# LabWindows®

## Guidelines for Interrupt and DMA Programming in Loadable Object Modules

**December 1992 Edition**

**Part Number 320412-01**

**National Instruments Corporate Headquarters**
6504 Bridge Point Parkway
Austin, TX 78730-5039
(512) 794-0100
(800) 433-3488 (toll-free U.S. and Canada)
Technical support fax:  (512) 794-5678

**Branch Offices:**
Australia 03 879 9422, Belgium 02 757 00 20, Canada (Ontario) 519 622 9310, Canada (Québec) 514 694 8521
Denmark 45 76 73 22, Finland 90 5272321, France 1 48 65 33 70, Germany 089 714 50 93, Italy 02 48301892,
Japan 03 3788 1921, Netherlands 01720 45761, Norway 03 846866, Spain 91 896 0675, Sweden 08 984970,
Switzerland 056 27 00 20, U.K. 0635 523545

# Contents

# Chapter 1
# Programming Guidelines for Installing Interrupts and Performing DMA

This chapter contains general guidelines for coding and installing interrupt handlers and performing DMA transfers from within LabWindows loadable object module code. The guidelines apply to object module code for both instrument drivers and external modules.

All of the functions discussed in this document are available both within the LabWindows interactive program and in the LabWindows standalone system libraries. Modules that follow the guidelines in this document can execute correctly both in the LabWindows interactive program and in standalone programs linked with the LabWindows libraries.

## Writing Interrupt Routines

**Warning**: Due to expected revisions in the DPMI (DOS Protected Mode Interface) standard, the methods outlined in this document for installing interrupts may change in the future.

Functions called by an interrupt routine must be in the same code segment as the interrupt routine.

There are two types of interrupt routines: passdown or passup. With *passdown interrupts*, the interrupt handler is always executed in real mode. If the selected interrupt occurs in protected mode, a mode switch occurs and the interrupt is processed in real mode. With *passup interrupts*, the interrupt handler is always executed in protected mode. If the selected interrupt occurs in real mode, a mode switch occurs and the interrupt is processed in protected mode.

### Accessing Memory from Interrupt Handlers

Extended memory is not accessible in real mode. Because passdown interrupts are handled in real mode, passdown handlers can only access data that is stored in conventional memory.

Furthermore, any data accessed by an interrupt handler must be locked in memory. The default data segment is locked in conventional memory by LabWindows. You can explicitly lock other segments in extended memory by using the `_lock_segment` function.

The rules for accessing memory in interrupt handlers are listed as follows:

*Passdown Handlers:* Any data accessed must be in the default data segment.

*Passup Handlers:* Any data accessed must either be located in the default data segment or locked in extended memory using `_lock_segment`.

**Note:** The *near* keyword in Microsoft C and Borland C is used to place data in the default data segment. The *far* keyword is used to place data in a far data segment. In addition, the Microsoft C `Gt` compiler flag and the Borland C `Ff` compiler flag can be used to force data items larger than a specified size into a far data segment. The near and far keywords override the `Gt` and Ff flags.

## Restrictions on Interrupt Handlers

Passup interrupt handlers are not allowed to chain to the previously installed interrupt handler. In general, the sharing of hardware interrupt lines is *not* supported.

A passdown interrupt handler should not be in the same code segment as a passup interrupt handler.

## Determining the Type of Interrupt to Use

With the exception of timer interrupt handlers, National Instruments recommends installing all interrupt handlers as passup interrupts. Timer tick interrupt handlers must be installed under interrupt number 08h as a passdown interrupt. Interrupt number 1Ch cannot be used.

### Timer Tick Interrupt

This section provides guidelines for installing and removing a passdown timer tick interrupt handler. Example source code showing the installation, coding, and removal of a timer tick interrupt handler is included at the end of this document.

#### Installing the Handler

Install the passdown interrupt routine using the following LabWindows function call:

```
int _install_interrupt (int vec_num,
                        void *rm_addr,
                        void *pm_addr,
                        void ***rm_chain_ptr,
                        void ***pm_chain_ptr);
```

The address of the timer tick interrupt handler must be passed in `rm_addr`. The parameters `pm_addr` and `pm_chain_ptr` must be NULL. `rm_chain_ptr` is the address of a pointer to a function pointer. The value returned in `rm_chain_ptr` must be stored in the default data segment and is used to chain to the previously installed timer interrupt handler. All timer tick handlers must chain to the previously installed interrupt handler.

#### Coding the Interrupt Routine

The first operation a timer interrupt handler should perform is chaining to the previously installed interrupt handler. Chaining ensures that the BIOS timer interrupt handler code is executed first.

A timer tick interrupt handler should not be in the same code segment as a passup interrupt handler.

All data accessed by a timer interrupt handler *must* be in the default data segment.

Never use the processor instruction `STI` to enable interrupt requests in a timer tick interrupt handler.

Never use the `SEG` operator in a passdown interrupt routine.

Removing the Handler

Use the LabWindows function call `_uninstall_interrupt` to restore the original real mode interrupt handler.

```
int _uninstall_interrupt  (int vec_num,
                           void *rm_addr,
                           void *pm_addr);
```

`vec_num`, `rm_addr`, and `pm_addr` must be the same values passed to `_install_interrupt`.

**Passup Interrupts**

This section provides guidelines for installing and removing a passup interrupt handler. Example source code showing the installation, coding, and removal of a passup interrupt handler is included at end of this document.

Installing the Passup Interrupt Handler

1. Lock into memory any variables accessed by the interrupt routine. These variables include data items explicitly declared or dynamically allocated within the object module as well as user variables that are passed to object module functions. Lock variables into memory using the following LabWindows function:

   ```
   long _lock_segment (void *pm_ptr)
   ```

   `pm_ptr` is a pointer to the variable being locked. If the function fails, a NULL is returned. Otherwise, ignore the return value.

   **Note:** Variables in the default data segment do not need to be locked.

2. Install the passup interrupt routine using the following LabWindows function call:

   ```
   int _install_interrupt (int vec_num,
                           void *rm_addr,
                           void *pm_addr,
                           void ***rm_chain_ptr,
                           void ***pm_chain_ptr);
   ```

   The address of the passup interrupt handler must be passed in `pm_addr`. The parameters `rm_addr`, `rm_chain_ptr` must be NULL. Passup interrupt handlers are not allowed to chain to the previously installed interrupt handler. Therefore, the parameter `pm_chain_ptr` must be NULL.

Removing the Passup Interrupt Handler

1. Restore the original protected mode interrupt handler by using the following LabWindows function call:

   ```
   int _uninstall_interrupt  (int vec_num,
                              void *rm_addr,
                              void *pm_addr);
   ```

   `vec_num`, `rm_addr`, and `pm_addr` must be the same values passed to `_install_interrupt`.

2. Unlock any previously locked variables by using the following LabWindows function:

```
void _unlock_segment (void * pm_ptr);
```

   The same pointer must be used to both lock and unlock the variable. If you called `_lock_segment` on the same pointer more than once, you must call `_unlock_segment` on the pointer an equal number of times. If a locked variable is dynamically allocated, you must unlock the variable before freeing it.

### Loss of Interrupts During Mode Switches

Interrupt servicing can be delayed if the interrupt occurs during a mode switch. Mode switches are caused by clock interrupts, mouse movement, DOS service calls (such as file I/O), and GPIB handler calls. In addition, a passup interrupt can cause a mode switch if the interrupt occurs when the processor is in real mode. Depending on the computer, the maximum potential delay for a passup interrupt can be as much as 4 milliseconds or as little as 200 microseconds. If more than one interrupt is received during a mode switch, all interrupts except the last one are lost.

## Accessing Absolute Addresses

LabWindows provides *transparent access* for a number of important real mode segments. Transparent access means that a segment value such as B000 or F000 addresses the same physical memory in either protected or real mode. There are 48 transparent segments:

A000, A200, A400, ..., FE00

An absolute address is accessed through a pointer constructed from one of the transparent segments. Before accessing an absolute address, its segment must be registered as either a code or data segment using the function:

```
int _set_absolute_access (unsigned int seg, int access);
```

The parameter `seg` must be one of the segment values listed above (A000...FE00). The parameter `access` indicates whether the segment is data or code. `0` indicates data, and `1` indicates code. `set_absolute_access` returns a zero if successful or a nonzero if failed.

