Field Descriptions

Bit	Field	Value	Description
31-16	Reserved	0x0000	Reserved. Read returns 0. Write has no effect.
15-8	FAULTID	0x00	Master ID number that caused the error
7-3	SEM_NUM	0x00	Semaphore peripheral ID number (0 to 31)
2-0	ERR		Semaphore error code.
		000	No semaphore access error has occurred.
		001	Master ID FAULTID attempted to free semaphore NUM when it was already free.
		010	Master ID FAULTID attempted to free semaphore NUM while not currently owned by FAULTID.
		011	Master ID FAULTID attempted to acquire semaphore NUM while it was already owned by FAULTID.
		100	Master ID FAULTID attempted to acquire semaphore NUM while FAULTID already had a request pending.

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Clocks and Timers	www.ti.com/clocks	Digital Control	www.ti.com/digitalcontrol
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
RFID	www.ti-rfid.com	Telephony	www.ti.com/telephony
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless