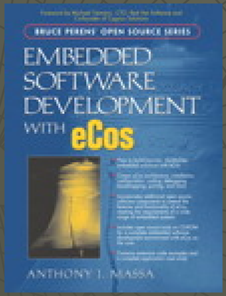
Embedded Software Development with eCos – Chapter3



中興大學資訊科學系

指導教授：張軒彬

學生：李佳翰

OUTLINE

- Exceptions
  - HAL and Kernel Exception Handling
  - Application Exception Handling
- Interrupts
  - eCos Interrupt
    - Interrupt and Scheduler Synchronization
  - Interrupt Configuration
  - Interrup Handling
  - Interrupt Control

OUTLINE (cont.)

- Interrupt Service Routine Management
- Interrupt State Management
- Interrupt Controller Mangement

Exceptions

- Exceptions can occurs in a system include those raised by HardWare those raised by software. (Reset,IRQ,資料錯誤,無定義指令,SWI….)
- After exception causes an interruption,the processor will jump to defined address and begin to run the instructions at that locations.(Address contains the excepton handling code to process the error)

Exceptions

- Two main method for exception handling in eCos
  - 1st：Hal and Kernal exception Handling。
  - 2nd：Application Exception Handling。
- All exception can be disabled in the system。

HAL and Kernel Exception Handling

- VSR (Vector Service Routine)table
  - Defined in each HAL package as hal_vsr_table。
  - An array of pointer to the exception handler。
  - VSR table is the first place the processor looks to determine where to jump to execute the exception handler。
  - The size and base address of the hal_vsr_table is architecture specific。
  - Link script files are used to define the location of the exception table，Located at a fixed memory location。
  - **The ARM architecture does not use hal_vsr_table，ARM architecture defines separate handler routines for each exception it supports。**

HAL and Kernel Exception Handling

- The job of the **default exception VSR** is to perform the common processing of all exceptions ,which includes：
  - saving the processor’s state
  - calling any kernel-level handler routine to perform additional processing
  - restoring the state of the processor prior to returning to normal program execution.