### Example

```
#include <dos.h>

struct IO_REGS
{
    int register1;
    int register2;
    int register3;
};

setup()
{
    struct IO_REGS *p;

/* construct pointer to segment C000, offset A00 */
    FP_SEG(p) = 0xC000;
    FP_OFF(p) = 0x0A00;
```

```
/* set segment C000 to access DATA */
    _set_absolute_access(0xC000, 0);
    .
    .
    .
/* move data into memory mapped I/O */
    p->register1 = 0xFFFF;
    .
    .
    .
}
```

## Performing DMA Operations

A DMA transfer buffer must be locked into memory while DMA transfers are in progress. Use the following LabWindows function to lock the buffer:

```
long _lock_segment (void *pm_ptr);
```

`_lock_segment` locks a buffer into extended memory. `pm_ptr` is a pointer to the buffer being locked. If successful, `_lock_segment` returns the absolute nonsegmented physical address of the locked buffer. Otherwise, it returns NULL.

If you need to obtain the absolute address of the buffer (or a point within the buffer) after you have locked it, use the following function:

```
long _get_absolute_address (void *pm_ptr);
```

After the DMA operations are completed, unlock the buffer using the following function:

```
void _unlock_segment (void * pm_ptr);
```

The same pointer must be used to lock and unlock the buffer. If you called `_lock_segment` on the same pointer more than once, you must call `_unlock_segment` on the pointer an equal number of times. If a locked buffer is dynamically allocated, you must unlock the buffer before freeing it.

## Providing Support for Stopping Asynchronous I/O

LabWindows must uninstall all interrupt handlers, terminate all DMA transfers, and unlock all variables and buffers on certain occasions, such as when the user wants to temporarily exit to a DOS shell. Object modules that install interrupt handlers, perform DMA transfers, or lock data in memory must provide a way for LabWindows to uninstall the handler, terminate the transfer, or unlock the data. These operations are accomplished through a `message` function and a `stop` function, which are registered with LabWindows by calling the following routine:

```
void _register_stop_functions (void (*message_function)(char *),
                               void (*stop_function)(void));
```

The `stop` function must perform the following actions:

- Stop any DMA transfers.
- Uninstall all previously installed interrupt handlers.
- Unlock all previously locked variables and buffers.

- Deregister the `message` and `stop` functions by calling the following routine:

```
void _deregister_stop_functions  (void (*message_function)(char *),
                                  void (*stop_function)(void));
```

Before calling the `stop` function, LabWindows sometimes displays a dialog box describing the activity that must be terminated and giving the user the option of canceling the request. The `message` function is used to help create this dialog box. The `message` function takes the address of a character buffer as a parameter and places a `message` into the buffer, up to a maximum of 60 characters. The message must describe the type of activity that will be terminated when the `stop` function is called.

## Suspending Asynchronous I/O

There may be situations when you want to suspend asynchronous I/O rather than stop it. For example, the LabWindows RS-232 Library suspends all RS-232 I/O through a `suspend` function activity while a user is shelled out to DOS. Any open communication ports are temporarily closed before the user exits to the DOS shell. When the user returns from the DOS shell, the communication ports are reopened through a `restart` function. Object modules can register their own `suspend` and `restart` functions with LabWindows by calling the following routine:

```
void _register_suspend_functions (void (*suspend_function)(int restart),
                                  void (*restart_function)(void));
```

The `suspend` function must perform the following actions:

- Stop any DMA transfers.
- Uninstall all previously installed interrupt handlers.
- Unlock all previously locked variables and buffers.

LabWindows passes an integer value to the `suspend` function indicating whether LabWindows will call the `restart` function at a later time. A nonzero value indicates that the `restart` function will be called and the I/O should merely be suspended. A zero indicates that the `restart` function will not be called and the I/O should be stopped.

When suspending I/O, the `suspend` function must keep track of all currently installed interrupts and locked variables and buffers.

When stopping I/O, the `suspend` function must deregister the `suspend` and `restart` functions by calling the following routine:

```
void _deregister_suspend_functions  (void (*suspend_function) (int),
                                     void (*restart_function) (void));
```

The `restart` function must perform the following actions:

- Lock all the variables and buffers unlocked by the `suspend` function.
- Install all the interrupt handlers uninstalled by the `suspend` function.

Asynchronous I/O should be suspended only if `_register_stop_functions` does not meet your needs.

**Note**: An object module should not register both a `stop` and a `suspend` function. Only one method should be used to stop asynchronous I/O.

