CAPI FUNCTIONAL SPECIFICATION VERSION 3.4

Configuration Application Programming Interface for Chaparral External RAID Controllers and Intelligent Storage Routers

Document Revision Date: 20 Sep 2002

Chaparral Network Storage, Inc.
7420 East Dry Creek Parkway
Longmont, Colorado 80503
USA
http://www.chaparralnet.com

Changes
The material in this document is subject to change without notice. While reasonable efforts have been made to ensure the accuracy of this document, Chaparral Network Storage, Inc. assumes no liability resulting from errors or omissions in this publication, or from the use of the information contained herein.
Chaparral reserves the right to make changes in the product design without reservation and without notification to its users. Comments and suggestions can be sent to the address listed above.

Technical Support
If after reviewing this specification, you still have questions about installing or using your Chaparral product, please contact us at (303) 845-3200 or by e-mail at support@chaparralnet.com.

Chaparral Part Number
07-0003-340

NEW!

Differences from Previous Versions
The graphic "NEW! " has been inserted in this document to highlight differences between CAPI 3.1 and CAPI 3.2.
The graphic "NEW! in CAPI 3.3" has been inserted in this document to highlight differences between CAPI 3.2 and CAPI 3.3. CAPI 3.3 was introduced with RIO 1.0.
The graphic "NEW! in CAPI 3.4" has been inserted in this document to highlight differences between CAPI 3.3 and CAPI 3.4. CAPI 3.4 was introduced with RIO 1.1.

See also *CAPI Versions* on page 29.

Table of Contents

Index of Tables and Figures

1

INTRODUCTION

This document is intended for software developers writing their own configuration and management applications for Chaparral external RAID controllers and intelligent storage routers (referred to collectively as “controllers” in this document).

Please see the note on the back of the title page for information about versions of CAPI.

CAPI Overview

The Configuration Application Programming Interface (CAPI) is a set of high-level functions that allow users of Chaparral's external controllers (routers and RAID controllers) to quickly develop custom applications that perform set up, configuration, and management tasks. The custom applications run on a separate processor from the external controller. Chaparral has developed a single API and two implementations of the communications software that underlies that API. this communications software is referred to as an “LMX” (Link Manager Exchange). One LMX communicates with the controller through an “out-of-band” RS-232 (serial) interface as shown in Figure 1-1. The other LMX communicates “in-band” through the SCSI or Fibre Channel interface as shown in Figure 1-2. Both versions present the same API so that a user’s application can run on either implementation without changes.

Chaparral provides a Software Developer’s Kit (SDK) that includes this document and sample code files. These code files are provided as C source code. Some of the code files should be usable by an application developer with little or no change. You simply compile these and link them with your application to provide the interface to the Chaparral controllers. One of the code files provides the function interface for all the CAPI functions defined in Chapter 5, *CAPI Function Reference*, and this should be usable with no changes. If you are developing your application for a Windows NT 4 or Windows 2000 PC, then you should be able to use the LMXs provided with the SDK with few or no changes. Some customization may be required for other platforms, such as UNIX systems that use a different “endian” convention from a PC. The SDK also includes a sample command-line-interface application. For a complete list of the code files that are included with the SDK, see page 12.

Some developers prefer not to use our SDK, but instead choose to develop their own interface to Chaparral controllers. We generally recommend against this, especially if you are developing a CAPI app that will run on Windows NT or 2000 (since the sample code has been tested on Windows NT) or if you are developing a complex application that will use many of the commands defined in the Function Reference in Chapter 5. But if you prefer to design your own interface, see Chapter 18 for some information that will be useful to you.

Figure 1-1. Controller System With RS-232 Configuration API

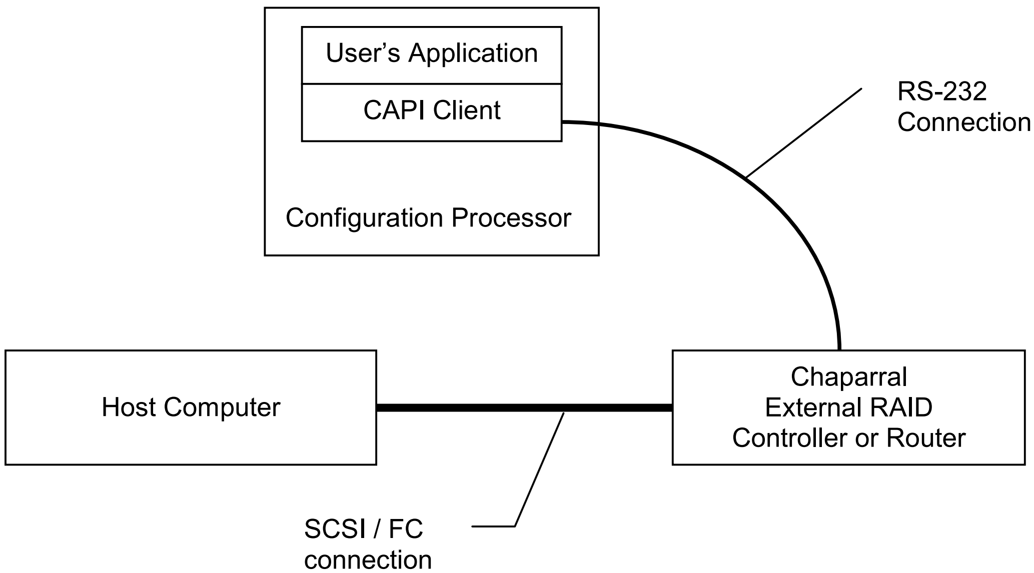


Figure 1-1 shows the RS-232 implementation of CAPI and a user's storage management application running on a separate management processor, such as the enclosure manager for a remote storage enclosure. The API can also run on the host system with a separate RS-232 connection from the host.

Figure 1-2. Controller System With SCSI Configuration API

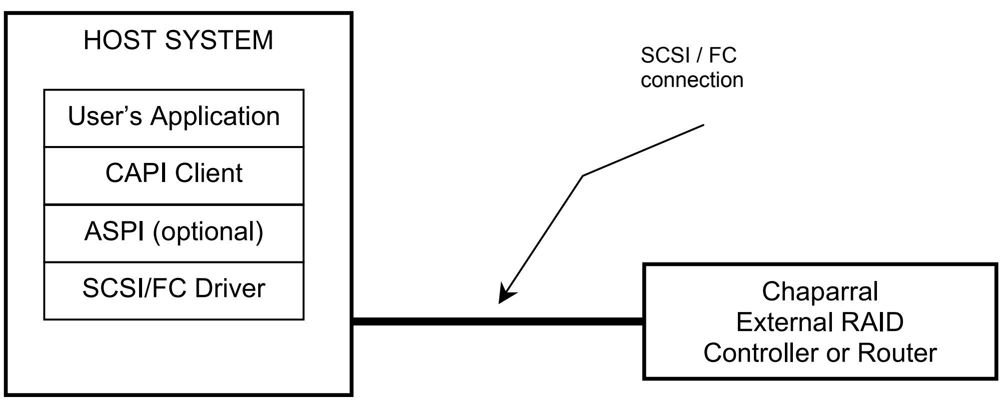


Figure 1-2 shows a system using the SCSI protocol implementation of CAPI. This implementation provides a storage management interface to the external controller via the same SCSI or Fibre Channel interface that the host uses for data so that no separate RS-232 interface is required.

Document Overview

- Chapter 2, CAPI Programming Concepts, provides an overview of key CAPI programming concepts, including a description of the behavior of asynchronous CAPI functions and associated callbacks and a description of how to port CAPI into various processors.
- Chapter 3, Typedefs and Defines, provides a listing of the typedefs used in the CAPI SDK.
- Chapter 4, Data Structures, provides a detailed description of the data structures that are passed across the interface.
- Chapter 5, CAPI Function Reference, provides detailed descriptions of each CAPI function including 'C' prototypes and descriptions of each function parameter.
- Chapter 6, Reply Code Reference, describes the replies received by the configuration application through the callback routine.
- Chapter 7, Event Code Reference, describes the event codes received by the configuration application through the callback routine.
- Chapter 8, Return Code Reference, describes the return codes received by the configuration application through the callback routine.
- Chapter 9, Error Code Reference, describes the error codes.
- Chapter 10, Link Manager Exchange (LMX), describes the communications layer that is used to match CAPI function calls to the appropriate underlying data exchange layer.
- Chapter 11, SCSI LMX, describes one of the LMXs provided with the SDK. It may be used to transport CAPI messages between host computer and controller over either Fibre Channel or parallel SCSI physical transports.
- Chapter 12, RS-232 LMX, describes another LMX that is included with the SDK. It may be used to transport CAPI messages between a management computer and controller over an RS-232 interface.
- Chapter 13, Simplified RS-232 LMX, describes another RS-232 LMX.
- Chapter 14, Changes since CAPI2.x, describes some of the changes since the last major revision to CAPI.
- Chapter 15, Capabilities, provides CAPI capability discussions related to some specific products.
- Chapter 16, Failover Notes, provides some notes related to failover.
- Chapter 17, Non-CAPI Pass Through Feature, describes a facility for communicating with back-end devices through a Chaparral controller, but bypassing CAPI.
- Chapter 18, CAPI Interface Without Using the CAPI SDK, describes the messaging interface used to communicate with CAPI.

◇ ◇ ◇ 2

CAPI PROGRAMMING CONCEPTS

CAPI Basics

CAPI application programs (commonly referred to as "CAPI apps") communicate with the CAPI Client (provided by Chaparral as sample code) via function calls and a callback function, which are compiled and linked with the customer's code. The CAPI Client uses the Link Manager Exchange code layer (LMX) and possibly a data exchange layer (low-level transport) to communicate over a remote link to the controller. The LMX is customized to the appropriate communication protocol layer, which may be RS-232 serial, parallel SCSI, or Fibre Channel. By changing the CAPI LMX and re-linking, the application source can remain unchanged and independent of the link type used to communicate with the controller. (LMXs are discussed further in Chapters 10 through 13.)

The CAPI interface consists of a set of CAPI functions, also referred to as CAPI commands, which are passed by CAPI messages. The CAPI functions are defined in the CAPI Function Reference chapter. All CAPI applications operate in a master/slave mode (also referred to as client/server). The customer's management application (running on a host computer) is the master (client) and CAPI software in the Chaparral controller is the slave (server). The slave never sends an unexpected message to the master; it only responds to commands from the master.

All CAPI functions return a CAPI_RETURN_CODE (see page 339) from the host LMX. Six commands (CAPI_EnablePacketCompression, CAPI_EnablePacketCompressionMasterToSlave, CAPI_RegisterCallback, CAPI_Initialize, CAPI_FindNextController, and CAPI_TimerTick) do not send command packets to the controller; they are processed locally by the LMX on the host computer. CAPI_Initialize uses a callback function to indicate success or failure; the other five commands indicate if they have completed successfully with the function return code. All the other CAPI functions communicate with the controller over the remote link and use a callback function to provide status from the controller. For these commands, the return code indicates whether or not the command was sent out on the remote link; it does not indicate the command succeeded. The callback routine provides the application with a CAPI_REPLY_CODE (see page 328) and a CAPI_ERROR_CODE (see page 340). The reply code describes the command sent to the controller, and the error code indicates if it started or completed successfully.

At initialization time, the application provides the callback function to the LMX with the CAPI_RegisterCallback call. Obviously, no commands that communicate over the remote link can be sent before this function has been called. (The callback function is discussed further on page 6. Initialization is discussed further on page 15.)

Commands that take many seconds or longer for the controller to complete require the application, through the LMX, to periodically poll the controller for an event (by calling CAPI_GetLastEvent or CAPI_U_GetLastEvent) to determine completion of the command. These commands are designated "lengthy operation" in this specification. An example of a lengthy operation is CAPI_CreateArray; completion is indicated by the event CAPI_EVENT_CREATE_ARRAY_COMPLETE. Percent complete status for lengthy operations can be found using CAPI_GetPercentComplete. Note that lengthy operations will complete quickly if their callback error code indicates failure.

Note: *Creating an array with large disk drives can take many hours.*

All lengthy operations perform in the same manner by responding quickly via the callback function and providing the event status (CAPI_REPLY_CODE) and error code (CAPI_ERROR_CODE) as function parameters to the callback function. At approximately the same time as the callback function is called, an event is logged indicating that the command has started. Then, at some later time, a second event is logged indicating that the command has completed. The event information is provided via a data pointer to a data structure (CAPI_EVENT) containing the event code. (See the EVENT_CODE reference on page 332.) The event code name for the start of an operation will correspond to the reply code for that operation; for example, CAPI_REPLY_CREATE_ARRAY_START and CAPI_EVENT_CREATE_ARRAY_START.

The callback function's errorCode parameter must always be checked for command success. If the reply to a CAPI_CreateArray (which has replyCode CAPI_REPLY_CREATE_ARRAY_START) has an error, then the application should not expect to find the completion event (CAPI_EVENT_CREATE_ARRAY_COMPLETE would be expected). This is because the controller could not start the create-array process due to some error. (Lengthy operations are discussed further on page 7.)

Since a CAPI remote link can have only one outstanding operation at a time, if another thread (or interrupt context) in the application makes a CAPI call while the link is busy, it receives a return code (not reply code) of CAPI_STATUS_LINK_BUSY. This requires the application developer to coordinate CAPI calls if you are using a separate thread to make calls to CAPI_GetLastEvent. For example, a developer might have an interrupt timer set up to call CAPI_GetLastEvent every ten seconds using the method described in the CAPI Events section on page 7. If the CAPI_GetLastEvent tries to get an event while the main thread is making a CAPI call (such as CAPI_CreateArray) and gets a CAPI_STATUS_LINK_BUSY, it can sleep for ten seconds and try again. On the other hand, if the main thread is trying to make a CAPI call (such as CAPI_CreateArray) while CAPI_GetLastEvent has the link, then the main thread must retry the command or return an error to the user that the link is busy. As currently implemented in the sample code in the SDK, this paragraph applies to the SCSI LMX, but not the serial LMX. The serial LMX simply drops a second command rather than returning a CAPI_STATUS_LINK_BUSY error, so a timeout and retry are necessary if you are using a serial LMX. See the next paragraph.

As a general precaution against lost messages, you should have a timeout and retry a command if you don't receive a reply in a reasonable period of time. Good values to use for this timeout:

- 15 seconds for these commands: CAPI_SetControllerParams, CAPI_U_SetControllerParams, CAPI_GetDebugData, CAPI_U_GetDebugData.
- 40 seconds for these commands: CAPI_PutOffline, CAPI_U_PutOffline, CAPI_ForceOffline, CAPI_U_ForceOffline.
- 5 seconds for all other commands.

Unified CAPI

Unified CAPI (UCAPI) allows a CAPI application to interface with both controllers in a dual-controller system via an interface to just one of these controllers. In other words, a unified view of a dual-controller system is presented via the API, rather than requiring separate management interfaces to each controller. This simplifies design of CAPI applications. Unified CAPI was introduced with CAPI 3.4. This approach is recommended for CAPI applications being designed for RIO, Stratis RAID S3300 (Project "Rottweiler"), and other dual-controller products that support CAPI 3.4. All of the "non-unified" commands (that is, function calls) defined in this document continue to be supported in CAPI 3.4. For the most part, there is a one-to-one correspondence of the non-unified (CAPI 3.2 and CAPI 3.3) commands with the unified versions. The unified commands follow the non-unified commands in Chapter 5. We recommend that the unified command be used whenever there is one available; that is, avoid mixing unified and non-unified commands in the same application. Note that there is no unified version of the following commands because they only go to the LMX on the host machine; they are not sent to a controller: CAPI_EnablePacketCompression, CAPI_EnablePacketCompressionMasterToSlave, CAPI_RegisterCallback, CAPI_Initialize, CAPI_FindNextController, and CAPI_TimerTick.

Reply to Function Calls

One of the first steps that an application must perform when initializing CAPI is to provide a pointer to its callback function by calling CAPI_RegisterCallback. This requires a pointer to the callback function as a parameter and the function must be declared with the format described below. In the supplied example application, the callback function is myCallBack.

Callback Function

The prototype for the callback function is as follows:

```
void appCallBack( CAPI_REPLY_CODE    replyCode,
                  CAPI_ERROR_CODE    errorCode,
                  CAPI_IDENTIFIER  * identifier,
                  CAPI_U32           param1,
                  CAPI_U32           param2,
                  CAPI_U32           param3,
                  CAPI_U32           param4,
                  void             * pDataPtr );
```

Table 2-1. CAPI callback function parameters

Parameter	Description
replyCode	This specifies the reason for the callback. Every callback is a reply to a previous command from the application. Since all CAPI communications are synchronous (that is, the app must wait for a reply to one command before sending a second command), this member can be ignored if you wish; it just provides a “sanity check” that the reply matches the command.
errorCode	If this value is equal to anything other that CAPI_NO_ERROR, there was an error executing the command. This is the key member of this struct that you should look at for every callback. If the value is anything other that CAPI_NO_ERROR, your application should not attempt to access any data pointed to by pDataPtr.
identifier	This is a pointer to a data structure that provides the handle of the controller that sent this callback (see *Controller Handle* on page 16). This structure may also identify the *arrayIndex* and/or *channelIndex* and/or *driveIndex*, depending on the command (see the comments next to the *identifier* item in the **Callback** table for each command in Chapter 5 to determine which members of this struct are valid for each command). These three indices are not used by Unified CAPI applications; array and drive serial numbers are used instead.
param1	This parameter provides additional information, if available, based on the reply code as specified in the **Callback** table in Chapter 5.
param2	This parameter provides additional information, if available, based on the reply code as specified in the **Callback** table in Chapter 5.
param3	This parameter provides additional information, if available, based on the reply code as specified in the **Callback** table in Chapter 5.
param4	This parameter provides additional information, if available, based on the reply code as specified in the **Callback** table in Chapter 5.
pDataPtr	This parameter points to a data structure. This is used by CAPI commands that return data. See in the CAPI Function Reference, Chapter 5, the *dataPtr* item in the **Callback** table for each function to determine if it returns a data structure. The data structures are defined in the Data Structures chapter. The structure referred to by this pointer is valid only for the duration of the callback. The application should copy the data before returning from the callback function.

CAPI Events

For completion status of lengthy operations, the application must poll for new events by calling CAPI_GetLastEvent. The reply to this call is received by the application callback with replyCode equal to CAPI_REPLY_GET_LAST_EVENT. The parameter, dataPtr, is a pointer of type CAPI_EVENT. The application should check the sequenceNumber of the event to verify if the event is new. If several events have occurred since the last call to CAPI_GetLastEvent (the application should save the sequenceNumber of the last processed event), then the application can make calls to CAPI_GetEvent to fill in the gaps. The application should poll for events at least every ten seconds and, if a new event is discovered, should immediately poll again to expedite the processing of multiple events.

Because there can only be one outstanding CAPI call at a time, the application programmer is responsible for coordinating calls to CAPI_GetLastEvent and to user-initiated CAPI calls.

> ***Note****: Do not assume that the sequence of events is guaranteed. The only exception is operation start events, which precede operation complete events. For example, CAPI_EVENT_CREATE_ARRAY_START will always precede CAPI_EVENT_CREATE_ARRAY_COMPLETE.*

> ***Note:*** *The error code in the event structure contains the information needed to determine if lengthy operations (such as create array) are completed without errors. See CAPI_EVENT for structure details.*

Lengthy Operations

Every CAPI command is followed by a quick reply (within seconds) via the callback function. If a command cannot complete the operation in this amount of time, it is referred to as a lengthy operation. Lengthy operations reply within seconds, but only to communicate that the operation started (such as CAPI_REPLY_CREATE_ARRAY_START). Also, the operation only starts if the error code on the reply is CAPI_NO_ERROR. Operation completion should only be determined via the get event mechanism while the CAPI_GetPercentComplete command can be used to find percent complete. The completion event (such as CAPI_EVENT_CREATE_ARRAY_COMPLETE) error code indicates if the operation was successful. Only one lengthy operation can be performed on an array at a time. But multiple lengthy operations can be performed simultaneously; for example, CAPI_VerifyArray on two different arrays. To allow a CAPI application to associate a command with a completion event, a uniqueId parameter is included with the operation-started message and the same uniqueId is logged with the completion event.

Obtaining Information on the Health of a System Via CAPI

There are two ways of monitoring the health of a system from a CAPI application: by examining the event log and by examining the contents of various controller data structures. In all cases where a fault occurs, information is available through both the event log and the controller structures. For some types of information, no specific event occurs, so there is no event logged, but this information can be obtained from the controller data structures. One example is environmental values (such as voltages and temperatures) that are within normal operating ranges. Another example is drive error statistics; these can be obtained by a call to CAPI_GetDriveErrorStatistics (new with RIO). (In this second example, if the drive error rate exceeds a programmed threshold, an event *will* be logged.)

Event Log: See CAPI_GetEvent, CAPI_GetFirstEvent, and CAPI_GetLastEvent. When a fault occurs, an event will be logged (with the **criticality** member of the CAPI_EVENT struct set to either CAPI_EVENT_CRITICALITY_WARNING or CAPI_EVENT_CRITICALITY_ERROR). If and when a fault is resolved, an event will also be logged (with **criticality** set to CAPI_EVENT_CRITICALITY_INFORMATIONAL) in most *(but not all)* cases. Details for specific types of faults are discussed below. All CAPI event types are defined with #define statements that begin CAPI_EVENT_. A developer of a CAPI application will probably want to examine this list of event types and decide which ones that developer wants to monitor, if any.

Structures: Additional health information is available in various controller structures, as detailed below. A developer of a CAPI application will probably want to study these structures and decide which members of these structures that developer wants to monitor, if any. When event CAPI_EVENT_CONFIGURATION_HAS_CHANGED is logged, this is an indication that this is a good time to examine these structures for health information, but it is more typical to write a CAPI application that checks these structures on a regular basis (such as every 10 seconds) instead of monitoring CAPI_EVENT_CONFIGURATION_HAS_CHANGED.

Here are details on some key types of health information and how to obtain it:

Environmentals:

By struct: Environmentals are available in the CAPI_CONTROLLER_ENVIRONMENTALS struct (and CAPI_ADVANCED_ENVIRONMENTALS on more recent products), which may be obtained by calling CAPI_UpdateController.

By event log: If an environmental value goes out of range, a warning (CAPI_EVENT_AD_WARNING) or error (CAPI_EVENT_AD_FAILURE) event will be logged. If and when it goes back in range, a CAPI_EVENT_AD_OK will be logged.

Array status:

By struct: Overall status for an array is available in the **health** member of the CAPI_ARRAY struct, which may be examined by calling CAPI_GetArrayList.

By event log: For array failures, an event will be logged: CAPI_EVENT_ARRAY_OFFLINE or CAPI_EVENT_ARRAY_CRITICAL. If and when this condition ends, an event will be logged: CAPI_EVENT_RECONSTRUCT_ARRAY_COMPLETE or CAPI_EVENT_VERIFY_ARRAY_COMPLETE.

Channel status:

By struct: Overall status for a host or drive channel is available in the **health** and **healthReason** members of the CAPI_CHANNEL struct. (These are new members beginning with RIO. Prior to RIO, there was no explicit monitoring of channel health.)

By event log: If a disk channel fault is detected, an event will be logged: CAPI_EVENT_DISK_CHANNEL_DEGRADED. There is no event logged when a host channel fails, nor when the health of a channel goes back to normal. However, beginning with RIO, CAPI_EVENT_CONFIGURATION_HAS_CHANGED is logged whenever **health** or **healthReason** changes.

Module status (only for products with replaceable modules – only RIO at this writing):

By struct: For RIO, there are multiple modules in a system, and a failure of any of those modules results in a status change that is available by calling CAPI_UpdateController and examining the moduleStatus member of the struct for each replaceable module. (Search for moduleStatus in capi3.h to understand details of this member.)

By event log: A module failure also results in logging an event, CAPI_EVENT_MODULE_HAS_FAILED. When the module goes online (normally when the bad module is replaced), an event will be logged: CAPI_EVENT_USER_PUT_ONLINE_COMPLETE, CAPI_EVENT_USER_FORCE_ONLINE_COMPLETE,

CAPI_EVENT_SYSTEM_PUT_ONLINE_COMPLETE, or
CAPI_EVENT_SYSTEM_FORCE_ONLINE_COMPLETE.

Board status (pre-RIO active-active RAID systems):

By struct: Board status is available by calling CAPI_UpdateController and examining **failover.failedOver**. If this value is "TRUE" then the other controller has failed over and you can get additional information by examining **failover.failoverReason** and **failover.otherState**.

By event log: One of the following events will indicate a board failure:
CAPI_EVENT_KILL_OTHER_CONTROLLER (this controller has detected a failure in the other controller and has killed it), CAPI_EVENT_SHUTDOWN_CONTROLLER (this controller has shut itself down), or CAPI_EVENT_FAILOVER (the other controller failed and its resources have been transferred to this controller). If and when the other controller goes online (normally when the bad board is replaced), the other controller will resume responsibility for its resources and both this controller and the other controller will log CAPI_EVENT_FAILBACK.

Various other kinds of failures are logged. Some examples:
CAPI_EVENT_BATTERY_FAILURE,
CAPI_EVENT_BATTERY_END_OF_LIFE,
CAPI_EVENT_EMP_FAILURE,
CAPI_EVENT_BUFFER_CORR_ECC_ERR,
CAPI_EVENT_BUFFER_UNCORR_ECC_ERR,
CAPI_EVENT_DISK_DETECTED_ERROR,
CAPI_EVENT_SPARE_DRIVE_FAILURE, etc.

Controller Structure Updates

> ***Note to CAPI 2.x users:*** *To update information in CAPI 3.x may require a call to one or more CAPI functions. Where this was accomplished with one call to CAPI_UpdateController in CAPI2.x now requires calls to CAPI_UpdateController, CAPI_GetArrayList, and/or CAPI_GetDriveList.*
>
> *CAPI_GetConfigSequenceNumber has been added to aid in determining if an update is necessary.*

A CAPI application program is typically designed to set configuration data (also known as parameters) on a controller with the following sequence:

- Get current values of parameters from the controller with CAPI "get" commands. Parameters are returned in C data structures.
- Modify one or more parameters in those structures via a user interface.
- Pass those structures back to the controller with CAPI "set" commands.

CAPI_UpdateController or CAPI_U_GetControllerData should be called to get current information for a controller. CAPI_GetArrayList (or CAPI_U_GetArrayList) and CAPI_GetDriveList (or CAPI_U_GetDriveList) should be called to get current information about associated RAID arrays and drives. CAPI_GetArrayPartitions (or CAPI_U_GetArrayPartitions) and CAPI_GetFreeArrayPartitions (or CAPI_U_GetFreeArrayPartitions) should be called to get current information about the partitions within an array. If a CAPI call is made that can potentially change the configuration of the controller, CAPI verifies that the configuration request is made with up-to-date information. (Discussion of how this is done is in the next section.) If the data is out-of-date, it returns an errorCode of CAPI_ERROR_FAILURE_DUE_TO_CONFIG_CHANGE. A call to CAPI_UpdateController and/or CAPI_GetArrayList and/or CAPI_GetDriveList (or the equivalent Unified CAPI commands) is then necessary before a configuration change can be made.

For a list of which commands require which data to be up-to-date, see the **CSN checking** column in the table in capicmdsup.c. (This column uses values enumerated and explained in capicmdsup.h.)

A controller may also log CAPI_EVENT_CONFIGURATION_HAS_CHANGED when a configuration change occurs on the controller (such as create array start, create array complete, reconstruct complete, and so on). A call to CAPI_UpdateController and/or CAPI_GetArrayList and/or CAPI_GetDriveList (or the equivalent Unified CAPI commands) should then be made to get the latest information.

Typically, a CAPI application includes a process that periodically (for example, once every 10 seconds) updates its copy of controller data. However, since external events (such as Fibre Channel LIP) can cause the configuration to change between updates, a CAPI application should be designed to gracefully handle a CAPI_ERROR_FAILURE_DUE_TO_CONFIG_CHANGE. The simplest approach is probably to report the failure to the user and require the user to re-enter the parameters. This is the approach taken by the Disk Array Administrator (also known as Menu User Interface or MUI). A more sophisticated approach would be to get updated data from the controller and then re-apply the user's changes to the structures and try again.

Controller Configuration Sequence Number

All three data structures (CAPI_CONTROLLER structure, CAPI_ARRAY list, and CAPI_DRIVE list) must be current for a configuration change to take effect, or CAPI will reply with CAPI_ERROR_FAILURE_DUE_TO_CONFIG_CHANGE. To determine this, the controller maintains a configuration sequence number (CSN). A call to CAPI_GetConfigSequenceNumber can be made to determine if the information is up-to-date.

Note that controllers that support active-active (AA) mode maintain unique configuration sequence numbers for each controller. Controller A uses odd numbers and B uses even numbers, even when operated in standalone mode. This method provides a simple way for CAPI applications to detect controller failover: the configuration sequence number will change from odd to even or vice-versa, indicating the application is now communicating with the other controller.

For Unified CAPI commands that return a CSN for both controllers, if the controller that the CAPI app is communicating with is unable to communicate with the other controller then the CSN of the other controller will be returned as 0xFFFFFFFF.

Each of the three main CAPI calls to retrieve data (CAPI_UpdateController, CAPI_GetArrayList, and CAPI_GetDriveList, or the three Unified CAPI equivalents) return the configuration sequence number from the controller. This is an unsigned 32-bit value that the controller increases by a value of one (or two, in products that support AA mode) every time the configuration changes on the controller. The controller only keeps one configuration sequence number (there is not a separate number for the controller structure, the RAID arrays, and drive list). This number is reset to zero (for controller B) or one (for controller A) when the controller reboots.

Param2 in the callback from CAPI_UpdateController, CAPI_GetArrayList, and CAPI_GetDriveList contains the configuration sequence number. The configurationSequenceNumber can also be found as member fields in CAPI_CONTROLLER, CAPI_ARRAY, and CAPI_DRIVE to assist the developer in further verifying that the information is valid.

SDK Code Assists with Current Configuration Information

The low-level code in the CAPI SDK will assist in making sure that configuration changes are not attempted with incorrect or outdated data structures. When a CAPI_UpdateController (or CAPI_GetArrayList or CAPI_GetDriveList) call is made to the controller, the controller will respond with a new CAPI_CONTROLLER (or CAPI_ARRAY list or CAPI_DRIVE list). In addition, the low-level packet header will contain the current configuration sequence number (it is also returned in param2 in the callback for convenience and embedded in the structures). The CAPI SDK code will save this CSN in the CONTROLLER_CONTEXT structure that the application program provided space for in the initialization sequence. This is also done for CAPI_GetArrayList and CAPI_GetDriveList. When a subsequent command is sent to the controller, the CAPI SDK code will copy all three of the configuration sequence numbers it has (one for CAPI_UpdateController, one for CAPI_GetDriveList, and one for CAPI_GetArrayList) from the CONTROLLER_CONTEXT structure and insert them in the packet header (which is a structure of type CAPI_PACKET). If the CSNs are not the same or if the CSNs don't match the current CSN on the controller, a CAPI_ERROR_FAILURE_DUE_TO_CONFIG_CHANGE error will result. The application program should then do an entire update of all three data structures/lists (call CAPI_UpdateController, CAPI_GetDriveList, and CAPI_GetArrayList). (See the table in capicmdsup.c for details of exactly which structs must be up-to-date for each CAPI command.) For Unified CAPI applications, CAPI_U_GetControllerData, CAPI_U_GetArrayList, and CAPI_U_GetDriveList use these same three configuration sequence numbers.

Portability

The CAPI sample code is written in strict ANSI C and the source is provided as part of the CAPI Software Developer's Kit (SDK) so that developers may build the API into any environment that has an ANSI-compatible C compiler. To be able to successfully use the data structures described in this document, the following conditions must be met:

- Your compiler must pack structures to the byte, and not insert any extra alignment padding between fields (**THIS REQUIRES A COMPILER OPTION IN MOST CASES**).
- Your structures must be built in little-endian byte order (least significant bytes at lower addresses.) For machines using big-endian byte ordering, you must change the way CAPI packets are built (in the file capi2pak.c) so that the actual structures sent to the controller are in little-endian byte order. For example, the structure below:

```
typedef struct
{
      CAPI_U8       a;
      CAPI_U16      b;
      CAPI_U32      c;
} TEST;

TEST test;
test.a = 0xAA;
test.b = 0xABCD;
test.c = 0x12345678;
```

  would be represented in memory (low addresses first) as:

```
AA CD AB 78 56 34 12
```

  and sizeof(TEST) is exactly 7 bytes.

SDK Contents

The Chaparral Network Storage CAPI Software Development Kit (SDK) consists of:

- CAPI Functional Specification (this document)
- CAPI Client source code
- CAPI sample application source code

The following files are included in the CAPI Client code:

aspi.c	CAPI Internals. (16-bit SCSI support using ASPI. Not recommended for new development.)
aspidefs.h	CAPI Internals. (16-bit SCSI support using ASPI. Not recommended for new development.)
aspi.h	CAPI Internals. (16-bit SCSI support using ASPI. Not recommended for new development.)
capi.h	Basic CAPI definitions. You may want to customize this for your app. Include in all CAPI apps.
capi_end.h	Endian transformation macros. You may find these useful if you need to do endian conversion.
capi_event_reply.h	CAPI_EVENT_ defines (see Chapter 7) and CAPI_REPLY_ defines (see Chapter 6). Include in all CAPI apps.
capi2.h	Basic CAPI definitions. (DEPRECATED – use capi3.h for new development.)
capi2lmx.c	CAPI Internals. Include in all CAPI apps.
capi2lmx.h	CAPI Internals. Include in all CAPI apps.
capi2pak.c	CAPI Internals. (CAPI functions that are defined in Chapter 5.) Include in all CAPI apps.
capi3.h	Basic CAPI definitions. Include in all CAPI apps.
capicmdsup.c	For reference only. Do not include in CAPI apps.
capicmdsup.h	For reference only. Do not include in CAPI apps.
capicust.h	Customizing file. Customize this file for your environment and include in all CAPI apps.
capipak.h	CAPI Internals. Include in all CAPI apps.
capiu_defs.h	Basic CAPI definitions for Unified CAPI. Include in all Unified CAPI apps.
capiu_v1.h	Basic CAPI definitions for Unified CAPI. Include in all Unified CAPI apps.
commport.c	16-/32-bit Windows RS-232 support
commport.h	16-/32-bit Windows RS-232 support
Devioctl.h	CAPI Internals. (Microsoft header file included in lmx_sc32.c.)
dlm.c	CAPI Internals. (RS-232 protocol)
dlm.h	CAPI Internals. (RS-232 support)
dlmj.c	CAPI Internals. (Simplified RS-232 protocol)
dlmj.h	CAPI Internals. (Simplified RS-232 support)
environ.h	CAPI Internals. (typedefs). Customize this file for your environment and include in all CAPI apps.
lmx.h	CAPI Internals. Include in all CAPI apps.
lmx232.c	CAPI Internals. (RS-232 protocol)
lmx232.h	CAPI Internals. (RS-232 support)
lmx232j.c	CAPI Internals. (Simplified RS-232 protocol)
lmx232j.h	CAPI Internals. (Simplified RS-232 support)
lmxscsi.c	CAPI Internals. (16-bit SCSI using ASPI. Not recommended for new development.)
lmxscsi.h	CAPI Internals. (16-bit SCSI using ASPI. Not recommended for new development.)
lmx_sc32.c	CAPI Internals. (32-bit SCSI for Windows – uses IOCTL, not ASPI. Recommended sample code for in-band LMX.)
lmx_sc32.h	CAPI Internals. (32-bit SCSI for Windows – uses IOCTL, not ASPI. Recommended sample code for in-band LMX.)
mt_call.h	CAPI Internals. (RS-232 support. Included in dlm.h.)
ntdddisk.h	CAPI Internals. (Microsoft header file included in lmx_sc32.c.)

- ntddscsi.h CAPI Internals. (Microsoft header file included in lmx_sc32.c.)
- ntddstore.h CAPI Internals. (Microsoft header file included in ntdddisk.h.)
- scsidefs.h CAPI Internals. (SCSI definitions. Included in several other files.)
- sc_def32.h CAPI Internals. (SCSI definitions.)
- sizetest.c Prints sizes of various CAPI structs. For reference only; not intended to be included as part of CAPI apps.
- wnaspi32.h CAPI Internals. (32-bit SCSI support using ASPI. Not recommended for new development.)
- wnaspi32.lib CAPI Internals. (32-bit SCSI support using ASPI. Not recommended for new development.)

The following files are included in the CAPI sample application, and are provided to demonstrate a CAPI application that is a command-line interface (CLI):

- capicli.c Part of sample command-line interface CAPI app.
- capicli.h
- capimntc.c Sample code to implement call to CAPI_ScsiMaintenance.
- capitest.c Main part of sample command-line interface CAPI app.
- capitest.h
- makefile Borland makefile (to build the serial version).
- *.dsp Microsoft makefiles (project files). You will most likely want to use one of these two project files:
- scsi32.dsp Microsoft project file for Visual C++ 6.0 to build an in-band CAPI app for Windows that will work for parallel SCSI or Fibre Channel (uses DeviceIoControl calls, a.k.a. IOCTL, not ASPI). Although the user interface is not very user-friendly, this project will enable you to quickly generate and try a command-line user interface that communicates over the host SCSI or Fibre Channel connection to your controller.
- rs232.dsp Microsoft project file for Visual C++ 6.0 to build an RS-232 CAPI app for Windows (uses the simplified RS-232 LMX). Although the user interface is not very user-friendly, this project will enable you to quickly generate and try a command-line user interface that communicates over the serial connection to your controller.

Compiler Settings

The following compiler defines must be defined in the developer's build environment:

- Always add CAPI, CAPI_MASTER, and CAPI3.
- Add USE_SERIAL_LMX and DLM (or, for the simplified RS-232 LMX, DLMJ) processor defines for serial port (RS-232) support.
- Add USE_SCSI_LMX processor define for 16-bit ASPI SCSI support. (Not recommended for new development. Use the IOCTL SCSI interface instead.)
- Add USE_SCSI32_LMX processor define for 32-bit IOCTL SCSI support.
- Add USE_CAPI_DLL, CAPI_DLL, and CAPI_WINDOWS_DLL processor defines to make a DLL.
- Add USE_CAPI_DLL and CAPI_WINDOWS_DLL processor defines to make the sample program to communicate with the DLL.
- Add FIRMWARE_DOWNLOAD_ONLY if you want to compile the sample app as a program to just do firmware downloads. NOTE: The code generated with this option will prompt for the controller that you want to download to: A, B, or both. Only the "both" option is supported for in-band CAPI if you are downloading to an active-active system. (This is because the controller has to be in a shutdown state to accept new firmware, but for A-A systems, once you shut down a controller, the host connection to it is lost since the surviving controller assumes the identity of the shut down contoller.)

SDK Notes

- For hints of places in the SDK code that you may wish to customize, search for the word "customize" in the code.

- You must increase MAX_CONTROLLERS in capitest.c (the sample CAPI app) for multiple-controller support.
- When using RS-232 communications, CAPI always returns a handle for the serial port implementation even if a suitable controller is not connected. The connection can be verified by the first command sent.
- No prompt is given for comport speed in the DLL example.
- The sample CAPI application was tested using Borland 16-bit/32-bit compilers and Microsoft Visual C++ 4.1/5.0/6.0 32-bit console application compilers.
- Windows95 caveat: If you are using an RS-232 link and you compiled with a Microsoft Windows compiler (such as Visual C++), WIN32 serial communications are used; otherwise, direct serial UART communications are used (see commport.c). If you are using the direct UART serial comm in a Windows95 DOS system, you might get serial port overruns because it won't be able to poll often enough (also, port will become unusable when you exit).
- If your serial CAPI application leaves the controller's RS-232 port in CAPI mode, you can put the controller back into terminal (MUI) mode by either typing CTRL-P and CTRL-Z, or by rebooting the controller while holding your hand on the spacebar of the terminal and running option 5.
- It has been observed that libraries made with Visual C++ 6.0 are not compatible with Visual C++ 5.0.

Link Manager Exchange

CAPI communicates to different links by means of a software layer referred to as the Link Manager Exchange (LMX). LMX layers are modular and you only need to link in the LMX that you have selected for your CAPI application (a .h file and a .c file). Appropriate compiler flags must also be set for each LMX (see *Compiler Settings*, above). Chaparral provides the following LMXs as examples:

- Windows 32-bit SCSI (see Chapter 11)
- Windows 32-bit RS-232 (see Chapter 12)
- Windows 32-bit RS-232, simplified version (see Chapter 13)

See Chapter 10, *Link Manager Exchange,* on page 343 for information about writing additional custom LMXs, such as UNIX SCSI, Macintosh SCSI, direct UART, RS-232, and so on.

Primitive Data Types

The CAPI must build data packets with specific size fields regardless of the native word size of the configuration processor. The API source uses custom types for primitive data types which have specific sizes of 8, 16, or 32 bits. These typedefs are contained in a separate header file called capicust.h as shown Figure 2-1. This file is included in the SDK and developers must modify these types as appropriate for their target hardware platform.

Figure 2-1. Primitive Typedefs Customized by the Developer

```
typedef char           CAPI_S8;      /* signed byte    - exactly 8 bits  */
typedef unsigned char  CAPI_U8;      /* unsigned byte  - exactly 8 bits  */
typedef short          CAPI_S16;     /* signed word    - exactly 16 bits */
typedef unsigned short CAPI_U16;     /* unsigned word  - exactly 16 bits */
typedef long           CAPI_S32;     /* signed dword   - exactly 32 bits */
typedef unsigned long  CAPI_U32;     /* unsigned dword - exactly 32 bits */
typedef CAPI_U8        CAPI_BOOL;    /* TRUE or FALSE, - exactly 8 bits  */
typedef char           CAPI_CHAR;    /* ASCII character- exactly 8 bits  */
typedef unsigned long  CAPI_TIME;    /* number of seconds since 1/1/1970 */
```

Initialization Details

The recommended calling sequence for initializing the CAPI API includes the following:

1. If using serial port transport, initialize serial port hardware.
2. CAPI_EnablePacketCompression – optional
3. CAPI_EnablePacketCompressionMasterToSlave – optional
4. CAPI_RegisterCallback
5. CAPI_Initialize
6. Wait for initialization complete callback.
7. CAPI_FindNextController

- Continue calling CAPI_FindNextController until *lastTime equals TRUE. If *handle equals CAPI_NULL_ID, then a controller was not found; otherwise, it is a valid handle and you can now make regular API calls. (See below for a discussion of the handle.)
- For each call to CAPI_FindNextController, allocate memory for the controller and pass a pointer to a CAPI_CONTROLLER_CONTEXT. This structure is used by the CAPI internals.
- Allocate a buffer at least as large as the size of a CAPI_RECEIVE_GENERAL_BUFFER_SIZE and pass a pointer in capiBuffer. This is the buffer that CAPI receives data in. You can use the same buffer for all controllers or allocate separate buffers. This buffer is returned as the CAPI general receive buffer.
- Another option is to allocate a separate buffer for receiving events (at least as large as CAPI_RECEIVE_EVENT_BUFFER_SIZE) or pass the same pointer as capiBuffer.

> **Note:** The serial RS-232 version of LMX cannot determine if a controller is attached. The application must determine this by attempting a CAPI API call such as CAPI_UpdateController after CAPI_FindNextController is complete.

> **Note:** The serial RS-232 version currently uses a bi-sync protocol that requires CAPI_Initialize to be recalled if the controller is rebooted (such as in a firmware update procedure).

Controller Handle

When function CAPI_FindNextController finds a controller, it returns a handle of type CAPI_HANDLE. This handle should be viewed as an arbitrary 32-bit number. It must then be passed as a parameter with each call to the CAPI functions defined in Chapter 5. This handle allows your application to tell the LMX which controller in a dual-controller system you want the message to go to. If your CAPI application is designed to manage multiple controllers, then this handle will be used to distinguish between the multiple controllers. Note that for Unified CAPI commands, you should use the handle of the controller that you are communicating with, not the handle of the controller that will implement the command; for those commands that allow an application to specify the controller that will implement the command, that is specified with the *controllerId* parameter on the function call. Normally, for Unified CAPI, you will only be communicating with one of the two controllers in a dual-controller system. If you wish to have your application establish a communications path with both controllers so you can continue managing your system even when there is a failover, you should design your application so that it only uses the second controller for management in the event of a failover.

When your application gets a callback from a controller, the handle that was passed with the command is echoed back as the *identifier.controllerHandle* parameter passed to your callback function. (See *Reply to Function Calls* on page 6.)

When your application gets events from a controller, the handle that was passed with the command is echoed back as the *id.controllerHandle* member of the CAPI_EVENT struct. Note that for CAPI_U_GetFirstEvent, CAPI_U_GetLastEvent, and CAPI_U_GetEvent, the returned handle is the one that you passed with the command, which is not necessarily the handle of the controller that the events came from.

CAPI Timer Tick

The application must call CAPI_TimerTick every ½ second (an interrupt timer can be used for this purpose). This allows the internal LMX layer to time out on link errors. Note that this is not used by all LMXs; this is not required for the SCSI LMX (lmx_sc32.c), but is required for the two serial LMXs (lmx232.c and lmx232j.c).

Finding Controllers Example

After initialization, the application must repetitively call CAPI_FindNextController to obtain handles to connected controllers until CAPI_NULL_ID is returned in the *handle* parameter. The first call needs to pass TRUE in the firstTime parameter; otherwise, it should be FALSE. For this function, the application does not need to wait for the callback function.

The application programmer can store all controller information in their own structure, such as demonstrated in the following example:

```
typedef struct
{
    CAPI_HANDLE             controllerHandle;
    CAPI_CONTROLLER         controller;
    CAPI_CONTROLLER_CONTEXT controllerContext;

    // Define a receive buffer for data from the remote link.
    CAPI_U8  capiBuffer[CAPI_RECEIVE_GENERAL_BUFFER_SIZE];

    // An optional second buffer for receiving CAPI events so as not to
    // disturb the CAPI_CONTROLLER structure in the other buffer.
    CAPI_U8  eventBuffer[CAPI_RECEIVE_EVENT_BUFFER_SIZE];
} RAID_CONTROLLER;

/*==========================================================================*/
void FindControllers( int *numFound, RAID_CONTROLLER *raidControllers )
/*==========================================================================*/
{
    CAPI_BOOL        firstTime;
    CAPI_BOOL        lastcontroller;
    int              I;
    CAPI_RETURN_CODE rc;

    for( I=0; I<MAX_CONTROLLERS; I++ ) // User defined MAX_CONTROLLERS
        raidController[I].controllerHandle = CAPI_NULL_ID;

    I = 0;
    printf( “Searching for external controllers...” );
    firstTime = TRUE;   // Restart search with the first controller.
    lastController = FALSE;

    while( lastController == FALSE && I < MAX_CONTROLLERS )
    {
        rc = CAPI_FindNextController( firstTime, &lastController,
             &raidController[I].controllerHandle,
             &raidController[I].controllerContext,
             raidController[I].capiBuffer,
             raidController[I].eventBuffer );

        firstTime = FALSE;  // Keep working down list of controllers.
        if( rc != CAPI_STATUS_GOOD ) // If the command failed when sending
            break;
        if( raidController[I].controllerHandle != CAPI_NULL_ID )
            I++;
    }
    *numFound = I;
    if( numFound == 0 )
    {
        printf( “Could not find any controllers!\n” );
        exit(0);
    }
    else
    {
        for( I=0; I<numFound; I++ )
            printf( “Found controller %d with Handle %x \n”,
                    I, raidController[I].controllerHandle );
    }
    return;
}
```

◊ ◊ ◊ 3

TYPEDEFS AND DEFINES

The following section lists the typedefs and defines used in the data structures that will be described in chapter 4. Each typedef is followed by the legal values for that type. Typedefs and defines are used instead of enums to better maintain portability between different compilers. If a different compiler is used for a CAPI application on a host computer from what is used for compiling the Chaparral firmware, the handling of enums may be different in the two compilers, whereas defines are more portable.

> **Note:** *This list has not been updated for CAPI 3.3 and CAPI 3.4. We believe that this list is of limited utility to a CAPI application developer since it is not an easily searchable table. We recommend that you use your development environment to search the .h files in the SDK for any #define that you have an interest in.*

```
/*****************************************************************/
/*                 Constants:                                    */
/*****************************************************************/
#define CAPI_HEADER_FILE_REV_CONTROL_VERSION    "$Revision:: 147 $"

#define CAPI_VERSION_MAJOR                      3  /* ie. v3.x */
#define CAPI_VERSION_MINOR                      1

#define CAPI_ENVIRON_MAX_INQUIRY_BYTES          256
#define CAPI_ENVIRON_MAX_SENSE_BYTES            200
#define CAPI_ENVIRON_MAX_ENVIRON_DATA_LENGTH    256
#define CAPI_FC_WWID_SIZE                       8
#define CAPI_HIGHEST_USABLE_UNIT_NUM            63
#define CAPI_INQ_MODEL_LEN                      17
#define CAPI_INQ_REV_LEN                        5
#define CAPI_INQ_VENDOR_LEN                     9
#define CAPI_MAX_ARRAY_NAME                     32
#define CAPI_MAX_BYTES_FOR_EVENT_CDB            16
#define CAPI_MAX_BYTES_FOR_EXTRA_EVENT_DATA     64
#define CAPI_MAX_DEVICES_FC_LOOP                128
#define CAPI_MAX_DIGITAL_KEY_BYTES              16 NEW!
#define CAPI_MAX_DRIVE_CHANNELS_PER_CONTROLLER   8  /* Max back-end channels  */
#define CAPI_MAX_DRIVES_PER_ARRAY               68  /* 64 + 4 dedicated spares */
#define CAPI_MAX_DRIVES_PER_CHANNEL             125
#define CAPI_MAX_DRIVES_PER_CONTROLLER          250
#define CAPI_MAX_ENVIRON_DEVICES                10
#define CAPI_MAX_EXPAND_DRIVES                  4
#define CAPI_MAX_HOST_CHANNELS_PER_CONTROLLER    8  /* Max front-end channels */
#define CAPI_MAX_HOST_NAME                      16
#define CAPI_MAX_HOST_TABLE                     16
#define CAPI_MAX_NETWORK_STRING                 32 NEW!
#define CAPI_MAX_NUMBER_OF_ARRAY_STAT_BUCKETS   12
#define CAPI_MAX_PASSWORD_BYTES                 8
#define CAPI_MAX_POOL_SPARES_PER_CONTROLLER     8
#define CAPI_MAX_RAID_LEVELS                    12
#define CAPI_MAX_SERIAL_NUMBER_BYTES            32
```

```
#define CAPI_MAX_SINGLES_PER_CONTROLLER        10
#define CAPI_MAX_SPARES_PER_ARRAY              4
#define CAPI_MAX_STRING                        20
#define CAPI_MAX_UNIT_NUM                      (CAPI_HIGHEST_USABLE_UNIT_NUM + 1)
#define CAPI_MAX_UNIT_MAP                     128
#define CAPI_SYSTEM_STRING_MAX                 80 NEW!

#define CAPI_MAX_ARRAYS_PER_CONTROLLER         32
/* see maxArrays in CAPI_RAID for actual number supported by the particular RAID
   product. This is the max arrays *per* bank of arrays */
#define CAPI_MAX_PARTITIONS_PER_ARRAY            16
#define CAPI_MAX_ARRAY_PARTITIONS_PER_CONTROLLER  128  /* The max number of LUNs. */

/* This the maximum number of free partition area in an array: */
#define CAPI_MAX_FREE_PARTITIONS_PER_ARRAY (CAPI_MAX_PARTITIONS_PER_ARRAY + 1)

#define CAPI_PERFORMANCE_TUNING_FLAG_DUAL_FIBRE 0x00000001
#define CAPI_RECEIVE_EVENT_BUFFER_SIZE        (sizeof(CAPI_EVENT)+sizeof(CAPI_PACKET))
#define CAPI_RECEIVE_GENERAL_BUFFER_SIZE  (sizeof(CAPI_EXTRA_DATA)+sizeof(CAPI_PACKET))

/* Indicates the target ID is currently set to "soft", meaning that the
 * system chooses the target ID's value dynamically.  Only supported for Fibre
 * Channel target IDs.
 */
#define CAPI_SOFT_TARGET_ID                    0xFF

/*-------------------------------------------------------------------------*/
typedef CAPI_U8                            CAPI_AD_ALARM_SIGNAL;
/*-------------------------------------------------------------------------*/
#define CAPI_AD_ALARM_SIGNAL_UNKNOWN               0
#define CAPI_AD_ALARM_SIGNAL_VCC                   1
#define CAPI_AD_ALARM_SIGNAL_BACK                  2
#define CAPI_AD_ALARM_SIGNAL_V12                   3
#define CAPI_AD_ALARM_SIGNAL_V3                    4
#define CAPI_AD_ALARM_SIGNAL_TEMPERATURE           5
#define CAPI_AD_ALARM_SIGNAL_CPU_TEMPERATURE       6
#define CAPI_AD_ALARM_SIGNAL_TEMPERATURE_2         7
#define CAPI_AD_ALARM_SIGNAL_V25_MAIN              8 NEW!
#define CAPI_AD_ALARM_SIGNAL_V25_LAN               9 NEW!

/*-------------------------------------------------------------------------*/
typedef CAPI_U8                            CAPI_ADDRESSING_METHOD;
/*-------------------------------------------------------------------------*/
#define CAPI_ADDR_MODE_PERIPHERAL_DEVICE     0
#define CAPI_ADDR_MODE_LOGICAL_UNIT          1

/*-------------------------------------------------------------------------*/
typedef CAPI_U8                            CAPI_ARRAY_HEALTH;
/*-------------------------------------------------------------------------*/
#define CAPI_ARRAY_FAULT_TOLERANT                  0
#define CAPI_ARRAY_FAULT_TOLERANT_WITH_DOWN_DRIVES 1  /* RAID10 applicable */
#define CAPI_ARRAY_NOT_FAULT_TOLERANT              2
#define CAPI_ARRAY_OFFLINE                         3

/*-------------------------------------------------------------------------*/
typedef CAPI_U8                            CAPI_BATTERY_STATE;
/*-------------------------------------------------------------------------*/
#define CAPI_BATTERY_STATE_RESET                   0
#define CAPI_BATTERY_STATE_FAST_CHARGE_INITIATED   1
#define CAPI_BATTERY_STATE_FAST_CHARGE_ACTIVE      2
#define CAPI_BATTERY_STATE_FAST_CHARGE_VERIFY      3
#define CAPI_BATTERY_STATE_CHARGER_PENDING         4
#define CAPI_BATTERY_STATE_FULL_CHARGED            5
#define CAPI_BATTERY_STATE_FAILURE                 6
```

```
#define CAPI_BATTERY_STATE_TRICKLE_PENDING         7

/*----------------------------------------------------------------------*/
typedef CAPI_U8                             CAPI_BATTERY_STATUS;
/*----------------------------------------------------------------------*/
#define CAPI_BATTERY_STATUS_GOOD                     0
#define CAPI_BATTERY_STATUS_DEAD                     1
#define CAPI_BATTERY_FAILURE_FAST_CHARGE             2
#define CAPI_BATTERY_FAILURE_OVERCHARGE              3
#define CAPI_BATTERY_FAILURE_OVERCURRENT             4
#define CAPI_BATTERY_FAILURE_CHARGER                 5
#define CAPI_BATTERY_FAILURE_PACK_TEMP_OUT_OF_RANGE   6
#define CAPI_BATTERY_FAILURE_SYSTEM_TEMP_OUT_OF_RANGE 7
#define CAPI_BATTERY_FAILURE_VOLTAGE_OUT_OF_RANGE     8
#define CAPI_BATTERY_FAILURE_UNDER_VOLTAGE           9
#define CAPI_BATTERY_FAILURE_OVER_VOLTAGE            10
#define CAPI_BATTERY_FAILURE_PACK_NOT_INSTALLED      11
#define CAPI_BATTERY_FAILURE_PACK_SHORT_CIRCUIT      12

/*----------------------------------------------------------------------*/
typedef CAPI_U8                             CAPI_BUS_TYPE;
/*----------------------------------------------------------------------*/
#define CAPI_BUS_UNKNOWN                           0
#define CAPI_BUS_SE                                1
#define CAPI_BUS_HVD                               2
#define CAPI_BUS_LVD                               3
#define CAPI_BUS_FC1                               4
#define CAPI_BUS_FC2                               5
#define CAPI_BUS_FC                                6  /* Fibre Channel generic type */
#define CAPI_BUS_SCSI                              7  /* SCSI generic type */

/*----------------------------------------------------------------------*/
typedef CAPI_U8                             CAPI_CHANNEL_STATE;
/*----------------------------------------------------------------------*/
#define CAPI_CHANNEL_ACTIVE                        0
#define CAPI_CHANNEL_PAUSED                        1  /* active, but paused */
#define CAPI_CHANNEL_PASSIVE                       2  /* passive (for A-A)  */

/*----------------------------------------------------------------------*/
typedef CAPI_U8                             CAPI_CHANNEL_TYPE;
/*----------------------------------------------------------------------*/
#define CAPI_CHANNEL_TYPE_HOST                     0
#define CAPI_CHANNEL_TYPE_DRIVE                    1

/*----------------------------------------------------------------------*/
typedef CAPI_U8                             CAPI_CONTROLLER_ID;
/*----------------------------------------------------------------------*/
#define CAPI_CONTROLLER_B                          0
#define CAPI_CONTROLLER_A                          1
#define CAPI_CONTROLLER_BOTH                       2
#define CAPI_CONTROLLER_UNKNOWN                    3

/*----------------------------------------------------------------------*/
typedef CAPI_U8                             CAPI_CONTROLLER_MODE;  NEW!
/*----------------------------------------------------------------------*/
#define CAPI_CONTROLLER_MODE_UNKNOWN               0 /* unknown or invalid mode */
#define CAPI_CONTROLLER_MODE_STANDALONE_SINGLE_PORT 1 /* single controller, */
                                                    /* single host port */
#define CAPI_CONTROLLER_MODE_STANDALONE_DUAL_PORT   2 /* single controller, */
                                                    /* dual host ports */
#define CAPI_CONTROLLER_MODE_AA_SINGLE_PORT        3 /* active/active, single */
                                                    /* host port */
#define CAPI_CONTROLLER_MODE_AA_DUAL_PORT          4 /* active/active, dual */
                                                    /* host ports */
```

```
#define CAPI_CONTROLLER_MODE_ACTPAS_DUAL_PORT       5 /* active/passive, dual */
                                                      /* host ports */
#define CAPI_CONTROLLER_MODE_AA_DUAL_PORT_MULTI_ID  6 /* active/active, dual */
                                                      /* port with multi id */

/*--------------------------------------------------------------------*/
typedef CAPI_U8                             CAPI_DIRECTION;
/*--------------------------------------------------------------------*/
#define CAPI_DIRECTION_NONE                         0
#define CAPI_DIRECTION_IN                           1
#define CAPI_DIRECTION_OUT                          2

/*--------------------------------------------------------------------*/
typedef CAPI_U8                             CAPI_DISK_SETTING;
/*--------------------------------------------------------------------*/
#define CAPI_DISK_SETTING_DONT_TOUCH                0
#define CAPI_DISK_SETTING_ENABLE                    1
#define CAPI_DISK_SETTING_DISABLE                   2

/*--------------------------------------------------------------------*/
typedef CAPI_U8                             CAPI_DRIVE_STATE;
/*--------------------------------------------------------------------*/
#define CAPI_DRIVE_ONLINE                           1
#define CAPI_DRIVE_OFFLINE                          2
#define CAPI_DRIVE_MISSING                          3

/*--------------------------------------------------------------------*/
typedef CAPI_U8                             CAPI_DRIVE_TYPE;
/*--------------------------------------------------------------------*/
#define CAPI_DRIVE_TYPE_DISK                        0
#define CAPI_DRIVE_TYPE_TAPE                        1
#define CAPI_DRIVE_TYPE_PRINTER                     2
#define CAPI_DRIVE_TYPE_PROCESSOR                   3
#define CAPI_DRIVE_TYPE_WRITE_ONCE                  4
#define CAPI_DRIVE_TYPE_CDROM                       5
#define CAPI_DRIVE_TYPE_SCANNER                     6
#define CAPI_DRIVE_TYPE_OPTICAL_MEMORY              7
#define CAPI_DRIVE_TYPE_MEDIUM_CHANGER              8
#define CAPI_DRIVE_TYPE_COMMUNICATIONS              9
#define CAPI_DRIVE_TYPE_GRAPHIC_1                   10
#define CAPI_DRIVE_TYPE_GRAPHIC_2                   11
#define CAPI_DRIVE_TYPE_CONTROLLER                  12
#define CAPI_DRIVE_TYPE_ENCLOSURE                   13
#define CAPI_DRIVE_TYPE_SIMPLIFIED_DISK             14
#define CAPI_DRIVE_TYPE_OPTICAL_CARD                15

/*--------------------------------------------------------------------*/
typedef CAPI_U8                             CAPI_DRIVE_USAGE;
/*--------------------------------------------------------------------*/
#define CAPI_DRIVE_AVAILABLE                        0
#define CAPI_DRIVE_MEMBER_OF_ARRAY                  1
#define CAPI_DRIVE_DEDICATED_SPARE                  2
#define CAPI_DRIVE_POOL_SPARE                       3
#define CAPI_DRIVE_SINGLE                           4
#define CAPI_DRIVE_LEFTOVER                         5

/*--------------------------------------------------------------------*/
typedef CAPI_U8                             CAPI_EVENT_CRITICALITY;
/*--------------------------------------------------------------------*/
#define CAPI_EVENT_CRITICALITY_INFORMATIONAL        0
#define CAPI_EVENT_CRITICALITY_WARNING              1
#define CAPI_EVENT_CRITICALITY_ERROR                2

/*--------------------------------------------------------------------*/
typedef CAPI_U8                             CAPI_EVENT_PROGRESS;
```

```
/*----------------------------------------------------------------------*/
#define CAPI_EVENT_PROGRESS_INITIATED              1
#define CAPI_EVENT_PROGRESS_COMPLETED              2

/*----------------------------------------------------------------------*/
typedef CAPI_U8                             CAPI_FLEX_TYPE;
/*----------------------------------------------------------------------*/
#define CAPI_FLEX_TYPE_SCSI                       0x01
#define CAPI_FLEX_TYPE_FC_LOOP_ID                 0x02
#define CAPI_FLEX_TYPE_FC_ADDR                    0x04
#define CAPI_FLEX_TYPE_FC_WWN_NODE                0x08
#define CAPI_FLEX_TYPE_FC_WWN_PORT                0x10
#define CAPI_FLEX_TYPE_LUN                        0x20
#define CAPI_FLEX_TYPE_BRIDGE_LUN                 0x40
#define CAPI_FLEX_TYPE_ENVIRON_LUN                0x80

/*----------------------------------------------------------------------*/
typedef CAPI_U8                             CAPI_FORMAT_TYPE;
/*----------------------------------------------------------------------*/
#define CAPI_FORMAT_TYPE_NO_FORMAT                  0
#define CAPI_FORMAT_TYPE_ZERO_INIT_ONLY             1
#define CAPI_FORMAT_TYPE_ZERO_AND_LOWLEVEL          2
#define CAPI_FORMAT_TYPE_ONLINE_INIT                3

/*----------------------------------------------------------------------*/
typedef CAPI_U32                            CAPI_HANDLE;
/*----------------------------------------------------------------------*/
#define CAPI_NULL_ID                              0xFFFFFFFF

/* Indicates the LUN's value is currently unassigned (i.e. it's unavailable) */
#define CAPI_LUN_UNASSIGNED                       0xFF

/*----------------------------------------------------------------------*/
typedef CAPI_U8                             CAPI_LINK_SPEED;
/*----------------------------------------------------------------------*/
#define CAPI_LINK_SPEED_1GB                         0
#define CAPI_LINK_SPEED_2GB                         1
#define CAPI_LINK_SPEED_AUTO                        2

/*----------------------------------------------------------------------*/
typedef CAPI_U32                            CAPI_MAINT_COMMAND;
/*----------------------------------------------------------------------*/
#define CAPI_MAINT_USE_CDB                          1
#define CAPI_MAINT_INQUIRY                          2
#define CAPI_MAINT_TUR                              3
#define CAPI_MAINT_FORMAT_UNIT                      4
#define CAPI_MAINT_VERIFY                           5
#define CAPI_MAINT_START_UNIT                       6
#define CAPI_MAINT_STOP_UNIT                        7
#define CAPI_MAINT_SEND_DIAGNOSTIC                  8
#define CAPI_MAINT_MODE_SENSE                       9
#define CAPI_MAINT_MODE_SELECT                      10
#define CAPI_MAINT_CLEAR_METADATA                   11

/*----------------------------------------------------------------------*/
typedef CAPI_U32                             CAPI_MAPPING_MODE;
/*----------------------------------------------------------------------*/
#define CAPI_MAPPING_MODE_AUTO                      0
#define CAPI_MAPPING_MODE_FIXED                     1

/*----------------------------------------------------------------------*/
typedef CAPI_U8                             CAPI_RAID_LEVEL;
/*----------------------------------------------------------------------*/
#define CAPI_RAID0                                  0
#define CAPI_RAID1                                  1
```

```
#define CAPI_RAID2                                      2
#define CAPI_RAID3                                      3
#define CAPI_RAID4                                      4
#define CAPI_RAID5                                      5
#define CAPI_RAID_VOLUME_SET                            6
#define CAPI_RAID30                                     7
#define CAPI_RAID50                                     8
#define CAPI_RAID10                                     10

/*----------------------------------------------------------------------*/
typedef CAPI_U8                                 CAPI_SCAN_SEQUENCE;
/*----------------------------------------------------------------------*/
#define CAPI_SCAN_SEQUENCE_0                            0   /* Channel 0, 1, 2, 3 ... */
#define CAPI_SCAN_SEQUENCE_1                            1   /* Channel 1, 0, 2, 3 ...
   customer special */
#define CAPI_SCAN_SEQUENCE_NONE                      0xFF   /* No channel scan */

/*----------------------------------------------------------------------*/
typedef CAPI_U8 CAPI_SNMP_NOTIFICATION_FILTER; NEW!
/*----------------------------------------------------------------------*/
#define CAPI_SNMP_NOTIFICATION_FILTER_INFO              0 /* give all events     */
#define CAPI_SNMP_NOTIFICATION_FILTER_WARN              1 /* just warn and error */
#define CAPI_SNMP_NOTIFICATION_FILTER_ERR               2 /* just errors         */

/*----------------------------------------------------------------------*/
typedef CAPI_U8                                 CAPI_TOPOLOGY;
/*----------------------------------------------------------------------*/
#define CAPI_TOPOLOGY_LOOP                              0
#define CAPI_TOPOLOGY_POINT_TO_POINT                    1
#define CAPI_TOPOLOGY_AUTO                              2

/*----------------------------------------------------------------------*/
typedef CAPI_U8                                 CAPI_UNIT_TYPE;
/*----------------------------------------------------------------------*/
#define CAPI_UNIT_TYPE_ARRAY_LUN                        1
#define CAPI_UNIT_TYPE_BRIDGE_LUN                       2
#define CAPI_UNIT_TYPE_SAFTE_LUN                        3
#define CAPI_UNIT_TYPE_SES_LUN                          4

/*----------------------------------------------------------------------*/
typedef CAPI_U8                                 CAPI_UTILITY_PRIORITY;
/*----------------------------------------------------------------------*/
#define CAPI_UTILITY_PRIORITY_HIGH                      0
#define CAPI_UTILITY_PRIORITY_MEDIUM                    1
#define CAPI_UTILITY_PRIORITY_LOW                       2

/*----------------------------------------------------------------------*/
typedef CAPI_U8                                 CAPI_UTILITY_RUNNING;
/*----------------------------------------------------------------------*/
#define CAPI_NO_UTILITY_RUNNING                         0
#define CAPI_UTIL_LOW_LEVEL_FORMAT                      1
#define CAPI_UTIL_ZERO_INITIALIZE                       2
#define CAPI_UTIL_RECONSTRUCT                           3
#define CAPI_UTIL_VERIFY                                4
#define CAPI_UTIL_EXPAND                                5

/*----------------------------------------------------------------------*/
typedef CAPI_U8                                 CAPI_WB_CACHE_STATE;
/*----------------------------------------------------------------------*/
#define CAPI_WB_CACHE_STATE_ENABLED                     0
#define CAPI_WB_CACHE_STATE_DISABLED                    1

/*----------------------------------------------------------------------*/
typedef CAPI_U8                                 CAPI_OTHER_STATE;
/*----------------------------------------------------------------------*/
```

```
#define CAPI_OS_ACTIVE_ACTIVE                    0
#define CAPI_OS_DOWN                             1
#define CAPI_OS_NOT_INSTALLED                    2
#define CAPI_OS_UNDEFINED                        3

/*----------------------------------------------------------------------*/
typedef CAPI_U8                            CAPI_FR_FAILOVER_REASON;
/*----------------------------------------------------------------------*/
#define CAPI_FR_NA                               0
#define CAPI_FR_FIRMWARE_INCOMPATIBLE            1
#define CAPI_FR_MODEL_INCOMPATIBLE               2
#define CAPI_FR_HEARTBEAT_LOST                   3
#define CAPI_FR_MSG_TO_OTHER_FAILED              4
#define CAPI_FR_OTHER_NOT_PRESENT                5
#define CAPI_FR_CAPI_REQUESTED                   6
#define CAPI_FR_FOC_REGISTER_ERROR               7
#define CAPI_FR_MEMORY_SIZE_INCOMPATIBLE         8
#define CAPI_FR_BOOT_HANDSHAKE_TIMEOUT           9
#define CAPI_FR_FIRMWARE_UPDATE                  10
#define CAPI_FR_SHUTDOWN                         11
#define CAPI_FR_REBOOTING                        12
#define CAPI_FR_WRITE_UNIQUE_DATA                13
#define CAPI_FR_OTHER_ORPHAN_DIRTY               14
#define CAPI_FR_LOCK_MGR_LOST_COMM               15
#define CAPI_FR_SAME_SERIAL_NUMBER               16
#define CAPI_FR_CPLD_REVISION_MISMATCH           17
#define CAPI_FR_UNKNOWN                          0x7f

/*----------------------------------------------------------------------*/
typedef CAPI_U8                            CAPI_INFOSHIELD_ACCESS;
/*----------------------------------------------------------------------*/
#define CAPI_INFOSHIELD_ACCESS_ALL                     0
#define CAPI_INFOSHIELD_ACCESS_NONE                    1
#define CAPI_INFOSHIELD_ACCESS_INCLUDE_LIST            2
#define CAPI_INFOSHIELD_ACCESS_EXCLUDE_LIST            3

/*----------------------------------------------------------------------*/
typedef CAPI_U8                            CAPI_MIB_PORT_STATE;
/*----------------------------------------------------------------------*/
#define CAPI_MIB_PORT_STATE_UNKNOWN              1
#define CAPI_MIB_PORT_STATE_ONLINE               2
#define CAPI_MIB_PORT_STATE_OFFLINE              3
#define CAPI_MIB_PORT_STATE_BYPASSED             4

/*----------------------------------------------------------------------*/
typedef CAPI_U8                            CAPI_MIB_PORT_STATUS;
/*----------------------------------------------------------------------*/
#define CAPI_MIB_PORT_STATUS_UNKNOWN             1
#define CAPI_MIB_PORT_STATUS_UNUSED              2
#define CAPI_MIB_PORT_STATUS_OK                  3
#define CAPI_MIB_PORT_STATUS_WARNING             4
#define CAPI_MIB_PORT_STATUS_FAILURE             5
#define CAPI_MIB_PORT_STATUS_NOTPARTICIPATING    6
#define CAPI_MIB_PORT_STATUS_INITIALIZING        7
#define CAPI_MIB_PORT_STATUS_BYPASS              8

/*----------------------------------------------------------------------*/
typedef CAPI_U8                            CAPI_SENSOR_STATUS;
/*----------------------------------------------------------------------*/
#define CAPI_SENSOR_STATUS_UNKNOWN               1
#define CAPI_SENSOR_STATUS_OTHER                 2
#define CAPI_SENSOR_STATUS_OK                    3
#define CAPI_SENSOR_STATUS_WARNING               4
#define CAPI_SENSOR_STATUS_FAILED                5
```

```
/*----------------------------------------------------------------------*/
typedef CAPI_U32                            CAPI_ENCLOSURE_FEATURES;  NEW!
/*----------------------------------------------------------------------*/
        /* The internal Fibre Channel internal hubs are disabled. */
#define CAPI_ENCL_DISABLE_FC_INTERNAL_HUBS          1
        /* The internal Fibre Channel internal hubs will be connected */
        /* together into one large loop if a controller fails.        */
#define CAPI_ENCL_ENABLE_CONNECT_INTERNAL_HUBS_ON_FO 2

/*----------------------------------------------------------------------*/
typedef CAPI_U32                            CAPI_ENCLOSURE_CAPABILITY;  NEW!
/*----------------------------------------------------------------------*/
/* If set, this controller resides in a separately supplied */
/* enclosure (as opposed to being a stand-alone product).   */
#define CAPI_ENCL_CAPABILITY_RESIDES_IN_ENCLOSURE           0x0001

/* If set, the internal enclosure internal hubs can be configured */
/* to allow connection directly to the controller's ports.        */
#define CAPI_ENCL_CAPABILITY_CHANGE_INTERNAL_HUBS           0x0002

/* If set, the internal enclosure internal hubs can be configured */
/* to be connected together into one large loop when a            */
/* controller fails.                                              */
#define CAPI_ENCL_CAPABILITY_CONNECT_INTERNAL_HUBS_ON_FO    0x0004

/* The Fibre Channel Port Bypass Circuits in the enclosure can  */
/* be configured to run at either 1Gb/s or 2Gb/s.               */
#define CAPI_ENCL_CAPABILITY_CHANGE_FC_SPEED                0x0008

/*----------------------------------------------------------------------*/
/* Fibre Channel Hardware Version   NEW!                                */
/*----------------------------------------------------------------------*/
#define CAPI_FC_HW_VERSION_EMERALD      0x00000000
#define CAPI_FC_HW_VERSION_RIO          0x00000001

/*----------------------------------------------------------------------*/
/* Commands use in CAPI_EnvironRead and CAPI_EnvironWrite               */
/*----------------------------------------------------------------------*/
typedef enum
{
    SAFTE_READ_ENCLOSURE_CFG_CMD    = 0x00,
    SAFTE_READ_ENCLOSURE_STATUS_CMD = 0x01,
    SAFTE_READ_USAGE_STATS_CMD      = 0x02,
    SAFTE_READ_DEV_INSERTIONS_CMD   = 0x03,
    SAFTE_READ_DEV_SLOT_STATUS_CMD  = 0x04,
    SAFTE_READ_GLOBAL_FLAGS_CMD     = 0x05,
    SAFTE_WRITE_DEV_SLOT_STATUS_CMD = 0x10,
    SAFTE_SET_SCSI_ID_CMD           = 0x11,
    SAFTE_PERFORM_SLOT_OPERATION_CMD= 0x12,
    SAFTE_SET_FAN_SPEED_CMD         = 0x13,
    SAFTE_ACTIVATE_POWER_SUPPLY_CMD = 0x14,
    SAFTE_SEND_GLOBAL_FLAGS_CMD     = 0x15
/*----------------------------------------------------------------------*/
} SAFTE_CMD_ENUM;
/*----------------------------------------------------------------------*/
```

The following typedefs are used in nearly all of the CAPI functions and their associated values are listed in later sections.

```
typedef CAPI_U32                            CAPI_EVENT_CODE;
typedef CAPI_U32                            CAPI_RETURN_CODE;
typedef CAPI_U32                            CAPI_REPLY_CODE;
typedef CAPI_U32                            CAPI_ERROR_CODE;
```

4

DATA STRUCTURES

The API communicates information about the configuration of controllers, channels, RAID arrays, and devices to the application through a set of data structures. These structures are listed in this chapter and may not contain the reserved fields used for byte alignment. The actual structures are in the CAPI SDK files. Byte alignment fields are added to make the data structures portable between different processors and different compilers used for your CAPI application running on a host computer and the Chaparral firmware.