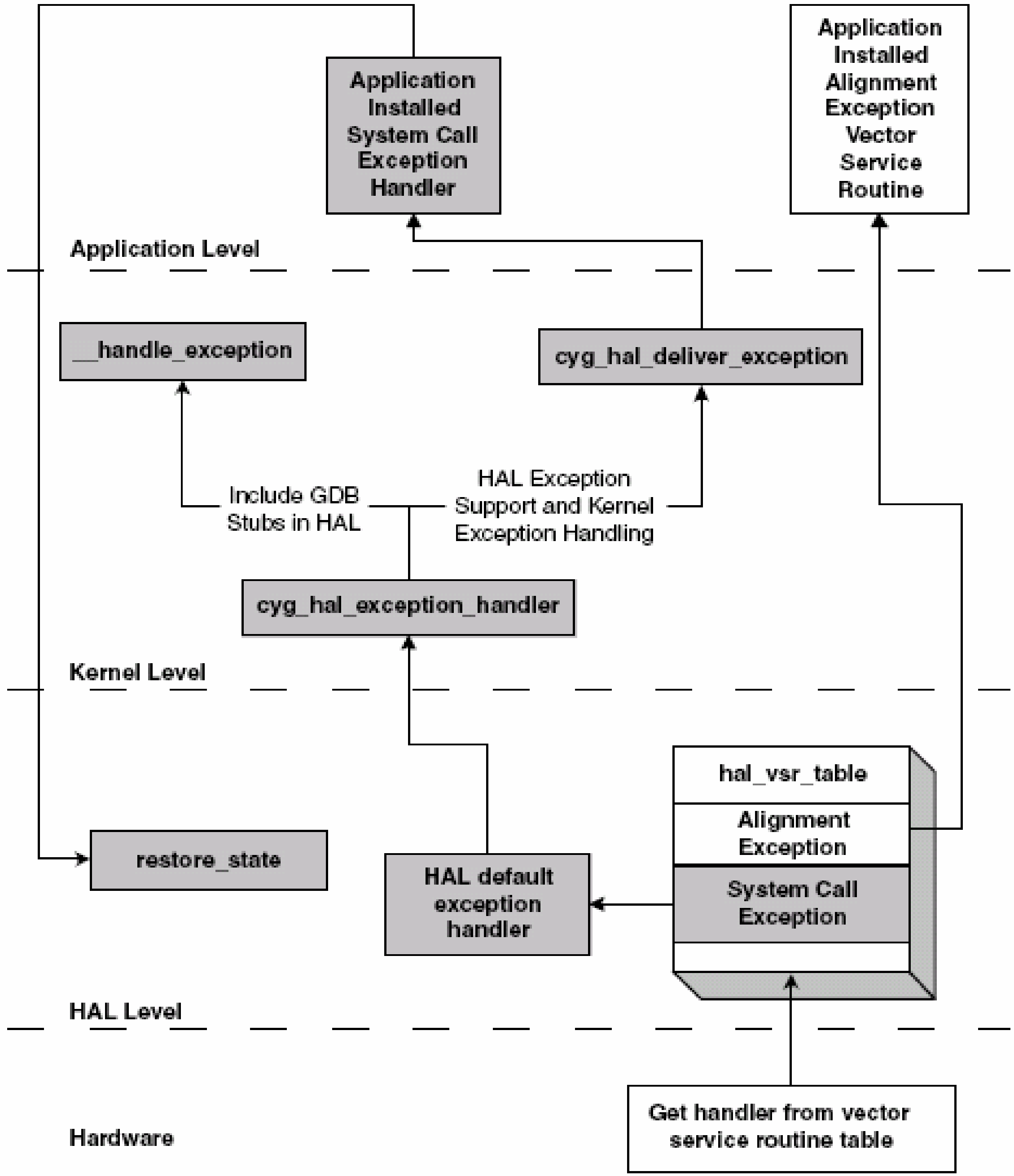


Figure 3.1 eCos exception handling execution flow.

HAL and Kernel Exception Handling

Kernel Exception Handler API Functions :

```
externC void cyg_exception_set_handler(
    cyg_code_t                 exception_number, // exception number
    cyg_exception_handler_t    *new_handler,     // handler function
    cyg_addrword_t             new_data,         // data argument
    cyg_exception_handler_t    **old_handler,    // handler function
    cyg_addrword_t             *old_data
)
{
    Cyg_Thread::register_exception(
        exception_number,
        (cyg_exception_handler *)new_handler,
        (CYG_ADDRWORD)new_data,
        (cyg_exception_handler **)old_handler,
        (CYG_ADDRWORD *)old_data
        );
}
```

HAL and Kernel Exception Handling

```
//Clear exception handler to default
externC void cyg_exception_clear_handler(
    cyg_code_t  exception_number)
{
    Cyg_Thread::deregister_exception( exception_number );
// Exception deregistation. Revert the handler for the exception number to the default.

}


/* Invoke exception handler */
externC void cyg_exception_call_handler(
    cyg_handle_t  thread,
    cyg_code_t    exception_number,
    cyg_addrword_t  error_code
)
{
    Cyg_Thread *t = (Cyg_Thread *)thread;

    t->deliver_exception( exception_number, error_code );
// Exception delivery. Call the appropriate exception handler.
}
```

Application Exception Handling

- The Hal defines macro to give the application access to the VSR table directly.
- HAL Exception Vector Service Routine Macros：

Syntax: **HAL_VSR_GET**(vector_,pvsr_)

Description: Get the current VSR set in the hal_vsr_table and return it in the location pointed to by pvsr_.

Syntax: **HAL_VSR_SET**(vector_,vsr_,poldvsr_)

Description: Replace the routine in the hal_vsr_table with the new vector service routine.

Application Exception Handling

- When installing a vector service routine, it is not necessary to call the HAL set and get macros directly. The eCos kernel API defines functions for the application to use instead
- Kernel Exception Vector Service Routine API Functions, cyg_interrupt_get_vsr calls the HAL macro HAL_VSR_GET：