### Standalone Applications

In a standalone application, the `stop` and `suspend` (without `restart`) functions are called only when the standalone application terminates.

## Example–Installing a Timer Tick Interrupt Handler

This section contains example source code for installing, coding, and removing a timer tick interrupt handler.

### C Routines for Timer Tick Interrupt Handler

```
/* next_timer_ptr and cnt are placed in the    */
/* default data segment                        */
void (interrupt ** near next_timer) (void);

extern int near cnt;
extern interrupt timer_tick ();
extern int _install_interrupt (int, void *, void *, void ***, void ***);
extern int _uninstall_interrupt (int, void *, void *);

int install_timer_tick (void)
{
  if (_install_interrupt (8, timer_tick, NULL, &next_timer, NULL) < 0)
         return (ERR);
  return(OK);
}


void remove_timer_tick(void)
{
  _uninstall_interrupt (8, timer_tick, NULL);
}
```

### Assembly Code for the Interrupt Handler

```
.MODEL LARGE
.DATA

extrn _next_timer:dword

         public _cnt
         _cnt dw 0
```

```
.CODE

        public _timer_tick
_timer_tick proc
        push   bx
        push   ds
; load ds with the default data segment
        mov    bx,DGROUP
        mov    ds,bx
; chain to the previously installed handler
        push   ds
        lds    bx,dword ptr [_next_timer]
        pushf
        call   dword ptr [bx]
        pop    ds
; perform timer handler operations
        inc    _cnt
        pop    ds
        pop    bx
        iret
_timer_tick endp

end
```

## Example–Installing a Passup Interrupt Handler

This section contains example source code for installing, coding, and removing a passup interrupt handler.

### C Routines for Installing and Removing the Interrupt

```
extern interrupt pm_int ();
extern int _install_interrupt (int, void *, void *, void ***, void ***);
extern int _uninstall_interrupt (int, void *, void *);

/* place the pointers in the default data segment*/
void * near user_buffer_ptr;
void * near internal_interrupt_table;


int install_passup(int int_num)
{
  if ((internal_interrupt_table = malloc (SIZE)) == NULL)
        return (ERR);
/* the interrupt table is locked into extended memory */
  if (_lock_segment (internal_interrupt_table) == NULL)
        return (ERR);
/* the user buffer is locked into extended memory */
  if (_lock_segment (user_buffer_ptr) == NULL)
        return (ERR);
```

```
  if (_install_interrupt (int_num, NULL, pm_int, NULL, NULL) < 0)
        return (ERR);
  return (OK);
}


void remove (int int_num)
{
  _uninstall_interrupt (int_num, NULL, pm_int);
  _unlock_segment (user_buffer_ptr);
  _unlock_segment (internal_interrupt_table);
}
```

### Assembly Code for the Interrupt Handler

```
.MODEL LARGE
.DATA

extrn _user_buffer_ptr:dword

.CODE

        public _pm_int
_pm_int proc
        push   es
        push   bx
        push   ds
        mov    bx,DGROUP
        mov    ds,bx
; load the user buffer address into es:bx
        les    bx,dword ptr [_user_buffer_ptr]
        .
        .
        .
        pop    ds
        pop    bx
        pop    es
        iret
_pm_int endp
end
```

## LabWindows Callable Functions

### _deregister_stop_functions

**Syntax**: `void _deregister_stop_functions (void (*message_function)(char *), void (*stop_function)(void));`

**Action**: Deregisters the previously registered `message` and `stop` functions.

**Remarks**:

Input parameters:

| | |
|---|---|
| `message_function` | `function pointer to the message function` |
| `stop_function` | `function pointer to the stop function` |

---

### _deregister_suspend_functions

**Syntax**: `void _deregister_suspend_functions (void (*suspend_function)(int), void (*restart_function)(void));`

**Action**: Deregisters the previously registered `suspend` and `restart` functions.

**Remarks**:

Input parameters:

| | |
|---|---|
| `suspend_function` | `function pointer to the suspend function` |
| `restart_function` | `function pointer to the restart function` |

---

### _get_absolute_address

**Syntax**: `long _get_absolute_address (void *pm_ptr);`

**Action**: Returns the absolute nonsegmented physical address of the buffer pointed to by `pm_ptr`. The absolute address remains constant as long as the buffer is locked. Do not call `_get_absolute_address` unless the buffer has been locked.

