Z-Stack
OS Abstraction Layer
Application Programming Interface

Document Number: F8W-2003-0002

Texas Instruments, Inc.
San Diego, California USA
(619) 542-1200

Version	Description	Date
1.0	Initial ZigBee v1.0 release.	04/08/2005
1.1	Added note on NV initialization to NV Memory API introduction.	07/22/2005
1.2	Modified discussion of .the Task Management API.	08/25/2005
1.3	Changed logo on title page, changed copyright on page footer.	02/27/2006
1.4	Modified power management API.	11/27/2006

TABLE OF CONTENTS

1. INTRODUCTION 1

1.1 PURPOSE 1
1.2 SCOPE 1
1.3 ACRONYMS 1

2. API OVERVIEW 2

2.1 OVERVIEW 2

***3.* MESSAGE MANAGEMENT API 3**

3.1 INTRODUCTION 3
3.2 OSAL_MSG_ALLOCATE ( ) 3
3.2.1 Description 3
3.2.2 Prototype 3
3.2.3 Parameter Details 3
3.2.4 Return 3
3.3 OSAL_MSG_DEALLOCATE( ) 3
3.3.1 Description 3
3.3.2 Prototype 3
3.3.3 Parameter Details 3
3.3.4 Return 3
3.4 OSAL_MSG_SEND( ) 4
3.4.1 Description 4
3.4.2 Prototype 4
3.4.3 Parameter Details 4
3.4.4 Return 4
3.5 OSAL_MSG_RECEIVE( ) 5
3.5.1 Description 5
3.5.2 Prototype 5
3.5.3 Parameter Details 5
3.5.4 Return 5

4. TASK SYNCHRONIZATION API 6

4.1 INTRODUCTION 6
4.2 OSAL_SET_EVENT( ) 6
4.2.1 Description 6
4.2.2 Prototype 6
4.2.3 Parameter Details 6
4.2.4 Return 6

5. TIMER MANAGEMENT API 7

5.1 INTRODUCTION 7
5.2 OSAL_START_TIMER( ) 7
5.2.1 Description 7
5.2.2 Prototype 7
5.2.3 Parameter Details 7
5.2.4 Return 7
5.3 OSAL_START_TIMEREX( ) 8
5.3.1 Description 8
5.3.2 Prototype 8
5.3.3 Parameter Details 8
5.3.4 Return 8

5.4 OSAL_STOP_TIMER( ) 8
5.4.1 Description 8
5.4.2 Prototype 8
5.4.3 Parameter Details 8
5.4.4 Return 9
5.5 OSAL_STOP_TIMEREX( ) 9
5.5.1 Description 9
5.5.2 Prototype 9
5.5.3 Parameter Details 9
5.5.4 Return 9
5.6 OSAL_GETSYSTEMCLOCK( ) 9
5.6.1 Description 9
5.6.2 Prototype 9
5.6.3 Parameter Details 9
5.6.4 Return 10

6. INTERRUPT MANAGEMENT API 11

6.1 INTRODUCTION 11
6.2 OSAL_INT_ENABLE( ) 11
6.2.1 Description 11
6.2.2 Prototype 11
6.2.3 Parameter Details 11
6.2.4 Return 11
6.3 OSAL_INT_DISABLE( ) 11
6.3.1 Description 11
6.3.2 Prototype 11
6.3.3 Parameter Details 11
6.3.4 Return 12

7. TASK MANAGEMENT API 13

7.1 INTRODUCTION 13
7.2 OSAL_INIT_SYSTEM() 13
7.2.1 Description 13
7.2.2 Prototype 13
7.2.3 Parameter Details 13
7.2.4 Return 13
7.3 OSAL_START_SYSTEM() 13
7.3.1 Description 13
7.3.2 Prototype 13
7.3.3 Parameter Details 13
7.3.4 Return 13
7.4 OSAL_SELF() 14
7.4.1 Description 14
7.4.2 Prototype 14
7.4.3 Parameter Details 14
7.4.4 Return 14
7.5 OSALTASKADD () 14
7.5.1 Description 14
7.5.2 Prototype 14
7.5.3 Parameter Details 15
7.5.4 Return 15

8. MEMORY MANAGEMENT API 16

8.1 INTRODUCTION 16
8.2 OSAL_MEM_ALLOC() 16

8.2.1 Description *16*
8.2.2 Prototype *16*
8.2.3 Parameter Details *16*
8.2.4 Return *16*
8.3 OSAL_MEM_FREE() 16
8.3.1 Description *16*
8.3.2 Prototype *16*
8.3.3 Parameter Details *16*
8.3.4 Return *16*
9. POWER MANAGEMENT API 17
9.1 INTRODUCTION 17
9.2 OSAL_PWRMGR_DEVICE() 17
9.2.1 Description *17*
9.2.2 Prototype *17*
9.2.3 Parameter Details *17*
9.2.4 Return *17*
9.3 OSAL_PWRMGR_TASK_STATE() 18
9.3.1 Description *18*
9.3.2 Prototype *18*
9.3.3 Parameter Details *18*
9.3.4 Return *18*
10. NON-VOLATILE MEMORY API 19
10.1 INTRODUCTION 19
10.2 OSAL_NV_ITEM_INIT() 19
10.2.1 Description *19*
10.2.2 Prototype *20*
10.2.3 Parameter Details *20*
10.2.4 Return *20*
10.3 OSAL_NV_READ() 20
10.3.1 Description *20*
10.3.2 Prototype *20*
10.3.3 Parameter Details *20*
10.3.4 Return *20*
10.4 OSAL_NV_WRITE() 21
10.4.1 Description *21*
10.4.2 Prototype *21*
10.4.3 Parameter Details *21*
10.4.4 Return *21*
10.5 OSAL_OFFSETOF() 21
10.5.1 Description *21*
10.5.2 Prototype *21*
10.5.3 Parameter Details *21*

1. Introduction

1.1 Purpose

The purpose of this document is to define the OS Abstraction Layer (OSAL) API. This API allows the software components in the Z-stack to be written independently of the specifics of the operating system, kernel or tasking environment (including control loops or connect-to-interrupt systems). The OSAL is implemented on the target.

1.2 Scope

This document enumerates all the function calls provided by the OSAL. The function calls are specified in sufficient detail so as to allow a programmer to implement them.

1.3 Acronyms

API	Application Programming Interface
OSAL	Operating System (OS) Abstraction Layer
PC	Personal Computer
SPI	Serial Port Interface

2. API Overview

2.1 Overview

The OS abstraction layer is used to shield the Z-Stack software components from the specifics of the processing environment. It provides the following functionality in a manner that is independent of the processing environment.

1. Task registration, initialization, starting
2. Message exchange between tasks
3. Task synchronization
4. Interrupt handling
5. Timers
6. Memory allocation

3. Message Management API

3.1 Introduction

The message management API provides a mechanism for exchanging messages between tasks or processing elements with distinct processing environments (for example, interrupt service routines or functions called within a control loop).The functions in this API enable a task to allocate and de-allocate message buffers, send command messages to another task and receive reply messages.

3.2 osal_msg_allocate ( )

3.2.1 Description

This function is called by a task to allocate a message buffer, the task/function will then fill in the message and call `osal_msg_send()` to send the message to another task. If the buffer can not be allocated, msg_ptr will be set to NULL.

NOTE: Do not confuse this function with `osal_mem_alloc()`, this function is used to allocate a buffer to send messages between tasks [using `osal_msg_send()`]. Use `osal_mem_alloc()` to allocate blocks of memory.

3.2.2 Prototype

```
byte *osal_msg_allocate( uint16 len )
```

3.2.3 Parameter Details

`len` is the length of the message.

3.2.4 Return

The return value is a pointer to the buffer allocated for the message. A NULL return indicates the message allocation operation failed.

3.3 osal_msg_deallocate( )

3.3.1 Description

This function is used to de-allocate a message buffer. This function is called by a task (or processing element) after it has finished processing a received message.

3.3.2 Prototype

```
byte osal_msg_deallocate( byte *msg_ptr )
```

3.3.3 Parameter Details

`msg_ptr` is a pointer to the message buffer that needs to be de-allocated.

3.3.4 Return

Return value indicates the result of the operation.

RETURN VALUE	DESCRIPTION
ZSUCCESS	De-allocation Successful
INVALID_MSG_POINTER	Invalid message pointer
MSG_BUFFER_NOT_AVAIL	Buffer is queued

3.4 osal_msg_send( )

3.4.1 Description

Th osal_msg_send function is called by a task to send a command or data message to another task or processing element. The destination_task identifier field must refer to a valid system task. Task identifiers are assigned when osal_create_task ( ) is called to start a task. The osal_msg_send() function will also set the SYS_EVENT_MSG event in the destination tasks event list.

