The On-Line Addition/Replacement (OLA/R) is a required feature for high-availability servers. This chapter describes writing I/O device drivers on HP-UX 11i for OLA/R support. This chapter mainly focuses on additions to WSIO interface, modifications required to drivers for supporting OLA/R, and handling OLA/R events at the driver. Pseudo code is given wherever required for a better understanding of the concepts under discussion.

In the examples and pseudo code present in this chapter “driver_name” is used. This can be replaced with your driver name when you actually develop a driver for OLA/R support.

The reader of this chapter is assumed to have a good understanding of:

- HP-UX I/O Subsystem
- WSIO driver development environment
- Writing a WSIO driver
- High-availability issues
- Application level impact on OLA/R of a driver of interest
Introduction

The ability to insert device controller cards and replace such cards while a system is being used, without the interruption of services to users not directly affected by the device resource, is a necessary capability for high-availability machines. On PCI based HP 9000 servers, this capability is provided as On-Line Addition/Replacement (OLA/R) of I/O cards. Support for OLA/R is implemented at — hardware, firmware and software. Specified here are the implementation details to provide support for OLA/R in the driver software of an I/O card.

The major functional areas included in this chapter are:

- Overview of the required modifications to the driver for OLA/R support.
- The additions to the WSIO interface for OLA/R support.
- Details of the enhancement/modification steps for OLA/R.
- Details of handling OLA at the driver, MP-safe issues, resource allocation failure issues, and suspend and resume event handling for OLR.
- Miscellaneous code changes at the driver for OLA/R.
- Performing OLA/R of PCI I/O cards.

Supported Hardware and Software

Servers on which OLA/R is supported:

- L-Class, N-Class

Firmware Updates:

- Any required f/w updates

OS Version:

- HP-UX 11i and above
Overview of Driver Modifications Required for OLA/R

A driver can be enhanced or developed with OLA/R support by following these three steps:

1. Register a generic event handler. To support OLA/R functionality, each driver has to register an event handling function with the WSIO. WSIO calls this event handler to service OLA/R events like suspend and resume on the I/O adapter card the driver controls. The generic event handling function is registered in the driver install routine.

2. Register a capability mask. The driver has to register an event mask with the WSIO. The event mask specifies the capabilities of the driver in OLA/R event handling. This allows the same driver to support various capabilities on different instances of I/O adapter. The event capability mask is registered in the driver attach routine.

3. Driver Event Handler. To support OLA/R functionality in the driver, the driver has to handle suspend and resume events that are generated by the WSIO. On-Line Addition (OLA) does not generate any event. However since OLA can be done at any time once the machine is up and running, the attach and init routines of the driver must be made MP-safe and should back out of any errors during allocation of resources.

Detailed discussion on how to implement these three steps are present in later sections.

Since the OLA/R enhancements are not in the driver's main performance path, there should not be any performance penalty on the driver's regular data path.
WSIO Interfaces

WSIO provides a new set of data structures and services that allow driver enhancements for OLA/R support.

wsio_drv_info_t

A new field is added to the `wsio_drv_info_t` structure indicating the version of WSIO. The new `wsio_drv_info_t` structure is given:

```c
typedef struct wsio_drv_info {
    drv_info_t * drv_info;
    drv_ops_t * drv_ops;
    wsio_drv_data_t * drv_data;
    unsigned int driver_version;
} wsio_drv_info_t;
```

A driver would then use the following definition when it calls `wsio_install_driver`, `isc_claim` and `wsio_install_drv_event_handler`.

```c
static wsio_drv_info_t driver_name_wsio_drv_info = {
    &driver_name_info,
    &driver_name_ops,
    &driver_name_wsio_drv_data,
    WSIO_DRV_CURRENT_VERSION
};
```

The macro `WSIO_DRV_CURRENT_VERSION` is defined in `wsio.h`.