**Remarks**:

Input parameter:

| | |
|---|---|
| `pm_ptr` | `pointer to the data item` |

Return value:

`Absolute address of the data item or NULL if the function failed.`

---

## _install_interrupt

**Syntax**:
```
int _install_interrupt  (int vec_num,
                         void *rm_addr,
                         void *pm_addr,
                         void ***rm_chain_ptr,
                         void ***pm_chain_ptr);
```

**Action**: Installs an interrupt handler as either passdown or passup depending on the parameters `rm_addr` and `pm_addr`. A passdown interrupt is installed by passing the address of the real mode interrupt handler in `rm_addr` and a NULL in `pm_addr`. A passup interrupt is installed by passing the address of the protected mode interrupt handler in `pm_addr` and a NULL in `rm_addr`.

Along with `rm_addr` and `pm_addr`, an associated `rm_chain_ptr` and `pm_chain_ptr` must be passed to `_install_interrupt`. `rm_chain_ptr` and `pm_chain_ptr` contain the address of a pointer to a function pointer. These function pointers are used to chain to the previously installed interrupt handler.

**Remarks**:

Input parameters:

```
vec_num         interrupt vector number
rm_addr         address of real mode interrupt routine
                or NULL
pm_addr         address of protected mode interrupt
                routine or NULL
rm_chain_ptr    address of a pointer to a function pointer used to chain
                to previously installed real mode interrupt handler
pm_chain_ptr    address of a pointer to a function pointer used to chain
                to previously installed protected mode interrupt handler
```

Return values:

```
 0  success
-1  general failure
-2  invalid parameter(s)
-3  limit of 50 installed interrupts exceeded
-4  insufficient conventional or extended memory
-5  limit of installed passup interrupts exceeded
-6  interrupt type (passdown or passup) conflicts with the interrupt
handler type  previously installed at the interrupt vector
```

---

## _lock_segment

**Syntax**: `long _lock_segment (void *pm_ptr);`

**Action**: Locks the segment associated with `pm_ptr` into memory (either extended or conventional memory) and returns the absolute nonsegmented physical address of the buffer pointed to by `pm_ptr`.

**Remarks**:

Input parameter:

`pm_ptr` `pointer to the buffer being locked into memory.`

Return value:

`Absolute address of the locked buffer or NULL if failed.`

---

## _register_stop_functions

**Syntax**:
```
void _register_stop_functions  (void (*message_function)(char *),
                                void (*stop_function)(void));
```

**Action**: Registers a `message` and `stop` function. The `message` function passes a pointer to a character buffer and must fill the buffer with a message (no greater than 60 characters) pertaining to the operation of the `stop` function. The `stop` function must perform the following actions:

- Stop any DMA transfers.
- Uninstall all previously installed interrupt handlers.
- Unlock all previously locked variables and buffers.
- Deregister the `message` and `stop` functions by calling the function `_deregister_stop_functions`.

**Note**: Only one set of `stop` and `message` functions is registered when `_register_stop_functions` is called multiple times with the same parameters.

**Remarks**:

Input parameters:

| | |
|---|---|
| `message_function` | `function pointer to the message function` |
| `stop_function` | `function pointer to the stop function` |

---

## _register_suspend_functions

**Syntax**: `void _register_suspend_functions (void (*suspend_function)(int), void (*restart_function)(void));`

**Action**: Registers a `suspend` and `restart` function. The `suspend` function must perform the following actions:

- Stop any DMA transfers.
- Uninstall all previously installed interrupt handlers.
- Unlock all previously locked variables and buffers.

The `suspend` function is passed an integer value indicating whether the `restart` function will be called at a later time. A nonzero indicates that the `restart` function will be called and that the I/O should merely be suspended. A zero indicates that the `restart` function will not be called and that the I/O should be stopped.

When suspending I/O, the `suspend` function must keep track of all currently installed interrupts and locked variables and buffers.

When stopping I/O, the `suspend` function must deregister the `suspend` and `restart` functions by calling the routine `_deregister_suspend_functions`.

The `restart` function must perform the following actions:

- Lock all the variables and buffers unlocked by the `suspend` function.
- Install all the interrupt handlers uninstalled by the `suspend` function.

**Note**: Only one set of `suspend` and `restart` functions is registered when `_register_suspend_functions` is called multiple times with the same parameters.