> **Note:** Developers familiar with CAPI2.x should observe that there is now another level of indirection to get to CAPI_DRIVE (physical drive) from CAPI_MEMBER_DRIVE (logical array drive).

> **Note:** See Chapter 2 for important information on how to get these structures and how Configuration Sequence Numbers keep the data current. See, especially, the sections of Chapter 2 titled *Controller Structure Updates, Controller Configuration Sequence Number,* and *SDK code assists with current configuration information*.

The main structure is the CAPI_CONTROLLER (or, for unified commands, the CAPI_UNIFIED_CONTROLLER), which describes a controller. It contains information such as the number of initiator channels (disk) and target channels (host), cache size, firmware revision, current cache parameters, and much more.

CAPI_CHANNEL structures are found in the CAPI_CONTROLLER structure. They describe both front-end (target or host) channels, and back-end (initiator or device) channels. Provisions are made for both traditional SCSI devices and Fibre Channel devices. The CAPI_CHANNEL struct contains information such as the bus type and the current and maximum number of devices attached to the bus. This structure contains an array of bytes called *driveReference.* Each element of this array is an index into the current CAPI_DRIVE list (retrieved with a call to CAPI_GetDriveList or CAPI_U_GetDriveList). CAPI_DRIVE describes each physical storage drive attached to that bus. CAPI_DRIVE provides information such as the vendor and model name, drive type, capacity, SCSI ID, and Logical Unit Number (LUN) for each drive.

CAPI_ARRAY structures (retrieved with a call to CAPI_GetArrayList or CAPI_U_GetArrayList) describe each RAID array on that controller. This structure contains all of the information that describes a RAID array including a list of CAPI_MEMBER_DRIVE structures. This includes CAPI_DRIVE_LOCATION identifiers for each member that identifies which of the drives in the CAPI_DRIVE array are logical members of the array.

These data structures, along with additional structures used in CAPI function calls, are described in detail in the remainder of this chapter.

Note: The CAPI_DRIVE_LOCATION includes a channelIndex field and a driveIndex field. While the channelIndex field is the physical channel number, the driveIndex field is not the physical drive ID (SCSI ID) of the drive, but rather an index value into the channel structure. For example, if you have three drives on channel 0, and one drive on channel 1, their respective CAPI_DRIVE_LOCATION values are (0,0), (0,1), (0,2), and (1,0) regardless of their SCSI ID's.

NEW!

The index into the channel structure is further used to get the driveReference value that corresponds with the location in the drive list retrieved with a call to CAPI_GetDriveList.

NEW! ***in CAPI 3.4:***

Note that as of CAPI 3.4, new, Unified CAPI commands have been added. These commands all take parameters that are array serial numbers instead of array indices and drive serial numbers instead of channel and drive indices. This simplifies design of CAPI applications by eliminating most of the need to be concerned with the drive and array indices.

The Unified CAPI commands make use of newly organized Unified CAPI data structures that divide the data that are gotten with CAPI_U_GetCotrollerData and the parameters that are set with CAPI_U_SetControllerParams into two classes: data/parameters that are common to both controllers in a dual-controller system and data/parameters that can be unique per controller. To understand this organization, see struct CAPI_UNIFIED_CONTROLLER and the substructures that are members of this structure. (See, also, *Unified CAPI* on page 5.)

Controller Structure Diagram

<u>CAPI_ARRAY list</u>
Retrieved with call to
CAPI_GetArrayList

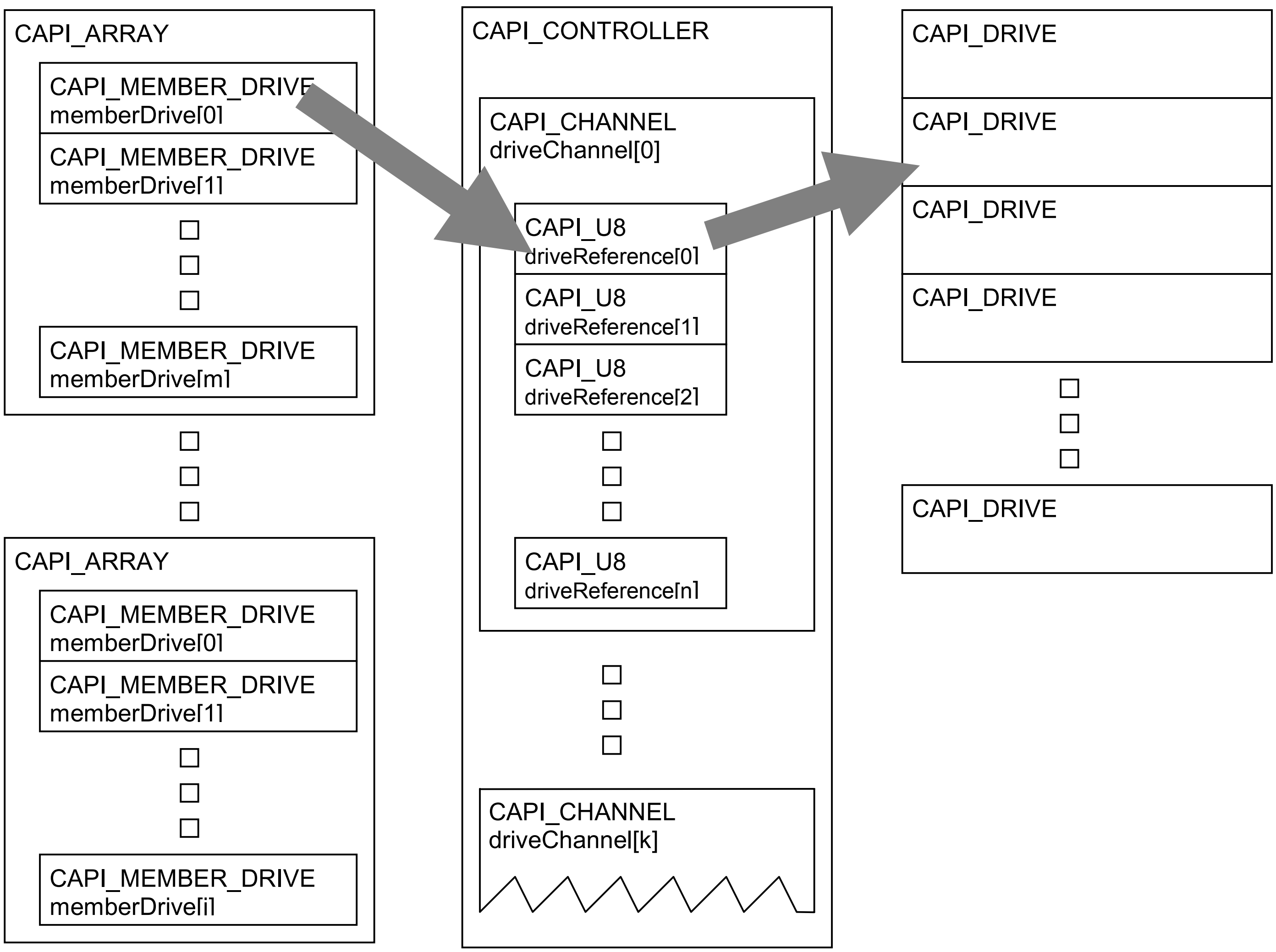
```
callback:
pDataPtr= first CAPI_ARRAY
param1  = number of RAID arrays
param2  = config. seq. number
```

<u>CAPI_CONTROLLER structure</u>
Retrieved with call to
CAPI_UpdateController

```
callback:
pDataPtr = CAPI_CONTROLLER
param1 = undefined
param2 = config. seq. number
```

<u>CAPI_DRIVE list</u>
Retrieved with call to
CAPI_GetDriveList

```
callback:
pDataPtr = first CAPI_DRIVE
param1 = number of drives
param2 = config. seq. number
```



The memberDrive specifies a CAPI_DRIVE_LOCATION, which specifies a channelIndex and a driveIndex.
The channelIndex is used to index into the driveChannel array in the CAPI_CONTROLLER structure.
The driveIndex is used to index into the driveReference array in the particular driveChannel.
The driveReference is used to index into the CAPI_DRIVE list.

CAPI Versions

Different Chaparral products support different versions of CAPI. If the major version (for example, the 3 in version 3.4) is the same for two products, you should be able to easily design a single CAPI app that manages both products, although the product with the lower minor version number (for example, the 4 in version 3.4) will not have all the features of the product with the higher minor version number. Specifically, there may be additional CAPI commands added for a higher version number and there may be additional members in some of the data structures passed to and from CAPI commands, but the members that existed in the lower version of CAPI are still in the same locations in the same structures.

When a new version of the CAPI SDK is released, typically the most recent Chaparral product(s) are released simultaneously with the SDK and the version of CAPI that is running in the controller corresponds to the SDK version. However, developers of new CAPI apps can use a more recent version of the CAPI SDK to talk to older products, provided the major version number matches, and, of course, the app cannot use the newest features that have been added to CAPI which are not supported on that product.

Conversely, if a CAPI app was developed using an older version of the CAPI SDK (for example, CAPI 3.2), it can still be used for managing a newer product that supports a newer version of CAPI (for example, CAPI3.4), but of course the app will not be able to take advantage of the newer features that have been added to CAPI 3.4 in the product but which are not in the CAPI 3.2 SDK.

As of this writing (September 2002), the following products support the corresponding version of CAPI:

- CAPI 2.x: G5312, G7313, all Kxxxx.
- CAPI 3.4: RIO, Stratis RAID S3300 (JFF224). (Thus, the features marked "NEW! in CAPI 3.3" and "NEW! in CAPI 3.4" in this document are supported by these products only.)
- CAPI 3.2: All other Chaparral products.

CAPI Capabilities

Different controller models have different capabilities that are reflected by capability masks found in the CAPI_CONTROLLER structure (obtained with CAPI_UpdateController) and in the CAPI_UNIFIED_CONTROLLER_UNIQUE_DATA structure (obtained with CAPI_U_GetControllerData). Specifically, the members of these structures that indicate the capabilities are *capabilities, capabilities2, capabilities3,* and *enclosureCapabilities*. A TRUE bit indicates the feature is supported. For example, to determine if the controller supports the array statistics feature after receiving an updated controller pointer via the CAPI_UpdateController call and resultant callback, this code fragment could be used:

```
appCallBack( replyCode, errorCode, id, param1, param2, param3, param4, dataPtr )
{
  CAPI_CONTROLLER *pController = (CAPI_CONTROLLER*)dataPtr;
  if (pController->capabilities & CAPI_CAPABILITY_ARRAY_STATS)
  {
      /* array statistics are supported */
  }
  else
  {
      /* array statistics are not supported */
  }
}
```

Some other members of these structures also indicate capabilities that your CAPI app may be interested in.; specifically: *numHostChannels, numDriveChannels, raidCapable,* and *routerCapable*.

You may also wish to look at the revision numbers in these structs, such as *firmwareRevision* and *boardRevision* since different versions of a product may require variations in the user interface. (See the release notes for the product(s) that you are interfacing with.)

You may also have to do different things in your CAPI app if you are designing it to interface with products that support different versions of CAPI. See above under *CAPI Versions* (page 29).

Capabilities supported by some specific products are listed in Chapter 15.

Table 4-1. CAPI Capabilities

Capability Bit	Description
CAPI_CAPABILITY_ARRAY_NAME	The naming of arrays with both the CAPI_CreateArray and the CAPI_ChangeArrayName commands.
CAPI_CAPABILITY_ARRAY_STATS	Array statistics (a limited set defined by product-specific documentation).
CAPI_CAPABILITY_AUTO_VERIFY_FIX	Automatically fixing of parity mismatches during a verify operation. Generally, controllers assume the parity to be wrong and not the data; however, this is product-specific.
CAPI_CAPABILITY_DEDICATED_SPARE	Dedicated spares. That means that the spare drive is available only to an array that it is assigned to.
CAPI_CAPABILITY_DRIVE_STATS	Drive statistics (a limited set defined by product-specific documentation).
CAPI_CAPABILITY_FORMAT_AT_CREATION	Array initialization at array creation time.
CAPI_CAPABILITY_LAST_VERIFY_LAST_RECON	Filling in the time stamps of the last verify and reconstruct operations.
CAPI_CAPABILITY_NO_FORMAT_AT_CREATION	The controller supports the ability to disable array initialization at array creation time.
CAPI_CAPABILITY_NO_VERIFY_FIX_OPTION	The use of the disableAutoFix parameter in the CAPI_VerifyArray command. This ensures that the controller does not fix parity inconstancies if any are found.
CAPI_CAPABILITY_ONLINE_CAPACITY_EXPAND	Online capacity expansion feature.
CAPI_CAPABILITY_RAID0	RAID level 0, as defined by the RAID Advisory Board (RAB).
CAPI_CAPABILITY_RAID1	RAID level 1, as defined by the RAB.
CAPI_CAPABILITY_RAID2	RAID level 2, as defined by the RAB
CAPI_CAPABILITY_RAID3	RAID level 3, as defined by the RAB.
CAPI_CAPABILITY_RAID4	RAID level 4, as defined by the RAB.
CAPI_CAPABILITY_RAID5	RAID level 5, as defined by the RAB.
CAPI_CAPABILITY_RAID10	RAID level 10, as defined by the RAB.
CAPI_CAPABILITY_RAID30	RAID level 30, as defined by the RAB.
CAPI_CAPABILITY_RAID50 NEW!	RAID level 50, as defined by the RAB.
CAPI_CAPABILITY_RAID_VOLUME_SET	Volume sets, which is a concatenation of disks.
CAPI_CAPABILITY_READ_AHEAD_CACHE	Controller read-ahead caching is supported.
CAPI_CAPABILITY_SAFTE	SAF-TE environmental processors.
CAPI_CAPABILITY_SES	SCSI-3 Enclosure Services command set.
CAPI_CAPABILITY_SOFTWARE_TERMINATION	Software SCSI termination settings.
CAPI_CAPABILITY_SPARE_POOL	Pool spares. This means that spare drives are available to any array that needs it, provided the spare drive is large enough.
CAPI_CAPABILITY_WRITE_BACK_CACHE	Controller write-back caching is supported.
CAPI_CAPABILITY_2_2GB_FC_SPEED_SUPPORT NEW!	2 GB Fibre Channel is supported.
CAPI_CAPABILITY_2_ABORT_CREATE_ARRAY	Allows the user to abort an array creation in progress.
CAPI_CAPABILITY_2_ADVANCED_NETWORK_INTF	Internal Chaparral use only.
CAPI_CAPABILITY_2_ADVANCED_UNIT_MAPPING	Ability to use CAPI_SetAdvancedUnitMapping and CAPI_GetAdvancedUnitMapping to map front end channels to back end devices or arrays.
CAPI_CAPABILITY_2_ARRAY_PARTITIONS NEW!	Allows partitioning (i.e. subdividing) the storage with an array into separate "partitions," each of which have their own LUN.
CAPI_CAPABILITY_2_AUTO_FC_SPEED_SUPPORT	Automatic detection of Fibre Channel Speed is supported.
CAPI_CAPABILITY_2_AUTO_FC_TOPOLOGY_SUPPORT NEW!	Auto Fibre Channel Topology detection is supported.
CAPI_CAPABILITY_2_BATTERY_PERCENT_CHARG.ED	The filling in of batteryPercentCharged in the CAPI_CONTROLLER_ENVIRONMENTALS structure.
CAPI_CAPABILITY_2_DEV_MEM_EXPORT_PROTOCOL	Supports configuring the memory buffer space used by the SCSI

Capability Bit	Description
NEW!	Device Memory Export Protocol (DMEP). In a nutshell, DMEP is a SCSI protocol mechanism to support file sharing (i.e. clustering) on a SCSI device by multiple hosts.
CAPI_CAPABILITY_2_DISK_SMART_SUPPORT	Support SMART on the disk channel devices.
CAPI_CAPABILITY_2_DRIVE_SERIAL_NUMBERS	Filling in of unique serial numbers for physical drives.
CAPI_CAPABILITY_2_DYNAMIC_POOL_SPARES NEW!	If enabled, drives marked as “Available” by the controller may be picked up automatically by the controller and used as pool spares, if a critical array needs a spare drive and no dedicated spare or pool spare is available.
CAPI_CAPABILITY_2_FAILOVER_ACTIVE_ACTIVE	The controller is capable of active/active fault-tolerant configuration.
CAPI_CAPABILITY_2_FIRMWARE_DOWNLOAD	CAPI_UpdateFirmware.
CAPI_CAPABILITY_2_HOST_SMART_SUPPORT	Report SMART events to the host.
CAPI_CAPABILITY_2_INFOSHIELD NEW!	Allows access to controller LUNs to be managed using lists of host World Wide Names (WWNs). Only supported on products with Fibre Channel front ends.
CAPI_CAPABILITY_2_MANUAL_RECONSTRUCT	The CAPI_ReconstructArray command. Many controllers do not need to support this command because they allow a reconstruct utility to start automatically if a spare drive is available. To manually start a reconstruct operation on these controllers, you must issue a CAPI_AddDedicatedSpare or CAPI_AddPoolSpare (if supported).
CAPI_CAPABILITY_2_MAP_SINGLE_DEVICE	True pass-through operation to back-end devices without putting metadata on the device.
CAPI_CAPABILITY_2_MULTIPLE_HOST_CHANNELS	Capable of supporting multiple host channels.
CAPI_CAPABILITY_2_MULTIPLE_HOST_ID	Capable of supporting multiple host target ids.
CAPI_CAPABILITY_2_NEW_ARRAY_AVAIL_IMMED	Redundant arrays are available for I/O before init completes
CAPI_CAPABILITY_2_ONLINE_ARRAY_INIT	Controller has online array initialization feature.
CAPI_CAPABILITY_2_PAUSE_INDIVIDUAL_BUS	The ability to pause individual disk buses.
CAPI_CAPABILITY_2_RESCAN_INDIVIDUAL_BUS	The ability to rescan individual disk buses.
CAPI_CAPABILITY_2_SCSI_MAINT_COMMANDS	CAPI_ScsiMaintenance is supported.
CAPI_CAPABILITY_2_SECURITY_LOG_IN_OUT	CAPI_LogIn and CAPI_LogOut are supported.
CAPI_CAPABILITY_2_SOFT_DOWN_DRIVE	Unused
CAPI_CAPABILITY_2_TEST_SPARES	CAPI_TestSpares is supported.
CAPI_CAPABILITY_2_UNIT_AUTO_SETTING NEW!	Controller can automatically assign unit numbers (LUNs).
CAPI_CAPABILITY_3_FC_BACKEND NEW! in CAPI 3.4	Fibre Channel drives.
CAPI_CAPABILITY_3_MASTER_TO_SLAVE_COMPRESSION NEW! in CAPI 3.4	Controller can uncompress CAPI commands received from a CAPI application.
CAPI_CAPABILITY_3_RAID51 NEW! in CAPI 3.4	RAID level 51 as defined by the RAID Advisory Board (RAB).
CAPI_CAPABILITY_3_REPLACEABLE_MODULE NEW! in CAPI 3.4	Controller is used as part of a system with replaceable modules (for example, RIO).
CAPI_CAPABILITY_3_SUPPORT_16_ENVIRON_LUNS NEW! in CAPI 3.4	Controller can handle up to 16 Enclosure Management Processors. If this bit is not set, controller can handle up to 10 Enclosure Management Processors.
CAPI_ENCL_CAPABILITY_RESIDES_IN_ENCLOSURE	If set, this controller resides in a separately supplied enclosure (as opposed to being a stand-alone product).
CAPI_ENCL_CAPABILITY_CHANGE_INTERNAL_HUBS	If set, the internal enclosure internal hubs can be configured to allow connection directly to the controller's ports.
CAPI_ENCL_CAPABILITY_CONNECT_INTERNAL_HUBS_ON_FO	If set, the internal enclosure internal hubs can be configured to be connected together into one large loop when a controller fails.
CAPI_ENCL_CAPABILITY_CHANGE_FC_HOST_SPEED	If set, the Fibre Channel host port circuits in the enclosure can be configured to run at either 1Gb/s or 2Gb/s.
CAPI_ENCL_CAPABILITY_CHANGE_FC_DRIVE_SPEED NEW! in CAPI 3.4	If set, the Fibre Channel drive port circuits in the enclosure can be configured to run at either 1Gb/s or 2Gb/s. (Note that Rottweiler supports drive speed change via a hardware enclosure setting, not via CAPI, so this capability bit is not set for Rottweiler.)

CAPI_ADD_ARRAY_STRUCT

The CAPI_ADD_ARRAY_STRUCT structure is used to pass parameters when creating and expanding arrays. For non-unified commands, this structure is used, but it is hidden from the CAPI application since the members of this struct are loaded from the parameters of the CAPI_CreateArray or CAPI_ExpandArray function. For unified commands, this structure is passed into the CAPI_U_CreateArray and CAPI_U_ExpandArray functions as part of the CAPI_UNIFIED_CREATE_ARRAY_STRUCT structure, which is a parameter to these two functions.

```
typedef struct
{
    CAPI_DRIVE_LOCATION    driveList[CAPI_MAX_DRIVES_PER_ARRAY];
    CAPI_U32               numDrives;
    CAPI_U32               numSpares;
    CAPI_RAID_LEVEL        raidLevel;
    CAPI_U32               minDriveSize;
    CAPI_U32               dataChunkSize;
    CAPI_U32               numDrivesPerLowLevelContainer;
    CAPI_U32               unitNum;
    CAPI_CONTROLLER_ID     preferredOwner;
    CAPI_UTILITY_PRIORITY  priority;
    CAPI_FORMAT_TYPE       formatType;
    CAPI_CHAR              arrayName[CAPI_MAX_ARRAY_NAME];
    CAPI_CACHE_PARAMS      cacheParams;
} CAPI_ADD_ARRAY_STRUCT;
```

Table 4-2. CAPI_ADD_ARRAY_STRUCT fields.

Parameter	Description
driveList[]	List of drives to include in or add to the array. Note that CAPI_MAX_DRIVES_PER_ARRAY includes the dedicated spares. The number of drives in driveList must be *numDrives* + *numSpares*. The first drives in the list must be the drives to use in the array, and the last drives in the list must be the spare drives. This member is *not used* for Unified CAPI commands; instead, the *driveList* member of CAPI_UNIFIED_CREATE_ARRAY_STRUCT is used.
numDrives	Number of member drives to include in or add to the array. Does not include spare drives.
numSpares	Number of spare drives to include or add.
raidLevel	See legal values for CAPI_RAID_LEVEL in capi3.h. (Not used for CAPI_U_ExpandArray.)
minDriveSize	The size of each member in the array, in 512-byte blocks. The size of the smallest drive in the array determines the maximum value for this field, but a smaller value may be used. A value of 0 uses the default (the smallest drive in the array). (Not used for CAPI_U_ExpandArray.)
dataChunkSize	Data chunk size in kilobytes in a RAID 3, 4, or 5 array. (Chunk size is the stripe size on one drive.) Must be one of: 16, 32, 64. (Not used for CAPI_U_ExpandArray.)
numDrivesPerLowLevelContainer	Specifies the number of member drives in the lower-level container. This is only applicable to RAID 30 and RAID 50; a value of 0 can be used for other RAID levels. Lower-level containers are the underlying RAID 5 (for RAID 50) or RAID 3 (for RAID 30) arrays that are striped together to make a two-level RAID 50 or RAID 30 array. All of the lower-level containers within a two-level array must have the same number of drives. (Not used for CAPI_U_ExpandArray.)
unitNum	Logical unit number (LUN). If a valid unused LUN is specified, the array will be created with one partition that uses all of the space

	in the array (this is done for backward compatibility with CAPI applications that don't support array partitions). If CAPI_NULL_ID is specified, then the array will be created without any partitions; to use the free area in the array, partitions must be added using the CAPI_AddArrayPartition or CAPI_U_AddArrayPartition function. (Not used for CAPI_U_ExpandArray.)
preferredOwner	Specifies which controller should be the preferred owner of this array. (Not used for CAPI_U_ExpandArray.)
priority	Specifies the priority that the utility that this struct is passed to should have. (Not used.)
formatType	See under *formatType* in the description of the CAPI_CreateArray function for a list of allowed format types. (Not used for CAPI_U_ExpandArray.)
arrayName	A NULL-terminated string containing the name of the array. If a valid LUN is specified, then the single partition created for the array will have the same name as the array. Although there are CAPI_MAX_ARRAY_NAME characters in this string (32 at this writing), strings longer than CAPI_MAX_STRING (20 at this writing) are truncated. (Not used for CAPI_U_ExpandArray.)
cacheParams	This member is not used. It should be set to all zeros. Use CAPI_SetCacheParams, CAPI_U_SetCacheParams, CAPI_SetArrayPartitionCacheParams, or CAPI_U_SetArrayPartitionCacheParams to set cache parameters.

CAPI_ADVANCED_NETWORK_INTERFACE NEW!

```
typedef struct _CAPI_ADVANCED_NETWORK_INTERFACE
{
    CAPI_NETWORK_INTERFACE  netIf;

    /*
     * VALUES SET BY THE LAN SUBSYSTEM ONLY.
     * These values should not be set by any customer-developed CAPI
     * application.
     *
     * Note that the first 4 members below are common, per-system,
     * and their values can be read by a CAPI application by calling
     * CAPI_GetAdvancedNetworkInterface (or by calling CAPI_U_GetControllerData
     * and then examining the CAPI_NETWORK_INTERFACE_COMMON_DATA struct
     * in the CAPI_UNIFIED_CONTROLLER_COMMON_DATA struct).
     */
    CAPI_U8     snmpVersionMajor;
    CAPI_U8     snmpVersionMinor;
    CAPI_U8     snmpVersionMinorMinor;
    CAPI_U8     snmpVersionChar;

    /*
     * The following members are unique per controller
     * and their values can be read by a CAPI application by calling
     * CAPI_GetAdvancedNetworkInterface (or by calling CAPI_U_GetControllerData
     * and then examining the equivalent values for controllers A and B in
     * the CAPI_NETWORK_INTERFACE_UNIQUE_DATA struct in the
     * CAPI_UNIFIED_CONTROLLER_UNIQUE_DATA struct).
     */
    CAPI_CHAR   firmwareRevisionString[CAPI_MAX_STRING];
    CAPI_CHAR   firmwareBuildTimeDate[CAPI_MAX_NETWORK_STRING]; /* this format:
                                                         Apr  5 2001  13:17:07 */
    CAPI_CHAR   firmwareBaselevel[CAPI_MAX_STRING];
    CAPI_CHAR   lanLoaderRevision[CAPI_MAX_STRING];
    CAPI_U8     fwRevisionMajor;
    CAPI_U8     fwRevisionMinor;
    CAPI_U8     fwRevisionMinMin;

    /* End of values set by the LAN Subsystem only. */


    /*
     *  VALUES THAT MAY BE MODIFIED BY CAPI APPLICATIONS
     *  (using CAPI_SetAdvancedNetworkInterface or CAPI_U_SetControllerParams).
     */
    CAPI_BOOL   snmpTrapsEnable;
    CAPI_IP_ADDRESS_MODE ipAddressMode;
    CAPI_CHAR   snmpWriteCommunity[CAPI_MAX_NETWORK_STRING];
    CAPI_CHAR   snmpReadCommunity[CAPI_MAX_NETWORK_STRING];
    CAPI_SNMP_NOTIFICATION_FILTER  snmpEventFilter;
    CAPI_SNMP_NOTIFICATION_FILTER  snmpTrapFilter;
    CAPI_U32    snmpEventMaxToDisplay;
    CAPI_CHAR   systemName[CAPI_SYSTEM_STRING_MAX];
    CAPI_CHAR   systemContact[CAPI_SYSTEM_STRING_MAX];
    CAPI_CHAR   systemLocation[CAPI_SYSTEM_STRING_MAX];
    CAPI_CHAR   systemInfo[CAPI_SYSTEM_STRING_MAX];
    CAPI_CHAR   ftpUser[CAPI_MAX_NETWORK_STRING];
    CAPI_CHAR   ftpPassword[CAPI_MAX_NETWORK_STRING];
    CAPI_BOOL   ftpFwDownloadDisable;
    CAPI_CHAR   telnetPassword[CAPI_MAX_NETWORK_STRING];
    CAPI_U8     telnetTimeout;
    CAPI_BOOL   telnetDisable;
    CAPI_BOOL   dhcpEnable;
    CAPI_U8     pollInterval;
    CAPI_BOOL   httpDisable;
    CAPI_BOOL   snmpDisable;
```

```
    CAPI_BOOL   debugEnable;

    /* remote notification */
    CAPI_U32    monitoredEvents[CAPI_NUM_MONITORED_EVENTS];
    CAPI_CHAR   email1[CAPI_SYSTEM_STRING_MAX];
    CAPI_CHAR   email2[CAPI_SYSTEM_STRING_MAX];
    CAPI_CHAR   email3[CAPI_SYSTEM_STRING_MAX];
    CAPI_CHAR   email4[CAPI_SYSTEM_STRING_MAX];
    CAPI_CHAR   comment[CAPI_NUM_COMMENT_LINES*CAPI_SYSTEM_STRING_MAX];
    CAPI_U32    pollingPeriod;
    CAPI_U8     numberOfMessagesSentPerEvent;
    CAPI_BOOL   remoteNotificationEnable;
    CAPI_REMOTE_NOTIFICATION_SELECTION  remoteNotificationSelection;
    CAPI_U8     remoteNotificationTimeZone;
    CAPI_CHAR   serverName[CAPI_MAX_NETWORK_STRING];
    CAPI_U32    serverPort;

    /* wbi passwords */
    CAPI_CHAR   wbiMonitorPassword[CAPI_MAX_NETWORK_STRING];
    CAPI_CHAR   wbiManagePassword[CAPI_MAX_NETWORK_STRING];

    /* domain name (for remote notification) */
    CAPI_CHAR   domainName[CAPI_MAX_NETWORK_STRING];
} CAPI_ADVANCED_NETWORK_INTERFACE;
```

Table 4-3. CAPI_ADVANCED_NETWORK_INTERFACE fields.

Parameter	Description
snmpVersionMajor	Reports SNMP version supported by this code (v2.00c). Major is the first char = 2. This field is read only.
snmpVersionMinor	Reports SNMP version supported by this code (v2.00c). Minor is the second char = 0. This field is read only.
snmpVersionMinorMinor	Reports SNMP version supported by this code (v2.00c). MinorMinor is the third char = 0. This field is read only.
snmpVersionChar	Reports SNMP version supported by this code (v2.00c). Char is the fourth char = 'c'. This field is read only.
firmwareRevisionString	Reports the LAN firmware version. Example: "rff288_M311R06". This field is read only.
firmwareBuildTimeDate	Reports the LAN firmware time and Date. Example: "Apr 5 2001 13:17:07". This field is read only.
firmwareBaselevel	Reports the LAN firmware baselevel version. Example: "rff288_M311R06". This will differ from the firmwareRevisionString if the code release is a patch release rather than a general availability release. This field is read only.
lanLoaderRevision	Reports the LAN loader version. Example: "M7012". This field is read only.
fwRevisionMajor	Reports the first char of the revision number. Example: for rff288_M314R06 it is 3. This field is read only.
fwRevisionMinor	Reports the second char of the revision number. Example: for rff288_M314R06 it is 1. This field is read only.
fwRevisionMinMin	Reports the third char of the revision number. Example: for rff288_M314R06 it is 4. This field is read only.
snmpTrapsEnable	The control to allow SNMP traps to be enabled. If SNMP traps are enabled, then the snmpTrapHost should be set to the IP address that traps will be sent to. FALSE = traps disabled. TRUE = traps enabled.
ipAddressMode	Controls the IP address mode to be used. Currently not implemented. Only two-IP-address mode is supported, meaning that each controller has its own unique IP address.
snmpWriteCommunity	SNMP community string for writes. Used as password protection for SNMP sets to the controller. Default is "private".
snmpReadCommunity	SNMP community string for reads. Used as password protection for

	SNMP gets to the controller. Default is “public”.
snmpEventFilter	Controls what events are kept for the SNMP event table. Possible settings are: CAPI_EVENT_CRITICALITY_INFORMATIONAL 0 CAPI_EVENT_CRITICALITY_WARNING 1 CAPI_EVENT_CRITICALITY_ERROR 2 Default is: CAPI_EVENT_CRITICALITY_INFORMATIONAL
snmpTrapFilter	Controls what events will cause traps to be sent. Possible settings are the same as snmpEventFilter. Default is: CAPI_EVENT_CRITICALITY_ERROR
snmpEventMaxToDisplay	Controls number of events to save and display with SNMP. Not implemented.
systemName	Text field used to add description of the system. Also used for SNMP.
systemLocation	Text field used to add location of system. Also used for SNMP.
systemContact	Text field used to add contact for system. Also used for SNMP.
systemInfo	Text field used to add other information for the system. Also used for SNMP.
ftpUser	User name for logging in to FTP. Default user name is “flash”.
ftpPassword	Password for logging into FTP. Default password is “flash”.
ftpFwDownloadDisable	Control to allow or disable FTP firmware download. FALSE is enabled. TRUE is disabled. Default is enabled.
telnetPassword	Password for logging into telnet server. Default is empty string.
telnetTimeout	Timeout in minutes for automatically logging out a user if there has been no activity. Default is 60 minutes. Allowable range is 0-255. 0 means don’t ever timeout.
telnetDisable	Control to disable telnet access. FALSE = enabled. TRUE = disabled. Default is enabled.
dhcpEnable	Control to enable/disable DHCP. Not implemented.
pollInterval	Control to allow setting polling interval for LAN updated. Not implemented.
httpDisable	Control to enable/disable HTTP access. FALSE = enabled. TRUE = disabled. Default is enabled.
snmpDisable	Control to enable/disable SNMP access. FALSE = enabled. TRUE = disabled. Default is enabled.
debugEnable	Control is to enable/disable debug access over Ethernet. TRUE = enabled. FALSE = disabled. Default is disabled.
monitoredEvents	List of monitored events used to cause Remote Notification.
email1	First e-mail address to send Remote Notification to.
email2	Second address to send Remote Notification to.
email3	Third e-mail address to send Remote Notification to.
email4	Fourth e-mail address to send Remote Notification to.
comment	Text field where message may be entered that will be sent with the Remote Notification message.
pollingPeriod	Control to allow changing the polling period for remote notification. Not implemented.
numberOfMessagesSentPerEvent	Control to allow sending multiple messages per Remote Notification event. Not implemented. For a remote notification event, one email is sent per event.
remoteNotificationEnable	Control to enable or disable Remote Notification. TRUE = enabled. FALSE = disabled. Default is disabled.
remoteNotificationSelection	Control to allow Remote Notification on any major category of event. Event categories are: INFORMATIONAL 0x01 WARNING 0x02 ERROR 0x04

	These are bit fields so that any or all may be set simultaneously.
remoteNotificationTimeZone	This is the time zone setting for inclusion with the Remote Notification email. There are 72 possible settings, 0-71.
serverName	Name or IP address of mail server to use for Remote Notification. Example: 172.22.1.31. Note: some systems will not resolve external mail server correctly because there is no authentication.
serverPort	Port number to use for Remote Notification. Not used.
domainName	Domain name to use for Remote Notification. Not used for some hosts (for example, Microsoft Windows).
wbiMonitorPassword	Password used to access the RAIDar Web browser interface for monitor mode. Default password is “monitor”.
wbiManagePassword	Password used to access the RAIDar Web browser interface for monitor/manage mode. Default password is “manage”.

CAPI_ARRAY

The CAPI_ARRAY structure describes one RAID array on an external RAID controller.

```
typedef struct
{
    CAPI_U64                blockCapacity;
    CAPI_U32                unitNum;
    CAPI_U32                numDrives;
    CAPI_U32                numSpares;
    CAPI_U32                dataChunkSize;
    CAPI_U32                minDriveSize;
    CAPI_TIME               creationTimeStamp;
    CAPI_CACHE_PARAMS       cacheParams;
    CAPI_ARRAY_STATS        arrayStats;
    CAPI_MEMBER_DRIVE       memberDrive[CAPI_MAX_DRIVES_PER_ARRAY];
    CAPI_CHAR               name[CAPI_MAX_ARRAY_NAME];
    CAPI_U8                 serialNumber[CAPI_MAX_SERIAL_NUMBER_BYTES];
    CAPI_U8                 serialNumberLength;
    CAPI_ARRAY_HEALTH       health;
    CAPI_UTILITY_RUNNING    utilityRunning;
    CAPI_RAID_LEVEL         raidLevel;
    CAPI_U8                 targetId;
    CAPI_CONTROLLER_ID      preferredOwner;
    CAPI_U8                 containerNumber;
    CAPI_U8                 numArrayPartitions;
    CAPI_U32                configSequenceNumber;
    CAPI_U64                largestFreePartitionSpace;
    CAPI_U8                 numDrivesPerLowLevelArray;
    CAPI_CONTROLLER_ID      currentOwner;
} CAPI_ARRAY;
```

Table 4-4. CAPI_ARRAY fields.

Parameter	Description
blockCapacity	The capacity of the RAID array in 512-byte blocks.
unitNum	Identifies the SCSI LUN that is presented to the host, if the array has one and only one partition that occupies the entire array.
numDrives	The number of member drives in the array excluding spares.
numSpares	The number of spare drives dedicated to this array.
dataChunkSize	The stripe size on one drive in Kbytes for a RAID 0, 10, 3, 4, 5, or 50 array.
minDriveSize	The minimum size drive, in sectors, in this array.
creationTimeStamp	The time the array was created (seconds since 1/1/1970)
cacheParams	The CACHE_PARAMS structure associated with this array. Use CAPI_SetCacheParams to change this. This information is only valid if the array has one and only one partition that occupies the entire array.
arrayStats	The statistics for the array. This information is only valid if the array has one and only one partition that occupies the entire array.
memberDrive	A list of CAPI_MEMBER_DRIVE structures that include member and spare drive CAPI_DRIVE_LOCATION structures.
name	The ASCII character name of the array assigned by the user during array creation (null terminated string).
serialNumber	The serial number for the array that is assigned by the controller during array creation and uniquely identifies the array. Not null terminated.
serialNumberLength	The valid number of serial number bytes.
health	The current state of the array.
utilityRunning	Indicates whether a utility is currently running on the array and if so, which one.
raidLevel	Indicates the type of RAID array.
targetId	Identifies the targetId or FC id presented to the host

preferredOwner	Indicates whether this array prefers to be owned by controller A or B
containerNumber NEW!	For internal use by Chaparral controller software.
numArrayPartitions NEW!	The number of Array Partitions contained in this array.
configSequenceNumber	Identifies the controller configuration sequence number this array information is current for.
largestFreePartitionSpace NEW!	Size of the largest free partition area in the array in (512 byte) logical blocks.
numDrivesPerLowLevelArray NEW!	The number of drives per low level (i.e. subordinate) array in this array. Currently this indicates the number of drives in each subordinate RAID-5 array contained in a RAID-50 array.
currentOwner NEW! in CAPI 3.4	Current owner of the array. One of: CAPI_CONTROLLER_A or CAPI_CONTROLLER_B.

NOTE: The list of drives in the memberDrive field contains array member drives, immediately followed by dedicated spare drives. Spares run the same utilities.

CAPI_ARRAY_PARTITION

The CAPI_ARRAY_PARTITION structure describes one "partition" (or piece) of an array.

```
typedef struct
{
    CAPI_U64              startLba;
    CAPI_U64              sizeLba;
    CAPI_U32              unitNum;
    CAPI_CHAR             name[CAPI_MAX_ARRAY_NAME];
    CAPI_U8               arraySerialNumber[CAPI_MAX_SERIAL_NUMBER_BYTES];
    CAPI_U8               partitionSerialNumber[CAPI_MAX_SERIAL_NUMBER_BYTES];
    CAPI_ARRAY_STATS      stats;
    CAPI_CACHE_PARAMS     cacheParams;
    CAPI_U32              containerOffset;
    CAPI_U32              realContainerNumber;
    CAPI_U8               targetId;
    CAPI_CONTROLLER_ID    preferredOwner;
    CAPI_U8               arrayIndex;
    CAPI_INFOSHIELD_ACCESS  infoShieldAccess;
} CAPI_ARRAY_PARTITION;
```

Table 4-5. CAPI_ARRAY_PARTITION fields.

Parameter	Description
startLba	The starting Logical Block Address (LBA) relative to the start of the array. This value is in 512 byte blocks.
sizeLba	The size of the partition in logical (512 byte) blocks.
unitNum	Identifies the SCSI LUN that is presented to the host for this partition.
name	The ASCII character name of the partition assigned by the user during partition creation (null terminated string).
arraySerialNumber	The serial number for the array to which this partition belongs. Not null terminated.
partitionSerialNumber	The serial number for thepartition that is assigned by the controller during partition creation and uniquely identifies the partition. Not null terminated.
stats	The statistics for the partition.
cacheParams	The CACHE_PARAMS structure associated with this partition. Use CAPI_SetCacheParams to change these settings for all partitions in the array.
containerOffset	For internal use by Chaparral software.
realContainerNumber	For internal use by Chaparral software.
targetId	Identifies the targetId or FC id presented to the host.
preferredOwner	Indicates whether this array prefers to be owned by controller A or B.
arrayIndex	Relative index of array that owns this partition. This value may change as other arrays are added and deleted.
infoShieldAccess	InfoShield access type for the LUN associated with this partition. Valid only for products with a Fibre Channel front-end. CAPI_INFOSHIELD_ACCESS_ALL, CAPI_INFOSHIELD_ACCESS_NONE, CAPI_INFOSHIELD_ACCESS_INCLUDE_LIST, and CAPI_INFOSHIELD_ACCESS_EXCLUDE_LIST are the InfoShield access types.

CAPI_ARRAY_STATS

The CAPI_ARRAY_STATS structure describes the I/O statistics of an array or array partition. It is included as a member of structures CAPI_ARRAY and CAPI_ARRAY_PARTITION.

```
typedef struct
{
    CAPI_U32               numReads;
    CAPI_U32               numWrites;
    CAPI_U32               totalSectorsRead;
    CAPI_U32               totalSectorsWritten;
    CAPI_U32               readBuckets[ CAPI_MAX_NUMBER_OF_ARRAY_STAT_BUCKETS ];
    CAPI_U32               writeBuckets[ CAPI_MAX_NUMBER_OF_ARRAY_STAT_BUCKETS ];
    CAPI_ARRAY_STATS_HOST hostStat[CAPI_MAX_HOST_CHANNELS_PER_CONTROLLER];
} CAPI_ARRAY_STATS;
```

Table 4-6. CAPI_ARRAY_STATS fields.

Parameter	Description
numReads	Specifies the number of host read requests.
numWrites	Specifies the number of host write requests.
totalSectorsRead	Specifies the total number of blocks read on the array/partition.
totalSectorsWritten	Specifies the total number of blocks written to the array/partition.
readBuckets	Provides a histogram of the number of read I/Os of various I/O sizes.
writeBuckets	Provides a histogram of the number of write I/Os of various I/O sizes.
hostStat	Provides detail for the specified host channel.

Note: *Certain RAID controllers may support only a subset of array statistics. There is a capability bit to determine if array statistics are supported. See the controller's documentation to determine which particular parameters are supported.*

The controller keeps track of the number and size of host read and write requests in the readBuckets and writeBuckets fields. The buckets record the following:

bucket 0 – number of 1-sector I/O requests
bucket 1 – number of I/O requests between 2 and 3 sectors, inclusive
bucket 2 – number of I/O requests between 4 and 7 sectors, inclusive
bucket 3 – number of I/O requests between 8 and 15 sectors, inclusive

and so on. Each bucket records a number of I/O requests starting at 2 to the power of the bucket index, up through I/O requests 1 less than the next bucket's starting sector. The last bucket holds all requests >= its starting size.

CAPI_ARRAY_STATS_HOST

The CAPI_ARRAY_STATS_HOST structure describes the I/O statistics on the array/partition for the specified host channel.

```
typedef struct
{
    CAPI_U16            queueDepth;
    CAPI_U32            lastRequestSize;
} CAPI_ARRAY_STATS_HOST;
```

Table 4-7. CAPI_ARRAY_STATS_HOST fields.

Parameter	Description
queueDepth	Specifies the number of active host I/Os for this array.
lastRequestSize	Size in sectors of the last host I/O request to this array. Zero if last host I/O request was a command other than read or write.

CAPI_CACHE_PARAMS

The CAPI_CACHE_PARAMS structure describes the write-back and read look-ahead cache parameters that can be modified per array.

```
typedef struct
{
    CAPI_U32            readAheadSize;
    CAPI_U32            flushPeriod;
    CAPI_BOOL           readAheadEnable;
    CAPI_BOOL           writeBackEnable;
} CAPI_CACHE_PARAMS;
```

Table 4-8. CAPI_CACHE_PARAMS fields.

Parameter	Description
readAheadSize	The amount of additional data, in bytes, that is pre-fetched into cache on read commands.
flushPeriod	The maximum number of milliseconds that data remains in write-back cache before it is written back to the array.
readAheadEnable	Enable read ahead cache. TRUE = enable, FALSE = disable.
writeBackEnable	Enable write-back cache. TRUE = enable, FALSE = disable.

> ***Note:*** *Not all controllers support changing or reporting these settings. Fields will return CAPI_NULL_ID if not supported.*

CAPI_CHANNEL

The CAPI_CHANNEL structure describes a front-end or back-end SCSI or Fibre Channel interface.

> NEW! ***Users of CAPI 2.x should note:***
>
> *CAPI_CHANNEL is used for both front-end (target or host) channels and back-end (initiator or disk) channels. Channel indices will not change except with reboot.*
>
> *CAPI_DRIVEs are not contained in CAPI_CHANNEL, rather an array of bytes (driveReference) is used to index into a drive list retrieved with a call to CAPI_GetDriveList.*
>
> *maxSpeed is no longer a #define but rather a value which is the number of megabytes transferred per second.*

```
typedef struct
{
    CAPI_BOOL                   enabled;
    CAPI_BOOL                   canDisable;
    CAPI_U8                     driveReference[CAPI_MAX_DRIVES_PER_CHANNEL];
    union
    {
        CAPI_SCSI_INFO          scsiInfo;
        CAPI_FC_INFO            fibreInfo;
    } i;
    CAPI_CHANNEL_PARAMS         params;
    CAPI_CHANNEL_PARAMS         currentParams;
    CAPI_BUS_TYPE               busType;
    CAPI_CHANNEL_STATE          state;
    CAPI_U8                     maxDrives;
    CAPI_U8                     numDrives;
    CAPI_U8                     hwChannelNumber;
    CAPI_U8                     hwModuleDisplayNumber;
    CAPI_U8                     hwModuleNumber;
    CAPI_U32                    iosCounter;
    CAPI_U32                    sectorsCounter;
    CAPI_CHANNEL_HEALTH         health;
    CAPI_CHANNEL_HEALTH_REASON  healthReason;
} CAPI_CHANNEL;
```

Table 4-9. CAPI_CHANNEL fields.

Parameter	Description
enabled NEW!	TRUE if channel is enabled. (See also the *disable* member of CAPI_CHANNEL_PARAMS.)
canDisable NEW!	TRUE if channel is capable of being disabled (always TRUE for host channels, FALSE for drive channels). (See also the *disable* member of CAPI_CHANNEL_PARAMS.)
driveReference NEW!	An array of bytes used to index into list of drives retrieved via CAPI_GetDriveList.
scsiInfo NEW!	SCSI-only information pertaining to this channel only.
fibreInfo NEW!	Fibre Channel-only information pertaining to this channel only.
params NEW!	Settings for this channel that can be changed with a call to CAPI_SetChannelParams.
currentParams NEW!	Some settings may require a reboot before becoming effective. This structure is the settings that are in effect now. Use the *params* structure above for read-modify-write calls to CAPI_SetChannelParams, not this structure.

busType	Describes the bus type (type of SCSI or FC interface) for this channel.
state	Specifies if the channel is active or if it is paused for drive insertion/removal.
maxDrives	Specifies the maximum number of drives that can be supported on the bus.
numDrives	Specifies the number of drives currently connected to the bus.
hwChannelNumber NEW!	Specifies the actual controller disk channel number. For example, the CAPI channel index may be zero, but the hardware silk screen may refer to this as channel one. Hardware disk channel numbers do not have to be zero based or contiguous.
hwModuleDisplayNumber NEW! in CAPI 3.4	The hardware module number (silk-screened) for this channel. This is only valid if the hardware supports replaceable modules.
hwModuleNumber NEW! in CAPI 3.4	The internal module number (zero based) for this channel. This is only valid if the hardware supports replaceable modules.
iosCounter NEW! in CAPI 3.4	A CAPI drive I/O statistics counter: Count of I/O commands received by this device.
sectorsCounter NEW! in CAPI 3.4	A CAPI drive I/O statistics counter: Transfer count for this device (in units of 512 bytes).
health NEW! in CAPI 3.4	See the #defines for CAPI_CHANNEL_HEALTH in capi3.h.
healthReason NEW! in CAPI 3.4	See the #defines for CAPI_CHANNEL_HEALTH_REASON in capi3.h.

CAPI_CHANNEL_COMMON_DATA NEW! in CAPI 3.4

This structure is used for Unified CAPI as part of the data that are gotten with CAPI_U_GetControllerData. This struct contains variables that are equivalent to variables with the same names that are in the CAPI_CHANNEL struct that are the same for both controllers. See CAPI_CHANNEL for details of the members of this struct.

```
typedef struct
{
    CAPI_BUS_TYPE           busType;
    CAPI_U8                 maxDrives;
} CAPI_CHANNEL_COMMON_DATA;
```

CAPI_CHANNEL_PARAMS

This structure contains settable channel configuration information for the given channel. This structure is used for both drive and host channels, but not all members are used for both drive and host channels.

Some of the members of this structure are unique per channel and some are common to all channels. The description field in the table below states which members are common and which are unique. This struct is a member of the CAPI_CHANNEL struct. Two arrays of CAPI_CHANNEL structs are included in the CAPI_CONTROLLER struct; one for drive channels and one for host channels. The common members contain the same data in every element of this array. The common members apply to host channels only; no common members are used for drive channels. When using this struct to set channel parameters for drives via the CAPI_SetChannelParams function, these common members are ignored.

(This struct is not used for Unified CAPI commands; the members of this struct are included in separate common and unique parameters structures.)

Note: CAPI 3 originally supported a maximum of 10 Enclosure Management Processors (also referred to as EMPs or environmental processors or environmental units or environmental devices). Then, with CAPI 3.3, we identified a requirement for supporting at least 16 EMPs (needed for the RIO product). The support for 16 EMPs breaks the backward compatibility with applications written for the old (10 EMP) structure. We have added a capability bit to the CAPI_CONTROLLER structure to indicate that the product supports 16 LUNs for the EMPs (CAPI_CAPABILITY_3_SUPPORT_16_ENVIRON_LUNS). Any product that supports 16 EMP LUNs will set this bit. If 16 EMPs are supported, you must use the bitmaps below (environUnitEnableBitmap and environUnitAutoSettingEnableBitmap) to set the EMPs, otherwise only 10 EMPs are supported and you must use the arrays of BOOLs below (environUnitEnable[] and environUnitAutoSettingEnable[]) to set the EMPs.

```
typedef struct
{
    union
    {
        CAPI_SCSI_PARAMS    scsiParams;
        CAPI_FC_PARAMS      fibreParams;
    } p;

    CAPI_U8                 id;
    CAPI_U8                 numIds;

    CAPI_BOOL               capiUnitEnable;
    CAPI_BOOL               capiUnitAutoSettingEnable;
    CAPI_U16                capiUnitNum;

    CAPI_U8                 capiTargetId;
    CAPI_BOOL               disable;

    CAPI_BOOL               environUnitEnable[CAPI_MAX_ENVIRON_DEVICES];
    CAPI_U16                environUnitEnableBitmap;
    CAPI_BOOL               environUnitAutoSettingEnable[CAPI_MAX_ENVIRON_DEVICES];
    CAPI_U16                environUnitAutoSettingEnableBitmap;
    CAPI_U8                 environUnitNum[CAPI_NEW_MAX_ENVIRON_DEVICES];
} CAPI_CHANNEL_PARAMS;
```

Table 4-10. CAPI_CHANNEL_PARAMS fields.

Parameter	Description
scsiParams	Unique. SCSI applicable settable parameters.
fibreParams	Unique. Fibre Channel applicable settable parameters.
id	Unique. Settable ID (SCSI or Fibre Loop): SCSI target ID or Fibre Channel target loop ID for host channels; SCSI initiator ID or Fibre Channel initiator loop ID for disk channels.
numIds	Unique. Number of ids (for channels that support multiple ids). (Used only for reverse router products.)
capiUnitEnable	Common. Enable CAPI LUN.
capiUnitAutoSettingEnable	Common. Enable automatic setting for CAPI LUN.
capiUnitNum	Common. Use to set the LUN which CAPI communicates via SCSI/FC. (Called “CAPI LUN” or “controller LUN” or “bridge LUN” various places in the code and documentation; all these terms are synonyms.)
capiTargetId	Common. Use to set the SCSI ID which CAPI communicates via SCSI/FC. (Used only for reverse router products.)
disable	Unique. Set to TRUE to disable this channel only if *canDisable* in CAPI_CHANNEL is TRUE.
environUnitEnable	Common. Enable EMP pass-through LUN (for SAF-TE or SES). Use this if only 10 EMP LUNs are supported.
environUnitEnableBitmap NEW! in CAPI 3.4	Common. Enable EMP pass-through LUN (for SAF-TE or SES). Use this if 16 EMP LUNs are supported.
environUnitAutoSettingEnable	Common. Enable automatic setting for EMP LUN. Use this if only 10 EMP LUNs are supported.
environUnitAutoSettingEnableBitmap NEW! in CAPI 3.4	Common. Enable automatic setting for EMP LUN. Use this if 16 EMP LUNs are supported.
environUnitNum	Common. EMP LUN number. Only the first 10 elements of this array are valid unless 16 EMP LUNs are supported.

CAPI_CHANNEL_UNIQUE_DATA NEW! in CAPI 3.4

This structure is used for Unified CAPI as part of the data that are gotten with CAPI_U_GetControllerData. This struct contains variables that are equivalent to variables with the same names that are in the CAPI_CHANNEL struct that are different for the two controllers. Exception: *hwChannelDisplayNumber* here is equivalent to *hwChannelNumber* in CAPI_CHANNEL. See CAPI_CHANNEL for details of the members of this struct.

Note that this struct does not contain the "params" or "currentParams" members of CAPI_CHANNEL since they are in the CAPI_UNIFIED_CONTROLLER_UNIQUE_PARAMS struct for Unified CAPI.

```
typedef struct
{
    CAPI_BOOL               enabled;
    CAPI_BOOL               canDisable;
    union
    {
        CAPI_SCSI_INFO          scsiInfo;
        CAPI_FC_INFO            fibreInfo;
    } i;
    CAPI_CHANNEL_STATE      state;
    CAPI_U8                 numDrives;
    CAPI_U8                 hwChannelDisplayNumber;
    CAPI_U8                 hwModuleDisplayNumber;
    CAPI_U8                 hwModuleNumber;
    CAPI_CHANNEL_HEALTH     health;
    CAPI_CHANNEL_HEALTH_REASON  healthReason;
} CAPI_CHANNEL_UNIQUE_DATA;
```

CAPI_CHANNEL_UNIQUE_PARAMS NEW! in CAPI 3.4

This structure is used for Unified CAPI as part of the parameters that are passed with CAPI_U_SetControllerParams. This struct contains variables that are equivalent to variables with the same names that are in the CAPI_CHANNEL_PARAMS struct. However, note that the per-channel parameters are in an array here (see struct CAPI_PER_CHANNEL_PARAMS) and note that the data type used for capiUnitNum and environUnitNum is larger to support possible future expansion.

Note that all CAPI channel parameters are unique per controller, so there is no "CAPI_CHANNEL_COMMON_PARAMS" struct.

```
typedef struct
{
    CAPI_PER_CHANNEL_PARAMS hostPerChannelParams[CAPI_MAX_HOST_CHANNELS_PER_CONTROLLER];
    CAPI_PER_CHANNEL_PARAMS drivePerChannelParams[CAPI_MAX_DRIVE_CHANNELS_PER_CONTROLLER];

    /*
     * The remainder of the members of this structure are common for all
     * host channels owned by a controller board.
     * These members do not apply to drive channels; there are no common
     * parameters for drive channels.
     */

    CAPI_BOOL               capiUnitEnable;
    CAPI_BOOL               capiUnitAutoSettingEnable;
    CAPI_U32                capiUnitNum;  /* This is U8 or U16 in other places,
                                           * but we make it U32 for possible
                                           * future support of max possible
                                           * value. */

    /*
     * Note that environUnitNum uses U32 instead of U8 for possible
     * future support of the maximum possible unit number value.
     */
    CAPI_U16                environUnitEnableBitmap;
    CAPI_U16                environUnitAutoSettingEnableBitmap;
    CAPI_U32                environUnitNum[CAPI_NEW_MAX_ENVIRON_DEVICES];

} CAPI_CHANNEL_UNIQUE_PARAMS;
```

CAPI_CONTROLLER

The CAPI_CONTROLLER describes a Chaparral controller. This is the primary structure used for getting information about a controller. For Unified CAPI commands, this structure has been replaced by CAPI_UNIFIED_CONTROLLER.

> ***Note to CAPI 2.x users:*** *CAPI_CONTROLLER no longer contains CAPI_DRIVE structures. Those are retrieved separately with calls to the CAPI_GetDriveList function. Host channels are now of type CAPI_CHANNEL (same as disk channels). activeEnvironUnitNum and activeCapiUnitNum are contained in CAPI_CHANNEL_PARAMS.*

```
typedef struct
{
    CAPI_U32                configSequenceNumber;
    CAPI_U32                cacheSize;
    CAPI_U32                numHostChannels;
    CAPI_U32                numDriveChannels;
    CAPI_TIME               timeDate;
    CAPI_CAPABILITY         capabilities;
    CAPI_CAPABILITY         capabilities2;
    CAPI_MEMORY             memorySizeSlotA;
    CAPI_MEMORY             memorySizeSlotB;
    CAPI_MEMORY             memorySizeSlotC;
    CAPI_MEMORY             memorySizeSlotD;
    CAPI_CONTROLLER_PARAMS              controllerParams;
    CAPI_CONTROLLER_PARAMS              currentControllerParams;
    CAPI_CONTROLLER_ENVIRONMENTALS      environmentals;
    CAPI_FAILOVER           failover;
    CAPI_CHANNEL            hostChannel[CAPI_MAX_HOST_CHANNELS_PER_CONTROLLER];
    CAPI_CHANNEL            driveChannel[CAPI_MAX_DRIVE_CHANNELS_PER_CONTROLLER];
    CAPI_CHAR               manufacturer[ CAPI_MAX_STRING ];
    CAPI_CHAR               model[ CAPI_MAX_STRING ];
    CAPI_CHAR               firmwareRevision[ CAPI_MAX_STRING ];
    CAPI_CHAR               baselevelRevision[ CAPI_MAX_STRING ];
    CAPI_CHAR               boardRevision[ CAPI_MAX_STRING ];
    CAPI_CHAR               cpldRevision[ CAPI_MAX_STRING ];
    CAPI_CHAR               loaderRevision[ CAPI_MAX_STRING ];
    CAPI_U8                 serialNumber[ CAPI_MAX_SERIAL_NUMBER_BYTES ];
    CAPI_U8                 serialNumberLength;
    CAPI_U32                aaVersion;
    CAPI_U8                 backplaneType;
    CAPI_U8                 daughterBoard0Type;
    CAPI_U8                 daughterBoard1Type;
    CAPI_U8                 linkType;
    CAPI_BOOL               raidCapable;
    CAPI_BOOL               routerCapable;
    CAPI_RAID               raid;
    CAPI_ROUTER             router;
    CAPI_CHAR               cpld2Revision[CAPI_MAX_STRING];
    CAPI_U32                maxDmepMemoryBufferSize;
    CAPI_U32                swFeaturesAllowed;
    CAPI_ENCLOSURE_CAPABILITY  enclosureCapabilities;
    CAPI_CAPABILITY         capabilities3;
    CAPI_PRODUCT_SPECIFIC_UNION  productSpecific;
    CAPI_U8                 currentNodeWWN[CAPI_FC_WWID_SIZE];
    CAPI_U8                 sfpPresent;
    CAPI_U8                 hostRXSignalOK;
    CAPI_U8                 hostTXSignalOK;
} CAPI_CONTROLLER;
```

Table 4-11. CAPI_CONTROLLER fields.

Parameter	Description
configSequenceNumber	
cacheSize	Specifies the controller's cache size in Kbytes.
numHostChannels	Specifies the number of front-end disk channels.
numDriveChannels	Specifies the number of back-end disk channels.
timeDate	Time of creation in time_t format.
capabilities	A bit mask that specifies capabilities of the controller.
capabilities2	An extension of the capabilities field above to allow for more bit masks.
memorySizeSlotA	Specifies the amount of memory in slot A in Mbytes.
memorySizeSlotB	Specifies the amount of memory in slot B in Mbytes.
memorySizeSlotC	Specifies the amount of memory in slot C in Mbytes.
memorySizeSlotD	Specifies the amount of memory in slot D in Mbytes.
controllerParams	Dynamic variables (user settable): Controller parameter information that may have been updated by CAPI, and therefore may not reflect the current, in-use parameters. This is the "pending" configuration. When setting parameters, some parameters go into effect immediately, others do not take effect until a reboot. Once all parameters have gone into effect, controllerParams and currentControllerParams will contain identical data. When a CAPI app sets parameters, it should use this struct as the starting point, then change some of these parameters as desired, then write this struct back to the controller by calling CAPI_SetControllerParams.
currentControllerParams	Controller Parameter information for the currently executing cofiguration. This is sometimes known as the "active" params.
environmentals	Read-only enclosure environmental statistics.
failover	A structure describing details of failover and other controller status.
hostChannel	Front-end host channels on this controller - array indexing # is NOT the actual hardware channel number, use hostChannel[].hwChannelNumber
driveChannel	Back-end drive channels on this controller - array indexing # is NOT the actual hardware channel number, use hostChannel[].hwChannelNumber
manufacturer	Contains the manufacturer as a null terminated ASCII string.
model	Contains the model as a null terminated ASCII string.
firmwareRevision	Contains the firmware revision as a null terminated ASCII string.
baselevelRevision	Contains the base-level revision as a null terminated ASCII string.
boardRevsion	Contains the board revision as a null terminated ASCII string.
cpldRevsion	Contains the cpld revision as a null terminated ASCII string.
loaderRevsion	Contains the loader revision as a null terminated ASCII string.
serialNumber	Contains the serial number as a null terminated ASCII string.
serialNumberLength	Valid number of serial number characters.
aaVersion	Active-Active compatibility version number
backplaneType	Identifies the type of backplane.
daughterBoard0Type	Identifies the type of daughter board 0.
daughterBoard1Type	Identifies the type of daughter board 1.
linkType	Type of the CAPI link
raidCapable	TRUE if capable of performing as a RAID product
routerCapable	TRUE if capable of performing as a ROUTER product
raid	CAPI_RAID information for raid product
router	CAPI_ROUTER information for router product
cpld2Revision	Contains the cpld revision as a null terminated ASCII string.
maxDmepMemoryBuffer Size	Maximum memory buffer size (in bytes) for SCSI Device Memory Export Protocol (DMEP)
swFeaturesAllowed	Software features allowed.
enclosureCapabilities	Capabilities of this enclosure, as determined by the backplane type of the enclosure.
capabilities3 NEW! in CAPI 3.4	An extension of the *capabilities* and *capabilities2* fields above to allow for more bit masks.

productSpecific NEW! in CAPI 3.4	This is a union of structs, where there is one struct defined for each product and this struct contains product-specific information.
currentNodeWWN NEW! in CAPI 3.4	Node WWN for this controller.
sfpPresent NEW! in CAPI 3.4	Host channel SFP is present. A bitmask where each bit is a boolean where 1 = TRUE; bit 0 (the LSB) corresponds to host channel 0 and bit 1 corresponds to host channel 1. Currently used only for Rottweiler.
hostRXSignalOK NEW! in CAPI 3.4	Signal from host is being received OK. A bitmask where each bit is a boolean where 1 = TRUE; bit 0 (the LSB) corresponds to host channel 0 and bit 1 corresponds to host channel 1. Currently used only for Rottweiler.
hostTXSignalOK NEW! in CAPI 3.4	Transmitter for sending signal to host is OK. A bitmask where each bit is a boolean where 1 = TRUE; bit 0 (the LSB) corresponds to host channel 0 and bit 1 corresponds to host channel 1. Currently used only for Rottweiler.

CAPI_CONTROLLER_CONTEXT

CAPI_CONTROLLER_CONTEXT is used by the CAPI internals to keep controller-specific data in the application's data space. The application writer does not need to understand how it is used, but must allocate the space and provide a pointer via CAPI_FindNextController.

```
typedef struct
{
    LMX_CONTEXT            lmxContext;
    LMX_IOB                lmxIob;
    CAPI_BOOL              linkBusy;
    CAPI_U8               *receiveCapiBuffer;
    CAPI_U8               *receiveEventBuffer;
    CAPI_U32               configSequenceNumber;
    CAPI_U8                linkType;
    void                  *firmwareImage;
    CAPI_S32               firmwareRemaining;
    CAPI_U32               firmwareChunkSize;
    CAPI_U32               firmwareIteration;
    CAPI_U32               driveListConfigSequenceNumber;
    CAPI_U32               arrayListConfigSequenceNumber;
} CAPI_CONTROLLER_CONTEXT;
```

Table 4-12. CAPI_CONTROLLER_CONTEXT fields.

Parameter	Description
lmxContext	Used for LMX internals.
lmxIob	Used for LMX internals.
linkBusy	Specifies the link control.
receiveCapiBuffer	Specifies the receive buffer pointer.
receiveEventBuffer	Specifies the receive buffer pointer.
configSequenceNumber	Specifies the last configuration sequence number.
linkType	Specifies the link type.
firmwareImage	This void pointer points to the beginning of a buffer containing the firmware image to be downloaded.
firmwareRemaining	Tells how much of the buffer remains to be transferred.
firmwareChunkSize	This is the size of the chunk of the image file sent to the controller during each data phase.
firmwareIteration	Used to keep track of firmware chunks when downloading code.
driveListConfigSequenceNumber	Specifies the last configuration sequence number.
arrayListConfigSequenceNumber	Specifies the last configuration sequence number.

CAPI_CONTROLLER_ENVIRONMENTALS

This structure contains environmental information for the given controller.

```
typedef struct
{
    CAPI_U32              gpioBits;
    CAPI_U16              voltageA;
    CAPI_U16              voltageB;
    CAPI_U16              voltageC;
    CAPI_U16              voltageD;
    CAPI_U16              voltageE;
    CAPI_S16              temperatureA;
    CAPI_S16              temperatureB;
    CAPI_S16              temperatureC;
    CAPI_S16              temperatureD;
    CAPI_S16              temperatureE;
    CAPI_U16              batteryVoltage;
    CAPI_BATTERY_STATUS   batteryStatus;
    CAPI_BATTERY_STATE    batteryState;
    CAPI_WB_CACHE_STATE   wbCacheState;
    CAPI_U8               batteryMonthsOld;
    CAPI_SENSOR_STATUS    voltageAstatus;
    CAPI_SENSOR_STATUS    voltageBstatus;
    CAPI_SENSOR_STATUS    voltageCstatus;
    CAPI_SENSOR_STATUS    voltageDstatus;
    CAPI_SENSOR_STATUS    voltageEstatus;
    CAPI_SENSOR_STATUS    temperatureAstatus;
    CAPI_SENSOR_STATUS    temperatureBstatus;
    CAPI_SENSOR_STATUS    temperatureCstatus;
    CAPI_SENSOR_STATUS    temperatureDstatus;
    CAPI_SENSOR_STATUS    temperatureEstatus;
    CAPI_SENSOR_STATUS    batteryVoltageStatus;
} CAPI_CONTROLLER_ENVIRONMENTALS;
```

Table 4-13. CAPI_CONTROLLER_ENVIRONMENTALS fields.

Parameter	Description
gpioBits	State of the back-plane's general purpose I/O bits.
voltageA,	System voltages, in tenths. (e.g., 105 equals 10.5 volts).
voltageB,	same
voltageC,	same
voltageD,	same
voltageE	same
temperatureA,	System temperatures, in tenths, Celsius. (e.g., 300 equals 30 degrees).
temperatureB,	same
temperatureC,	same
temperatureD,	same
temperatureE	same
batteryVoltage	On-board battery voltage, in hundredths.
batteryStatus	Status of an on-board battery.
batteryState	State of the on-board battery.
wbCacheState	Specifies if write-back caching is enabled.
batteryMonthsOld	Age of battery in months.
voltageAstatus	sensor status of type CAPI_SENSOR_STATUS
voltageBstatus	sensor status of type CAPI_SENSOR_STATUS
voltageCstatus	sensor status of type CAPI_SENSOR_STATUS
voltageDstatus	sensor status of type CAPI_SENSOR_STATUS
voltageEstatus	sensor status of type CAPI_SENSOR_STATUS
temperatureAstatus	sensor status of type CAPI_SENSOR_STATUS

temperatureBstatus	sensor status of type CAPI_SENSOR_STATUS
temperatureCstatus	sensor status of type CAPI_SENSOR_STATUS
temperatureDstatus	sensor status of type CAPI_SENSOR_STATUS
temperatureEstatus	sensor status of type CAPI_SENSOR_STATUS
batteryVoltageStatus	sensor status of type CAPI_SENSOR_STATUS

CAPI_CONTROLLER_PARAMS

This structure contains the user-settable parameters for a controller. To set new parameters, the application should read the values using CAPI_UpdateController and write new values using CAPI_SetControllerParams.

```
typedef struct
{
    CAPI_U32               environPollInterval;
    CAPI_U32               performanceTuningFlags;
    CAPI_BOOL              externalTargetIdControl;
    CAPI_BOOL              environTemperatureEnable;
    CAPI_BOOL              environAutoSlotFlags;
    CAPI_BOOL              environAutoGlobalFlags;
    CAPI_BOOL              alarmMute;
    CAPI_BOOL              disableBatteryOption;
    CAPI_UTILITY_PRIORITY  utilityPriority;
    CAPI_DISK_SETTING      driveWriteBackCache;
    CAPI_DISK_SETTING      driveSMART;
    CAPI_BOOL              standAlone;
    CAPI_BOOL              dualPort;
    CAPI_BOOL              cacheLock;
    CAPI_BOOL              routerEnable;
    CAPI_BOOL              raidEnable;
    CAPI_CONTROLLER_MODE   controllerMode;
    CAPI_CONTROLLER_RAID_PARAMS     cpRaid;
    CAPI_CONTROLLER_ROUTER_PARAMS   cpRouter;
    CAPI_NETWORK_INTERFACE          net;
    CAPI_U32               debugLogConfig;
    CAPI_U32               dmepMemoryBufferSize;
    CAPI_U32               swFeaturesDisabled;
    CAPI_ENCLOSURE_FEATURES   enclosureFeatureFlags;
    CAPI_FULL_POPULATED_CONFIG  fullPopConfig;
} CAPI_CONTROLLER_PARAMS;
```

Table 4-14. CAPI_CONTROLLER_PARAMS fields.

Parameter	Description
environPollInterval	Displays the environmental processor (also known as Enclosure Management Processor or EMP – SAF-TE "SEP" or SES "ESP") polling intervals in seconds.
performanceTuningFlags	Product specific.
externalTargetIdControl	Set to TRUE if the external enclosure is providing the SCSI target ID.
environTemperatureEnable	Insert the controller's temperature in the environmental package.
environAutoSlotFlags	Allows Enclosure Management Processor (EMP – SAF-TE "SEP" or SES "ESP") to set slot flags.
environAutoGlobalFlags	Allows Enclosure Management Processor (EMP – SAF-TE "SEP" or SES "ESP") to set global flags.
alarmMute	Enable/disable the controller's onboard alarm.
disableBatteryOption	If TRUE, ignore condition of battery and run in write-back mode.
utilityPriority	Priority of all utilities on this controller.
driveWriteBackCache	Indicates the global disk drive write-back cache setting.
driveSMART	Indicates the global disk drive SMART setting.
standAlone	Set to TRUE if this is a stand-alone controller. Ignored for "CAPI_COMMAND_SET_CONTROLLER_PARAMS" unless "controllerMode" is set to "CAPI_CONTROLLER_MODE_UNKNOWN".
dualPort	Set to TRUE is this controller supports dual host ports. Ignored for "CAPI_COMMAND_SET_CONTROLLER_PARAMS" unless "controllerMode" is set to "CAPI_CONTROLLER_MODE_UNKNOWN".
cacheLock	Set to TRUE to lock "mode page Write-Back" alteration for all devices.

routerEnable	Set to TRUE to enable ROUTER functionality (may not be supported).
raidEnable	Set to TRUE to enable RAID functionality (may not be supported).
controllerMode NEW!	Controller operating mode (see typedefs).
cpRaid	Settable parameters applicable to RAID products.
cpRouter	settable parameters applicable to ROUTER products
net	settable parameters applicable to products with LAN connectivity
debugLogConfig	internal use only
dmepMemoryBufferSize	Memory buffer size (in bytes) for SCSI Device Memory Export Protocol (DMEP)
swFeaturesDisabled	Software features disabled.
enclosureFeatureFlags	Current controller enclosure features that can be enabled or disabled.
fullPopConfig NEW! in CAPI 3.4	Used to indicate the configuration of systems that have different configuration options. For RIO, this is used to indicate if a fully configured system has 2 or 4 Data Gates.

CAPI_CONTROLLER_RAID_PARAMS

RAID product settable parameters.

```
typedef struct
{
    CAPI_U32            createArrayBackoffPercent;
    CAPI_U8             dynamicSpare;
} CAPI_CONTROLLER_RAID_PARAMS;
```

Table 4-15. CAPI_CONTROLLER_RAID_PARAMS fields.

Parameter	Description
createArrayBackoffPercent	The controller can support the ability to make RAID arrays smaller than the full size of the member drives. This value is the percentage (in tenths of percentage) the array should be made smaller. When creating RAID arrays, the application can apply this percentage when filling in the minDriveSize parameter in the CAPI_CreateArray call. The RAID controller doesn't actually apply the percentage for the application, it just remembers the percentage amount.
dynamicSpare	0 = off, else = interval (min.) between rescans

CAPI_CONTROLLER_ROUTER_PARAMS

Router product settable parameters.

```
typedef struct
{
    CAPI_MAPPING_MODE       mappingMode;
    CAPI_BOOL               scanSequenceValid;
    CAPI_ADDRESSING_METHOD  addressingMethod;
    CAPI_U8                 scanDelay;
    CAPI_U8                 scanSequence [CAPI_MAX_DRIVE_CHANNELS_PER_CONTROLLER];
} CAPI_CONTROLLER_ROUTER_PARAMS;
```

Table 4-16. CAPI_CONTROLLER_ROUTER_PARAMS fields.

Parameter	Description
mappingMode	Front-end LUN mapping mode
scanSequenceValid	Set to TRUE if *scanSequence* is valid.
addressingMethod	0 = peripheral device, 1 = logical unit
scanDelay	Backend Channel scan delay
scanSequence	Back-end channel scan sequence or order.

CAPI_DRIVE

This structure contains the physical drive description.

```
typedef struct
{
    CAPI_U32              blockCapacity;
    CAPI_U8               serialNumber[ CAPI_MAX_SERIAL_NUMBER_BYTES ];
    CAPI_U8               serialNumberLength;
    CAPI_CHAR             vendor[ CAPI_INQ_VENDOR_LEN ];
    CAPI_CHAR             model[ CAPI_INQ_MODEL_LEN ];
    CAPI_CHAR             revision[ CAPI_INQ_MODEL_LEN ];
    CAPI_DRIVE_USAGE      howUsed;
    CAPI_U8               channel;
    CAPI_U8               secondaryChannel;
    CAPI_U8               containerIndex;
    CAPI_U8               memberIndex;
    CAPI_DRIVE_TYPE       driveType;
    CAPI_UTILITY_RUNNING  utilityRunning;
    CAPI_BOOL             blinking;
    CAPI_U16              busSpeed;
    CAPI_U8               scsiId;
    CAPI_U8               lun;
    CAPI_BOOL             smartCapable;
    CAPI_BOOL             dualPorted;      /* TRUE if dual ported       */
    CAPI_BOOL             seeErrorStats;   /* drive error stats contains interesting
    data */
    CAPI_U8               fcControlBits;   /* FC only: byte3 of FC interface mode
    page (19h) */
    CAPI_FLEX_ID          Fcid1;
    CAPI_FLEX_ID          Fcid2;
    CAPI_U32              configSequenceNumber;
    CAPI_U32              iosCounter;
    CAPI_U32              sectorsCounter;
    CAPI_CONTROLLER_ID    currentOwner;
    CAPI_U8               driveIndex;
} CAPI_DRIVE;
```

Table 4-17. CAPI_DRIVE fields.

Parameter	Description
blockCapacity	Contains the size of the drive, in 512-byte blocks, 2 terabyte maximum.
serialNumber	ASCII drive serial number (null-terminated string).
serialNumberLength	Valid number of serial number bytes.
vendor	Contains the drive's vendor name as a null-terminated ASCII string.
model	Contains the drive's model name as a null-terminated ASCII string.
revision	Contains the drive's firmware revision as a null-terminated ASCII string.
howUsed	Set to usage type (available, member of array, pool spare, etc.)
channel	Identifies the channel on the controller where the drive resides.
secondaryChannel NEW!	Fibre Channel channelIndex (for dual ported device)
containerIndex	Identifies the array index if the drive is a member of the array.
memberIndex	Identifies the member index if the drive is a member of the array.
driveType	Specifies the SCSI device type.
utiliityRunning	Identifies the utility running on the array.
blinking NEW! in CAPI 3.4	Drive is blinking because CAPI_BlinkDrive or CAPI_U_BlinkDrive has been called.
busSpeed NEW!	Speed of transfers in MB/Sec. (1MB = 1,000,000 bytes)
scsiId	Contains the SCSI ID of the drive.
lun	Contains the SCSI LUN of the drive.
smartCapable	Identifies if the drive supports SMART (Self Monitoring and Reporting Technology).

dualPorted NEW! in CAPI 3.4	TRUE if drive is dual-ported, else FALSE. If drive is capable of being connected as dual-ported but only one port is connected, this will be FALSE.
seeErrorStats NEW! in CAPI 3.4	Drive error stats contain interesting data. (Get this data by calling CAPI_GetDriveErrorStats or CAPI_U_GetDriveErrorStats.)
fcControlBits NEW! in CAPI 3.4	Fibre Channel only: byte 3 of FC interface mode page (19h).
Fcid1 NEW!	Fibre Channel id
Fcid2 NEW!	Fibre Channel id (for dual-ported device)
configSequenceNumber NEW!	The controller configuration sequence number that this drive information is current for.
iosCounter NEW! in CAPI 3.4	A CAPI drive I/O statistics counter: Count of I/O commands received by this device.
sectorsCounter NEW! in CAPI 3.4	A CAPI drive I/O statistics counter: Transfer count for this device (in units of 512 bytes).
currentOwner NEW! in CAPI 3.4	One of: CAPI_CONTROLLER_A, CAPI_CONTROLLER_B.
driveIndex NEW! in CAPI 3.4	Identifies the drive number within the channel where the drive resides. In other words, this is equivalent to an index into the driveReference array in the driveChannel array in the CAPI_CONTROLLER struct. (See also the channel member, above.) For Unified CAPI applications, there should be no need to use this member since drives are always referenced by serial number.

CAPI_DRIVE_ERROR_STATS NEW! in CAPI 3.3

This structure contains error statistics for a single drive and is used by CAPI_GetDriveErrorStats and CAPI_U_GetDriveErrorStats.

```
typedef struct
{
    struct
    {
        union
        {
            CAPI_FC_DRIVE_ERRORS     fc;
            CAPI_SCSI_DRIVE_ERRORS   scsi;
        } u;
        CAPI_U32    smartEventCount;
        CAPI_U32    ioTimeoutCount;
        CAPI_U32    noResponseCount;
    } port[2];
} CAPI_DRIVE_ERROR_STATS;
```

Table 4-18. CAP_DRIVE_ERROR_STATS fields.

Parameter	Description
fc	See definition of struct CAPI_FC_DRIVE_ERRORS.
scsi	Currently there are no members defined in struct CAPI_SCSI_DRIVE_ERRORS.
smartEventCount	The number of SMART events the drive has reported. SMART stands for Self Monitoring And Reporting Technology. A SMART event is an impending error condition detected by a disk drive. The drive reports SCSI sense data with a sense code of 0b (hex) or 5d (hex).
ioTimeoutCount	The number of times the drive accepted an I/O request but did not complete it in the allotted time. The allotted time depends on the product; an appropriate time applies to each product.
noResponseCount	The number of times the drive has not responded to an I/O request. This is different from ioTimeoutCount because to get an I/O timeout, the drive must at least accept the I/O request initially.
port[2]	This is an array of size 2 because all these statistics are recorded per drive port and the drive may be dual-ported.

CAPI_DRIVE_LOCATION

This structure provides the index to a SCSI drive relative to its controller.

```
typedef struct
{
    CAPI_U32             channelIndex;
    CAPI_U32             driveIndex;
} CAPI_DRIVE_LOCATION;
```

Table 4-19. CAP_DRIVE_LOCATION fields.

Parameter	Description
channelIndex	The physical channel number that specifies the index into the channel array (driveChannel[ ]) in the CAPI_CONTROLLER structure for the back-end drive channel that the drive is connected to.
driveIndex	Specifies the index into the drive array (driveReference[ ]) in the CAPI_CHANNEL structure for the drive. Note: This is not the SCSI ID.

CAPI_ENVIRON_PROCESSOR_DATA

This structure describes the data that is returned in the callback from the CAPI_EnvironRead and CAPI_U_EnvironRead functions and pointed to by dataPtr. It also defines the data structure that is used as the *buffer* parameter in the CAPI_EnvironWrite and CAPI_U_EnvironWrite functions.

```
typedef struct
{
    CAPI_U8     data[ CAPI_ENVIRON_MAX_ENVIRON_DATA_LENGTH ];
} CAPI_ENVIRON_PROCESSOR_DATA;
```

Table 4-20. CAPI_ENVIRON_PROCESSOR_DATA fields.

Parameter	Description
data	Contains the data received from the calls to CAPI_EnvironRead. It also is used to pass data into the CAPI_EnvironWrite function.

For a complete description of the layout of the SAF-TE data in this structure, please refer to the SCSI Accessed Fault-Tolerant Enclosures Interface Specification. (This implies that the data would only be SAF-TE data. I think that is could be SAF-TE or SES, depending on the capability of that particular Chaparral controller. If true, then maybe this section could use a bit more info written…)

When using this structure in the CAPI_EnvironWrite and CAPI_U_EnvironWrite commands as the *buffer* parameter, the first byte of this structure is the first byte of write data. That is, the first byte does *not* contain the write buffer's Operation Code. The write buffer's Operation Code is passed as the *environCommand* parameter to the CAPI function and is inserted into the actual command sent to the EMP by the controller.

CAPI_ENVIRON_PROCESSOR_INFO

This structure describes the data that is returned by the CAPI_FindNextEnvironProcessor and CAPI_U_FindNextEnvironProcessor functions.

```
typedef struct
{
    CAPI_U8         empId;
    CAPI_U8         busId;
    CAPI_U8         targetId;
    CAPI_U8         lun;
    union
    {
        CAPI_U8         inquiry[ CAPI_ENVIRON_MAX_INQUIRY_BYTES ];
        struct
        {
            CAPI_U8     scsiStatus;
            CAPI_U16    controllerStatus;
            CAPI_U8     senseData[ CAPI_ENVIRON_MAX_SENSE_BYTES ];
        } e;
    } u;
} CAPI_ENVIRON_PROCESSOR_INFO;
```

Table 4-21. CAPI_ENVIRON_PROCESSOR_INFO fields.

Parameter	Description
empId	This is the CAPI index of the Enclosure Management Processor (EMP) used in the CAPI_FindNextEnvironProcessor call.
busId	Controller disk bus on which EMP is connected.
targetId	SCSI ID or Loop ID of the environ processor (also know as Enclosure Management Processor or EMP).
lun	This EMP's LUN.
inquiry	If the CAPI_FindNextEnvironProcessor command returns CAPI_NO_ERROR, this field contains the standard inquiry page's data. As defined by SCSI, the first byte defines the device type (SES or SAF-TE).
scsiStatus	If the CAPI_FindNextEnvironProcessor command returns CAPI_ERROR_COMMAND_FAILED, this field contains the SCSI status byte from the EMP.
controllerStatus	Unused.
senseData	If the CAPI_FindNextEnvironProcessor command returns CAPI_ERROR_COMMAND_FAILED, this field contains sense data from the EMP.

CAPI_EVENT

CAPI_EVENT describes an event that occurred on the external controller. The event may have happened asynchronously (such as a drive failure) or might be the result of a command issued by the application (such as create array complete).

> **Note:** *arrayIndex in the CAPI_EVENT should never be used because it is only valid for a particular configuration.*

```
typedef struct
{
    CAPI_U32                 sequenceNumber;
    CAPI_TIME                timeStamp;
    CAPI_EVENT_CODE          eventCode;
    CAPI_ERROR_CODE          errorCode;
    CAPI_EVENT_CRITICALITY   criticality;
    CAPI_IDENTIFIER          id;
    CAPI_U32                 deviceId;
    CAPI_U32                 param1;
    CAPI_U32                 param2;
    CAPI_U32                 param3;
    CAPI_U32                 param4;
    union
    {
        CAPI_U8              extraEventData[CAPI_MAX_BYTES_FOR_EXTRA_EVENT_DATA];
        CAPI_SERIAL_NUMS     serialNumbers;
        CAPI_FW_REVS         fwRevs;
    }u;
    CAPI_U8                  cdb[CAPI_MAX_BYTES_FOR_EVENT_CDB];
    CAPI_U32                 uniqueId;
    CAPI_U32                 unitNum;
    CAPI_CONTROLLER_ID       controller;
} CAPI_EVENT;
```

Table 4-22. CAPI_EVENT fields.

Parameter	Description
sequenceNumber	The controller applies an ever increasing ordinal number to each event that occurs on the controller. A value of zero indicates an empty event log.
timeStamp	Number of seconds since January 1, 1970.
eventCode	Lists the event code, such as CAPI_EVENT_CREATE_ARRAY_COMPLETE.
errorCode	If errorCode equals CAPI_NO_ERROR, then the operation completed successfully; otherwise, see Error Code Reference on page 8-1.
criticality	Specifies if the event is either informational, a warning, or an error.
id	controllerHandle always valid; channelIndex, arrayIndex, and driveIndex sometimes valid (if not equal to CAPI_NULL_ID).
deviceId	The CAPI_IDENTIFIER does not specify an actual SCSI ID but rather an index into a channel array, The deviceId can be used to obtain the SCSI ID that identifies the device that generated the event, which is useful because the CAPI_IDENTIFIER may not be valid after a configuration change.
param1 – param4	Four event-specific data fields.
extraEventData	Event-specific data.
serialNumbers	A structure containing array and drive serial numbers for those events pertaining to arrays and drives.
fwRevs NEW!	A structure containing the main and baselevel version strings (for some events)
Cdb	Contains the CDB that triggered the event. This is only valid for CAPI_EVENT_DISK_CHANNEL_ERROR and CAPI_EVENT_DISK_DETECTED_ERROR.
uniqueId	This field is the value that is set in the uniqueId parameter in certain commands. It allows a CAPI application to direct events received to the appropriate controller

	sub-application.
unitNum	The LUN associated with this event. Useful with downed drive events.
controller NEW! in CAPI 3.4	Identifies which controller had this event in its event log. Used for unified commands. One of: CAPI_CONTROLLER_A, CAPI_CONTROLLER_B.

CAPI_FAILOVER

This structure is used to hold Active-Active controller failover knowledge. Most of the information in this structure is about the other controller. The other controller is in the opposite ID slot (A or B).

```
typedef struct
{
    CAPI_CONTROLLER_ID      failoverId;
    CAPI_BOOL               failedOver;
    CAPI_U8                 otherCapiUnitNum;
    CAPI_U8                 placeholderUnitNum;
    CAPI_U8                 otherSerialNumber[CAPI_MAX_SERIAL_NUMBER_BYTES];
    CAPI_CHAR               otherFirmwareRevision[CAPI_MAX_STRING];
    CAPI_CHAR               otherLoaderRevision[CAPI_MAX_STRING];
    CAPI_CHAR               otherModel[CAPI_MAX_STRING];
    CAPI_U8                 otherTargetId;
    CAPI_CONTROLLER_ID      otherId;
    CAPI_OS_OTHER_STATE     otherState;
    CAPI_FR_FAILOVER_REASON failoverReason;
    CAPI_U32                otherAAVersion;
    CAPI_U8                 otherEnvironUnitNum[CAPI_MAX_ENVIRON_DEVICES];
    CAPI_U8                 otherNodeWWN[CAPI_FC_WWID_SIZE];
    CAPI_U8                 otherPortWWN
                  [CAPI_MAX_HOST_CHANNELS_PER_CONTROLLER / 2][CAPI_FC_WWID_SIZE];
    CAPI_BOOL               placeholderUnitValid;
    CAPI_U8                 newOtherEnvironUnitNum[CAPI_NEW_MAX_ENVIRON_DEVICES];
} CAPI_FAILOVER;
```

Table 4-23. CAPI_FAILOVER fields.

Parameter	Description
FailoverId	Identifies this controller; one of: CAPI_CONTROLLER_A or CAPI_CONTROLLER_B.
FailedOver	TRUE if the other controller has failed and this controller is servicing its LUNs. (This may not happen until several seconds after otherState goes to CAPI_OS_DOWN.)
otherCapiUnitNum	Contains the value of the controller LUN for the other controller.
PlaceholderUnitNum	Contains the placeholder LUN for the other controller's controller LUN. A placeholder LUN is used when a surviving controller does not know the value of the other controller's controller LUN.
OtherSerialNumber	Contains the serial number of the other controller as a null terminated ASCII string.
otherFirmwareRevision	Contains the revision of firmware running in the other controller as a null terminated ASCII string.
otherLoaderRevision	Contains the revision of loader code running in the other controller as a null terminated ASCII string.
otherModel	Contains the model of the other controller as a null terminated ASCII string.
OtherTargetId	Identifies the SCSI target ID or FC ID presented to the host for LUNs belonging to the other controller.
otherId	Identifies the other controller (either A or B)
otherState	Contains the Active-Active related state of the controller.
failoverReason	If the other controller's state is "CAPI_OS_DOWN", contains the reason the other controller is in this state (if known).
otherAAVersion	Contains the Active-Active compatibility version number of the other controller. This value must match the value in the other controller to run Active-Active controllers.
otherEnvironUnitNum	Array of environmental LUN (also know as EMP LUN) values for the other controller. Use this if only 10 environmental LUNs are supported; that is, if

	CAPI_CAPABILITY_3_SUPPORT_16_ENVIRON_LUNS is *not* set.
otherNodeWWN	Contains Node WWN of other controller
otherPortWWN	Contains Port WWN's of other controller. Divide total host channels by 2, since to support failover, half must be assigned to the other controller:
placeholderUnitValid	TRUE if placeholder LUN in use, else FALSE. See Failover Notes in Chapter 16.
newOtherEnvironUnitNum NEW! in CAPI 3.4	Array of environmental LUN (also known as EMP LUN) values for the other controller. Use this if 16 environmental LUNs are supported; that is, if CAPI_CAPABILITY_3_SUPPORT_16_ENVIRON_LUNS is set.

CAPI_FC_DRIVE_ERRORS NEW! in CAPI 3.3

This structure contains error data specific to Fibre Channel drives. Used as a member of CAPI_DRIVE_ERROR_STATS.

```
typedef struct
{
    CAPI_FC_LESB_DATA  lesb;
    CAPI_U32           protocolErrorCount;
} CAPI_FC_DRIVE_ERRORS;
```

Table 4-24. CAPI_FC_DRIVE_ERRORS fields.

Parameter	Description
lesb	Link Error Status Block. This is a data structure defined by the FC-FS (Fibre Channel Framing and Signaling) specification. It contains very-low-level Fibre Channel error information maintained by the drive. See CAPI_FC_LESB_DATA in capi3.h.
protocolErrorCount	A count of frame and CRC errors. This is a count kept by the controller and may not match the LESB data

CAPI_FC_INFO

This structure contains Fibre Channel information.

```
typedef struct
{
    CAPI_BOOL              FCLinkUp;
    CAPI_U8                FCActiveTopology;
    CAPI_U8                FCConfigTopology;
    CAPI_U8                FCClassOfService;
    CAPI_FC_LOOP_POSITION  loopPositionalMap;
    CAPI_U16               maxSpeed;
    CAPI_U8                FCLoopId;
    CAPI_U32               FCAddr;
    CAPI_U8                FCNodeWWN[ CAPI_FC_WWID_SIZE ];
    CAPI_U8                FCPortWWN[ CAPI_FC_WWID_SIZE ];
    CAPI_U8                HardwareVersion;
    CAPI_U8                HardwareVariant;
    CAPI_U8                FCLibRevMajor;
    CAPI_U8                FCLibRevMinor;
    CAPI_U8                FCLibRevFix;
    CAPI_LINK_SPEED        FCActiveLinkSpeed;
    CAPI_LINK_SPEED        FCConfigLinkSpeed;
    CAPI_MIB_PORT_STATE    mibState;
    CAPI_MIB_PORT_STATUS   mibStatus;
    CAPI_BOOL              externalFCLinkUp;
} CAPI_FC_INFO;
```

Table 4-25. CAPI_FC_INFO fields.

Parameter	Description
FCLinkUp	Fibre channel link state
FCActiveTopolgy	Currently active Fibre Channel topology
FCConfigTopolgy	Configured Fibre Channel topology
FCClassOfService	FCC class of service
loopPositionalMap	An array of loop ID's reflecting position in loop
maxSpeed	max capable speed in MB/sec.
FCLoopId	Currently active loop ID.
FCAddr	Currently active fibre channel address.
FCNodeWWN	Contains the node's world-wide name.
FCPortWWN	Contains the port's world-wide name.
HardwareVersion	Chaparral use only (vendor's FC chip version)
HardwareVariant	Chaparral use only (vendor's FC chip variant)
FCLibRevMajor	Chaparral use only
FCLibRevMinor	Chaparral use only
FCLibRevFix	Chaparral use only
FCActiveLinkSpeed	Current active FC Link Speed: 0 = 1G, 1 = 2G, 2 = AUTO
FCConfigLinkSpeed	Configured FC Link Speed: 0 = 1G, 1 = 2G, 2 = AUTO
mibState	Port state as defined by FibreAlliance MIB 2.2
mibStatus	Port status as defined by FibreAlliance MIB 2.2
externalFCLinkUp NEW! in CAPI 3.4	TRUE if there is a live external device.

CAPI_FC_LOOP_POSITION

This structure is used with the CAPI_FC_INFO structure.

```
typedef struct
{
    CAPI_U8                 numIDs;
    CAPI_U8                 loopMasterID;
    CAPI_U8                 map[CAPI_MAX_DEVICES_FC_LOOP];
} CAPI_FC_LOOP_POSITION;
```

Table 4-26. CAPI_FC_LOOP_POSITION fields.

Parameter	Description
numIDs	The number of IDs on the loop.
loopMasterID	The ID of the loop master.
map[]	Array of loop IDs that show the physical topology of the loop. That is, the order of the IDs in this array matches the order of the devices on the loop. The number of valid elements in this array is *numIDs*.

CAPI_FC_PARAMS

Settable Fibre Channel parameters.

```
typedef struct
{
    CAPI_BOOL          forcePrivateLoop;
    CAPI_TOPOLOGY      topology;
    CAPI_LINK_SPEED    linkSpeed;
    CAPI_U8            multiTargetId[16];
} CAPI_FC_PARAMS;
```

Table 4-27. CAPI_FC_INFO fields.

Parameter	Description
forcePrivateLoop	Set to TRUE to force private loop. This is not currently supported on any Chaparral products.
topology	Topology of Fibre Channel connection. Applies to both host and disk channels, but current Chaparral disk products only support loop mode on disk channels.
linkSpeed	0 = 1G, 1 = 2G, 2 = AUTO. Applies to host channels and disk channels.
multiTargetId	128-bit bit map of enabled IDs (if multiple IDs are supported). Applies to host channels only, not disk channels. This is not currently supported on any Chaparral products.

CAPI_FLEX_ID

The CAPI_FLEX_ID structure is used to describe Fibre Channel and SCSI devices and hosts.

```
typedef struct
{
    CAPI_FLEX_TYPE        type;
    CAPI_U32              channelIndex;
    CAPI_U32              deviceId;
    CAPI_U8               FCNodeWWN[CAPI_FC_WWID_SIZE];
    CAPI_U8               FCPortWWN[CAPI_FC_WWID_SIZE];
    CAPI_U32              unitNum;
} CAPI_FLEX_ID;
```

Table 4-28. CAPI_FLEX_ID fields.

Parameter	Description
type	A bit-mask of CAPI_FLEX_TYPE that describes which fields are valid
channelIndex	A CAPI channel index.
deviceId	Either a traditional SCSI ID or an FC ADDRESS (see 'type')
FCNodeWWN	World-wide Fibre Channel Node ID
FCPortWWN	World-wide Fibre Channel Port ID
unitNum	SCSI LUN value

CAPI_FW_REVS NEW!

The CAPI_FW_REVS structure is used to describe controller firmware revisions within a CAPI_EVENT.

Note that this is only the Storage Controller firmware version and does not include the LAN Subsystem firmware version nor the loader firmware version.

```
typedef struct
{
    CAPI_U8                 fwVersion[16];
    CAPI_U8                 baseVersion[16];
} CAPI_FW_REVS;
```

Table 4-23. CAPI_FW_REVS fields.

Parameter	Description
fwVersion	Storage Controller firmware version. The letters B and A in this field refer to Beta and Alpha code, respectively.
baseVersion	The base Storage Controller firmware version. In pre-release builds, this is usually the same string as *fwVersion*. In released builds, this string shows the release candidate number, while *fwVersion* does not.

CAPI_HOST_DESCRIPTOR

The CAPI_HOST_DESCRIPTOR is used to describe a host. This struct is used for arrays of known hosts and for arrays of host nicknames.

```
typedef struct
{
    CAPI_FLEX_ID        hostId;
    CAPI_U8             name[CAPI_MAX_HOST_NAME];
    CAPI_U32            age;
    CAPI_CONTROLLER_ID  controllerId;
} CAPI_HOST_DESCRIPTOR;
```

Table 4-29. CAPI_HOST_DESCRIPTOR fields.

Parameter	Description
hostId	A flexible ID that describes the host.
name	A symbolic name that the user may assign to the host (nickname).
age	A value used to keep track of when this instance of CAPI_HOST_DESCRIPTOR was added to an array of CAPI_HOST_DESCRIPTOR structs. This is a timestamp (seconds since January 1, 1970). NEW! in CAPI 3.4 (Prior to CAPI 3.4, this was a counter that incremented with each addition rather than a timestamp.)
controllerId NEW! in CAPI 3.4	Used to indicate whether controller A, B, or both knows about this host. (Valid only when CAPI_U_GetKnownHosts is called with *controllerId* set to CAPI_CONTROLLER_BOTH.)

CAPI_HOST_NICKNAMES NEW! in CAPI 3.3

The CAPI_HOST_NICKNAME structure is used to identify the hosts that have been assigned nicknames by calling CAPI_AddHostNickname or CAPI_U_AddHostNickname.

```
typedef struct
{
    CAPI_U8               numHosts;
    CAPI_HOST_DESCRIPTOR  host[CAPI_MAX_HOST_TABLE];
} CAPI_HOST_NICKNAMES;
```

Table 4-30. CAPI_HOST_NICKNAMES fields.

Parameter	Description
numHosts	The number of hosts in the list.
host	The list of host IDs with their nicknames.

CAPI_HOST_TABLE

The CAPI_HOST_TABLE is used to include or exclude a host from access to a particular LUN.

```
typedef struct
{
    CAPI_BOOL       include;
    CAPI_BOOL       all;
    CAPI_U8         numHosts;
    CAPI_BOOL       portInfoShield;
    CAPI_U8         portNumber;
    CAPI_FLEX_ID    hostId[CAPI_MAX_HOST_TABLE];
} CAPI_HOST_TABLE;
```

Table 4-31. CAPI_HOST_TABLE fields.

Parameter	Description
include	If TRUE, this list is a list of hosts to include for access to the LUN, otherwise this is a list of hosts to exclude.
all	If TRUE, the list is ignored and all hosts are either included or excluded.
numHosts	The number of hosts in the list.
portInfoShield	If TRUE, then the *portNumber* must be taken into account for determining access (i.e. only requests on the specified port number will qualify).
portNumber	The port number that the request must come in on.
hostId	A flexible ID that describes the host.

CAPI_IDENTIFIER

This structure is passed to the application's callback routine to specify a combination of controller, channel, array, or drive related to an event.

```
typedef struct
{
    CAPI_HANDLE          controllerHandle;
    CAPI_U32             arrayIndex;
    CAPI_U32             channelIndex;
    CAPI_U32             driveIndex;
} CAPI_IDENTIFIER;
```

Table 4-32. CAPI_IDENTIFIER fields.

Parameter	Description
controllerHandle	The CAPI_HANDLE received during initialization from calls to CAPI_FindNextController.
arrayIndex	The index into the CAPI_ARRAY structure array in the CAPI_CONTROLLER structure for the related RAID array. CAPI_NULL_ID if no array is specified.
channelIndex	The physical channel number. This is the index into the CAPI_CHANNEL array in the CAPI_CONTROLLER structure for the related drive channel. CAPI_NULL_ID if no channel is specified.
driveIndex	The index into the CAPI_DRIVE array in the CAPI_CHANNEL structure for the related drive. CAPI_NULL_ID if no drive is specified. NOTE: This is not the SCSI ID.

CAPI_KNOWN_HOSTS

The CAPI_KNOWN_HOSTS structure is used to identify the hosts that are known to the controller as a result of a host accessing the controller.

> ***Note:*** *As of CAPI 3.4, CAPI_MAX_HOST_TABLE has been increased from 16 to 64.*

```
typedef struct
{
    CAPI_U8               numHosts;
    CAPI_HOST_DESCRIPTOR  host[CAPI_MAX_HOST_TABLE];
} CAPI_KNOWN_HOSTS;
```

Table 4-33. CAPI_KNOWN_HOSTS fields.

Parameter	Description
numHosts	The number of hosts in the list.
host	The list of hosts that are known to the controller.

CAPI_MAINT_CDB

This structure is used by the CAPI_ScsiMaintRetrieveData and CAPI_U_GetScsiMaintenanceData functions to pass a SCSI command descriptor block to a back-end device.

```
typedef struct
{
    CAPI_U8              cdbArray[16];
} CAPI_MAINT_CDB;
```

Table 4-34. CAPI_MAINT_CDB fields.

Parameter	Description
cdbArray	The SCSI CDB.

CAPI_MAINT_DATA_STRUCT

This structure is used with the CAPI_ScsiMaintRetrieveData and CAPI_U_GetScsiMaintenanceData functions.

```
typedef struct _CAPI_MAINT_DATA_STRUCT
{
    CAPI_U8                data[ CAPI_MAX_MAINT_DATA_SIZE ];
} CAPI_MAINT_DATA_STRUCT;
```

Table 4-35. CAPI_MAINT_DATA_STRUCT fields.

Parameter	Description
data	Contains the SCSI maintenance data.

CAPI_MEMBER_DRIVE

This structure describes the physical channel and index (not the SCSI ID) of the drive in the channel structure.

```
typedef struct
{
    CAPI_DRIVE_LOCATION   driveLocation;
    CAPI_UTILITY_RUNNING  utilityRunning;
    CAPI_DRIVE_STATE      state;
} CAPI_MEMBER_DRIVE;
```

Table 4-36. CAPI_MEMBER_DRIVE fields.

Parameter	Description
driveLocation	Specifies the physical drive.
utilityRunning	Indicates whether a utility is currently running on the array and if so, which one.
state	Drive state.

CAPI_MEMORY

This structure describes the current memory setup of a given memory slot.

```
typedef struct
{
    CAPI_U16              size;
    CAPI_BOOL             ECCprotected;
} CAPI_MEMORY;
```

Table 4-37. CAPI_MEMORY fields.

Parameter	Description
size	Memory size in MB (1MB = 1,048,576 bytes)
ECCprotected	Set to true if ECC protection on this memory

CAPI_MIN_MAX_DRIVES_PER_RAID_LEVEL

This structure describes the minimum and maximum number of drives that are allowed based on the RAID level.

```
typedef struct
{
    CAPI_U8              minDrives;
    CAPI_U8              maxDrives;
} CAPI_MIN_MAX_DRIVES_PER_RAID_LEVEL;
```

Table 4-38. CAPI_MIN_MAX_DRIVES_PER_RAID_LEVEL fields.

Parameter	Description
minDrives	Specifies the minimum number of drives allowed per the RAID level.
maxDrives	Specifies the maximum number of drives allowed per the RAID level.

CAPI_NETWORK_INTERFACE

This structure describes the controller's LAN Subsystem (also known as LAN processor) configuration.

Note: field names have been changed since CAPI 3.1 (underscores removed to be consistent with standard CAPI naming conventions). NEW!

```
typedef struct
{
    CAPI_BOOL   connection;
    CAPI_BOOL   status;
    CAPI_U8     hwRevision;
    CAPI_U8     fwRevisionChar;
    CAPI_U8     fwRevisionMajor;
    CAPI_U8     fwRevisionMinor;
    CAPI_U8     physicalAddr[6];
    CAPI_U8     currentIp[4];
    CAPI_U8     defaultIp[4];
    CAPI_U8     currentMask[4];
    CAPI_U8     defaultMask[4];
    CAPI_U8     snmpTrapHostIp[4];
    CAPI_U8     gateway[4];
} CAPI_NETWORK_INTERFACE;
```

Table 4-39: CAPI_NETWORK_INTERFACE fields.

Parameter	Description
connection	Chaparral internal use only
status	0 = LAN Subsystem not ready or not installed; 1 = LAN Subsystem installed
hwRevision	Chaparral internal use only
fwRevisionChar	Chaparral internal use only
fwRevisionMajor	LAN Subsystem major firmware revision
fwRevisionMinor	LAN Subsystem minor firmware revision
physicalAddr	LAN Subsystem's MAC address (read only – not settable by CAPI apps)
currentIp	LAN Subsystem's current IP address (read only)
defaultIp	"Pending" value used by CAPI apps for setting the *currentIp*
currentMask	LAN Subsystem's current IP subnet mask (read only)
defaultMask	"Pending" value used by CAPI apps for setting the *currentMask*
snmpTrapHost	IP address that SNMP traps will be sent to (settable by CAPI apps)
gateway	gateway IP address (settable by CAPI apps)

CAPI_NETWORK_INTERFACE_COMMON_DATA

NEW! in CAPI 3.4

This structure is used for Unified CAPI as part of the data that is gotten with CAPI_U_GetControllerData. The members of this structure are equivalent to members of CAPI_NETWORK_INTERFACE and CAPI_ADVANCED_NETWORK_INTERFACE that are common for both controller boards. See the descriptions of those structures for details of the members of this struct.

Some members of this struct are set by the LAN Subsystem but they are not settable by customer CAPI applications.

```
typedef struct
{
    CAPI_BOOL               connection;
    CAPI_U8                 snmpVersionMajor;
    CAPI_U8                 snmpVersionMinor;
    CAPI_U8                 snmpVersionMinorMinor;
    CAPI_U8                 snmpVersionChar;
} CAPI_NETWORK_INTERFACE_COMMON_DATA;
```

CAPI_NETWORK_INTERFACE_COMMON_PARAMS

NEW! in CAPI 3.4

This structure is used for Unified CAPI as part of the parameters that are passed with CAPI_U_SetControllerParams. The members of this structure are equivalent to members of CAPI_NETWORK_INTERFACE and CAPI_ADVANCED_NETWORK_INTERFACE that are common for both controller boards. See the descriptions of those structures for details of the members of this struct.

The members of this struct are settable by the LAN Subsystem and by customer CAPI applications via CAPI_U_SetControllerParams.

```
typedef struct
{
    CAPI_IP_ADDRESS_MODE    ipAddressMode;
    CAPI_BOOL               snmpTrapsEnable;
    CAPI_U8                 snmpTrapHostIp[4];
    CAPI_CHAR               snmpWriteCommunity[CAPI_MAX_NETWORK_STRING];
    CAPI_CHAR               snmpReadCommunity[CAPI_MAX_NETWORK_STRING];
    CAPI_SNMP_NOTIFICATION_FILTER  snmpEventFilter;
    CAPI_SNMP_NOTIFICATION_FILTER  snmpTrapFilter;
    CAPI_U32                snmpEventMaxToDisplay;
    CAPI_CHAR               systemName[CAPI_SYSTEM_STRING_MAX];
    CAPI_CHAR               systemContact[CAPI_SYSTEM_STRING_MAX];
    CAPI_CHAR               systemLocation[CAPI_SYSTEM_STRING_MAX];
    CAPI_CHAR               systemInfo[CAPI_SYSTEM_STRING_MAX];
    CAPI_CHAR               ftpUser[CAPI_MAX_NETWORK_STRING];
    CAPI_CHAR               ftpPassword[CAPI_MAX_NETWORK_STRING];
    CAPI_BOOL               ftpFwDownloadDisable;
    CAPI_CHAR               telnetPassword[CAPI_MAX_NETWORK_STRING];
    CAPI_U8                 telnetTimeout;
    CAPI_BOOL               telnetDisable;
    CAPI_BOOL               dhcpEnable;
    CAPI_U8                 pollInterval;
    CAPI_BOOL               httpDisable;
    CAPI_BOOL               snmpDisable;
    CAPI_BOOL               debugEnable;
    CAPI_U32                monitoredEvents[CAPI_NUM_MONITORED_EVENTS];
    CAPI_CHAR               email1[CAPI_SYSTEM_STRING_MAX];
    CAPI_CHAR               email2[CAPI_SYSTEM_STRING_MAX];
    CAPI_CHAR               email3[CAPI_SYSTEM_STRING_MAX];
    CAPI_CHAR               email4[CAPI_SYSTEM_STRING_MAX];
    CAPI_CHAR               comment[CAPI_NUM_COMMENT_LINES * CAPI_SYSTEM_STRING_MAX];
    CAPI_U32                pollingPeriod;
    CAPI_U8                 numberOfMessagesSentPerEvent;
    CAPI_BOOL               remoteNotificationEnable;
    CAPI_U8                 remoteNotificationSelection;
    CAPI_U8                 remoteNotificationTimeZone;
    CAPI_CHAR               serverName[CAPI_MAX_NETWORK_STRING];
    CAPI_U32                serverPort;
    CAPI_CHAR               domainName[CAPI_MAX_NETWORK_STRING];
    CAPI_CHAR               wbiMonitorPassword[CAPI_MAX_NETWORK_STRING];
    CAPI_CHAR               wbiManagePassword[CAPI_MAX_NETWORK_STRING];
} CAPI_NETWORK_INTERFACE_COMMON_PARAMS;
```

CAPI_NETWORK_INTERFACE_UNIQUE_DATA

NEW! in CAPI 3.4

This structure is used for Unified CAPI as part of the data that is gotten with CAPI_U_GetControllerData. The members of this structure are equivalent to members of CAPI_NETWORK_INTERFACE and CAPI_ADVANCED_NETWORK_INTERFACE that are unique for each controller board. See the descriptions of those structures for details of the members of this struct.