Application Exception Handling

```
externC void cyg_interrupt_set_vsr(
    cyg_vector_t vector,         /* vector to set*/
    cyg_VSR_t    *vsr            /* vsr to set */
)
{
    Cyg_Interrupt::set_vsr( (cyg_vector)vector, (cyg_VSR *)vsr);
   // Install a vector service routine
}

void
Cyg_Interrupt::set_vsr (cyg_vector vector, cyg_VSR *vsr, cyg_VSR **old)
{
    CYG_REPORT_FUNCTION();

    CYG_REPORT_FUNCARG3( "vector = %d, new vsr is at %08x, mem to put "
                         "old VSR in is at %08x", vector, vsr, old);

    CYG_INSTRUMENT_INTR(SET_VSR, vector, vsr);

    CYG_ASSERT( vector >= CYGNUM_HAL_VSR_MIN, "Invalid vector");
    CYG_ASSERT( vector <= CYGNUM_HAL_VSR_MAX, "Invalid vector");
```

Application Exception Handling

```
    CYG_INTERRUPT_STATE old_ints;

    HAL_DISABLE_INTERRUPTS(old_ints);

    HAL_VSR_SET( vector, vsr, old );

    HAL_RESTORE_INTERRUPTS(old_ints);

    CYG_REPORT_RETURN();
}

externC void cyg_interrupt_get_vsr(      // Get the current service routine
    cyg_vector_t  vector,                /* vector to get */
    cyg_VSR_t    **vsr                   /* vsr got */
)
{
    Cyg_Interrupt::get_vsr( (cyg_vector)vector, (cyg_VSR **)vsr);
}
```

Interrupts

- eCos interrupt processing is divided into two parts：
  - ISR(interrupt Service Routine)
  - DSR(Defered Service Routine)
- In some cases,very little or short interrupt processing needs to be done, the interrupt can be handled in the ISR completely.If more complex servicing is required,DSR should be used.
- A thread has locked the scheduler,the DSR is delayed until the thread unlocks it.
- The priority scheme is that ISRs have absolute priority over

  DSRs, and DSRs have absolute priority over threads.
- Some HAL implementations offer an interrupt nesting scheme.

Interrupt and Scheduler Synchronization

- First, **ISRs** cannot make any scheduler-related synchronization function calls (semaphores , mutexes, condition variables)
- The eCos interrupt scheme allows these synchronization calls to be made from the **DSR.**
- Typically, the DSR will execute immediately after the ISR finishes.
- DSRs must not make a synchronization call that blocks , Blocking code execution must wait for a resource to be released.

Interrupt Configuration

- Configuration options for interrupts are available at the HAL and kernel level.
- HAL Interrupt Configuration Options：
  - Option Name Use Separate Stack For Interrupts
    - CDL Name ： CYGIMP_HAL_COMMON_INTERRUPTS_USE_INTERRUPT_STACK
    - Description：

      Allows a separate stack, maintained by the HAL, during interrupt processing，This option is enabled by default.

  - Option Name Interrupt Stack Size
    - CDL Name：

      CYGNUM_HAL_COMMON_INTERRUPTS_STACK_SIZE
    - Description：Specifies, in bytes, the stack size for the interrupt stack. This is the stack that will be used for all interrupts when Use Separate Stack For Interrupts is enabled，The default value for this option is 4096.

Interrupt Configuration

- Option Name **Allow Nested Interrupts**
  - CDL Name CYGSEM_HAL_COMMON_INTERRUPTS_ALLOW_NESTING
  - Description :

    This allows other interrupts, typically of higher priority, to occur and be processed.This option is disabled by default.

- Option Name **Save Minimum Context On Interrupt**
  - CDL Name CYGDBG_HAL_COMMON_INTERRUPTS_SAVE_MINIMUM_CONTEXT
  - Description :

    Permits the HAL interrupt handling code to use architecture-specific calling conventions, reducing the amount of state information saved. This improves performance and reduces codesize. The drawback is that debugging is often more difficult. This option is enabled by default.

Interrupt Configuration

- Option Name **Chain All Interrupts Together**
  - CDL Name： CYGIMP_HAL_COMMON_INTERRUPTS_CHAIN
  - Description：

    Allows all interrupt vectors to be chained, requiring each handler to check if it needs to process the interrupt. The default for this option is disabled, allowing interrupts to be attached to individual vectors.

- Option Name **Ignore Spurious Interrupts**
  - CDL Name

    CYGIMP_HAL_COMMON_INTERRUPTS_IGNORE_SPURIOUS
  - Description：

    Specifies whether to ignore an interrupt that might occur from the hardware source not being properly de-bounced or interrupts coming from glitches. This option is disabled by default.