WSIO Event Handling Structures

WSIO and the driver that supports OLA/R interact through events and event completion callbacks. A new event data structure is defined in WSIO:

```c
typedef enum {
    WSIO_EVENT_SUSPEND   = 1<<0,
    WSIO_EVENT_RESUME    = 1<<1,
    WSIO_EVENT_REMOVE    = 1<<2,
    WSIO_EVENT_DEV_ERROR = 1<<3,
    WSIO_EVENT_BUS_ERROR = 1<<4,
    WSIO_EVENT_SELF_TEST = 1<<5,
} wsio_event_t;
```

For OLA/R functionality, only the first two events `WSIO_EVENT_SUSPEND` and `WSIO_EVENT_RESUME` are of interest.

```c
typedef unsigned int wsio_event_id_t;
```

The `event_id` is a tag to identify a call to a driver’s event handler and its completion. A driver will return it when it calls the completion call back. It is a number that WSIO uses to match up requests with replies. The callback function is defined as:

```c
typedef int (*generic_complete_callback_t)(
    struct isc_table_type *,
    wsio_event_id_t,
    void *);
```

The event status is returned to WSIO using the third argument (void *) in the above `generic_complete_callback` definition. Although it is defined as a pointer, WSIO expects only one of the WSIO status values described:
typedef enum {
    WSIO_OK = 0,
    WSIO_ERROR = -1,
    WSIO_INFO_NULL = -2,
    WSIO_HANDLER_NULL = -3,
    WSIO_DRV_NOT_FOUND = -4,
    WSIO_INVALID_ISC = -5,
    WSIO_INVALID_EVENT = -6,
    WSIO_NO_DRV_HANDLER = -7,
    WSIO_INVALID_COMBIN_EVENTS = -8,
    WSIO_UNSUPPORTED_EVENT = -9,
    WSIO_HA_NA = -10, /* Not an OLA/R or HA capable system */
    WSIO_DRV_FUNC_NULL = -11,
    WSIO_UNKNOWN_FUNC_TYPE = -12,
} wsio_ret_code_t;

The generic event structure wsio_generic_event_t is defined as follows:

typedef struct wsio_generic_event {
    wsio_event_t event; /* suspend, resume, and so on */
    wsio_event_id_t event_id;
    struct isc_table_type *isc;
    generic_complete_callback_t wsio_completion_cb;
    void *arg;
} wsio_generic_event_t;

The wsio_generic_event_t is passed to the driver event handler function. A driver event handler is a function that WSIO will call when an event, such as suspend, occurs.

The driver's event handler is defined as:

typedef void (*wsio_drv_event_handler_t) (wsio_generic_event_t *);
Driver Modifications for OLA/R Support

The driver must perform the following four steps to support OLA/R functionality:

1. Register its event handler.
2. Register its event capability mask.
3. Set the suspend and resume event timeout values.
4. Driver’s event handler routine.

Driver Registering Its Event Handler

The driver registers a generic event handler function by calling `wsio_install_drv_event_handler`. This should be called after installing the driver by calling `wsio_install_driver` in the driver’s install routine. This service is passed with two arguments:

```c
int wsio_install_drv_event_handler(
    wsio_drv_info_t * drv_info,
    wsio_drv_event_handler_t event_handler);
```

This is called in the `driver_install` routine. Pseudo code for a typical driver install routine is as follows:

```c
int driver_name_install(void)
{
    if (return_value = wsio_install_driver(&driver_name_drv_info)) {
        saved_attach = pci_attach;
        pci_attach = driver_name_attach;
    } else { /* Install Failure */
        return 0;
    }

    /* driver_name_event_handler() is the generic event
     * handler implemented in the driver.
     */
    if (wsio_install_drv_event_handler(
            &driver_name_drv_info,
            driver_name_event_handler) != WSIO_OK) {
        /* Driver not registered its event handler.
         * Driver’s normal operation may not be affected.
         */
    }

    return return_value;
}
```

Driver Registering Its Event Mask

This entry point is for a driver to call inside its attach routine after calling `isc_claim()`. There will be two parameters for this call; `isc` and event mask. Event mask is a `uint64_t` representing a possibility of 64 operations associated with the `isc`.

```c
typedef uint64_t wsio_event_mask_t; /* This is an OR of
                                  wsio_event_t described in last section. */

int wsio_reg_drv_capability_mask(
```
struct isc_table_type *isc,
wsio_event_mask_t event_mask);

The following is pseudo code for a typical driver attach routine:

driver_name_attach()
{
    ...
    wsio_event_mask_t driver_name_event_mask = 0;
    /* There is no specific event for OLA. OLA does not use
     * the driver event handler. It uses the normal attach
     * and init routines. To support OLR, the driver
     * requires to handle two events - suspend and resume.
     */
    driver_name_event_mask = WSIO_EVENT_SUSPEND |
                           WSIO_EVENT_RESUME;
    .... normal attach processing ....
    isc_claim(isc, &driver_name_wsio_drv_info);
    /* Register driver event capability mask.
     * NOTE: This should be called only after
     *   * isc_claim() is called.
     */
    return_value = wsio_reg_drv_capability_mask(isc,
                                              driver_name_event_mask);
    if(return_value != WSIO_OK && return_value !=
       WSIO_HA_NA) {
        /* Registering driver event mask failed.
         * Continue with normal processing.
         */
    }
    .... normal processing ...
}

Driver Event Timeout Values

When WSIO issues an event request to a driver by calling the driver's event handling function, it expects the
driver to complete the event, and to call the callback function within a specified period of time. If the driver
fails to complete the event within that period of time, the event goes into a timeout state. It is undesirable for
any event to enter a timeout state, so a driver must set the timeout value for each event such that the timeout
period will never be exceeded. However, if a driver timeouts for some reason during a suspend or resume
event it still can reply anytime later to change the status. The default timeout limit is ten seconds. If a driver
needs more than this to complete an OLA/R event, it should set the timeout limit to the required value using
the WSIO service wsio_set_parm. The current value of a specific parameter can be obtained from
wsio_get_parm.

The data structure wsio_parm_t defines the valid parameters for wsio_set_parm() and wsio_get_parm().

typedef enum {
    WSIO_HW_SUSPEND_TIMEOUT, /* Hardware Suspend Timeout param */
    WSIO_HW_RESUME_TIMEOUT,  /* Hardware Resume Timeout param */
    WSIO_HW_REMOVE_TIMEOUT,  /* Hardware Remove Timeout param */
    WSIO_HW_ERROR_TIMEOUT,   /* Hardware Error Timeout param */
    WSIO_IDENTIFY_CHILD,     /* Function that identifies if a
                              * child is an interface */
}
WSIO service wsio_set_parm() takes three arguments.

```c
int wsio_set_parm (struct isc_table_type * isc,
                   wsio_parm_t parm,
                   void * value);
```

Even though the third parameter “value” is declared as a pointer to a void, WSIO will read it as a value and NOT as a pointer. So do not use a pointer to a value; instead, use a defined value. The value specified is in microseconds.

This can be called at any place in the driver.

In the following example, the timeout values are set in the driver attach routine after registering the driver event capability mask.

Pseudo code for a typical attach routine is shown:

```c
{ 
    driver_name_attach()
    
    ubit32_t olar_timeout = 15000000;
    /* timeout set to 15 sec */
    
    ... normal processing ...

    isc_claim();
    
    if(wsio_reg_drv_capability_mask(isc,
                                    event_mask) == WSIO_OK) { 
        /* Set the suspend and resume events
         * timeout period */
        if(wsio_set_parm(isc, WSIO_HW_SUSPEND_TIMEOUT,
                         (void *) olar_timeout) != WSIO_OK)
            wsio_reg_drv_capability_mask(isc, 0);
        else if(wsio_set_parm(isc, WSIO_HW_RESUME_TIMEOUT,
                              (void *) olar_timeout) != WSIO_OK)
            wsio_reg_drv_capability_mask(isc, 0);
    }
    
    ... normal processing ...
}
```
Event Handling Function

WSIO generates an event to a driver when it receives an OLA/R request (from SAM or rad) on an I/O card the driver controls. WSIO calls the driver’s event handling function which the driver has registered with the WSIO during the driver’s installation time. It is the driver’s responsibility to make sure that an OLA/R event is never timed out. OLA does not generate any event and follows the normal attach and init routines path. So the events supported by the driver would be for suspend and resume.