**Remarks**:

Input parameters:

```
suspend_function        function pointer to the suspend function
restart_function        function pointer to the restart function
```

---

## _set_absolute_access

**Syntax**: `int _set_absolute_access (unsigned seg, int access);`

**Action**: Register a transparent segment as either code or data. LabWindows provides 48 transparent segments:

A000, A200, A400, ..., FE00

**Remarks**:

Input parameters:

```
seg         transparent segment value
access      access type (0 = data, 1 = code)
```

Return value:

```
Zero if successful or nonzero if failed.
```

---

## _uninstall_interrupt

**Syntax**:
```
int _uninstall_interrupt  (int vec_num,
                           void *rm_addr,
                           void *pm_addr);
```

**Action**: Removes a passdown or passup interrupt handler by restoring the original interrupt handler. The parameters `rm_addr` and `pm_addr` must match the values used to install the interrupt handler using `_install_interrupt`.

**Remarks**:

Input parameters:

```
vec_num     interrupt vector number
rm_addr     address of real mode interrupt routine or NULL
pm_addr     address of protected mode interrupt routine or NULL
```

Return values:

```
 0          success
-1          failure
```

---

## _unlock_segment

**Syntax**: `void _unlock_segment (void *pm_ptr);`

**Action**: Unlocks a previously locked variable buffer, allowing the Virtual Memory Manager to swap the segment in and out of memory. If `_lock_segment` has been called on the same pointer more than once, `_unlock_segment` must be called on the pointer an equal number of times.

Any dynamically allocated memory that has been locked using `_lock_segment` must be unlocked before the memory is freed.

**Note:** The segment is not unlocked if any other variables or buffers in the segment are currently locked.

**Remarks**:

Input parameter:

```
pm_ptr      pointer to the buffer being unlocked.  The same pointer used
            to lock the buffer must also be used to unlock the buffer.
```

---

# Chapter 2
# Programming Guidelines for Installing Bimodal Interrupt Handlers

This chapter contains general guidelines for coding and installing bimodal interrupt handlers within LabWindows loadable object modules.

Bimodal interrupt handlers are used in LabWindows to reduce the maximum potential delay in servicing interrupts. Interrupt servicing can be delayed when an interrupt occurs during a mode switch. Mode switches are caused by clock interrupts, DOS service calls (such as file I/O), and GPIB handler calls. Interrupt servicing can also be delayed if the processor must switch modes to service the interrupt. For instance, when a passup interrupt occurs in real mode, the processor must switch to protected mode to service the interrupt. Depending on the computer, the maximum potential delay for a passup interrupt can be as long as 4 milliseconds or as brief as 200 microseconds.

A bimodal interrupt handler establishes routines for processing the interrupt in both real and protected mode. In this way, no mode switches are required to service a bimodal interrupt. However, the potential delay is not entirely eliminated. In particular, if the bimodal interrupt occurs during a clock interrupt, the bimodal interrupt is not serviced until the clock interrupt handler and any related mode-switching have completed. In general, bimodal interrupt handlers reduce the maximum interrupt latency period by a factor of two, approximately.

Like passup handlers, bimodal handlers are not allowed to chain to the previously installed interrupt handler.

## Accessing Memory from Bimodal Interrupt Handlers

Bimodal interrupt handlers must *not* access user variables or buffers (that is, variables or buffers allocated within a user's program and passed to a library function). These items are allocated in the same segments as other variables and buffers that may have to be locked in extended memory. Bimodal handlers can only access data that is statically or dynamically allocated within the loadable object module.

Because of the differences in real and protected mode addressing, interrupt routines must be aware of the address type (real, transparent, or protected) being used to access data. In real mode, bimodal interrupt handlers must use real mode or transparent addresses. In protected mode, bimodal interrupt handlers must use protected mode or transparent addresses. In addition, the data must be reside in conventional memory.

Ideally, a bimodal interrupt routine should access only a small number of bytes (that is, less than 100) and the data should be in the default data segment. Data items in the default data segment have transparent addresses. No other data items have transparent addresses.

If the number of bytes is too large to place in the default data segment, the following alternatives are available.