Some members of this struct are set by the LAN Subsystem but they are not settable by customer CAPI applications.

```
typedef struct
{
    CAPI_BOOL               status;
    CAPI_U8                 hwRevision;
    CAPI_U8                 fwRevisionChar;
    CAPI_U8                 physicalAddr[6];
    CAPI_CHAR               firmwareRevisionString[CAPI_MAX_STRING];
    CAPI_CHAR               firmwareBuildTimeDate[CAPI_MAX_NETWORK_STRING];
    CAPI_CHAR               firmwareBaselevel[CAPI_MAX_STRING];
    CAPI_CHAR               lanLoaderRevision[CAPI_MAX_STRING];
    CAPI_U8                 fwRevisionMajor;
    CAPI_U8                 fwRevisionMinor;
    CAPI_U8                 fwRevisionMinMin;
} CAPI_NETWORK_INTERFACE_UNIQUE_DATA;
```

CAPI_NETWORK_INTERFACE_UNIQUE_PARAMS

NEW! in CAPI 3.4

This structure is used for Unified CAPI as part of the parameters that are passed with CAPI_U_SetControllerParams. The members of this structure are equivalent to members of CAPI_NETWORK_INTERFACE and CAPI_ADVANCED_NETWORK_INTERFACE that are unique for each controller board.

The members of this struct are settable by the LAN Subsystem and by customer CAPI applications via CAPI_U_SetControllerParams.

```
typedef struct
{
    CAPI_U8                 ipAddress[4];
    CAPI_U8                 ipSubnetMask[4];
    CAPI_U8                 gateway[4];
} CAPI_NETWORK_INTERFACE_UNIQUE_PARAMS;
```

Table 4-40: CAPI_NETWORK_INTERFACE_UNIQUE_PARAMS fields.

Parameter	Description
ipAddress	The LAN processor's IP address. When this struct is instantiated in the pendingControllerUniqueParams struct, *ipAddress* is equivalent to *defaultIp* in CAPI_NETWORK_INTERFACE. When this struct is instantiated in the currentControllerUniqueParams struct, ipAddress is equivalent to currentIp in CAPI_NETWORK_INTERFACE. Format is Big-Endian; ex. 172.22.2.1 = 0xAC160201
ipSubnetMask	The LAN processor's IP subnet mask. When this struct is instantiated in the pendingControllerUniqueParams struct, *ipSubnetMask* is equivalent to *defaultMask* in CAPI_NETWORK_INTERFACE. When this struct is instantiated in the currentControllerUniqueParams struct, *ipSubnetMask* is equivalent to *currentMask* in CAPI_NETWORK_INTERFACE. Format is Big-Endian; ex. 255.255.255.0 = 0xFFFFFF00
gateway	The LAN processor's IP gateway. When this struct is instantiated in the pendingControllerUniqueParams struct or the currentControllerUniqueParams struct, gateway is equivalent to gateway in CAPI_NETWORK_INTERFACE. In other words, when a CAPI application calls CAPI_U_GetControllerData it will get the same value for gateway in both the pending and unique params structures since there are not separate values for this member of CAPI_NETWORK_INTERFACE. Format is Big-Endian; ex. 172.22.2.1 = 0xAC160201

CAPI_PACKET

This structure is used as the header of all messages passed between a CAPI application and a controller. Normally, a CAPI app developer does not need to be concerned with this structure since they are removed from this by the LMX; that is, this structure is filled in automatically when an app calls one of the API functions defined in this document, and when a reply is received by an app from a controller, the key members of this structure are copied into parameters passed to the application's callback function.

However, if a CAPI app developer needs to develop an LMX or modify an existing one, this information may be useful.

```
typedef struct
{
    CAPI_U8                control;
    CAPI_U8                byteOrder;
    CAPI_U8                capiVersionMajor;
    CAPI_U8                capiVersionMinor;
    CAPI_COMPRESSION_TYPE  requestCompressionType;
    CAPI_COMPRESSION_TYPE  packetCompressionType;
    CAPI_U8                eventOrCommand;
    CAPI_U8                signatureString[4];
    CAPI_U32               includeStructType;
    CAPI_U32               commandCode;
    CAPI_IDENTIFIER        identifier;
    CAPI_U32               configSequenceNumber;
    CAPI_ERROR_CODE        errorCode;
    CAPI_U32               param1;
    CAPI_U32               param2;
    CAPI_U32               param3;
    CAPI_U32               param4;
    CAPI_U32               packetLength;
    CAPI_U32               arrayListConfigSequenceNumber;
    CAPI_U32               uniqueId;
    CAPI_U32               driveListConfigSequenceNumber;
} CAPI_PACKET;
```

Table 4-41: CAPI_PACKET fields.

Parameter	Description
control	Not used.
byteOrder	Not implemented.
capiVersionMajor	Major version (for example, the "3" in "3.4").
capiVersionMinor	Minor version (for example, the "4" in "3.4").
requestCompressionType	The type of compression that this command is requesting be used on the reply to this command.
packetCompressionType	The type of compression used on the data sent with this message.
eventOrCommand	0 = command (message going from host to controller); 1 = reply (message going from controller to host).
signatureString[4]	The ASCII string "CAPI" to aid in confirming that messages are CAPI commands.
includeStructType	Identifies the data type of the data (if any) that follows this packet header; one of the INCLUDE_… values defined in capipak.h.
commandCode	The command code passed with a CAPI command (host->controller) or the reply code passed with a CAPI reply (controller->host). See chapter 6 for a list of reply codes.
identifier	See structure definition for CAPI_IDENTIFIER. The *controllerHandle* member of this structure is used for every message, but the other members are only used for some

	commands and replies.
configSequenceNumber	See Controller Configuration Sequence Number on page 10.
errorCode	Success/failure code. Used for replies only. See Chapter 9 for a list of error codes.
param1 through param4	General purpose parameters used for both commands and replies.
packetLength	Total packet size, including both this header and any data that follows this header.
arrayListConfigSequenceNumber	See Controller Configuration Sequence Number on page 10.
uniqueId	Not implemented.
driveListConfigSequenceNumber	See Controller Configuration Sequence Number on page 10.

CAPI_PARTITION_REQUEST

This structure is used to describe an array partition when calling functions CAPI_AddArrayPartition, CAPI_U_AddArrayPartition, CAPI_ChangeArrayPartitionGeometry, and CAPI_U_ChangeArrayPartitionGeometry.

```
typedef struct
{
    CAPI_U64    startLba;
    CAPI_U64    sizeLba;
    CAPI_CHAR   name[CAPI_MAX_ARRAY_NAME];
    CAPI_U8     unitNum;
    CAPI_U8     arraySerialNumber[CAPI_MAX_SERIAL_NUMBER_BYTES];
    CAPI_U8     partitionSerialNumber[CAPI_MAX_SERIAL_NUMBER_BYTES];
} CAPI_PARTITION_REQUEST;
```

Table 4-42: CAPI_PARTITION_REQUEST fields.

Parameter	Description
startLba	Starting Logical Block Address (LBA) of the partition relative to the first LBA (i.e. 0) of the array. The starting LBA must reside in a free (i.e. unpartitioned) area of the array. (Used for both adding partition and changing partition geometry.)
sizeLba	Size of the partition in (512 byte) logical blocks. The size of the partition must be such that it resides completely within a free area of the array. (Used for both adding partition and changing partition geometry.)
name[]	The ASCII character name of the partition assigned by the user during array creation (null terminated string). (Used for adding partition only.)
unitNum	Identifies the SCSI LUN that is presented to the host. (Used for adding partition only.)
arraySerialNumber[]	The serial number of the array to which the partition belongs (or in which it will be created). Not null terminated. (Used for both adding partition and changing partition geometry.)
partitionSerialNumber[]	The serial number for the partition that is assigned by the controller when adding a partition and which uniquely identifies the partition. Not null terminated. (Used for changing partition geometry only.)

CAPI_PER_CHANNEL_PARAMS NEW! in CAPI 3.4

This structure is used for Unified CAPI as part of the parameters that are passed with CAPI_U_SetControllerParams. It contains variables that are equivalent to variables with the same names that are in the CAPI_CHANNEL_PARAMS struct. The CAPI_PER_CHANNEL_PARAMS structure was created for UCAPI since that seemed like a good time to clean up an idiosyncrasy: some of the members of CAPI_CHANNEL_PARAMS were per-channel but others were per-controller; the per-controller variables were unnecessarily repeated in each instance of CAPI_CHANNEL_PARAMS and this could lead to confusion.

See CAPI_CHANNEL_PARAMS for details of the members of this struct.

```
typedef struct
{
    union
    {
        CAPI_SCSI_PARAMS    scsiParams;
        CAPI_FC_PARAMS      fibreParams;
    } p;

    CAPI_U8                 id;
    CAPI_BOOL               disable;
} CAPI_PER_CHANNEL_PARAMS;
```

CAPI_RAID

This structure contains information on the current configuration of all of the arrays on the controller.

```
typedef struct
{
    CAPI_U32                    maxChunkSize;
    CAPI_U32                    minChunkSize;
    CAPI_U32                    numDrives;
    CAPI_U32                    numPoolSpares;
    CAPI_U32                    numArrays;
    CAPI_MIN_MAX_DRIVES_PER_RAID_LEVEL  minMaxPerRaidLevel[CAPI_MAX_RAID_LEVELS];
    CAPI_MEMBER_DRIVE           poolSpare[CAPI_MAX_POOL_SPARES_PER_CONTROLLER * 2];
    CAPI_U8                     maxOwnedArraysPerController;
    CAPI_U8                     maxArrays;
    CAPI_U8                     numArrayBanks;
    CAPI_U8                     maxArrayBanks;
} CAPI_RAID;
```

Table 4-43. CAPI_RAID fields.

Parameter	Description
maxChunkSize	Specifies the maximum chunk size (in KB) for RAID 0, 10, 3, 4 and 5.
minChunkSize	Specifies the minimum chunk size (in KB) for RAID 0, 10, 3, 4 and 5.
numDrives	Number of drives owned by this controller (that is, part of an array owned by this controller, including dedicated spares), plus pool spares, plus “available” drives.
numPoolSpares	Specifies the number of pool spare drives currently assigned.
numArrays	Specifies the current number of RAID arrays owned by this controller.
minMaxPerRaidLevel	Specifies the min and max number of drives allowable for each RAID level.
poolSpare	An array of CAPI_MEMBER_DRIVE structures for the pool spares active on this controller. The size allows room for the other controller's pool spares when a failover occurs.
maxOwnedArraysPerController	The maximum number of arrays this controller can own.
maxArrays	Maximum number of arrays this controller supports.
numArrayBanks	Number of array banks this controller is using (32 arrays per bank).
maxArrayBanks	Maximum number of array banks this controller is capable of.

CAPI_ROUTER

This structure will contain router-specific fields to be decided for future use.

```
typedef struct
{
    CAPI_U8              reserved[84];
} CAPI_ROUTER;
```

Table 4-44. CAPI_ROUTER fields.

Parameter	Description
reserved	TBD (future use)

CAPI_SCSI_INFO

This structure describes information specific to a SCSI channel.

> ***Note to CAPI 2.x users:*** *most of the structure members were simply moved from CAPI_CHANNEL.*

```
typedef struct
{
    CAPI_BUS_TYPE           activeType;
    CAPI_BUS_TYPE           type;
    CAPI_U16                maxSpeed;
    CAPI_U16                lastSpeed;
    CAPI_BOOL               lastDataValid;
    CAPI_U8                 lastOffset;
    CAPI_U8                 lastWidth;
    CAPI_U8                 numResets;
    CAPI_MIB_PORT_STATE     mibState;
    CAPI_MIB_PORT_STATUS    mibStatus;
} CAPI_SCSI_INFO;
```

Table 4-45. CAPI_SCSI_INFO fields.

Parameter	Description
activeType	Describes the bus transceiver mode (LVD, SE, etc) currently in use.
type	Default bus transceiver mode
maxSpeed NEW!	Don't get a speeding ticket! The max capable bus speed. (MB/s, where 1MB = 1,000,000 bytes)
lastSpeed	Last negotiated speed in MB/s
lastDataValid	Set to TRUE if the *lastSpeed, lastOffset* and *lastWidth* fields are valid.
lastOffset	Last negotiated req/ack offset
lastWidth	Last negotiated width, in bits
numResets	number of SCSI bus resets on this channel since power up
mibState	State of the port as defined by the FibreAlliance MIB 2.2
mibStatus	Status of the port as defined by the FibreAlliance MIB 2.2

CAPI_SCSI_PARAMS

This structure describes SCSI-specific information that can be changed.

> ***Note to CAPI 2.x users:*** *most of the structure members were simply moved from CAPI_CHANNEL_CONFIG. The word 'host' was also removed from* termination *and* terminationPower *because this could also pertain to a disk (initiator) channel.*

```
typedef struct
{
    CAPI_U16          busSpeed;
    CAPI_BOOL         termination;
    CAPI_BOOL         terminationPower;
    CAPI_U16          multiTargetId;
    CAPI_BOOL         domainValidationDisable;
    CAPI_BOOL         hostResetOnFailover;
} CAPI_SCSI_PARAMS;
```

Table 4-46. CAPI_SCSI_PARAMS fields.

Parameter	Description
busSpeed	Bus speed (MB/s, where 1MB = 1,000,000 bytes)
termination	SCSI termination enable
terminationPower	SCSI termination power enable
multiTargetId	Bit map of enabled ids (if multiple ids are supported)
domainValidationDisable	TRUE to disable domain validation. Default is FALSE.
hostResetOnFailover	If TRUE the controller will reset the SCSI bus when it enables a SCSI channel for failover or failback. The default is FALSE (no reset.)

CAPI_SERIAL_NUMS

This structure describes the serial numbers used to uniquely identify drives, arrays, and array partitions. The length of the drive serial number is specified in each CAPI_DRIVE structure. Neither serial number is null terminated.

```
typedef struct
{
    CAPI_U8          driveSerialNumber[CAPI_MAX_SERIAL_NUMBER_BYTES];
    CAPI_U8          arraySerialNumber[CAPI_MAX_SERIAL_NUMBER_BYTES];
} CAPI_SERIAL_NUMS;
```

Table 4-47. CAPI_SERIAL_NUMS fields.

Parameter	Description
driveSerialNumber	Serial number of the drive. (Not a null-terminated string.)
arraySerialNumber	Serial number of the array. If an array partition is being referenced, then this is the partition's serial number. (Not a null-terminated string.)

CAPI_UNIFIED_CONTROLLER NEW! in CAPI 3.4

This structure is used for Unified CAPI as the structure that is returned in the callback from CAPI_U_GetControllerData. It contains all the key information about both controllers in a dual-controller system. For Unified CAPI applications, it is used instead of CAPI_CONTROLLER.

```
typedef struct
{
    CAPI_UNIFIED_CONTROLLER_COMMON_DATA  common;
    CAPI_UNIFIED_CONTROLLER_UNIQUE_DATA  unique[CAPI_MAX_NUM_CONTROLLERS];
} CAPI_UNIFIED_CONTROLLER;
```

Table 4-48. CAPI_UNIFIED_CONTROLLER fields.

Parameter	Description
common	Information (read-only) and parameters (read/write) that are common to both controllers in a dual-controller system.
unique	Information (read-only) and parameters (read/write) that can be different between the two controllers in a dual-controller system.

CAPI_UNIFIED_CONTROLLER_COMMON_DATA

NEW! in CAPI 3.4

This structure is used for Unified CAPI as part of the data that are gotten with CAPI_U_GetControllerData. This struct contains data that are common for both controllers. The members of this struct are read-only except for *pendingControllerCommonParams*, which is settable with CAPI_U_SetControllerParams. See comments in the struct, below, for information about what structures the equivalent non-unified variables are in, then see those structs for details of the members.

```
typedef struct
{
    /*
     * The members of this structure are equivalent to members of
     * CAPI_NETWORK_INTERFACE and CAPI_ADVANCED_NETWORK_INTERFACE that are
     * common for both controller boards.
     */
    CAPI_NETWORK_INTERFACE_COMMON_DATA  netIfCommonData;


    /*
     * The following variables are equivalent to variables with the same names
     * that are in the CAPI_RAID struct.
     * (Other variables equivalent to ones in CAPI_RAID are unique and so are in
     * CAPI_UNIFIED_CONTROLLER_UNIQUE_DATA.)
     */
    CAPI_U32                maxChunkSize;
    CAPI_U32                minChunkSize;
    CAPI_U32                numPoolSpares;
    CAPI_MIN_MAX_DRIVES_PER_RAID_LEVEL  minMaxPerRaidLevel[CAPI_MAX_RAID_LEVELS];
    CAPI_U8                 maxOwnedArraysPerController;
    CAPI_U8                 maxArrays;
    CAPI_U8                 numArrayBanks;
    CAPI_U8                 maxArrayBanks;
    /* (End of variables equivalent to ones in CAPI_RAID.) */


    /*
     * The following variables are equivalent to variables in CAPI_CONTROLLER
     * with the same name.
     */
    CAPI_U32                cacheSize;
    CAPI_U32                numHostChannels;
    CAPI_U32                numDriveChannels;
    CAPI_TIME               timeDate;
    CAPI_MEMORY             memorySlotA;
    CAPI_MEMORY             memorySlotB;
    CAPI_MEMORY             memorySlotC;
    CAPI_MEMORY             memorySlotD;
    CAPI_CHANNEL_COMMON_DATA
                      hostChannelCommonData[CAPI_MAX_HOST_CHANNELS_PER_CONTROLLER];
    CAPI_CHANNEL_COMMON_DATA
                      driveChannelCommonData[CAPI_MAX_DRIVE_CHANNELS_PER_CONTROLLER];
    CAPI_U8                 linkType;
    CAPI_BOOL               raidCapable;
    CAPI_BOOL               routerCapable;
    CAPI_U32                maxDmepMemoryBufferSize;
    CAPI_U32                swFeaturesAllowed;
    CAPI_ENCLOSURE_CAPABILITY  enclosureCapabilities;
    CAPI_PRODUCT_SPECIFIC_UNION  productSpecific;
    /* (End of variables equivalent to ones in CAPI_CONTROLLER.) */


    /*
     * Controller parameter information that may have been updated by CAPI,
     * and therefore does not reflect the current, in-use parameters.  This
     * is the "pending" configuration which will take effect on a reboot.
     */
```

```
    CAPI_UNIFIED_CONTROLLER_COMMON_PARAMS  pendingControllerCommonParams;

    /*
     * Controller parameter information for the currently executing
     * configuration.  This is sometimes known as the "active" params.
     */
    CAPI_UNIFIED_CONTROLLER_COMMON_PARAMS  currentControllerCommonParams;
} CAPI_UNIFIED_CONTROLLER_COMMON_DATA;
```

CAPI_UNIFIED_CONTROLLER_COMMON_PARAMS

NEW! in CAPI 3.4

This structure is used for Unified CAPI as part of the parameters that are passed with CAPI_U_SetControllerParams. This struct contains those parameters that are common for both controllers. Except for *netIfCommonParams*, these variables are equivalent to variables with the same names that are in the non-unified structure CAPI_CONTROLLER_PARAMS; see comments in the struct definition below. See CAPI_CONTROLLER_PARAMS for details of the members.

```
typedef struct
{
    /*
     * The members of this structure are equivalent to the settable members of
     * CAPI_NETWORK_INTERFACE and CAPI_ADVANCED_NETWORK_INTERFACE that are
     * common for both controller boards.
     */
    CAPI_NETWORK_INTERFACE_COMMON_PARAMS  netIfCommonParams;


    /*
     * The following variables are equivalent to variables with the same names
     * that are in the CAPI_CONTROLLER_PARAMS struct.
     */

    CAPI_U32                environPollInterval;
    CAPI_U32                performanceTuningFlags;
    CAPI_BOOL               externalTargetIdControl;

    CAPI_BOOL               environTemperatureEnable;
    CAPI_BOOL               environAutoSlotFlags;
    CAPI_BOOL               environAutoGlobalFlags;

    CAPI_BOOL               alarmMute;
    CAPI_BOOL               disableBatteryOption;
    CAPI_UTILITY_PRIORITY   utilityPriority;

    CAPI_DISK_SETTING       driveWriteBackCache;
    CAPI_DISK_SETTING       driveSMART;

    CAPI_BOOL               standAlone;
    CAPI_BOOL               dualPort;
    CAPI_BOOL               cacheLock;

    CAPI_BOOL               routerEnable;
    CAPI_BOOL               raidEnable;
    CAPI_CONTROLLER_MODE    controllerMode;

    CAPI_CONTROLLER_RAID_PARAMS     cpRaid;
    CAPI_CONTROLLER_ROUTER_PARAMS   cpRouter;

    CAPI_U32                debugLogConfig;

    CAPI_U32                dmepMemoryBufferSize;

    CAPI_U32                swFeaturesDisabled;
    CAPI_ENCLOSURE_FEATURES enclosureFeatureFlags;
    CAPI_FULL_POPULATED_CONFIG  fullPopConfig;

    /* (End of variables equivalent to ones in CAPI_CONTROLLER_PARAMS.) */

} CAPI_UNIFIED_CONTROLLER_COMMON_PARAMS;
```

CAPI_UNIFIED_CONTROLLER_PARAMS

NEW! in CAPI 3.4

This structure is used for Unified CAPI as the structure that is passed with CAPI_U_SetControllerParams. It contains all the key configuration parameters that a user may want to set on one or both controllers in a dual-controller system.

```
typedef struct
{
    CAPI_UNIFIED_CONTROLLER_COMMON_PARAMS  commonParams;
    CAPI_UNIFIED_CONTROLLER_UNIQUE_PARAMS  uniqueParams[CAPI_MAX_NUM_CONTROLLERS];
} CAPI_UNIFIED_CONTROLLER_PARAMS;
```

Table 4-49. CAPI_UNIFIED_CONTROLLER_PARAMS fields.

Parameter	Description
commonParams	Parameters that are common to both controllers in a dual-controller system.
uniqueParams	Parameters that can be different between the two controllers in a dual-controller system.

CAPI_UNIFIED_CONTROLLER_UNIQUE_DATA

NEW! in CAPI 3.4

This structure is used for Unified CAPI as part of the data that are gotten with CAPI_U_GetControllerData. This struct contains data that are different for each controller. The members of this struct are read-only except for *pendingControllerUniqueParams*, which is settable with CAPI_U_SetControllerParams. See comments in the struct, below, for information about what structures the equivalent non-unified variables are in, then see those structs for details of the members.

When a failover occurs, the failed controller cannot provide this data, of course, but some of this data is maintained on the working controller and this data is returned in this structure for the failed controller. For example, if your CAPI app is communicating with the A controller and the B controller is failed, then the B copy of this struct will contain some valid members. The members that are valid for the failed controller are:

controllerStatus
serialNumber
firmwareRevision
loaderRevision
model
aaVersion
currentNodeWWN
hostChannelUniqueData[<all channels>].i.fibreInfo.FCPortWWN
currentControllerUniqueParams.channelUniqueParams.environUnitNum[<all LUNs>]
currentControllerUniqueParams.channelUniqueParams.capiUnitNum

```
typedef struct
{
    /*
     * The following variable is equivalent to the variable in CAPI_CONTROLLER
     * with the same name.
     */
    CAPI_U32                configSequenceNumber;


    /*
     * The members of this structure are equivalent to members of
     * CAPI_NETWORK_INTERFACE and CAPI_ADVANCED_NETWORK_INTERFACE that are
     * unique for each controller board.
     */
    CAPI_NETWORK_INTERFACE_UNIQUE_DATA  netIfUniqueData;


    /*
     * The following variables are equivalent to variables with the same names
     * that are in the CAPI_RAID struct.
     * (Other variables equivalent to ones in CAPI_RAID are common and so are in
     * CAPI_UNIFIED_CONTROLLER_COMMON_DATA.)
     */
    CAPI_U32                numDrives;
    CAPI_U32                numArrays;
    /* (End of variables equivalent to ones in CAPI_RAID.) */


    /*
     * Failover information for this controller.
     * The following variables are equivalent to variables in CAPI_FAILOVER
     * with the same name, except where the names have changed as noted in
     * the comments.
     * Note that most of the contents of that struct do not need to be included
     * here since the information for the other controller is available in
     * the appropriate substructures included in this struct
     * (CAPI_UNIFIED_CONTROLLER_UNIQUE_DATA) and there is a complete copy
```

```
     * of this structure maintained in CAPI_UNIFIED_CONTROLLER for both the
     * A and B controller.
     */

    /* This controller's ID (A or B).
     * (Equivalent to failoverId in CAPI_FAILOVER.) */
    CAPI_CONTROLLER_ID      controllerId;

    /* This controller's status (up, down, or unknown).
     * (Equivalent to otherState in CAPI_FAILOVER except it is the status of
     * THIS controller, not the other.) */
    CAPI_CONTROLLER_STATUS  controllerStatus;

    /* TRUE if the other controller has failed and this controller has taken
     * over its responsibilities.
     * Note that there will be a delay between when the controllerStatus of
     * the other controller goes to 'down' and this value is set to TRUE,
     * which represents the time that it takes for this controller to take
     * over the other controller's responsibilities. */
    CAPI_BOOL               failedOver;

    /* If failed over, what happened. */
    CAPI_FR_FAILOVER_REASON failoverReason;

    /* (End of variables equivalent to ones in CAPI_FAILOVER.) */


    /*
     * The following variables are equivalent to variables in CAPI_CONTROLLER
     * with the same name.
     */
    CAPI_CAPABILITY         capabilities;
    CAPI_CAPABILITY         capabilities2;
    CAPI_CAPABILITY         capabilities3;
    CAPI_CAPABILITY         spareCapabilities[5];
    /* Note that by including the following struct here, there is no need for a
     * "unified" version of CAPI_GetAdvancedEnvironmentals since this structure
     * can be gotten with CAPI_U_GetControllerData. */
    CAPI_ADVANCED_CONTROLLER_ENVIRONMENTALS  advancedEnvironmentals;
    CAPI_CHANNEL_UNIQUE_DATA
                     hostChannelUniqueData[CAPI_MAX_HOST_CHANNELS_PER_CONTROLLER];
    CAPI_CHANNEL_UNIQUE_DATA
                     driveChannelUniqueData[CAPI_MAX_DRIVE_CHANNELS_PER_CONTROLLER];
    CAPI_CHAR               manufacturer     [CAPI_MAX_STRING];
    CAPI_CHAR               model            [CAPI_MAX_STRING];
    CAPI_CHAR               firmwareRevision [CAPI_MAX_STRING];
    CAPI_CHAR               baselevelRevision[CAPI_MAX_STRING];
    CAPI_CHAR               boardRevision    [CAPI_MAX_STRING];
    CAPI_CHAR               cpldRevision     [CAPI_MAX_STRING];
    CAPI_CHAR               cpld2Revision    [CAPI_MAX_STRING];
    CAPI_CHAR               loaderRevision   [CAPI_MAX_STRING];
    CAPI_U8                 serialNumber     [CAPI_MAX_SERIAL_NUMBER_BYTES];
    CAPI_U8                 serialNumberLength;
    CAPI_U32                aaVersion;
    CAPI_U8                 backplaneType;
    CAPI_U8                 daughterBoard0Type;
    CAPI_U8                 daughterBoard1Type;
    CAPI_U8                 currentNodeWWN[CAPI_FC_WWID_SIZE];
    CAPI_U8                 sfpPresent;
    CAPI_U8                 hostRXSignalOK;
    CAPI_U8                 hostTXSignalOK;
    /* (End of variables equivalent to ones in CAPI_CONTROLLER.) */


    /*
     * Controller parameter information that may have been updated by CAPI,
     * and therefore does not reflect the current, in-use parameters.  This
     * is the "pending" configuration which will take effect on a reboot.
     */
    CAPI_UNIFIED_CONTROLLER_UNIQUE_PARAMS  pendingControllerUniqueParams;

    /*
```

```
     * Controller Parameter information for the currently executing
     * configuration.  This is sometimes known as the "active" params.
     */
    CAPI_UNIFIED_CONTROLLER_UNIQUE_PARAMS  currentControllerUniqueParams;
} CAPI_UNIFIED_CONTROLLER_UNIQUE_DATA;
```

CAPI_UNIFIED_CONTROLLER_UNIQUE_PARAMS

NEW! in CAPI 3.4

This structure is used for Unified CAPI as part of the parameters that are passed with CAPI_U_SetControllerParams. This struct contains those parameters that are different for each controller. See comments in the structure, below. See CAPI_NETWORK_INTERFACE_UNIQUE_PARAMS and CAPI_CHANNEL_UNIQUE_PARAMS.

```
typedef struct
{
    /*
     * The members of this structure are equivalent to the settable members of
     * CAPI_NETWORK_INTERFACE and CAPI_ADVANCED_NETWORK_INTERFACE that are
     * unique for each controller board.
     */
    CAPI_NETWORK_INTERFACE_UNIQUE_PARAMS  netIfUniqueParams;


    /*
     * The following struct contains variables that are equivalent to variables
     * with the same names that are in the CAPI_CHANNEL_PARAMS struct.
     * However, note that the per-channel parameters are in an array here.
     */
    CAPI_CHANNEL_UNIQUE_PARAMS  channelUniqueParams;

} CAPI_UNIFIED_CONTROLLER_UNIQUE_PARAMS;
```

CAPI_UNIFIED_CREATE_ARRAY_SERIAL_NUMBER_STRUCT

NEW! in CAPI 3.4

The CAPI_UNIFIED_CREATE_ARRAY_SERIAL_NUMBER_STRUCT is used in the callback to the CAPI_U_CreateArray function.

```
typedef struct
{
    CAPI_U8     arraySerialNumber[CAPI_MAX_SERIAL_NUMBER_BYTES];
} CAPI_UNIFIED_CREATE_ARRAY_SERIAL_NUMBER_STRUCT;
```

Table 4-50. CAPI_UNIFIED_CREATE_ARRAY_SERIAL_NUMBER_STRUCT fields.

Parameter	Description
arraySerialNumber	The array serial number assigned by the controller. See the description of CAPI_U_CreateArray for details of the makeup of the serial number.

CAPI_UNIFIED_CREATE_ARRAY_STRUCT

NEW! in CAPI 3.4

The CAPI_UNIFIED_CREATE_ARRAY_STRUCT is used by CAPI_U_CreateArray and CAPI_U_ExpandArray. The CAPI_CreateArray function uses separate parameters that are copied into CAPI_ADD_ARRAY_STRUCT to be passed to the controller, but the corresponding unified commands use this structure instead and it incorporates the old structure.

```
typedef struct
{
    CAPI_ADD_ARRAY_STRUCT  oldAddArray;
    CAPI_CONTROLLER_ID     preferredOwner;
    CAPI_U8                arraySerialNumber[CAPI_MAX_SERIAL_NUMBER_BYTES];
    CAPI_U8   driveList[CAPI_MAX_DRIVES_PER_ARRAY][CAPI_NUM_DRIVE_IDENTIFIER_BYTES];
} CAPI_UNIFIED_CREATE_ARRAY_STRUCT;
```

Table 4-51. CAPI_UNIFIED_CREATE_ARRAY_STRUCT fields.

Parameter	Description
oldAddArray	See definition of the CAPI_ADD_ARRAY_STRUCT structure.
preferredOwner	Deprecated. Not used. Specify the preferred owner with *oldAddArray.preferredOwner*.
arraySerialNumber	Used by CAPI_U_ExpandArray to identify the array to expand. (Not used for CAPI_U_CreateArray.)
driveList	Array of drive serial numbers. &&&& TRUE? Why isn't this subscripted by CAPI_MAX_SERIAL_NUMBER_BYTES instead of creating the new #define of CAPI_NUM_DRIVE_IDENTIFIER_BYTES? Note that CAPI_MAX_DRIVES_PER_ARRAY includes the dedicated spares. The number of drives in driveList must be *oldAddArray.numDrives* + *oldAddArray.numSpares*. The first drives in the list must be the drives to use in the array, and the last drives in the list must be the spare drives.

CAPI_UNIFIED_DRIVE NEW! in CAPI 3.4

The CAPI_UNIFIED_DRIVE struct was created because we ran out of reserved bytes in CAPI_DRIVE and needed to add arraySerialNumber for Unified CAPI.

```
typedef struct
{
    CAPI_DRIVE            oldCapiDrive;
    CAPI_U8               arraySerialNumber[CAPI_MAX_SERIAL_NUMBER_BYTES];
} CAPI_UNIFIED_DRIVE;
```

Table 4-52. CAPI_UNIFIED_DRIVE fields.

Parameter	Description
OldCapiDrive	See definition of CAPI_DRIVE.
arraySerialNumber	If this drive has a *howUsed* value of CAPI_DRIVE_MEMBER_OF_ARRAY or CAPI_DRIVE_DEDICATED_SPARE then this is the serial number of the array that the drive is a member of or a dedicated spare for; otherwise, this is all-zeros.

CAPI_UNIFIED_KNOWN_HOSTS NEW! in CAPI 3.4

The CAPI_UNIFIED_KNOWN_HOSTS struct is the same as the CAPI_KNOWN_HOSTS structure except that is is twice as big since it is used for returning a combined list of known hosts from controllers A and B to Unified CAPI apps.

```
typedef struct
{
    CAPI_U8               numHosts;
    CAPI_HOST_DESCRIPTOR  host[CAPI_MAX_HOST_TABLE * 2];
} CAPI_UNIFIED_KNOWN_HOSTS;
```

Table 4-53. CAPI_UNIFIED_KNOWN_HOSTS fields.

Parameter	Description
numHosts	The number of hosts in the list.
host	The list of hosts that are known to the controller.

CAPI_UNIT_MAP

The CAPI_UNIT_MAP structure is used to map front-end SCSI Logical Unit Numbers to back-end devices or RAID array partitions. This structure is used by CAPI_SetAdvancedUnitMapping and CAPI_GetAdvancedUnitMapping. See the capability bits to see if this functionality is supported by the target Chaparral product.

```
typedef struct
{
    CAPI_U16        hostChannelIndex;
    CAPI_U16        deviceChannelIndex;
    CAPI_FLEX_ID    hostId;
    CAPI_FLEX_ID    deviceId;
    CAPI_U8         arraySerialNumber[CAPI_MAX_SERIAL_NUMBER_BYTES];
    CAPI_U32        startLba;
    CAPI_U32        startLbaHi;
    CAPI_U32        size;
    CAPI_U32        sizeHi;
    CAPI_U8         mappingMode;
    CAPI_U8         lunMask;
} CAPI_UNIT_MAP;
```

Table 4-54. CAPI_UNIT_MAP fields.

Parameter	Description
hostChannelIndex	Front-end channel for which this LUN will be mapped
hostId	Flexible CAPI identifier that describes how the host channel is mapped
deviceChannelIndex	Back-end device channel, if mapping a device(not used for RAID array)
deviceId	Flexible CAPI identifier that describes a device (not used for RAID array)
arraySerialNumber	Serial number of a RAID array ***partition*** (not for mapping devices). This field is *not* currently supported by any Chaparral products, although it may be supported in the future (see capability bits).
startLba	Currently unused. May be supported in future products (see capability bits).
startLbaHi	Currently unused. May be supported in future products (see capability bits).
size	Currently unused. May be supported in future products (see capability bits).
sizeHi	Currently unused. May be supported in future products (see capability bits).
mappingMode NEW! in CAPI 3.4	Auto versus fixed. Used for routers.
lunMask NEW! in CAPI 3.4	0 = not masked, 1 = masked

◇ ◇ ◇ 5

CAPI FUNCTION REFERENCE

This section provides detailed descriptions of each of the CAPI functions. The non-unified functions are listed first, in alphabetical order, followed by the unified functions, also in alphabetical order. See page 5 for an introduction to Unified CAPI.

See Chapter 6, *Reply Code Reference*, and Chapter 7, *Event Code Reference*, for details on specific reply and event codes.

The *Callback* section of each function description provides details of which parameters of the callback function are valid and what they contain. See *Reply to Function Calls* on page 6.

NOTE: *CAPI_RC is used in this chapter as an abbreviation for CAPI_RETURN_CODE.*

The following table describes the attributes used to characterize each CAPI function. Each function includes a table of these attributes. A check mark indicates that the attribute applies to that function.

Attribute	Description
Lengthy Operation	Specifies if the function is a lengthy operation. See Lengthy Operations on page 7.
Need Current Configuration	The application requires current configuration information for the operation to succeed. If a function is called and configuration is not current, the callback function will receive an errorCode of CAPI_ERROR_FAILURE_DUE_TO_CONFIG_CHANGE. See Controller Structure Updates on page 9.
May Change Configuration	The function may change the controller's current configuration. In most cases, this means that if the function succeeds it will increment the configuration sequence number. See Controller Configuration Sequence Number on page 10.
See Capability Bits	See CAPI Capabilities on page 29 and refer to the controller's documentation to determine if the function is supported.

Note for CAPI 2.x users: The word SAFTE has been changed to ENVIRON to include other environmental processors, also known as Enclosure Management Processors or EMPs.