3.4.2 Prototype

```
byte osal_msg_send( byte destination_task, byte *msg_ptr )
```

3.4.3 Parameter Details

`destination_task` is the ID of the task to receive the message.

`msg_ptr` is a pointer to the buffer containing the message. Msg_ptr must be a pointer to a valid message buffer allocated via osal_msg_allocate ().

3.4.4 Return

Return value is a 1 byte field indicating the result of the operation.

RETURN VALUE	DESCRIPTION
ZSUCCESS	Message sent successfully
INVALID_MSG_POINTER	Invalid Message Pointer
INVALID_TASK	Destination_task is not valid

3.5 osal_msg_receive( )

3.5.1 Description

This function is called by a task to retrieve a received command message. The calling task must de-allocate the message buffer after processing the message using the osal_msg_deallocate() call.

3.5.2 Prototype

```
byte *osal_msg_receive( byte task_id )
```

3.5.3 Parameter Details

`task_id` is the identifier of the calling task (to which the message was destined).

3.5.4 Return

Return value is a pointer to a buffer containing the message or NULL if there is no received message.

4. Task Synchronization API

4.1 Introduction

This API enables a task to wait for events to happen and return control while waiting. The functions in this API can be used to set events for a task and notify the task once any event is set.

4.2 osal_set_event( )

4.2.1 Description

This function is called to set the event flags for a task.

4.2.2 Prototype

```
byte osal_set_event( byte task_id, UINT16 event_flag )
```

4.2.3 Parameter Details

`task_id` is the identifier of the task for which the event is to be set.

`event_flag` is a 2 byte bitmap with each bit specifying an event. There is only 1 system event (SYS_EVENT_MSG), the rest of the events/bits are defined by the receiving task.

4.2.4 Return

Return value indicates the result of the operation.

RETURN VALUE	DESCRIPTION
ZSUCCESS	Success
INVALID_TASK	Invalid Task

5. Timer Management API

5.1 Introduction

This API enables the use of timers by internal (Z-Stack) tasks as well as external (Application level) tasks. The API provides functions to start and stop a timer. The timers can be set in increments of 1millisecond.

5.2 osal_start_timer( )

5.2.1 Description

This function is called to start a timer. When the timer expires, the given event bit will be set. The event will be set in the task from which the osal_start_timer function is called. To explicitly specify the task id, use the `osal_start_timerEx()` function instead of `osal_start_timer`().

5.2.2 Prototype

```
byte osal_start_timer(UINT16 event_id, UINT16 timeout_value);
```

5.2.3 Parameter Details

`event_id` is a user defined event bit. When the timer expires, the calling task will be notified (event).

`timeout_value` is the amount of time (in milliseconds) before the timer event is set.

5.2.4 Return

Return value indicates the result of the operation.

RETURN VALUE	DESCRIPTION
ZSUCCESS	Timer Start Successful
NO_TIMER_AVAILABLE	Unable to start the timer

5.3 osal_start_timerEx( )

5.3.1 Description

This is very similar to `osal_start_timer()`, with the added parameter of `taskID`. This allows the caller to set the timer for another task. When in doubt, use this function over `osal_start_timer()`.

5.3.2 Prototype

```
    byte osal_start_timerEx( byte taskID, UINT16 event_id, UINT16
timeout_value);
```

5.3.3 Parameter Details

`tasked` is the task ID of the task that is to get the event when the timer expires..

`event_id` is a user defined event bit. When the timer expires, the calling task will be notified (event).

`timeout_value` is the amount of time (in milliseconds) before the timer event is set.

5.3.4 Return

Return value indicates the result of the operation.

RETURN VALUE	DESCRIPTION
ZSUCCESS	Timer Start Successful
NO_TIMER_AVAILABLE	Unable to start the timer

5.4 osal_stop_timer( )

5.4.1 Description

This function is called to stop a timer that has already been started. If successful, the function will cancel the timer and prevent the event associated with the timer from being set for the calling task. Use of the `osal_stop_timer()` function implies the timer is running in the context of the task calling `osal_stop_timer()`. To stop a timer in a different task, use `osal_stop_timerEx()` instead of `osal_stop_timer()`.