- You can access a dynamically allocated buffer if it contains 65,536 bytes. To allocate such a buffer, use the following call:

  ```
  ptr = _lw_malloc (0);
  ```

  To free the buffer, call:

  ```
  _lw_free (ptr);
  ```

- The interrupt handler can access data declared within the library code only if the total amount of data declared within the object module's data segment is greater than a LabWindows threshold. Currently, this threshold is set at 100 bytes. This condition guarantees that the data is allocated in its own segment. Please consult National Instruments before accessing data declared within the library module.

# Coding Bimodal Interrupts

This section provides guidelines for installing, coding, and removing a bimodal interrupt handler. A bimodal interrupt handler contains both a real mode routine and a protected mode routine.

## Installing the Bimodal Interrupt Handler

To install a bimodal interrupt handler, follow these steps:

1. Lock into conventional memory any variables used by the interrupt routines that are not in the default data segment. Use the following LabWindows function:

   ```
   void * _lock_segment_low (void *pm_ptr)
   ```

   If successful, `_lock_segment_low` returns a real mode pointer to the locked buffer. If unsuccessful, a NULL is returned. `pm_ptr` is a pointer to the variable being locked. The real mode address should be stored in the default data segment so the real mode interrupt routine can access it.

2. Install both the real mode and protected mode interrupt routines using the LabWindows call:

   ```
   int _install_interrupt (int vec_num,
                           void *rm_addr,
                           void *pm_addr,
                           void **rm_chain_ptr,
                           void ** pm_chain_ptr);
   ```

   The address of the real mode entry point for the interrupt handler should be passed in `rm_addr`, while the address of the protected mode entry point for the interrupt handler should be passed in `pm_addr`. `rm_chain_ptr` and `pm_chain_ptr` should be NULL because chaining is not permitted in bimodal interrupt handlers.

## Coding the Interrupt Handler

It may be possible to write a single interrupt handler that works in both real and protected mode. More than likely, separate entry points will be required. The majority of interrupt handler code can be shared by both real and protected mode handlers. The separate entry points can serve to set flags or make address adjustments.

A bimodal interrupt handler should not be in the same code segment as a passup interrupt handler.

### Removing the Bimodal Interrupt Handler

To remove a bimodal interrupt handler, follow these steps:

1. Use the LabWindows function `_uninstall_interrupt` to restore the original real mode and protected mode interrupt handlers.

```
int _uninstall_interrupt (int vec_num, void *rm_addr, void *pm_addr);
```

**Note:** The same parameters used to install the interrupt handler must also be used to uninstall the interrupt.

2. Unlock any previously locked variables. Use the following LabWindows function:

```
void _unlock_segment_low (void * pm_ptr);
```

   The same pointer used to lock the variable must also be used to unlock it. If you call `_lock_segment_low` on the same pointer more than once, you must call `_unlock_segment_low` on the pointer an equal number of times. If a locked variable is dynamically allocated, you must unlock the variable before freeing it.

## Example–Installing a Bimodal Interrupt Handler

This section contains example source code for installing, coding, and removing a bimodal interrupt handler.

### C Routines for Installing and Removing the Interrupt

```
extern interrupt rm_int ();
extern interrupt pm_int ();
extern void * _lock_segment_low (void *);
extern void _unlock_segment_low void *);
extern int _install_interrupt (int, void *, void *, void ***, void ***);
extern int _uninstall_interrupt (int, void *, void *);

/* iobuffer is allocated in its own segment */
int iobuffer[32768];
/* rmptr_iobuffer is placed in the default data */
/* segment by declaring it near                 */
void * near rmptr_iobuffer;


int install(int int_num)
{
  if ((rmptr_iobuffer = _lock_segment_low (iobuffer))
       == NULL)
          return (ERR);
  if (_install_interrupt (int_num, rm_int, pm_int,
                          NULL, NULL) < 0)
          return (ERR);
  return (OK);
}


void remove(int int_num)
{
```

```
      _uninstall_interrupt (int_num, rm_int, pm_int);
      _unlock_segment_low (iobuffer);
}
```