Abort Utility

Syntax:

```
CAPI_RC  CAPI_AbortUtility( CAPI_HANDLE  handle,
                            CAPI_U32     arrayIndex );
```

Description:

Aborts the configuration/management utility running on the specified array.

handle is the handle of the controller that executes the command.
arrayIndex is the index of the target array for which the utility should be aborted.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_UTILITY_ABORT**
errorCode	Completion status of the command.
identifier	arrayIndex and controllerHandle are valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_UTILITY_ABORT

Remarks:

Each RAID array can have a maximum of one configuration or management utility running at a time. This function aborts the utility; however, not all utilities may be aborted. See *CAPI Versions*
Different Chaparral products support different versions of CAPI. If the major version (for example, the 3 in version 3.4) is the same for two products, you should be able to easily design a single CAPI app that manages both products, although the product with the lower minor version number (for example, the 4 in version 3.4) will not have all the features of the product with the higher minor version number. Specifically, there may be additional CAPI commands added for a higher version number and there may be additional members in some of the data structures passed to and from CAPI commands, but the members that existed in the lower version of CAPI are still in the same locations in the same structures.

When a new version of the CAPI SDK is released, typically the most recent Chaparral product(s) are released simultaneously with the SDK and the version of CAPI that is running in the controller corresponds to the SDK version. However, developers of new CAPI apps can use a more recent version of the CAPI SDK to talk to older products, provided the major version number matches, and, of course, the app cannot use the newest features that have been added to CAPI which are not supported on that product.

Conversely, if a CAPI app was developed using an older version of the CAPI SDK (for example, CAPI 3.2), it can still be used for managing a newer product that supports a newer version of CAPI (for example, CAPI3.4), but of course the app will not be able to take advantage of the newer features that have been added to CAPI 3.4 in the product but which are not in the CAPI 3.2 SDK.

As of this writing (September 2002), the following products support the corresponding version of CAPI:

- CAPI 2.x: G5312, G7313, all Kxxxx.
- CAPI 3.4: RIO, Stratis RAID S3300 (JFF224). (Thus, the features marked “NEW! in CAPI 3.3” and “NEW! in CAPI 3.4” in this document are supported by these products only.)

- CAPI 3.2: All other Chaparral products.

CAPI Capabilities on page 29.

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

Add Array Partition

Syntax:

```
CAPI_RC  CAPI_AddArrayPartition( CAPI_HANDLE              handle,
                                 CAPI_U8                 *arraySerialNumber,
                                 CAPI_PARTITION_REQUEST  *addPartition );
```

Description:

Adds (i.e., creates) a new partition to an existing array.

handle is the handle of the controller that executes the command.
arraySerialNumber (*Not used* – the *arraySerialNumber* member of *addPartition* is used to specify the array serial number to which the partition will be added.)
addPartition is a pointer to the CAPI_PARTITION_REQUEST structure which is used to specify the characteristics of the partition to be created. All the members of CAPI_PARTITION_REQUEST must be specified except partitionSerialNumber.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_ADD_ARRAY_PARTITION**
errorCode	Completion status of the command.
identifier	controllerHandle and arrayIndex are valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_ADD_ARRAY_PARTITION_COMPLETE

Remarks :

This command is valid if the CAPI_CAPABILITY_2_ARRAY_PARTITIONS capability flag is set.

The maximum number of partitions supported by one array is given by CAPI_MAX_PARTITIONS_PER_ARRAY. The maximum number of partitions supported by a controller is given by CAPI_MAX_ARRAY_PARTITIONS_PER_CONTROLLER.

The partition serial number of the new partition is included with the event CAPI_EVENT_ADD_ARRAY_PARTITION_COMPLETE as u.serialNumbers.arraySerialNumber. This serial number can then be used as a parameter when calling other CAPI functions that require a partition serial number.

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_ChangeArrayPartitionGeometry()
CAPI_ChangeArrayPartitionLun()
CAPI_ChangeArrayPartitionName()

CAPI_DeleteArrayPartition()
CAPI_GetArrayPartitions()
CAPI_GetFreeArrayPartitions()
CAPI_ResetArrayPartitionStatistics()

Add Dedicated Spare

Syntax:

```
CAPI_RC  CAPI_AddDedicatedSpare( CAPI_HANDLE handle,
                                 CAPI_U32    arrayIndex,
                                 CAPI_U32    channelIndex,
                                 CAPI_U32    driveIndex );
```

Description:

This function adds an unused or free drive as a dedicated spare to a redundant array.

handle is the handle of the controller that executes the command.
arrayIndex is the index of the target array among the arrays on the specified controller.
channelIndex is the index of the channel on the specified controller.
driveIndex is the index of the drive on the specified channel.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_ADD_DEDICATED_SPARE**
errorCode	Completion status of the command.
identifier	arrayIndex and controllerHandle are valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_ADD_DEDICATED_SPARE

Remarks :

It is assumed that the calling routine has verified that the drive has sufficient capacity for the array. If the array has a down drive, a reconstruct utility immediately starts.

If a drive contains metadata from a previous array, you must clear the metadata using the CAPI_ScsiMaintenance or CAPI_U_DoScsiMaintenance command before adding the drive as a dedicated spare or pool spare. The controller will automatically rescan the bus when the metadata is cleared.

On some older RAID controller implementations, if a drive contains metadata from a previous array, you must clear the metadata using the CAPI_ScsiMaintenance command, *then issue a rescan by calling CAPI_RescanBus* before adding the drive as a dedicated spare or pool spare.

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_AddPoolSpare()
CAPI_DeleteSpare()
CAPI_ScsiMaintenance()

Add Host

Syntax:

```
CAPI_RC  CAPI_AddHost( CAPI_HANDLE   handle,
                       CAPI_U32      channelIndex,
                       CAPI_U8      *partitionSerialNumber,
                       CAPI_U32      unitNum,
                       CAPI_FLEX_ID  hostId,
                       CAPI_BOOL     allHosts,
                       CAPI_BOOL     accessMode );
```

Description:

This function adds a host to the list of hosts that may communicate with a specified *unitNum* or *partitionSerialNumber*. The list is either a list of hosts that are included for access to the LUN or a list of hosts that are excluded from access. The *allHosts* flag may be used to override the list and have all hosts either included or excluded.

handle is the handle of the controller that executes the command.
channelIndex host channel index that the array or device is being presented on.
partitionSerialNumber is the serial number of the partition; if partitions are not supported (capability bit CAPI_CAPABILITY_2_ARRAY_PARTITIONS not set), then this is an array serial number. (Applies to RAID only; not routers.)
unitNum LUN that this array or device is being presented as.
hostId Fibre Channel or SCSI ID of the host.
allHosts setting to TRUE causes the *accessMode* parameter to apply to all hosts; setting to FALSE causes the *accessMode* parameter to apply to this LUN's list of hosts that have access. (Applies to routers only; not RAID.)
accessMode setting to TRUE designates a list of hosts that are to be included for access; setting to FALSE designates a list of hosts that are to be excluded for access. (Applies to routers only; not RAID.)

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_ADD_HOST**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

Remarks:

See CAPI_U_AddHost for a discussion of how a typical application might best use this command.

Applications Errata for Router – The Router must be in FIXED mode or else the function will fail.

To change *allHosts* and *accessMode* for RAID products, use CAPI_ChangeInfoShieldType.

If *partitionSerialNumber* is not NULL, it will be used; if it is NULL, *channelIndex* and *unitNum* will be used.

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_U_AddHost()
CAPI_GetHostTable()
CAPI_RemoveHost()
CAPI_ChangeInfoShieldType()
CAPI_GetKnownHosts()
CAPI_GetHostNicknames()

Add Host Nickname NEW! in CAPI 3.3

Syntax:

```
CAPI_RC  CAPI_AddHostNickname( CAPI_HANDLE   handle,
                               CAPI_FLEX_ID  hostId,
                               CAPI_U8      *nickname );
```

Description:

This command allows a CAPI application to define a “nickname” that corresponds to the worldwide name for a host. This capability of CAPI is provided so a CAPI application can provide a mechanism for the user of that application to more conveniently refer to a host. The CAPI application can access these host nicknames via the CAPI_GetHostNicknames and CAPI_GetKnownHosts functions.

handle is the handle of the controller that executes the command.
hostId is the worldwide name of the host that this nickname applies to. In the CAPI_FLEX_ID struct, the CAPI_FLEX_TYPE may be set to either CAPI_FLEX_TYPE_FC_WWN_NODE or CAPI_FLEX_TYPE_FC_WWN_PORT and the corresponding field, *FCNodeWWN* or *FCPortWWN*, is then used.
nickname points to a null-terminated string provided by the CAPI application. Maximum number of characters allowed in this string is CAPI_MAX_HOST_NAME (15 characters plus NULL).

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_ADD_HOST_NICKNAME**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

Remarks:

This function can be used to change a nickname as well as add a new one.

Caution: This function performs no check that the nickname is unique. That is, it is possible for the same nickname to be used for two or more different worldwide names, with unpredictable results.

Note that nicknames can be added or changed via the Disk Array Administrator (MUI) or other user interfaces; there is a single table of nicknames. Thus, name changes and additions made via one user interface are visible via other user interfaces.

The list of nicknames is saved on both controllers in a dual-controller system. The list of nicknames is preserved through a reboot and through replacement of one of the two controller boards.

Nicknames can be deleted by using this function with the nickname defined as a null string (that is, first character in the string is 0).

This function requires capability bit CAPI_CAPABILITY_2_INFOSHIELD to be set.

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
✔	See Capability Bits

See also:

CAPI_GetHostNicknames()
CAPI_GetKnownHosts()

Add Pool Spare

Syntax:

```
CAPI_RC  CAPI_AddPoolSpare( CAPI_HANDLE handle,
                            CAPI_U32    channelIndex,
                            CAPI_U32    driveIndex );
```

Description:

This function adds an unused or free drive to the spare pool.

handle is the handle of the controller that executes the command.
channelIndex is the index of the channel on the specified controller.
driveIndex is the index of the drive on the specified channel.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_ADD_POOL_SPARE**
errorCode	Completion status of the command.
identifier	controllerHandle, arrayIndex, channelIndex, and driveIndex are valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_ADD_POOL_SPARE

Remarks :

It is assumed that the calling routine has verified that the drive has sufficient capacity for the array. If the array has a down drive, a Reconstruct utility immediately starts.

If a drive contains metadata from a previous array, you must clear the metadata using the CAPI_ScsiMaintenance or CAPI_U_DoScsiMaintenance command before adding the drive as a dedicated spare or pool spare. The controller will automatically rescan the bus when the metadata is cleared.

On some older RAID controller implementations, if a drive contains metadata from a previous array, you must clear the metadata using the CAPI_ScsiMaintenance command, *then issue a rescan by calling CAPI_RescanBus* before adding the drive as a dedicated spare or pool spare.

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_AddDedicatedSpare()
CAPI_DeleteSpare()
CAPI_ScsiMaintenance()

Blink Drive

Syntax:

```
CAPI_RC  CAPI_BlinkDrive( CAPI_HANDLE handle,
                          CAPI_U32    channelIndex,
                          CAPI_U32    driveIndex );
```

Description:

Blinks the drive activity light. The light is blinked by issuing a non-destructive command, such as a single sector read or a SCSI Test Unit Ready, at regular intervals.

handle is the handle of the controller that executes the command.
channelIndex is the index of the channel on the specified controller.
driveIndex is the index of the drive on the specified channel.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_DRIVE_BLINK**
errorCode	Completion status of the command.
identifier	controllerHandle, channelIndex, and driveIndex are valid.
param1	
param2	
dataPtr	

Events:

Remarks :

The controller continues blinking the drive light until a call to CAPI_UnblinkDrive.

	Lengthy Operation
✔	Need Current Configuration
	May Change Configuration
	See Capability Bits

See also:

CAPI_UnblinkDrive()

Cache Test

Syntax:

```
CAPI_RC  CAPI_CacheTest( CAPI_HANDLE  handle );
```

Description:

This command will test the controller's cache region.

handle is the handle of the controller that executes the command.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_CACHE_TESTED**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

Remarks :

Currently, this command in not implemented. It will return OK status, but do nothing.
When this is implemented, this command will clear the cache region. Make sure that the cache region has been flushed and that all I/O has been stopped.

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
	See Capability Bits

See also:

CAPI_FlushCache()

Change Array Name

Syntax:

```
CAPI_RC  CAPI_ChangeArrayName( CAPI_HANDLE   handle,
                               CAPI_U32      arrayIndex,
                               CAPI_CHAR    *name );
```

Description:

This command changes the array name.

handle is the handle of the controller that executes the command.
arrayIndex is the index of the target array on the specified controller.
name is a pointer to a NULL terminated string containing the new array name. Length must be less than or equal to CAPI_MAX_ARRAY_NAME.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_ARRAY_NAME_CHANGE**
errorCode	Completion status of the command.
identifier	controllerHandle and arrayIndex are valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_ARRAY_NAME_CHANGE

Remarks :

An error will occur if the string is too long.

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
	See Capability Bits

See also:

CAPI_CreateArray()

Change Array Partition Geometry

Syntax:

```
CAPI_RC CAPI_ChangeArrayPartitionGeometry(CAPI_HANDLE          controllerHandle,
                                          CAPI_U8             *partitionSerialNumber,
                                          CAPI_PARTITION_REQUEST *changePartition );
```

Description:

Changes the size of an existing array partition. Currently, the size of a partition may only be *increased*, not decreased.

handle is the handle of the controller that executes the command.
partitionSerialNumber is a pointer to the serial number of an existing partition.
changePartition is a pointer to the structure that is used to specify the new size of the partition. The members of this struct that must be specified are: *startLba* (must be the same as that specified when the partition was added), *sizeLba* (specifies the new size), and *arraySerialNumber.* The *partitionSerialNumber* member is filled in by the function; it copies the *partitionSerialNumber* function param to the *partitionSerialNumber* structure member. The *name* and *unitNum* members of this struct are ignored.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_ARRAY_PARTITION_GEOMETRY_CHANGE**
errorCode	Completion status of the command.
identifier	controllerHandle and arrayIndex are valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_ARRAY_PARTITION_GEOMETRY_CHANGE

Remarks :

This command is valid if the CAPI_CAPABILITY_2_ARRAY_PARTITIONS capability flag is set.

Note that the size of a partition may only be increased if the partition is immediately followed by a free partition area. If an array is expanded, this creates free space at the end of the array, allowing the last partition in an array to expand into this area.

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_AddArrayPartition()
CAPI_GetFreeArrayPartitions()

Change Array Partition LUN

Syntax:

```
CAPI_RC   CAPI_ChangeArrayPartitionLun( CAPI_HANDLE   controllerHandle,
                                        CAPI_U8      *partitionSerialNumber,
                                        CAPI_U8       lun );
```

Description:

Allows the application to change the LUN that a partition presents to the host.

handle is the handle of the controller that executes the command.
partitionSerialNumber is a pointer to the serial number of an existing partition.
lun is the new LUN value of the partition (this must be a currently unused LUN value).

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_ARRAY_PARTITION_LUN_CHANGE**
errorCode	Completion status of the command.
identifier	controllerHandle and arrayIndex are valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_ARRAY_PARTITION_LUN_CHANGE

Remarks :

This command is valid if the CAPI_CAPABILITY_2_ARRAY_PARTITIONS capability flag is set. No reboot is required for this change to take effect.

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_AddArrayPartition()

Change Array Partition Name

Syntax:

```
CAPI_RC  CAPI_ChangeArrayPartitionName( CAPI_HANDLE  controllerHandle,
                                        CAPI_U8     *partitionSerialNumber,
                                        CAPI_CHAR   *name );
```

Description:

Changes the name value of an existing array partition.

handle is the handle of the controller that executes the command.
partitionSerialNumber is a pointer to the serial number of an existing partition.
name is a pointer to a NULL terminated string containing the new partition name. Length must be less than or equal to CAPI_MAX_ARRAY_NAME.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_ARRAY_PARTITION_NAME_CHANGE**
errorCode	Completion status of the command.
identifier	controllerHandle and arrayIndex are valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_ARRAY_PARTITION_NAME_CHANGE

Remarks :

This command is valid if the CAPI_CAPABILITY_2_ARRAY_PARTITIONS capability flag is set.

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_AddArrayPartition()

Change InfoShield Type

Syntax:

```
CAPI_RC  CAPI_ChangeInfoShieldType( CAPI_HANDLE   handle,
                                    CAPI_U32      channelIndex,
                                    CAPI_U8      *partitionSerialNumber,
                                    CAPI_U32      unitNum,
                                    CAPI_BOOL     allHosts,
                                    CAPI_BOOL     include );
```

Description:

This function changes the type of access that a list of hosts has for a specified *unitNum* or *partitionSerialNumber*.

handle is the handle of the controller that executes the command.
channelIndex host channel index that the array or device is being presented on.
partitionSerialNumber is the serial number of the partition; if partitions are not supported (capability bit CAPI_CAPABILITY_2_ARRAY_PARTITIONS not set), then this is an array serial number. (Applies to RAID only; not routers.)
unitNum LUN that this array or device is being presented as.
allHosts setting to TRUE causes the *include* parameter to apply to all hosts; setting to FALSE causes the *include* parameter to apply to this LUN's list of hosts.
include setting to TRUE designates a list of hosts that are to be included for access; setting to FALSE designates a list of hosts that are to be excluded for access.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_CHANGE_INFOSHIELD_TYPE**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

Remarks:

The list of LUNs that applies when *allHosts* is FALSE is configured using the CAPI_AddHost and CAPI_RemoveHost commands.

If *partitionSerialNumber* is not NULL, it will be used; if it is NULL, *channelIndex* and *unitNum* will be used.

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_GetHostTable()
CAPI_AddHost()
CAPI_RemoveHost()

Change Utility Priority

Syntax:

```
CAPI_RC   CAPI_ChangeUtilityPriority( CAPI_HANDLE             handle,
                                      CAPI_U32                arrayIndex,
                                      CAPI_UTILITY_PRIORITY   priority );
```

Description:

Changes the priority of the utility running on the specified array.

handle is the handle of the controller that executes the command.
arrayIndex is the index of the target array on the specified controller.
priority is used to set the priority level of the utility running on the array.
Valid priorities are

CAPI_UTILITY_PRIORITY_HIGH	***0***
CAPI_UTILITY_PRIORITY_MEDIUM	***1***
CAPI_UTILITY_PRIORITY_LOW	***2***

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_ARRAY_UTIL_PRIORITY_CHANGE**
errorCode	Completion status of the command. **CAPI_ERROR_NO_UTILITY_TO_ABORT** may be returned if the is not a utility running on the array.
identifier	controllerHandle and arrayIndex are valid.
param1	
param2	
dataPtr	

Events:

Remarks :

This command may not be supported on current controller models.

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
	See Capability Bits

See also:

Clear Event Log

Syntax:

```
CAPI_RC  CAPI_ClearEventLog( CAPI_HANDLE  handle );
```

Description:

This command clears the non-volatile event log memory on the controller and resets the Event Log sequenceNumber.

handle is the handle of the controller that executes the command.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_LOG_CLEAR**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_LOG_CLEAR

Remarks :

This command should **only** be used to reset a controller to an empty log state before shipping to a customer. An application can clear its event log without actually clearing the event log on the controller by disregarding the last logged sequenceNumber and anything prior.

> ***WARNING:*** *This can cause problems for other attached applications currently polling for events.*

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
	See Capability Bits

See also:

CAPI_GetEvent()
CAPI_GetFirstEvent()
CAPI_GetLastEvent()

Create Array

Syntax:

```
CAPI_RC  CAPI_CreateArray( CAPI_HANDLE              handle,
                           CAPI_UTILITY_PRIORITY    priority,
                           CAPI_DRIVE_LOCATION     *driveList,
                           CAPI_U32                 numDrives,
                           CAPI_U32                 numDrivesPerLowLevelContainer,
                           CAPI_U32                 numSpares,
                           CAPI_RAID_LEVEL          raidLevel,
                           CAPI_U32                 minDriveSize,
                           CAPI_U32                 dataChunkSize,
                           CAPI_U32                 unitNum,
                           CAPI_CONTROLLER_ID       preferredOwner,
                           CAPI_FORMAT_TYPE         formatType,
                           CAPI_CHAR               *arrayName,
                           CAPI_CACHE_PARAMS       *cacheParams );
```

Description:

Creates a RAID array from a list of single drives.

handle is the handle of the controller that executes the command.
priority is not used.
driveList is a pointer to a list of CAPI_DRIVE_LOCATION structures that specify the member and spare drives in the array. The length of this list must equal *numDrives* plus *numSpares*.
numDrives specifies the number of member drives in the array.
numDrivesPerLowLevelContainer specifies the number of member drives in the lower-level container. This is only applicable to RAID 30 and RAID 50; a value of 0 can be used for other RAID levels. Lower-level containers are the underlying RAID 5 (for RAID 50) or RAID 3 (for RAID 30) arrays that are striped together to make a two-level RAID 50 or RAID 30 array. All of the lower-level containers within a two-level array must have the same number of drives.
numSpares is the number of spare drives assigned to the array. The last drives in the *driveList* are used as dedicated spares.
raidLevel specifies the type of array to create.
minDriveSize is the size of each member in the array, in 512-byte blocks. The size of the smallest drive in the array determines the maximum value for this field, but a smaller value may be used. A value of 0 uses the default (the smallest drive in the array).
dataChunkSize specifies the size, in KBytes, of the data chunk in a RAID 3, 4, or 5 array (chunk size is the stripe size on one drive). Must be one of: 16, 32, or 64.
unitNum If a valid unused LUN is specified, the array will be created with one partition that uses all of the space in the array (this is done for backward compatibility with CAPI applications that don't support array partitions). If CAPI_NULL_ID is specified, then the array will be created without any partitions; to use the free area in the array, partitions must be added using the CAPI_AddArrayPartition function.
preferredOwner specifies which controller should be the preferred owner of this array. NEW
formatType is one of the following;

CAPI_FORMAT_TYPE_NO_FORMAT	This will generate metadata but will leave all array partitioning and user data untouched.
CAPI_FORMAT_TYPE_ZERO_INIT_ONLY	This format type will zero all user data.
CAPI_FORMAT_TYPE_ZERO_AND_LOWLEVEL	This is unsupported.
CAPI_FORMAT_TYPE_ONLINE_INIT	This will zero the first 1KB of the user data and then initialize the array with good redundancy data. The array will be available for customer read/write access immediately.

arrayName specifies a NULL terminated string containing the name of the array. Names longer than CAPI_MAX_STRING (20 at this writing) will be truncated. If a valid LUN is specified, then the single partition created for the array will have the same name as the array.

cacheParams is not used. This should be set to NULL. Use CAPI_SetCacheParams, CAPI_U_SetCacheParams, CAPI_SetArrayPartitionCacheParams, or CAPI_U_SetArrayPartitionCacheParams to set cache parameters.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_CREATE_ARRAY_START**
errorCode	Completion status of the command.
identifier	controllerHandle and arrayIndex are valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_CREATE_ARRAY_START
CAPI_EVENT_CREATE_ARRAY_COMPLETE

Remarks :

The progress of the Create Array utility can be monitored by calling CAPI_GetPercentComplete. Completion status is obtained via calls to CAPI_GetLastEvent. The array serial number of the new partition is included with both events CAPI_EVENT_CREATE_ARRAY_START and CAPI_EVENT_CREATE_ARRAY_COMPLETE as u.serialNumbers.arraySerialNumber. This serial number can then be used as a parameter when calling other CAPI functions that require an array serial number. The array serial number can also be obtained from the CAPI_ARRAY struct returned by function CAPI_GetArrayList.

After event CAPI_EVENT_CREATE_ARRAY_START is logged, there is a slight delay (generally less than a second) before CAPI_GetArrayList will return the new array as part of its list of arrays. You can call other CAPI functions related to this array (such as CAPI_AddArrayPartition) once your app sees the new array in the list of arrays.

The array serial number is 12 bytes; 8 bytes is the controller serial number and 4 bytes is a timestamp. An example is shown here:

```
 0   1   2   3   4   5   6   7   8   9  10  11     byte #
--------------------------------------------------------
00  50  13  B0  30  00  00  00  2A  0F  58  3C     value
----------serial num----------  --time stamp--
```

In a typical application, this could be displayed as 0x005013B0300000002A0F583C.

In the Chaparral Disk Array Administrator and in the RAIDar web browser interface, only bytes 3-5 and 8-11 are displayed since bytes 0-2 are always 005013 and bytes 6 and 7 are always zeroes. Thus, the array serial number would display as B030002A0F583C.

In the case of a RAID 1, 10, 3, 4, or 5 array, the utility writes zeros to each LBA on each drive. The final step writes controller-specific information to the reserved sectors of each drive.

✔	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_DeleteArray()
CAPI_AddArrayPartition()

Delete Array

Syntax:

```
CAPI_RC  CAPI_DeleteArray( CAPI_HANDLE  handle,
                           CAPI_U8     *arraySerialNumber );
```

Description:

Removes information in the reserved sectors of an array's member drives so that they are no longer associated with a RAID array.

handle is the handle of the controller that executes the command.
arraySerialNumber is a pointer to the serial number of the target array on the specified controller.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_ARRAY_DELETE**
errorCode	Completion status of the command.
identifier	controllerHandle and arrayIndex are valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_ARRAY_DELETE

Remarks :

After completion of this utility, the array is no longer valid and is no longer visible to the host. The member drives become single, free drives that can be assigned for use in new arrays or as spare drives. The drives are not reformatted by this utility and are not visible to the host.

> ***Note:*** *CAPI will adjust the array indices of the remaining arrays after the CAPI_DeleteArray command so that they remain contiguous.*
>
> ***Note to CAPI 2.x users:*** *This differs from the CAPI2.x in that an array serial number is passed instead of an array index.*
>
> ***Warning****: All partitions contained in the array are automatically deleted when the array is deleted.*

	Lengthy Operation
	Need Current Configuration
✔	May Change Configuration
	See Capability Bits

See also:

CAPI_CreateArray()

Delete Array Partition

Syntax:

```
CAPI_RC  CAPI_DeleteArrayPartition( CAPI_HANDLE   controllerHandle,
                                    CAPI_U8      *partitionSerialNumber );
```

Description:

Permanently deletes an existing array partition. The area formerly occupied by the partition becomes a free partition area, which can be used for partition expansion or to add a new partition.

handle is the handle of the controller that executes the command.
partitionSerialNumber is a pointer to the serial number of an existing partition.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_DELETE_ARRAY_PARTITION**
errorCode	Completion status of the command.
identifier	controllerHandle and arrayIndex are valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_DELETE_ARRAY_PARTITION_COMPLETE

Remarks :

This command is valid if the CAPI_CAPABILITY_2_ARRAY_PARTITIONS capability flag is set. Note that once the partition is deleted, it *cannot* be recovered.

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_AddArrayPartition()
CAPI_GetArrayPartitions()

Delete Spare

Syntax:

```
CAPI_RC  CAPI_DeleteSpare( CAPI_HANDLE  handle,
                           CAPI_U32     channelIndex,
                           CAPI_U32     driveIndex );
```

Description:

This function changes the drive from spare drive to unused.

handle is the handle of the controller that executes the command.
channelIndex is the index of the channel of the target drive on the specified controller.
driveIndex is the index of the drive on the specified channel.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_ SPARE_DELETE**
errorCode	Completion status of the command.
identifier	controllerHandle, channelIndex, and driveIndex are valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_ SPARE_DELETE

Remarks :

The drive becomes an available drive, which can be assigned for use in new arrays or as another spare drive. This command can be used to delete both pool spares and dedicated spares.

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
	See Capability Bits

See also:

CAPI_AddDedicatedSpare()
CAPI_AddPoolSpare()

Down Drive

Syntax:

```
CAPI_RC  CAPI_DownDrive( CAPI_HANDLE  handle,
                          CAPI_U32     channelIndex,
                          CAPI_U32     driveIndex );
```

Description:

Disables a drive that is a member of an array and can cause the array to switch to degraded operation.

handle is the handle of the controller that executes the command.
channelIndex is the index of the channel on the specified controller that the drive is on.
driveIndex is the index of the drive on the specified channel.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_DOWN_DRIVE**
errorCode	Completion status of the command.
identifier	controllerHandle and driveIndex are valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_DRIVE_DOWN

Remarks:

This command should only be used for system testing. It will degrade an array to a critical state if one of the member drives is downed. Remember, after downing a drive, to use it again you must clear the metadata on the drive (with CAPI_ScsiMaintenance) and then rescan the bus.

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
	See Capability Bits

See also:

CAPI_RescanBus()
CAPI_ScsiMaintenance()

Enable Packet Compression

Syntax:

```
CAPI_RC  CAPI_EnablePacketCompression( CAPI_U8  *compressionBuffer );
```

Description:

Enables compression of data sent by a controller in reply to a CAPI command.

compressionBuffer is a pointer to a buffer of size CAPI_RECEIVE_GENERAL_BUFFER_SIZE.

Return Code:

Always returns CAPI_STATUS_GOOD.

Callback:

replyCode	None
errorCode	
identifier	
param1	
param2	
dataPtr	

Events:

Remarks :

This command will not invoke the application's callback function.

CAPI uses the HSZRLE (Horvath Simplified Zero Run Length Encoding) compression algorithm which compresses repeating zeros.

To disable compression, call this function with compressionBuffer set to NULL. This is the default state if your application never calls this function.

Use of compression for serial LMXs is recommended; this will greatly improve response time to commands that get large amounts of data such as CAPI_UpdateController, CAPI_U_GetControllerData, CAPI_GetDriveList, and CAPI_U_GetDriveList.

See *Initialization Details* on page 15.

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
	See Capability Bits

See also:

CAPI_EnablePacketCompressionMasterToSlave()

Enable Packet Compression Master To Slave

NEW! in CAPI 3.4

Syntax:

```
CAPI_RC  CAPI_EnablePacketCompressionMasterToSlave(
                                        CAPI_BOOL  enableCompressionMasterToSlave );
```

Description:

Enables compression of data sent from a CAPI app (master) to a controller (slave).

enableCompressionMasterToSlave if set to TRUE will enable compression; if set to FALSE will disable compression.

Return Code:

Always returns CAPI_STATUS_GOOD.

Callback:

replyCode	None
errorCode	
identifier	
param1	
param2	
dataPtr	

Events:

Remarks :

This command will not invoke the application's callback function.

CAPI uses the HSZRLE (Horvath Simplified Zero Run Length Encoding) compression algorithm which compresses repeating zeros.

Use of compression for serial LMXs is recommended; this will greatly improve response time to commands that send large amounts of data such as CAPI_SetControllerParams, CAPI_U_SetControllerParams, CAPI_ScsiMaintenance, and CAPI_U_DoScsiMaintenance.

This command *must not* be called to set enableCompressionMasterToSlave to TRUE if capability bit CAPI_CAPABILITY_3_MASTER_TO_SLAVE_COMPRESSION is not set. If this is done, compressed data will be sent to the controller but the controller will not be able to uncompress it and serious consequences may result; for example, garbage configuration will be loaded into the controller if this is done when calling CAPI_SetControllerParams.

The same compression buffer is used for both compression and uncompression. Thus, you must call CAPI_EnablePacketCompression to provide a buffer in order to have master-to-slave compression work.

See *Initialization Details* on page 15.

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
✔	See Capability Bits

See also:

CAPI_EnablePacketCompression()

Environ Read

Syntax:

```
CAPI_RC  CAPI_EnvironRead( CAPI_HANDLE  handle,
                           CAPI_U32     environProcessorIndex,
                           CAPI_U32     environCommand );
```

Description:

Requests data from an environmental processor (for either the SAF-TE or SES standard) attached to a controller.

handle is the handle of the controller that executes the command.
environProcessorIndex is the index of the environmental processor you are issuing the command to. This is the same as the index used in the CAPI_FindNextEnvironProcessor( ) function.
environCommand is the environmental command code. See list of valid commands below.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_ENVIRON_READ**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	The number of valid bytes of data in the data buffer pointed to by dataPtr.
param2	
dataPtr	Pointer to **CAPI_ENVIRON_PROCESSOR_DATA**.

Events:

Remarks :

param1 will be less than or equal to CAPI_ENVIRON_MAX_ENVIRON_DATA_LENGTH.

If errorCode is equal to CAPI_NO_ERROR, then the data buffer contains valid inquiry data. However, if it is equal to CAPI_ERROR_COMMAND_FAILED, then sense data is automatically returned; the first byte in the data buffer contains the SCSI status byte and the rest of the data buffer contains SCSI sense data.

In this document, the terms “environmental processor,” “environmental device,” “environmental unit,” “Enclosure Management Processor,” and “EMP” are used interchangeably.

Chaparral enclosure management is intended for disk array enclosures that comply with either of the following two standards for enclosure services:

- SAF-TE (SCSI Accessed Fault-Tolerant Enclosure) – commonly used in SCSI/SCSI RAID enclosures.
- SES (SCSI-3 Enclosure Services) – an ANSI standard used widely for Fibre/Fibre RAID controllers and for SCSI-ATA and Fibre-ATA RAID controllers.

Each of these two enclosure services use different terminology for the Enclosure Management Processors (EMPs) that provide the enclosure services:

- SEP (SAF-TE Enclosure Processor for SAF-TE)
- ESP (Enclosure Services Processor for SES)

The following SAF-TE commands are valid for the *environCommand* parameter above.

Command
SAFTE_READ_ENCLOSURE_CFG_CMD
SAFTE_READ_ENCLOSURE_STATUS_CMD
SAFTE_READ_USAGE_STATS_CMD
SAFTE_READ_DEV_INSERTIONS_CMD
SAFTE_READ_DEV_SLOT_STATUS_CMD
SAFTE_READ_GLOBAL_FLAGS_CMD

Read Enclosure Configuration should be issued first before issuing any other SAF-TE reads. Refer to the SAF-TE Specification for more details. Also note that some SEP vendors do not support all of the commands listed and may return error codes.

The following SES commands are valid for the *environCommand* parameter above:

Command
SES_RECV_SUPPORTED_DIAGS
SES_RECV_CONFIGURATION
SES_RECV_ENCLOSURE_STATUS
SES_RECV_HELP_TEXT
SES_RECV_STRING_IN
SES_RECV_THRESHOLD_IN
SES_RECV_ARRAY_STATUS
SES_RECV_ELEMENT_DESCRIPTOR
SES_RECV_SHORT_ENCLOSURE_STAT

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
✔	See Capability Bits

See also:

CAPI_FindNextEnvironProcessor()
CAPI_EnvironWrite()

Environ Write

Syntax:

```
CAPI_RC  CAPI_EnvironWrite( CAPI_HANDLE   handle,
                            CAPI_U32      environProcessorIndex,
                            CAPI_U32      environCommand,
                            CAPI_U8      *buffer,
                            CAPI_U32      length );
```

Description:

Sends data to an environmental processor (for either the SAF-TE or SES standard) attached to a controller.

handle is the handle of the controller that executes the command.
environProcessorIndex is the index of the environmental processor you are issuing the command to. This is the same as the index used in the CAPI_FindNextEnvironProcessor function.
environCommand is the environmental command code. See list of valid commands below.
buffer is a pointer to buffer containing the CAPI_ENVIRON_PROCESSOR_DATA structure.
length is the number of bytes to send to the EMP from the *buffer.*

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

A return code of CAPI_STATUS_INVALID_PARAM will be returned if *length* is greater than sizeof(CAPI_ENVIRON_PROCESSOR_DATA).

Callback:

replyCode	**CAPI_REPLY_ENVIRON_WRITE**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Remarks :

In this document, the terms “environmental processor,” “environmental device,” “environmental unit,” “Enclosure Management Processor,” and “EMP” are used interchangeably.

The following SAF-TE commands are valid for the *environCommand* parameter above:

Command
SAFTE_WRITE_DEV_SSLOT_STATUS_CMD
SAFTE_SET_SCSI_ID_CMD
SAFTE_PERFORM_SLOT_OPERATION_CMD
SAFTE_SET_FAN_SPEED_CMD
SAFTE_ACTIVATE_POWER_SUPPLY_CMD
SAFTE_SEND_GLOBAL_FLAGS_CMD

Note: The *buffer* parameter points to the structure that contains the write buffer command data only. It does not contain the write buffer Operation Code in the first byte as described in the SAF-TE Interface Specification. The Operation Code is inserted by the controller before the actual command is sent to the SEP, using the *environCommand* parameter.

The following SES commands are valid for the *environCommand* parameter above:

Command
SES_SEND_ENCLOSURE_CONTROL
SES_SEND_STRING_OUT
SES_SEND_THRESHOLD_OUT
SES_SEND_ARRAY_CONTROL

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
✔	See Capability Bits

See also:

CAPI_FindNextEnvironProcessor()
CAPI_EnvironRead()

Expand Array

Syntax:

```
CAPI_RC  CAPI_ExpandArray( CAPI_HANDLE            handle,
                           CAPI_U32               arrayIndex,
                           CAPI_DRIVE_LOCATION   *driveList,
                           CAPI_U32               numDrives,
                           CAPI_U32               numSpares );
```

Description:

This function adds a new drive to an existing array and begins online capacity expansion to increase the size of the array. The original array is indicated by the arrayIndex.

handle is the handle of the controller that executes the command.
arrayIndex is the index of the target array on the specified controller.
driveList is a pointer to a list of CAPI_DRIVE_LOCATION structures that specify the member and spare drives to be added. The length of this list must equal numDrives plus numSpares.
numDrives specifies the number of member drives to be added.
numSpares specifies the number of spare drives assigned to this array, which are at the end of the driveList.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_EXPAND_ARRAY_START**
errorCode	Completion status of the command.
identifier	controllerHandle, arrayIndex, channelIndex, and driveIndex are valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_EXPAND_ARRAY_START
CAPI_EVENT_EXPAND_ARRAY_COMPLETE

Remarks :

This function may not be supported by all external RAID controllers.