Interrupt Configuration

- The kernel-level interrupt suboptions available when using DSRs：
  - Option Name **Use Linked Lists For DSRs**
    - CDL Name CYGIMP_KERNEL_INTERRUPTS_DSRS_LIST
  - Option Name **Use Fixed-Size Table For DSRs**
    - CDL Name CYGIMP_KERNEL_INTERRUPTS_DSRS_TABLE
  - Option Name **Chain All Interrupts Together**
    - CDL Name CYGIMP_KERNEL_INTERRUPTS_CHAIN

Interrupt Handling

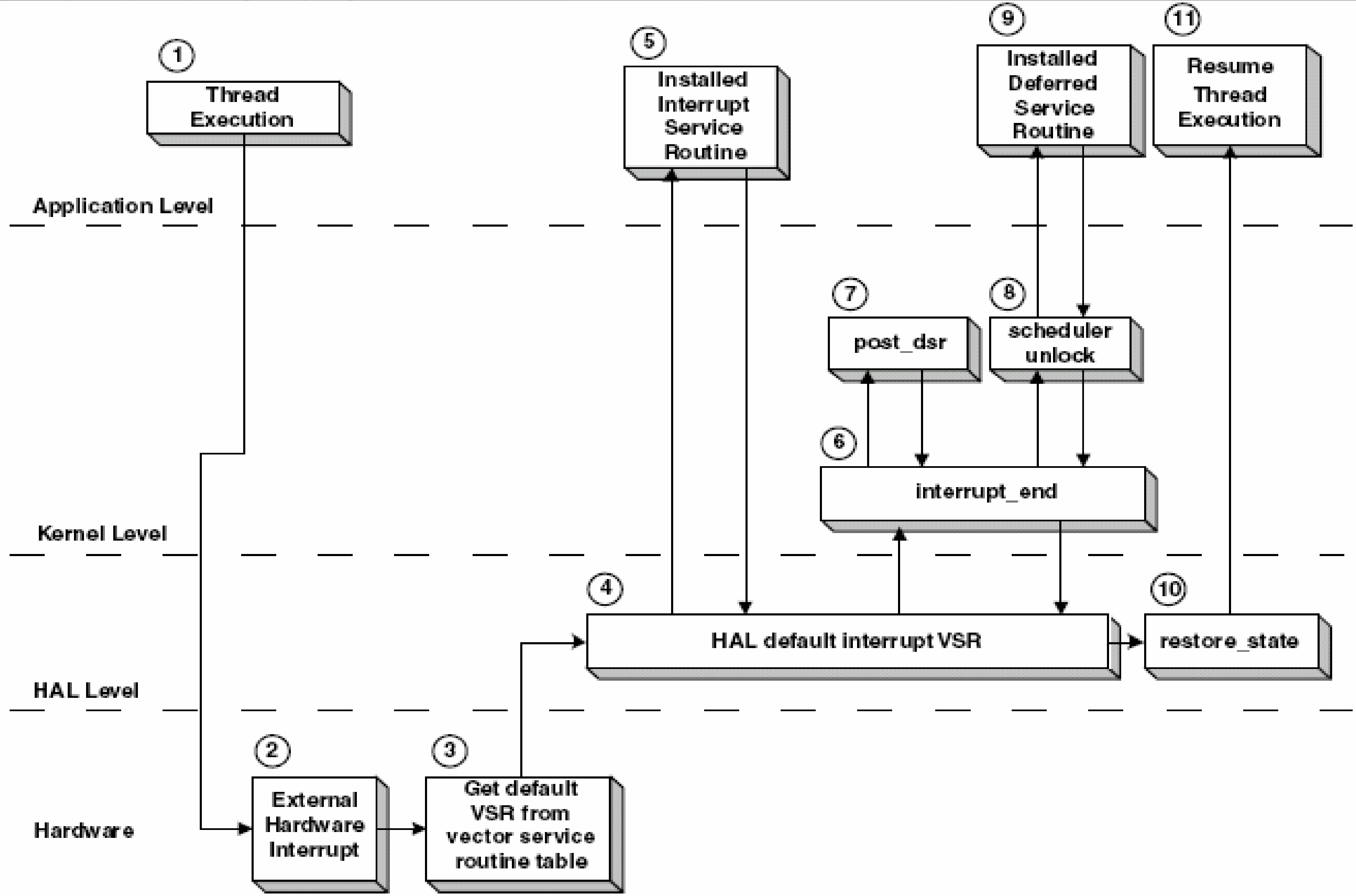


Figure 3.2 eCos interrupt handling execution flow.