## Assembly Code for the Interrupt Handler

```
.MODEL LARGE
.DATA

RM EQU 0
PM EQU 1

processor_mode    dw ?

extrn _iobuffer:dword
extrn _rmptr_iobuffer:dword

.CODE

        public _pm_int
_pm_int:
        push    bx
        push    ds
        mov     bx,DGROUP
        mov     ds,bx
; save the previous value of the processor mode
        push    word ptr [processor_mode]
        mov     word ptr [processor_mode],PM
; load es:bx with pointer to our I/O buffer
        les     bx,_iobuffer
        jmp     handler

_rm_int:
        push    bx
        push    ds
        mov     bx,DGROUP
        mov     ds,bx
; save the previous value of the processor mode
        push    word ptr [processor_mode]
        mov     byte ptr [processor_mode],RM
; load es:bx with pointer to our I/O buffer
        les     bx,_rmptr_iobuffer
        jmp     handler

handler:
; process the external hardware interrupt
```

```
; restore the previous processor mode value
        pop     word ptr [processor_mode]
        pop     ds
        pop     bx
        iret

end
```

# LabWindows Callable Functions

## _install_interrupt

**Syntax**:
```
int _install_interrupt  (int vec_num,
                         void *rm_addr,
                         void *pm_addr,
                         void ***rm_chain_ptr,
                         void ***pm_chain_ptr);
```

**Action**: Installs an interrupt handler as either passdown, passup, or bimodal depending on the parameters `rm_addr` and `pm_addr`. A passdown interrupt is installed by passing the address of the real mode interrupt handler in `rm_addr` and a NULL in `pm_addr`. A passup interrupt is installed by passing the address of the protected mode interrupt handler in `pm_addr` and a NULL in `rm_addr`. A bimodal interrupt is installed by passing the address of the real mode entry point in rm_addr and the address of the protected mode entry point in pm_addr.

Along with `rm_addr` and `pm_addr`, an associated `rm_chain_ptr` and `pm_chain_ptr` can be passed to `_install_interrupt`. `rm_chain_ptr` and `pm_chain_ptr` contain the address of a pointer to a function pointer. These function pointers are used to chain to the previously installed interrupt handler. `rm_chain_ptr` and `pm_chain_ptr` should be NULL for bimodal interrupts.

**Remarks**:

Input parameters:

```
vec_num           interrupt vector number
rm_addr           address of real mode interrupt routine or NULL
pm_addr           address of protected mode interrupt routine or NULL
rm_chain_ptraddress of a pointer to a function pointer used to
                  chain to a previously installed real mode interrupt
                  handler
pm_chain_ptraddress of a pointer to a function pointer used to
      chain to a previously installed protected mode
                  interrupt handler
```

Return values:

```
   0  success
  -1  general failure
  -2  invalid parameter(s)
  -3  limit of 50 installed interrupts exceeded
  -4  insufficient conventional or extended memory
  -5  limit of installed passup interrupts exceeded
  -6  interrupt type (passdown, passup, or bimodal) conflicts with the
interrupt handler type previously installed at the interrupt vector
```

## _lock_segment_low

**Syntax**: `void * _lock_segment_low (void *pm_ptr);`

**Action**: Locks the segment associated with `pm_ptr` into conventional memory and returns a real mode pointer to the buffer pointed to by `pm_ptr`.

**Remarks**:

Input parameter:

```
pm_ptr    pointer to the buffer being locked into conventional memory.
```

Return value:

```
Real mode pointer to the locked buffer or NULL if failed.
```

---

## _uninstall_interrupt

**Syntax**:
```
int _uninstall_interrupt  (int vec_num,
                           void *rm_addr,
                           void *pm_addr);
```

**Action**: Removes a passdown, passup, or bimodal interrupt handler by restoring the original interrupt handler. The parameters `rm_addr` and `pm_addr` must match the values used to install the interrupt handler using `_install_interrupt`.

**Remarks**:

Input parameters:

```
vec_num     interrupt vector number
rm_addr     address of real mode interrupt routine or NULL
pm_addr     address of protected mode interrupt routine or NULL
```

Return values:

```
 0  success
-1  failure
```

---

## _unlock_segment_low

**Syntax**: `void _unlock_segment_low (void *pm_ptr);`

**Action**: Unlocks a previously locked buffer by moving the entire segment into extended memory. If `_lock_segment_low` has been called on the same pointer more than once, `_unlock_segment_low` must be called on the pointer an equal number of times.

Any dynamically allocated memory that has been locked using `_lock_segment_low` must be unlocked before the memory is freed.

**Note:** The segment is not unlocked if any other variables or buffers in the segment are currently locked.

**Remarks**:

Input parameter:

```
pm_ptr   pointer to the buffer being unlocked.  The same pointer used
                to lock the buffer must also be used to unlock the buffer.
```

---