The following is pseudo code for a typical driver event handling routine:

```c
void driver_name_event_handler(wsio_generic_event_t *handler_arg) {
    /* Switch based on the event The timeout calls
     * in the switch cases below are to allow the
     * event handler to return to WSIO immediately.
     * The timeout value of 0 is used in those cases
     * where there is no need to delay execution of the
     * individual event handler.
     * / *
    switch (handler_arg->event) {
        case WSIO_EVENT_SUSPEND:
            * Save callback function and other info like
              event_id
              * Call timeout(driver_name_suspend,
                  driver_name_suspend_info, 0);
              * break;
        case WSIO_EVENT_RESUME:
            * Check if it is a like-for-like card
              * Save callback function and other info like
                event_id
              * Call timeout(driver_name_resume,
                  driver_name_resume_info, 0);
              * break;
        default:
            handler_arg->wsio_completion_cb(handler_arg->isc,
                handler_arg->event_id, WSIO_UNSUPPORTED_EVENT);
            break;
    }
    return;
}
```
Handling OLA/R

This section describes how On-Line Addition (OLA) and On-Line Replacement (OLR) have to be handled when designing the driver.

For detailed information on performing OLA/R operations, refer to the user manuals. The following information is only a description of the sequence of required operations necessary to write a driver with OLA/R support.

OLA

On-Line Addition of a card instance can be performed with the following steps:

1. Power-off the slot from System Administration Manager (SAM) or rad.
2. Insert the card into the slot.
3. Power-on the slot from SAM or rad.
4. Run ioscans, optionally passing the H/W path of the slot.

As mentioned earlier, On-Line addition of an I/O card does not generate any WSIO event that the driver event handler will handle. Instead, the driver's attach and init routines are called, just as they would be during the boot. There are two issues that a driver should handle to support OLA. They are:

- Driver's attach and init routines should be MP-safe
- Driver should handle resource allocation failures during the attach and init.

MP Safe

The attach and init routines must be MP-Safe; an OLA can occur at any time once the machine is up and running. If the driver has any global resources which are common to all instances, it is advisable to allocate them during the driver install time. If required global locks can be acquired and released during attach and init. routines to serialize access and/or to protect data.

Resource Allocation Failures

Since an OLA can occur at any time once the machine is up and running, resource allocation may fail because of resource shortage problem. So the driver should be able to retrace all the steps and release all the resources that are allocated up until the failure occurred. This will take the driver back to a clean state.

To facilitate the back out of a driver because of a failure during attach and init, a list of resource allocation services and their corresponding releasing services are given. A word of caution is, release or free the resources in LIFO order; i.e., the most recently allocated resources are released first.

<table>
<thead>
<tr>
<th>Resource Allocation/Acquire</th>
<th>Resource Free/Release</th>
</tr>
</thead>
<tbody>
<tr>
<td>kmalloc</td>
<td>kfree</td>
</tr>
<tr>
<td>wsio_map</td>
<td>wsio_unmap</td>
</tr>
<tr>
<td>wsio_map_port</td>
<td>wsio_unmap_port</td>
</tr>
<tr>
<td>wsio_allocate_shared_memory</td>
<td>wsio_free_shared_memory</td>
</tr>
</tbody>
</table>
All drivers must provide a return value from the `init` functions. The values must be either:

- `WSIO_OK` (0) on success.
- `WSIO_ERROR` (-1) on failure.

Refer to the ENET sample driver for an implementation example.

**OLR**

To replace a bad I/O card when the system is on-line, OLR can be used. Usually the following steps are followed while OLR of an I/O card:

1. Suspend the driver instance of the I/O card from SAM or `rad`.
2. Power-off the slot from SAM or `rad`.
3. Remove the card.
4. Insert a new I/O card.
5. Power-on the slot from SAM or `rad`.
6. Resume the driver instance of the I/O card from SAM or `rad`.

When writing a driver to support OLR, the driver should handle the events that happen during steps 1 and 6.

WSIO guarantees that suspend and resume requests will be single-threaded. Also, the underlying OLA/R infrastructure guarantees that no suspend or resume events are triggered to the same card instance until the previous event processing is completed. For OLR, the underlying infrastructure does not trigger a resume event if the replacement card is outright incompatible. The driver is expected to do a more detailed check, like comparing the PCI sub-system ID etc., before claiming the replacement card.

The following WSIO OLA/R device driver state diagram in Figure 15-1, “OLA/R State Diagram,” shows the driver states from the perspective of WSIO. The internal driver states will differ from driver to driver. The timeout states in the diagram should not occur if the driver sets an appropriate value in the corresponding timeout values using `wsio_set_parm()`. Since recovery from the timeout states cannot be fully defined, it is very important that these states are never reached.
The numbered state transition labels in the figure do not represent a sequence. The transitions are as follows:

- Normal suspend sequence is 1, 2.
- If a suspend fails, the sequence is 1, 5.

**NOTE**

A SUSPEND MUST NEVER FAIL.

- If a suspend timeout occurs, the sequence is 1, 6. From the SUSPENDING TIMEOUT state, 7 can occur if the driver calls the callback function with a status of `WSIO_OK` after a timeout has occurred. Transition 8 can occur if the driver calls the callback function with a status of `WSIO_ERROR` after the timeout has occurred.
- The normal resume sequence is 3, 4.
- If a resume fails, the sequence is 3, 9. From the RESUMING TIMEOUT state, 11 can occur if the driver calls the callback function with status `WSIO_OK` after a timeout has occurred. Transition 12 can occur if the driver calls the callback function with a status of `WSIO_ERROR` after the timeout has occurred.
Suspend

The sequence of events during a suspend is as follows:

1. Wait for the correct state. Definition of the “correct state” depends on a driver. But speaking in broad terms, no critical operation should be in progress like device reset, blocked on I/O request, etc. Pseudo code to perform this follows:

   ```c
   driver_name_event_handler(handler_arg)
   {
      ....
      switch(handler_arg->event) {
         ....
         case WSIO_EVENT_SUSPEND:
            if(driver_defined_state == DRIVER_DEFINED_RESET ||
               driver_defined_state == DRIVER_DEFINED_IOBLOCK
               || ...) {
               /* Set a timeout for suspend routine */
               timeout(driver_name_suspend_handler,
                       suspend_info, PREDEFINED_WAITING_TIME);
            } else {
               /* Call suspend handler. No need to wait here */
               timeout(driver_name_suspend_handler,
                       suspend_info, 0);
            }
            break;
         ....
      }
   ...}
   
   If the driver can not set any predetermined time as shown previously, the driver’s suspend handler can timeout itself for some number of clock ticks until the driver comes out of its critical operation state. Pseudo code to perform this follows:

   ```c
   driver_name_suspend_handler(suspend_info)
   {
      ....
      if(driver_defined_state == DRIVER_DEFINED_RESET ||
         driver_defined_state == DRIVER_DEFINED_IOBLOCK
         || ...) {
         /* Set a timeout for suspend routine */
         timeout(driver_name_suspend_handler,
                 suspend_info, HZ);
         return;
      }
      ....
   }
A suspend request should never occur while a driver is suspending or suspended. If one occurs, the driver should immediately call the callback function with a status of **WSIO_INVALID_EVENT**. While a driver is suspending or suspended, some control requests, such as a reset or abort (or any other request which would normally cause the driver to enter a different state, or interact directly with the adapter) should be rejected. In these cases, the control request path must check for the SUSPENDED state (or equivalent) as well as any state the driver may be in during suspension for certain requests. These are just guidelines and what to do while handling a control request during and after a suspend operation is driver dependent.

2. If the driver has any timers, (other than timers specifically used to time the suspension sequence), the timers must be cancelled using `untimeout()`.

3. Quiesce the device. This is driver dependent. After this, the device is not expected to perform any I/O operations or generate interrupts to the driver.

4. Save the required device information for a following resume operation. For example, when only like-for-like replacement is allowed during resume, comparing the vendor ID, subsystem ID, etc., can be used in identifying a suitable replacement I/O card.

5. Save any other required information. This is driver dependent. Generally speaking, this information would include the required state or configuration of the device the replacement I/O card should be brought to.

6. Call the callback function with a success.

Refer to the ENET sample driver for an implementation example.

**Resume**

A resume request should only occur when the driver is suspended, and not currently in the process of resuming. If the resume event occurs at any other time, the driver should call the callback function with a status of **WSIO_INVALID_EVENT**.

If resume fails, the driver must return to the SUSPENDED state, and the callback function must be called with a status of **WSIO_ERROR**.

While a driver is resuming, some control request, such as a reset or abort (or any other request which would normally cause the driver to enter a different state, or interact directly with the adapter) should be rejected. Therefore, the control request path must check for the RESUMING state (or equivalent) as well as any state the driver may be in during resuming for certain requests. These are just guidelines and what to do while handling a control request during and after a resume operation is driver dependent. The sequence of events during a resume is described:

1. Check for a like-for-like replacement. Currently, the PCI CDIO checks for Vendor and Device IDs. If like-for-like replacement conditions do not meet the requirements of the driver, the driver must return to the SUSPENDED state, and the callback function must be called with a status of **WSIO_ERROR**.

2. Initialize the device. Again, this is driver dependent and what is part of initialization depends on the device and the driver.

3. Restore the state of the device if required. This is driver dependent.

4. Configure the device with the information saved during the suspend. This is driver dependent.

5. Call the callback function with **WSIO_OK** status.
Pseudo code for a driver resume routine is as follows:

\begin{verbatim}
\texttt{driver\_name\_event\_handler(handler\_arg)}
{
    ...

    \texttt{switch(handler\_arg->event) \{}
        ....

        \texttt{case WSIO\_EVENT\_RESUME:}
            \texttt{/* Check if the driver is in suspended state*/}
            \texttt{/* Test of like-for-like replacement. */}
            \texttt{/* Call the driver resume handler */}
            \texttt{timeout(driver\_name\_resume\_handler,}
                \texttt{resume\_info, 0);}

            \texttt{break;}
            ....
        \}
    ...
}

\texttt{driver\_name\_resume\_handler(resume\_info)}
{
    ....

    \texttt{/* Turn on PCI memory access and bus master}
        \texttt{/* capability on host, if applicable */}

    \texttt{/* Initialize the device */}

    \texttt{/* Configure the device */}

    \texttt{/* Call callback function with WSIO\_OK status */}

    \texttt{return;}
}
\end{verbatim}

Refer to the ENET sample driver for an implementation example.


**Miscellaneous Changes Required at the Driver**

As mentioned earlier, while processing control path requests like device reset, configuring the device etc., checks are required to see whether the driver is in SUSPENDED or RESUME states, or in the process of suspending or resuming. A driver state flag can be included in the driver control structure to keep track of the driver’s OLA/R states.

If the driver is in SUSPENDED or RESUME states, the request should be reject with ENXIO. A portion of pseudo code for a driver reset is given:

```c
driver_name_reset(reset_info)
{
    ....
    /* Addition state check */
    if(driver_olar_state == SUSPENDED) {
        /* Return ENXIO saying that the device is busy */

        return ENXIO;
    }
    ....
}
```
Performing OLA/R of PCI I/O Cards

HP-UX provides two interfaces to perform OLA/R operations on PCI I/O cards:

- SAM (GUI)
- `/sbin/rad` (commandline)

Currently, only commandline tool `/sbin/rad` can be used to issue OLA/R requests on I/O cards that are controlled by third party drivers. SAM will be extended in later releases with support for performing OLA/R operation on I/O cards that are controlled by third party drivers.

Refer to the user documentation on *Managing PCI cards with OLA/R* for a detailed description of the use of these tools.
On-Line Addition/Replacement
Performing OLA/R of PCI I/O Cards