5.4.2 Prototype

```
      byte osal_stop_timer( UINT16 event_id );
```

5.4.3 Parameter Details

`event_id` is the identifier of the timer that is to be stopped.

5.4.4 Return

Return value indicates the result of the operation.

RETURN VALUE	DESCRIPTION
ZSUCCESS	Timer Stopped Successfully
INVALID_EVENT_ID	Invalid Event

5.5 osal_stop_timerEx( )

5.5.1 Description

This function is similar to osal_stop_timer with the exception that the task_id can be specified in a call to osal_stop_timerEx.

5.5.2 Prototype

```
byte osal_stop_timerEx( byte task_id, UINT16 event_id );
```

5.5.3 Parameter Details

`task_id` is the task to stop the timer for.

`event_id` is the identifier of the timer that is to be stopped.

5.5.4 Return

Return value indicates the result of the operation.

RETURN VALUE	DESCRIPTION
ZSUCCESS	Timer Stopped Successfully
INVALID_EVENT_ID	Invalid Event

5.6 osal_GetSystemClock( )

5.6.1 Description

This function is called to read the system clock

5.6.2 Prototype

```
uint32 osal_GetSystemClock( void );
```

5.6.3 Parameter Details

`None`.

5.6.4 Return

The system clock in millseconds.

6. Interrupt Management API

6.1 Introduction

This API enables a task to interface with external interrupts. The functions in the API allow a task to associate a specific service routine with each interrupt. The interrupts can be enabled or disabled. Inside the service routine, events may be set for other tasks.

6.2 osal_int_enable( )

6.2.1 Description

This function is called to enable an interrupt. Once enabled, occurrence of the interrupt causes the service routine associated with that interrupt to be called.

6.2.2 Prototype

```
byte osal_int_enable( byte interrupt_id )
```

6.2.3 Parameter Details

`interrupt_id` identifies the interrupt to be enabled.

6.2.4 Return

Return value indicates the result of the operation.

RETURN VALUE	DESCRIPTION
ZSUCCESS	Interrupt Enabled Successfully
INVALID_INTERRUPT_ID	Invalid Interrupt

6.3 osal_int_disable( )

6.3.1 Description

This function is called to disable an interrupt. When a disabled interrupt occurs, the service routine associated with that interrupt is not called.

6.3.2 Prototype

```
byte osal_int_disable( byte interrupt_id )
```

6.3.3 Parameter Details

`interrupt_id` identifies the interrupt to be disabled.

6.3.4 Return

Return value indicates the result of the operation.

RETURN VALUE	DESCRIPTION
ZSUCCESS	Interrupt Disabled Successfully
INVALID_INTERRUPT_ID	Invalid Interrupt

7. Task Management API

7.1 Introduction

This API is used to add and manage tasks in the OSAL system.

7.2 osal_init_system()

7.2.1 Description

This function initializes the OSAL system. The function must be called at startup prior to using any other OSAL function.

7.2.2 Prototype

```
byte osal_init_system( void )
```

7.2.3 Parameter Details

None.

7.2.4 Return

Return value indicates the result of the operation.

RETURN VALUE	DESCRIPTION
ZSUCCESS	Success

7.3 osal_start_system()

7.3.1 Description

This function is the main loop function of the task system. It will look through all task events and call the task event processor function for the task with the event. If there are events for particular task, the function will call the event process routine for that task to handle the events. Events are handled one at a time at the event process routine of the corresponding task. After an event is serviced, the remaining events will be returned back to the main loop for next time around. If there are no events (for all tasks), this function puts the processor into a Sleep mode.

7.3.2 Prototype

```
void osal_start_system( void )
```

7.3.3 Parameter Details

None

7.3.4 Return

None

7.4 osal_self()

7.4.1 Description

This function returns the task ID of the calling (current) task which happens to be the calling task. This function Returns the wrong results if called from within an interrupt service routine.

7.4.2 Prototype

```
byte osal_self( void )
```

7.4.3 Parameter Details

None

7.4.4 Return

Return value is the Task ID of the currently active task.

7.5 osalTaskAdd ()

7.5.1 Description

This function adds a task to the OSAL system. A task consists of 2 functions – init and messages processing. Messages processing function takes in events, then processes one of them and returns the rest back to the main loop.