Interrupt Control

- The HAL and kernel interrupt control functionality is broken down into three groups:
  - Interrupt Service Routine Management
  - Interrupt State Management
  - Interrupt Controller Management
- It is important to use the kernel API functions and avoid using the HAL macros directly.

Interrupt Service Routine Management

- Interrupt Service Routine Management, controls the attachment and detachment of interrupt service routines within the three HAL ISR tables (handlers, data, and objects).

Interrupt Service Routine Management

- HAL Interrupt Service Routine Management Macros :

```
Syntax: HAL_INTERRUPT_ATTACH(_vector_,_isr_,_data_,_object_)
#define HAL_INTERRUPT_ATTACH( _vector_, _isr_, _data_, _object_ )
    CYG_MACRO_START
    if( hal_interrupt_handlers[_vector_] == (CYG_ADDRESS)hal_default_isr )
    {
        hal_interrupt_handlers[_vector_] = (CYG_ADDRESS)_isr_;
        hal_interrupt_data[_vector_] = (CYG_ADDRWORD) _data_;
        hal_interrupt_objects[_vector_] = (CYG_ADDRESS)_object_;
    }
    CYG_MACRO_END

Syntax: HAL_INTERRUPT_DETACH(
_vector_,
_isr_
);
```

Interrupt Service Routine Management

- Kernel Interrupt Service Routine Management API Functions：

```
externC void cyg_interrupt_create(
    cyg_vector_t        vector,         /* Vector to attach to              */
    cyg_priority_t      priority,       /* Queue priority                   */
    cyg_addrword_t      data,           /* Data pointer                     */
    cyg_ISR_t           *isr,           /* Interrupt Service Routine        */
    cyg_DSR_t           *dsr,           /* Deferred Service Routine         */
    cyg_handle_t        *handle,        /* returned handle                  */
    cyg_interrupt       *intr           /* put interrupt here               */
){
    CYG_ASSERT_SIZES( cyg_interrupt, Cyg_Interrupt );
    Cyg_Interrupt *t = new((void *)intr) Cyg_Interrupt (
        (cyg_vector)vector,(cyg_priority)priority,
        (CYG_ADDRWORD)data,
        (cyg_ISR *)isr,(cyg_DSR *)dsr );
    t=t;
    CYG_CHECK_DATA_PTR( handle, "Bad handle pointer" );
    *handle = (cyg_handle_t)intr;
}
//Construct an interrupt object in memory .The interrupt is not attached to the vector,
   however until cyg_interrupt_attached is called.
```

Interrupt Service Routine Management

```
externC void cyg_interrupt_delete( cyg_handle_t interrupt)
{
    ((Cyg_Interrupt *)interrupt)->~Cyg_Interrupt();
}//Remove the interrupt object from memory,this code also detaches.
void cyg_interrupt_attach( cyg_handle_t interrupt )
{
    ((Cyg_Interrupt *)interrupt)->attach();
}//Attach the interrupt to the vector allowing interrupt to be delivered to
   the ISR，The interrupt is also set up in the chain list if the
   configuration option is enabled.
void cyg_interrupt_detach( cyg_handle_t interrupt )
{
    ((Cyg_Interrupt *)interrupt)->detach();
} //Detach the interrupt from the vector preventing interrupts from being
   delivered to the ISR，emoves the interrupt from the chain list if the
   configuration option is enabled.
```

Interrupt State Management

- Allows control over the state of the processor's interrupt mask mechanism by accessing the global interrupt enable found in the processor's register.
- HAL Interrupt State Management Macros :

```
#define HAL_DISABLE_INTERRUPTS(_old_)
    _old_ = hal_disable_interrupts(); //_old_– returned state of the
   interrupt mask

#define HAL_ENABLE_INTERRUPTS()
    hal_enable_interrupts();
```

Interrupt State Management

```
#define HAL_RESTORE_INTERRUPTS(_old_)
    hal_restore_interrupts(_old_);
/*Determine state of interrupt mask and return value */
#define HAL_QUERY_INTERRUPTS(_old_)
    _old_ = hal_query_interrupts();
```