7.5.2 Prototype

```
/*
 * Task Initialization function prototype
 */
typedef void (*pTaskInitFn)( unsigned char task_id );

/*
 * Event handler function prototype
 */
typedef unsigned short (*pTaskEventHandlerFn)( unsigned char task_id,
                                               unsigned short event );

void osalTaskAdd( const pTaskInitFn pfnInit,
                  const pTaskEventHandlerFn pfnEventProcessor,
                  const byte taskPriority);
```


7.5.3 Parameter Details

`pfnInit` pointer to the task's initialization function.

`pfnEventProcessor` pointer to the task's event processor function.

`taskPriority` priority level of the task. Value can be one of the following:

Priority	Value
OSAL_TASK_PRIORITY_LOW	50
OSAL_TASK_PRIORITY_MED	130
OSAL_TASK_PRIORITY_HIGH	230

7.5.4 Return

None.

8. Memory Management API

8.1 Introduction

This API represents a simple memory allocation system. These functions allow dynamic memory allocation.

8.2 osal_mem_alloc()

8.2.1 Description

This function is a simple memory allocation function that returns a pointer to a buffer (if successful).

8.2.2 Prototype

```
void *osal_mem_alloc( uint16 size );
```

8.2.3 Parameter Details

`size` – the number of bytes wanted in the buffer.

8.2.4 Return

A void pointer (which should be cast to the intended buffer type) to the newly allocated buffer. A NULL pointer is returned if there isn't enough memory to allocate.

8.3 osal_mem_free()

8.3.1 Description

This function frees the allocated memory to be used again. This only works if the memory had already been allocated with `osal_mem_alloc()`.

8.3.2 Prototype

```
void osal_mem_free( void *ptr );
```

8.3.3 Parameter Details

`ptr` – pointer to the buffer to be "freed". This buffer must have been previously allocated with `osal_mem_alloc()`.

8.3.4 Return

None.

9. Power Management API

9.1 Introduction

This section describes the OSAL's power management system. The system provides a way for the applications/tasks to notify OSAL when it's safe to turn off the receiver and external hardware, and put the processor in to sleep.

There are 2 functions to control the power management. The first, `osal_pwrmgr_device()`, is called to set the device level mode (power save or no power savings). Then, there is the task power state, each task can hold off the power manager from conserving power by calling `osal_pwrmgr_task_state( PWRMGR_HOLD )`. If a task "Holds" the power manager, it will need to call `osal_pwrmgr_task_state( PWRMGR_CONSERVE )` to allow the power manager to continue in power conserve mode.

By default, when the task is initialized, each task's power state is set to `PWRMGR_CONSERVE`, so if a task doesn't want to hold off power conservation (no change) it doesn't need to call `osal_pwrmgr_task_state()`.

The power manager will look at the device mode and the collective power state of all the tasks before going in to power conserve state.

9.2 osal_pwrmgr_device()

9.2.1 Description

This function is called on power- up or whenever the power requirements change (ex. Battery backed coordinator). This function sets the overall ON/OFF State of the device's power manager. This function should be called from a central controlling entity (like ZDO).

9.2.2 Prototype

```
void osal_pwrmgr_state( byte pwrmgr_device );
```

9.2.3 Parameter Details

`Pwrmgr_device` – changes or sets the power savings mode.

Type	Description
PWRMGR_ALWAYS_ON	With this selection there is no power savings and the device is most likely on mains power.
PWRMGR_BATTERY	Turns power savings on.

9.2.4 Return

None.

9.3 osal_pwrmgr_task_state()

9.3.1 Description

This function is called by each task to state whether or not this task wants to conserve power. The task will call this function to vote whether it wants the OSAL to conserve power or it wants to hold off on the power savings. By default, when a task is created, its own power state is set to conserve. If the task always wants to converse power, it doesn't need to call this function at all.

9.3.2 Prototype

```
byte osal_pwrmgr_task_state( byte task_id, byte state );
```

9.3.3 Parameter Details

`state` – changes a task's power state.

Type	Description
PWRMGR_CONSERVE	Turns power savings on, all tasks have to agree. This is the default state when a task is initialized.
PWRMGR_HOLD	Turns power savings off.

9.3.4 Return

Return value indicates the result of the operation.