- kernel API functions for Interrupt State Management：

```
/* CPU level interrupt mask  */
externC void cyg_interrupt_enable()
{
    Cyg_Interrupt::enable_interrupts();
}
externC void cyg_interrupt_disable()
{
    Cyg_Interrupt::disable_interrupts();
}
```

Interrupt Controller Management

- Interrupt Controller Management, provides control over any interrupt controller that might be present for a specific variant. Not all HAL architectures have an interrupt controller.
- HAL Interrupt Controller Management Macros：

```
#define HAL_INTERRUPT_MASK( _vector_ )
    hal_interrupt_mask( _vector_ )
  //Block the designated interrupt from occurring. This is typically a
   platform or variant specific implementation, which requires masking
   interrupts in the processor’s registers.

#define HAL_INTERRUPT_UNMASK( _vector_ )
    hal_interrupt_unmask( _vector_ )
```

Interrupt Controller Management

```
#define HAL_INTERRUPT_ACKNOWLEDGE( _vector_ )
    hal_interrupt_acknowledge( _vector_ )
//Acknowledges the current interrupt from the specified vector. This
   informs the processor that the interrupt was received and resets the
   interrupt to an inactive state. the HAL modifies the processor’s
   interrupt acknowledge bit in a register.
#define HAL_INTERRUPT_SET_LEVEL( _vector_, _level_ )
    hal_interrupt_set_level( _vector_, _level_ )
```

- The kernel API functions for the Interrupt Controller Management group：

```
externC void cyg_interrupt_mask(cyg_vector_t vector)
{
    Cyg_Interrupt::mask_interrupt( (cyg_vector)vector);
}
//  Program the interrupt controller to block delivery of
   interrupts for the vector specified.This function calls the
   HAL_INTERRUPT_MASK macro. All interrupts are disabled during
   this function call.
```

Interrupt Controller Management

```
void
Cyg_Interrupt::mask_interrupt(cyg_vector vector)
{
    CYG_REPORT_FUNCTION();
    CYG_REPORT_FUNCARG1("vector=%d", vector);

    CYG_ASSERT( vector >= CYGNUM_HAL_ISR_MIN, "Invalid vector");
    CYG_ASSERT( vector <= CYGNUM_HAL_ISR_MAX, "Invalid vector");

    CYG_INSTRUMENT_INTR(MASK, vector, 0);

    CYG_INTERRUPT_STATE old_ints;

    HAL_DISABLE_INTERRUPTS(old_ints);
    HAL_INTERRUPT_MASK( vector );
    HAL_RESTORE_INTERRUPTS(old_ints);

    CYG_REPORT_RETURN();
}
```

Interrupt Controller Management

```
externC void cyg_interrupt_mask_intunsafe(cyg_vector_t vector)
{
    Cyg_Interrupt::mask_interrupt_intunsafe( (cyg_vector)vector);
}
externC void cyg_interrupt_unmask(cyg_vector_t vector)
{
    Cyg_Interrupt::unmask_interrupt( (cyg_vector)vector);
}
externC void cyg_interrupt_unmask_intunsafe(cyg_vector_t vector)
{
    Cyg_Interrupt::unmask_interrupt_intunsafe( (cyg_vector)vector);
}
```

Interrupt Controller Management

```
externC void cyg_interrupt_acknowledge(cyg_vector_t vector)
{
    Cyg_Interrupt::acknowledge_interrupt( (cyg_vector)vector);
}

void
Cyg_Interrupt::acknowledge_interrupt(cyg_vector vector)
{
CYG_ASSERT( vector >= CYGNUM_HAL_ISR_MIN, "Invalid vector");
    CYG_ASSERT( vector <= CYGNUM_HAL_ISR_MAX, "Invalid vector");

    CYG_INSTRUMENT_INTR(ACK, vector, 0);

    HAL_INTERRUPT_ACKNOWLEDGE( vector );
}
```

```
//Acknowledge the current interrupt from the specified vector. This
function is called from within the ISR to cancel the interrupt request
from the processor, preventing a re-trigger of the same interrupt. This
function calls the HAL_INTERRUPT_ACKNOWLEDGE macro.
```