RETURN VALUE	DESCRIPTION
ZSUCCESS	Success
INVALID_TASK	Invalid Task

10. Non-Volatile Memory API

10.1 Introduction

This section describes the OSAL Non-Volatile (NV) memory system. The system provides a way for applications to store information into the device's memory persistently. It is also used by the stack for persistent storage of certain items required by the ZigBee specification. The NV functions are designed to read and write user-defined items consisting of arbitrary data types such as structures or arrays. The user can read or write an entire item or an element of the item by setting the appropriate offset and length. The API is independent of the NV storage medium and can be implemented for flash or EEPROM.

Each NV item has a unique ID. There is a specific ID value range for applications while some ID values are reserved or used by the stack or platform. If your application creates its own NV items it must select an ID from the Application value range. See the table below.

VALUE	USER
0x0000	Reserved
0x0001 – 0x0020	OSAL
0x0021 – 0x0040	NWK
0x0041 – 0x0060	APS
0x0061 – 0x0080	Security
0x0081 – 0x00A0	ZDO
0x00A1 – 0x0200	Reserved
0x0201 – 0x0FFF	Application
0x1000 -0xFFFF	Reserved

There are some important considerations when using this API:

1. These are blocking function calls and an operation may take several milliseconds to complete. This is especially true for NV write operations. In addition, interrupts may be disabled for several milliseconds. It is best to execute these functions at times when they do not conflict with other timing-critical operations. For example, a good time to write NV items would be when the receiver is turned off.
2. Try to perform NV writes infrequently. It takes time and power; also most flash devices have a limited number of erase cycles.
3. If the structure of one or more NV items changes, especially when upgrading from one version of Z-Stack to another, it is necessary to erase and re-initialize the NV memory. Otherwise, read and write operations on NV items that changed will fail or produce erroneous results.

10.2 osal_nv_item_init()

10.2.1 Description

Initialize an item in NV. This function checks for the presence of an item in NV. If it does not exist, it is created and initialized with the data passed to the function, if any.

This function must be called for each item before calling `osal_nv_read()` or `osal_nv_write()`.

10.2.2 Prototype

```
byte osal_nv_item_init( uint16 id, uint16 len, void *buf );
```

10.2.3 Parameter Details

`id` – User-defined item ID.

`len` – Item length in bytes.

`*buf` – Pointer to item initialization data. If no initialization data, set to NULL.

10.2.4 Return

Return value indicates the result of the operation.

RETURN VALUE	DESCRIPTION
ZSUCCESS	Success
NV_ITEM_UNINIT	Success but item did not exist
NV_OPER_FAILED	Operation failed

10.3 osal_nv_read()

10.3.1 Description

Read data from NV. This function can be used to read an entire item from NV or an element of an item by indexing into the item with an offset. Read data is copied into *buf.

10.3.2 Prototype

```
byte osal_nv_read( uint16 id, uint16 offset, uint16 len, void *buf );
```

10.3.3 Parameter Details

`id` – User-defined item ID.

`offset` – Memory offset into item in bytes.

`len` – Item length in bytes.

`*buf` – Data is read into this buffer.

10.3.4 Return

Return value indicates the result of the operation.

RETURN VALUE	DESCRIPTION
ZSUCCESS	Success
NV_ITEM_UNINIT	Item is not initialized
NV_OPER_FAILED	Operation failed

10.4 osal_nv_write()

10.4.1 Description

Write data to NV. This function can be used to write an entire item to NV or an element of an item by indexing into the item with an offset.

10.4.2 Prototype

```
byte osal_nv_write( uint16 id, uint16 offset, uint16 len, void *buf );
```

10.4.3 Parameter Details

`id` – User-defined item ID.

`offset` – Memory offset into item in bytes.

`len` – Item length in bytes.

`*buf` – Data to write.

10.4.4 Return

Return value indicates the result of the operation.

RETURN VALUE	DESCRIPTION
ZSUCCESS	Success
NV_ITEM_UNINIT	Item is not initialized
NV_OPER_FAILED	Operatoin failed

10.5 osal_offsetof()

10.5.1 Description

This macro calculates the memory offset in bytes of an element within a structure. It is useful for calculating the offset parameter used by NV API functions.

10.5.2 Prototype

```
osal_offsetof(type, member)
```

10.5.3 Parameter Details

`type` – Structure type.

`member` – Structure member.