> **Note:** *The new drives must be at least as large as the smallest existing member drive in the array.*

✔	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

Find LMX Of Type

Syntax:

```
CAPI_RC  CAPI_FindLmxOfType( CAPI_HANDLE                *handle,
                             CAPI_CONTROLLER_CONTEXT    *context,
                             CAPI_U8                    *capiBuffer,
                             CAPI_U8                    *eventBuffer,
                             CAPI_U8                     linkType );
```

Description:

This command is like CAPI_FindNextController except that it will only return the first entry of the particular type in the LmxTable (Master Table).

handle CAPI returns the controller handle for the found controller. This number is then used as the *handle* param for subsequent CAPI function calls. If no controller is found, then CAPI_NULL_ID is returned.
context Allocate a CAPI_CONTROLLER_CONTEXT for this controller and pass a pointer to it. CAPI uses this struct internally to store link routing information.
capiBuffer Allocate and pass a pointer to a buffer for CAPI to receive message packets from the controller. The size of the buffer should be at least the size of CAPI_RECEIVE_GENERAL_BUFFER_SIZE.
eventBuffer This buffer is used to receive CAPI_EVENT structures. The application can use the same capiBuffer as above (pass the same pointer) or can allocate a new buffer. The size of the buffer should be at least the size of CAPI_RECEIVE_EVENT_BUFFER_SIZE.
linkType The type of LMX to find.

Return Code:

Indicates if the request was successful (by returning CAPI_STATUS_GOOD) or, if not, provides an error status.

Callback:

replyCode	None
errorCode	
identifier	
param1	
param2	
dataPtr	

Events:

Remarks :

This command will not invoke the application's callback function.

See *Initialization Details* on page 15.

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
	See Capability Bits

See also:

CAPI_FindNextController()

Find Next Controller

Syntax:

```
CAPI_RC   CAPI_FindNextController( CAPI_BOOL                  firstTime,
                                   CAPI_BOOL                  *lastTime,
                                   CAPI_HANDLE                *handle,
                                   CAPI_CONTROLLER_CONTEXT    *context,
                                   CAPI_U8                    *capiBuffer,
                                   CAPI_U8                    *eventBuffer );
```

Description:

Finds the next attached external controller.

firstTime Should be set to TRUE the first time this function is called. Subsequent calls should set this to FALSE.
lastTime CAPI returns TRUE if this is the last controller found.
handle CAPI returns the controller handle for the found controller. If no controller was found, then CAPI_NULL_ID is returned.
context Allocate a CAPI_CONTROLLER_CONTEXT for each controller and pass a pointer to it. CAPI uses this struct internally to store link routing information.
capiBuffer Allocate and pass a pointer to a buffer for CAPI to receive message packets from the controller. The size of the buffer should be at least the size of CAPI_RECEIVE_GENERAL_BUFFER_SIZE.
eventBuffer This buffer is used to receive CAPI_EVENT structures. The application can use the same capiBuffer as above (pass the same pointer) or can allocate a new buffer. The size of the buffer should be at least the size of CAPI_RECEIVE_EVENT_BUFFER_SIZE.

Return Code:

Indicates if the request was successful (by returning CAPI_STATUS_GOOD) or, if not, provides an error status.

Callback:

replyCode	None
errorCode	
identifier	
param1	
param2	
dataPtr	

Events:

Remarks :

This command will not invoke the application's callback function.

See *Initialization Details* on page 15.

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
	See Capability Bits

See also:

CAPI_FindLmxOfType()

Find Next Environ Processor

Syntax:

```
CAPI_RC  CAPI_FindNextEnvironProcessor( CAPI_HANDLE  handle,
                                        CAPI_U32     environProcessorIndex );
```

Description:

Finds environmental devices (also known as Enclosure Management Processors or EMPs) that may be attached to the controller. The information that is returned in the **CAPI_ENVIRON_PROCESSOR_INFO** structure is the standard SCSI inquiry data.

handle is the handle of the controller that executes the command.
environProcessorIndex is the index of the EMP you are trying to find. This is a zero-based sequential index, so on the first call to this function, set index to zero. For the next call, set index to one and so on. When the callback returns a value of CAPI_ERROR_NO_SUCH_ENVIRON_PROCESSOR, you are finished.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_FIND_NEXT_ENVIRON_PROCESSOR**
errorCode	**CAPI_ERROR_NO_SUCH_ENVIRON_PROCESSOR** means no more EMPs.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	If an EMP is found (i.e., as long as error code is not CAPI_NO_SUCH_ENVIRON_PROCESSOR), this points to a **CAPI_ENVIRON_PROCESSOR_INFO** structure.

Events:

Remarks :

In this document, the terms “environmental processor,” “environmental device,” “environmental unit,” “Enclosure Management Processor,” and “EMP” are used interchangeably.

Call this function with an increasing index value, starting at 0, until you receive an error code of CAPI_ERROR_NO_SUCH_ENVIRON_PROCESSOR. Use the found index values in the CAPI_EnvironRead and CAPI_EnvironWrite function calls.

This command issues a SCSI Inquiry command to each EMP. If the Inquiry succeeds, the Callback contains errorCode = CAPI_NO_ERROR and *u.inquiry* in the CAPI_ENVIRON_PROCESSOR struct contains valid inquiry data. In the unlikely event that the Inquiry fails, the callback contains errorCode = CAPI_ERROR_COMMAND_FAILED and *u.e* in the CAPI_ENVIRON_PROCESSOR struct contains valid status and sense data. In either case, the *empId, busId, targetId* and *lun* members of CAPI_ENVIRON_PROCESSOR are valid.

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
	See Capability Bits

See also:

CAPI_EnvironRead()
CAPI_EnvironWrite()

Force Offline NEW! in CAPI 3.3

Syntax:

```
CAPI_RC   CAPI_ForceOffline( CAPI_HANDLE         handle,
                             CAPI_MODULE_TYPE    moduleType,
                             CAPI_MODULE_INDEX   moduleIndex,
                             CAPI_U8             param3 );
```

Description:

Forces the replaceable module (FRU) offline. The module will carry out this request even if it affects performance (for example, putting one Data Manager offline in an active-active RAID system) and even if it affects availability (for example, putting a Data Manager offline in a RAID system when the other Data Manager is already offline). If the request affects availability, this command returns an error code indicating the problem, but that error code will be returned in param1, not in errorCode.

handle is the handle of the controller that executes the command.
moduleType is the type of FRU that is being put offline. At this writing, only CAPI_MODULE_TYPE_DM and CAPI_MODULE_TYPE_DG are supported.
moduleIndex identifies the specific module. This must be one of 0 through 3 for Data Gates. It must be CAPI_MODULE_A or CAPI_MODULE_B for Data Managers.
param3 is reserved for possible future use.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_FORCE_OFFLINE**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	error code that would have been returned if this was a call to CAPI_PutOffline
param2	
dataPtr	

Events:

Remarks:

If the specified Data Manager (DM) that is to be forced offline is the other DM (not the one processing this command), this is accomplished by asserting the hardware reset line of that DM board to kill it.

If the specified DM that is to be forced offline is the one processing this command, this is accomplished by asking the other DM to kill this DM by asserting the hardware reset line.

But if the specified controller board that is to be forced offline is the one processing this command and the other controller board is offline, this is accomplished by gracefully shutting down the controller board via software (equivalent to CAPI_ShutDownController or CAPI_PutOffline).

This function requires capability bit CAPI_CAPABILITY_3_REPLACEABLE_MODULE to be set.

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_PutOffline()
CAPI_PutOnline()
CAPI_ForceOnline()

Force Online NEW! in CAPI 3.3

Syntax:

```
CAPI_RC   CAPI_ForceOnline( CAPI_HANDLE         handle,
                            CAPI_MODULE_TYPE    moduleType,
                            CAPI_MODULE_INDEX   moduleIndex,
                            CAPI_U8             param3 );
```

Description:

Forces the replaceable module (FRU) online ungracefully. Putting a module online ungracefully means not running full diagnostics and not running compatibility checks to see if the hardware and firmware of the FRU are compatible with the other FRUs. **This command is only for Chaparral internal use and it is available only in beta builds, not in customer builds.**

handle is the handle of the controller that executes the command.
moduleType is the type of FRU that is being put offline. At this writing, only CAPI_MODULE_TYPE_DM and CAPI_MODULE_TYPE_DG are supported.
moduleIndex identifies the specific module. This must be one of 0 through 3 for Data Gates. It must be CAPI_MODULE_A or CAPI_MODULE_B for Data Managers.
param3 is reserved for possible future use.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_FORCE_ONLINE**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

Remarks:

This function requires capability bit CAPI_CAPABILITY_3_REPLACEABLE_MODULE to be set.

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_ForceOffline()
CAPI_PutOffline()
CAPI_PutOnline()

Free Cache

Syntax:

```
CAPI_RC   CAPI_FreeCache( CAPI_HANDLE   controllerHandle,
                          CAPI_U8      *arraySerialNumber );
```

Description:

Frees memory used by the write-back cache in the controller for a specific array. Discards any data that is not flushed to the drive.

handle is the handle of the controller that executes the command.
arraySerialNumber serial number of array with orphan data(from CAPI_EVENT_ORPHAN_DATA)

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_CACHE_FREE**
errorCode	Completion status of the command.
identifier	controllerHandle, arrayIndex, channelIndex, and driveIndex are valid.
param1	
param2	
dataPtr	

Events:

Remarks :

In the event of a catastrophic array failure (such as a multiple drive failure under RAID 5), or if an array is moved from one controller to another, the controller is unable to flush cached write data to the array. To make this memory available to other arrays, free cache causes this memory to be made free for use to other arrays. The data is not written to the disks and is permanently lost. Use CAPI_EVENT_ORPHAN_DATA to trigger this command.

> ***Note to CAPI 2.x users:*** *The serial number of the array instead of the unit number is passed as a parameter now.*

	Lengthy Operation
✔	Need Current Configuration
	May Change Configuration
	See Capability Bits

See also:

CAPI_FlushCache()
CAPI_SetCacheParams()

Get Advanced Environmentals

Syntax:

```
CAPI_RC  CAPI_GetAdvancedEnvironmentals( CAPI_HANDLE  handle );
```

Description:

This function allows environmental status to be gotten. This function was added because we ran out of room in the CAPI_CONTROLLER_ENVIRONMENTALS structure in the CAPI_CONTROLLER structure.

handle is the handle of the controller that executes the command.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_GET_ADV_ENVIRONMENTALS**
errorCode	Completion status of the command.
identifier	ControllerHandle is valid.
param1	
param2	
dataPtr	This points to a **CAPI_ADVANCED_CONTROLLER_ENVIRONMENTALS** structure.

Events:

Remarks:

This command is not needed if you are using Unified CAPI commands since the CAPI_ADVANCED_CONTROLLER_ENVIRONMENTALS structure is included in the CAPI_UNIFIED_CONTROLLER structure obtained with CAPI_U_GetControllerData.

If you are developing a non-unified CAPI application, note that there may be additional environmental data in the CAPI_PRODUCT_SPECIFIC_UNION, which is part of the CAPI_CONTROLLER structure and can be obtained with CAPI_UpdateController.

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
	See Capability Bits

See also:

Get Advanced Network Interface NEW!

Syntax:

```
CAPI_RC CAPI_GetAdvancedNetworkInterface( CAPI_HANDLE handle );
```

Description:

This function allows configuration parameters to be gotten for the LAN processor.

handle is the handle of the controller that executes the command.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_GET_ADV_NETWORK_INTF**
errorCode	Completion status of the command.
identifier	ControllerHandle is valid.
param1	
param2	
dataPtr	This points to a **CAPI_ADVANCED_NETWORK_INTERFACE** structure.

Events:

Remarks:

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
✔	See Capability Bits

See also:

CAPI_SetAdvancedNetworkInterface()

Get Advanced Unit Mapping

Syntax:

```
CAPI_RC  CAPI_GetAdvancedUnitMapping( CAPI_HANDLE  handle );
```

Description:

This function returns the mapping of back-end devices or arrays to front-end LUNs.

handle is the handle of the controller that executes the command.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_GET_ADVANCED_UNIT_MAPPING**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	Number of CAPI_UNIT_MAP structs returned.
param2	Configuration sequence number.
dataPtr	Pointer to the first element of an array of **CAPI_UNIT_MAP** structures; there are param1 elements in the array.

Events:

Remarks:

This command is currently supported only on the Router products. RAID controllers will return CAPI_ERROR_NOT_SUPPORTED when sent this command.

Applications Errata for Router - This function currently returns an array of 64 CAPI_UNIT_MAP structures. Valid device-to-LUN mappings are indicated to CAPI Clients as follows:

unitMap[lun].hostId.type = CAPI_FLEX_TYPE_LUN;
unitMap[lun].hostChannelIndex = 0;
unitMap[lun].hostId.unitNum = lun;
unitMap[lun].deviceId.type = CAPI_FLEX_TYPE_SCSI | CAPI_FLEX_TYPE_LUN;
unitMap[lun].deviceChannelIndex = DeviceLunMap[lun].addr.channel;
unitMap[lun].deviceId.deviceId = DeviceLunMap[lun].addr.scsiId;
unitMap[lun].deviceId.unitNum = DeviceLunMap[lun].addr.lun;

The Router LUN is indicated to CAPI Clients as follows:

unitMap[lun].hostId.type = CAPI_FLEX_TYPE_LUN;
unitMap[lun].hostChannelIndex = 0;
unitMap[lun].hostId.unitNum = lun;
unitMap[lun].deviceId.type = 0;
unitMap[lun].deviceChannelIndex = CAPI_LUN_UNASSIGNED;
unitMap[lun].deviceId.deviceId = CAPI_LUN_UNASSIGNED;
unitMap[lun].deviceId.unitNum = CAPI_LUN_UNASSIGNED;

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_SetAdvancedUnitMapping()

Get Array List

Syntax:

```
CAPI_RC  CAPI_GetArrayList( CAPI_HANDLE  handle,
                            CAPI_U8      bankNumber );
```

Description:

This function returns an array of CAPI_ARRAY structures.

handle is the handle of the controller that executes the command.
bankNumber is unused and should be set to 0.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_GET_ARRAY_LIST**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	Number of CAPI_ARRAY structs returned.
param2	Configuration sequence number.
dataPtr	Pointer to the first element of an array of **CAPI_ARRAY** structures; there are param1 elements in the array.

Events:

Remarks:

The application developer needs to make sure that the configuration sequence number on their copy of the array list (an array of CAPI_ARRAY structures retrieved with a call to CAPI_GetArrayList) matches the configuration sequence number on their copy of CAPI_CONTROLLER (retrieved with a call to CAPI_UpdateController). A CAPI_ERROR_FAILURE_DUE_TO_CONFIG_CHANGE will occur if a configuration change is attempted with incompatible structures.

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
✔	See Capability Bits

See also:

Get Array Partitions

Syntax:

```
CAPI_RC  CAPI_GetArrayPartitions( CAPI_HANDLE   handle,
                                  CAPI_U8      *arraySerialNumber );
```

Description:

Gets a list of partitions contained in the specified array.

handle is the handle of the controller that executes the command.
arraySerialNumber is a pointer to the serial number of the array which contains the partitions.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_GET_ARRAY_PARTITIONS**
errorCode	Completion status of the command.
identifier	controllerHandle and arrayIndex are valid.
param1	Number of CAPI_ARRAY_PARTITION structs returned.
param2	Configuration sequence number.
dataPtr	Pointer to the first element of an array of **CAPI_ARRAY_PARTITION** structures; there are param1 elements in the array.

Events:

Remarks :

This command is valid if the CAPI_CAPABILITY_2_ARRAY_PARTITIONS capability flag is set. The maximum number of partitions supported by one array is given by CAPI_MAX_PARTITIONS_PER_ARRAY.

	Lengthy Operation
✔	Need Current Configuration
	May Change Configuration
✔	See Capability Bits

See also:

CAPI_AddArrayPartition()

Get Config Sequence Number

Syntax:

```
CAPI_RC  CAPI_GetConfigSequenceNumber( CAPI_HANDLE  handle );
```

Description:

Replies with the controller's current configuration sequence number which can be used to determine if a controller structures update is required (CAPI_UpdateController, CAPI_GetDriveList, CAPI_GetArrayList).

handle is the handle of the controller that executes the command.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_GET_CONFIG_SEQ_NUMBER**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	Configuration sequence number.
param2	
dataPtr	

Events:

Remarks:

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
✔	See Capability Bits

See also:

Get Debug Data NEW! in CAPI 3.3

Syntax:

```
CAPI_RC  CAPI_GetDebugData( CAPI_HANDLE             handle,
                            CAPI_DEBUG_DATA_REGION  region,
                            CAPI_U32                debugDataOffset );
```

Description:

This command allows a CAPI application to get the debug data that has been logged in the controller. Debug data is logged by many parts of the controller software. Data is in ASCII text format and consists of printable characters plus space, tab, and new-line characters. Many lines start with a time stamp.

handle is the handle of the controller that executes the command.
region is the portion of the debug data to get.
debugDataOffset is the offset (in bytes, 0-based) at which to start retrieving the debug data in the controller.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_GET_DEBUG_DATA**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	The number of characters that have been put in the data buffer pointed to by dataPtr.
param2	
dataPtr	CAPI_CHAR *

Events:

Remarks:

The data is not null-terminated; use param1 to determine how much data is available. There may be garbage characters in the data buffer after the valid data.

Notes on using *debugDataOffset*

The area on a controller that is dedicated to saving debug data is typically several hundred kilobytes. It is not possible to get all of this data in one call to this function, because of size limitations of the data buffer in the LMX. The maximum size of a block of data that will be returned by a call to this function is CAPI_MAX_DEBUG_DATA_PER_GET (defined as 32768 as of this writing). Your CAPI application should call this function repeatedly (with the region set to the same value) until it returns with param1 set to a value that is less than CAPI_MAX_DEBUG_DATA_PER_GET. Each time you call this function, you should increase the value of *debugDataOffset* by CAPI_MAX_DEBUG_DATA_PER_GET, starting with 0. For example, if a particular controller has debug data in the boot-up region that has a total size of 70000 bytes, the first time your app calls this command, *debugDataOffset* should be set to 0 and the callback will contain 32768 characters and param1 will be 32768. The second time the app calls this command, *debugDataOffset* should be set to 32768 and the callback will contain 32768 characters and param1 will be 32768. The third time the app calls this command, *debugDataOffset* should be set to 65536. This call will get the remaining 4464 characters (70000 – 65536 = 4464). The callback will contain 4464 characters and param1 will be 4464. Your app should concatenate these 3 blocks of data for display to a user.

If the *debugDataOffset* is beyond the end of valid debug data, 0 characters will be put in the data buffer and param1 will be 0.

When this function is called with offset = 0, a snapshot copy is made of the debug data in the specified region. Subsequent calls to this function with offset != 0 will retrieve data from that snapshot buffer.

WARNING: If more than one application is calling this function at the same time, there is the potential for interaction between the applications and the data that it retrieved may not be the desired data. (This is because the large buffer sizes involved require that all CAPI apps share a single, global snapshot buffer.)

Organization of the debug data into regions

The debug data is organized into 6 separate regions. They are:
- Boot-up prints (region = CAPI_DEBUG_DATA_REGION_BOOT_LOG)
- 4 crash-dump regions (region = CAPI_DEBUG_DATA_REGION_CRASH_LOG1 through 4)
- General debug prints (region = CAPI_DEBUG_DATA_REGION_PRINT_LOG)

Note that a CAPI application should not assume a region is any particular size, since this will vary from product to product and may vary with future releases of a product. Instead, the application should keep asking for data until param1 indicates all data has been retrieved, as discussed above. But to give you some idea as to the size, as of this writing the boot-up region is 20480 bytes; the other regions are each 102400 bytes.

Each region fills up from the lowest address. If the buffer has not filled up, param1 will indicate how many bytes of data you have received, and this number may even be 0. Once the buffer fills up, older data will be lost. The oldest line of debug data may be an incomplete line.

The 4 crash-dump regions wrap in this way: Crash-dump region 1 is used to save the first crash, then the second crash-dump region is used to save the second crash, and so on till all 4 are used, then the first crash-dump region is reused, then successive crash-dump regions are reused.

If a controller is gracefully shut down or put off line (for example, via CAPI_PutOffline, CAPI_RebootController, or CAPI_ShutDownController), all the debug data is cleared. If a controller is ungracefully shut down or forced off line (for example, killed by the other controller, or the power is shut off, or via CAPI_ForceOffline) then the debug data will be preserved in battery-backed RAM on the controller.

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
	See Capability Bits

See also:

Get Drive Error Statistics NEW! in CAPI 3.3

Syntax:

```
CAPI_RC  CAPI_GetDriveErrorStatistics( CAPI_HANDLE   handle,
                                       CAPI_U8      *driveNodeWWN,
                                       CAPI_U32      driveIndex );
```

Description:

This command gets drive error statistics for a specified disk drive.

handle is the handle of the controller that executes the command.
driveNodeWWN is a pointer to the drive node worldwide name, used for Fibre Channel-attached drives only. It is represented as a string of 8 bytes.
driveIndex is an index into an array of CAPI_DRIVE structures, used for SCSI-attached drives only.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_GET_DRIVE_ERROR_STATS**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	Pointer to a **CAPI_DRIVE_ERROR_STATS** structure.

Events:

Remarks:

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
	See Capability Bits

See also:

CAPI_ResetDriveErrorStatistics()

Get Drive List

Syntax:

```
CAPI_RC  CAPI_GetDriveList( CAPI_HANDLE  handle );
```

Description:

This returns an array of CAPI_DRIVE structures.

> ***Note to CAPI 2.x users:*** Up to 250 drives are supported. Drives are not listed by channel any more.

handle is the handle of the controller that executes the command.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_GET_DRIVE_LIST**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	Number of CAPI_DRIVE structs returned.
param2	Configuration sequence number.
dataPtr	Pointer to the first element of an array of **CAPI_DRIVE** structures; there are param1 elements in the array.

Events:

Remarks:

The application developer needs to make sure that the configuration sequence number on their copy of the drive list (an array of CAPI_DRIVE structures retrieved with a call to CAPI_GetDriveList) matches the configuration sequence number on their copy of CAPI_CONTROLLER (retrieved with a call to CAPI_UpdateController). A CAPI_ERROR_FAILURE_DUE_TO_CONFIG_CHANGE will occur if a configuration change is attempted with incompatible structures.

A controller does not have visibility to drives that are members of an array owned by the other controller nor to drives that are dedicated spares of an array owned by the other controller, and therefore does not return these drives in its list of drives.

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
✔	See Capability Bits

See also:

Get Event

Syntax:

```
CAPI_RC  CAPI_GetEvent( CAPI_HANDLE  handle,
                        CAPI_U32     eventNum );
```

Description:

Get event information from the controller.

handle is the handle of the controller that executes the command.
eventNum is the sequential number of the event to retrieve (zero is an invalid event number).

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_GET_EVENT**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	The requested event sequence number.
param2	The first event sequence number available on the controller.
param3	The last event sequence number available on the controller.
dataPtr	A pointer to a **CAPI_EVENT** structure.

Events:

Remarks:

Event numbers start at one. If the controller reports that the last event sequence number is zero, then this indicates an empty event log.

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
	See Capability Bits

See also:

CAPI_GetFirstEvent()
CAPI_GetLastEvent()

Get First Event

Syntax:

```
CAPI_RC   CAPI_GetFirstEvent( CAPI_HANDLE  handle );
```

Description:

Gets the first event information in the event queue from the controller.

handle is the handle of the controller that executes the command.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_GET_FIRST_EVENT**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	The first event sequence number available on the controller.
param2	The event sequence number of the last controller power up; that is, the most recent event that has an event code of CAPI_EVENT_POWER_UP.
param3	The last event sequence number available on the controller. (NEW! in CAPI 3.4)
dataPtr	A pointer to a **CAPI_EVENT** structure.

Events:

Remarks:

Event numbers start at one. If the controller reports that the last event sequence number is zero, then this indicates an empty event log. As the controller runs, the sequence numbers increment and the event trace will wrap. The first and last event numbers allow the application to determine how many events are in the event log.

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
	See Capability Bits

See also:

CAPI_GetEvent()
CAPI_GetLastEvent()

Get Free Array Partitions

Syntax:

```
CAPI_RC  CAPI_GetFreeArrayPartitions( CAPI_HANDLE   handle,
                                      CAPI_U8      *arraySerialNumber );
```

Description:

Gets the list of free array partitions contained in the specified array. These are essentially the unpartitioned or "free" areas on the array. Each of these free areas is a location where a new partition can be added or into which an adjacent (and physically lower) partition can be expanded.

handle is the handle of the controller that executes the command.
arraySerialNumber is a pointer to the serial number of the array that contains the free partitions.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_GET_FREE_ARRAY_PARTITIONS**
errorCode	Completion status of the command.
identifier	controllerHandle and arrayIndex are valid.
param1	Number of CAPI_ARRAY_PARTITION structs returned.
param2	Configuration sequence number.
dataPtr	Pointer to the first element of an array of **CAPI_ARRAY_PARTITION** structures; there are param1 elements in the array.

Events:

Remarks :

This command is valid if the CAPI_CAPABILITY_2_ARRAY_PARTITIONS capability flag is set. The maximum number of free partitions supported by one array is given by CAPI_MAX_FREE_PARTITIONS_PER_ARRAY. Note that the only fields of interest in the returned CAPI_ARRAY_PARTITION structure are *startLba* and *sizeLba*.

	Lengthy Operation
✔	Need Current Configuration
	May Change Configuration
✔	See Capability Bits

See also:

CAPI_AddArrayPartition()
CAPI_GetArrayPartitions()

Get Host Nicknames NEW! in CAPI 3.3

Syntax:

```
CAPI_RC  CAPI_GetHostNicknames( CAPI_HANDLE  handle );
```

Description:

This command allows a CAPI application to get a structure containing a list of all hosts that have nicknames defined. This structure maps worldwide names to nicknames. This mapping can be used by a CAPI application to allow a user to use nicknames instead of worldwide names.

handle is the handle of the controller that executes the command.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_GET_HOST_NICKNAMES**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	Pointer to a **CAPI_HOST_NICKNAMES** structure.

Events:

Remarks:

This function requires capability bit CAPI_CAPABILITY_2_INFOSHIELD to be set.

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
✔	See Capability Bits

See also:

CAPI_AddHostNickname()
CAPI_GetKnownHosts()

Get Host Table

Syntax:

```
CAPI_RC  CAPI_GetHostTable( CAPI_HANDLE  handle,
                            CAPI_U32     channelIndex,
                            CAPI_U32     unitNum,
                            CAPI_U8     *partitionSerialNumber );
```

Description:

This function returns the table of hosts that either do or do not have access to the specified *unitNum* or *partitionSerialNumber.*

handle is the handle of the controller that executes the command.
channelIndex host channel index that the array or device is being presented on.
unitNum LUN that this array or device is being presented as.
partitionSerialNumber is the serial number of the partition; if partitions are not supported (capability bit CAPI_CAPABILITY_2_ARRAY_PARTITIONS not set), then this is an array serial number. (Applies to RAID only; not routers.)

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_GET_HOST_TABLE**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	Pointer to a **CAPI_HOST_TABLE** structure.

Events:

Remarks:

If *partitionSerialNumber* is not NULL, it will be used; if it is NULL, *channelIndex* and *unitNum* will be used.

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_AddHost()
CAPI_RemoveHost()
CAPI_ChangeInfoShieldType()

Get Known Hosts

Syntax:

```
CAPI_RC  CAPI_GetKnownHosts( CAPI_HANDLE  handle );
```

Description:

This function returns the table of hosts that are known to have communicated with the controller.

handle is the handle of the controller that executes the command.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_GET_KNOWN_HOSTS**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	Pointer to a **CAPI_KNOWN_HOSTS** structure.

Events:

Remarks:

The list can contain up to 64 hosts; if more hosts contact the controller than 64, the oldest entries are dropped.

The list is returned sorted by time of first contact.

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_GetHostTable()

Get Last Event

Syntax:

```
CAPI_RC  CAPI_GetLastEvent( CAPI_HANDLE  handle );
```

Description:

Gets the last event information in the event queue from the controller.

handle is the handle of the controller that executes the command.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_GET_LAST_EVENT**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	The last event sequence number available on the controller.
param2	The first event sequence number available on the controller.
dataPtr	A pointer to a **CAPI_EVENT** structure.

Events:

Remarks:

Event numbers start at one. If the controller reports that the last event sequence number is zero, then this indicates an empty event log.

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
	See Capability Bits

See also:

CAPI_GetEvent()
CAPI_GetFirstEvent()

Get Percent Complete

Syntax:

```
CAPI_RC  CAPI_GetPercentComplete( CAPI_HANDLE  handle,
                                  CAPI_U32     arrayIndex );
```

Description:

Returns the percent complete of the currently running utility.

handle is the handle of the controller that executes the command.
arrayIndex is the index of the target array on the specified controller.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_UTILITY_PERCENT**
errorCode	Completion status. If successful, param1 contains a valid percentage.
identifier	controllerHandle and arrayIndex are valid.
param1	Contains the percent complete value as a 32-bit unsigned integer.
param2	Contains the **CAPI_UTILITY_RUNNING** type of utility running.
dataPtr	

Events:

Remarks :

If param2 equals CAPI_NO_UTILITY_RUNNING, then param1 is undefined.

	Lengthy Operation
✔	Need Current Configuration
	May Change Configuration
	See Capability Bits

See also:

Initialize

Syntax:

```
CAPI_RC  CAPI_Initialize( void );
```

Description:

Initializes the CAPI system.

Return Code:

Indicates if the initialization process can begin.

Callback:

replyCode	**CAPI_REPLY_INITIALIZE_COMPLETE**
errorCode	Indicates if the API successfully completed initialization or a status code if an error occurred.
identifier	
param1	
param2	
dataPtr	

Events:

Remarks :

Initializes the API. See *Initialization Details* on page 15.

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
	See Capability Bits

See also:

Kill Other

Syntax:

```
CAPI_RC  CAPI_KillOther( CAPI_HANDLE  handle );
```

Description:

This command forces the other controller (in a dual controller active/active configuration) into a reset and holds it there.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_KILLED_OTHER**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

Remarks :

If CAPI_CAPABILITY_3_REPLACEABLE_MODULE is *not* set, this function is supported.

	Lengthy Operation
	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_UnkillOther()

Log Event NEW! in CAPI 3.3

Syntax:

```
CAPI_RC  CAPI_LogEvent( CAPI_HANDLE  handle,
                        CAPI_EVENT  *event );
```

Description:

This command allows a CAPI application to make an entry in the event log that is maintained by and on a Chaparral controller board. **This command is for Chaparral internal use only.**

handle is the handle of the controller that executes the command.
event is a pointer to a structure containing the event data to be logged.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_LOG_EVENT**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

Remarks:

This command is intended for use by Chaparral's software only (specifically, to allow the LAN Subsystem to log events in the event log maintained by the Storage Controller processor). This function should not be used by external CAPI applications to avoid using up the limited space available for events (400 events at this writing).

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
	See Capability Bits

See also:

Log In

Syntax:

```
CAPI_RC  CAPI_LogIn( CAPI_HANDLE  handle,
                     CAPI_U8     *password );
```

Description:

This command registers a password with the API, which in turn sends it in all CAPI PACKETS until CAPI_LogOut. Any potentially destructive command will require this password. The controller will reject commands with bad passwords with CAPI_ERROR_BAD_PASSWORD. The password is set with the loader (can't set password with CAPI). If a password is not set with the loader then password is not required. The password will not be encrypted in this release of CAPI; its main purpose it to prevent accidents rather than malice.
handle is the handle of the controller that executes the command.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_LOG_IN**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

Remarks:

Not implemented if CAPI_CAPABILITY_2_SECURITY_LOG_IN_OUT is not set.
Not currently implemented.

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
✔	See Capability Bits

See also:

Log Out

Syntax:

```
CAPI_RC  CAPI_LogOut( CAPI_HANDLE  handle,
                      CAPI_U8     *password );
```

Description:

The CAPI API will stop sending the password to the controller.

handle is the handle of the controller that executes the command.
index is the index of the target array for which the utility should be aborted.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_LOG_OUT**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

Remarks:

Not implemented if CAPI_CAPABILITY_2_SECURITY_LOG_IN_OUT is not set.
Not currently implemented.

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
✔	See Capability Bits

See also:

Pause Bus

Syntax:

```
CAPI_RC  CAPI_PauseBus( CAPI_HANDLE  handle,
                        CAPI_U32     channelIndex );
```

Description:

Suspends I/O to all back-end SCSI buses.

handle is the handle of the controller that executes the command.
channelIndex is the index of the bus or channel on the specified controller. However, this parameter is not used at this time. By default, all buses will be paused.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_PAUSE_BUS**
errorCode	Status of the operation. If successful, the disk channels are paused.
identifier	controllerHandle and channelIndex are valid.
param1	
param2	
dataPtr	

Events:

Remarks :

While some connectors are designed to allow hot-plugging SCSI drives, most are not. In all cases, the SCSI bus should be paused to prevent corrupted data. If a SCSI drive is inserted or removed from the bus, the pins may disrupt the signals. This function can be used to pause I/O on the bus while drives are added or removed.

After a call to CAPI_PauseBus, the bus remains paused until a call to CAPI_UnpauseBus. When the pause is issued, any SCSI commands currently in progress are allowed to complete. Any SCSI commands received after the pause is issued are queued by the RAID controller. If the queue becomes full, a status of queue full is returned to the host via the SCSI interface. Pass CAPI_NULL_ID in *channelIndex* to pause all buses.
This command may not be implemented on this controller or you may not be able to pause individual buses. See *CAPI Versions*
Different Chaparral products support different versions of CAPI. If the major version (for example, the 3 in version 3.4) is the same for two products, you should be able to easily design a single CAPI app that manages both products, although the product with the lower minor version number (for example, the 4 in version 3.4) will not have all the features of the product with the higher minor version number. Specifically, there may be additional CAPI commands added for a higher version number and there may be additional members in some of the data structures passed to and from CAPI commands, but the members that existed in the lower version of CAPI are still in the same locations in the same structures.

When a new version of the CAPI SDK is released, typically the most recent Chaparral product(s) are released simultaneously with the SDK and the version of CAPI that is running in the controller corresponds to the SDK version. However, developers of new CAPI apps can use a more recent version of the CAPI SDK to talk to older products, provided the major version number matches, and, of course, the app cannot use the newest features that have been added to CAPI which are not supported on that product.

Conversely, if a CAPI app was developed using an older version of the CAPI SDK (for example, CAPI 3.2), it can still be used for managing a newer product that supports a newer version of CAPI (for example, CAPI3.4), but of course the app will not be able to take advantage of the newer features that have been added to CAPI 3.4 in the product but which are not in the CAPI 3.2 SDK.

As of this writing (September 2002), the following products support the corresponding version of CAPI:

- CAPI 2.x: G5312, G7313, all Kxxxx.
- CAPI 3.4: RIO, Stratis RAID S3300 (JFF224). (Thus, the features marked “NEW! in CAPI 3.3” and “NEW! in CAPI 3.4” in this document are supported by these products only.)
- CAPI 3.2: All other Chaparral products.

CAPI Capabilities on page 29. Requires CAPI_CAPABILITY_2_PAUSE_INDIVIDUAL_BUS set to pause an individual bus.

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
✔	See Capability Bits

See also:

CAPI_UnpauseBus()

Put Offline NEW! in CAPI 3.3

Syntax:

```
CAPI_RC   CAPI_PutOffline( CAPI_HANDLE         handle,
                           CAPI_MODULE_TYPE    moduleType,
                           CAPI_MODULE_INDEX   moduleIndex,
                           CAPI_U8             param3 );
```

Description:

Puts the replaceable module (FRU) offline gracefully. The controller will carry out this request even if it affects performance (for example, putting one Data Manager offline in an active-active RAID system), but will reject this request if it affects availability (for example, putting a Data Manager offline in a RAID system when the other Data Manager is already offline). If the request is rejected, this command returns an errorCode indicating the problem.

handle is the handle of the controller that executes the command.
moduleType is the type of FRU that is being put offline. At this writing, only CAPI_MODULE_TYPE_DM and CAPI_MODULE_TYPE_DG are supported.
moduleIndex identifies the specific module. This must be one of 0 through 3 for Data Gates. It must be CAPI_MODULE_A or CAPI_MODULE_B for Data Managers.
param3 is reserved for possible future use.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_PUT_OFFLINE**
errorCode	Completion status of the command.
identifier	ControllerHandle is valid.
param1	
param2	
dataPtr	

Events:

Remarks:

Putting a Data Manager (DM) offline is equivalent to shutting it down.

CAPI_ShutDownController provides similar functionality to this function. However, that function can only act on Data Managers and it can shut down both controller boards with a single function call.

Calling CAPI_PutOffline is equivalent to calling CAPI_ShutDownController for a single DM with *fwUpdate* set to FALSE except that CAPI_PutOffline does availability checking first.

This function requires capability bit CAPI_CAPABILITY_3_REPLACEABLE_MODULE to be set.

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_PutOnline()
CAPI_ForceOffline()
CAPI_ForceOnline()
CAPI_ShutDownController()

Put Online NEW! in CAPI 3.3

Syntax:

```
CAPI_RC  CAPI_PutOnline( CAPI_HANDLE        handle,
                         CAPI_MODULE_TYPE   moduleType,
                         CAPI_MODULE_INDEX  moduleIndex,
                         CAPI_U8            param3 );
```

Description:

Puts the replaceable module (FRU) online gracefully. Putting a module online gracefully means running its diagnostics and running compatibility checks to see if the hardware and firmware of the FRU are compatible with the other FRUs. If these checks do not pass, this command returns an errorCode indicating the problem.

handle is the handle of the controller that executes the command.
moduleType is the type of FRU that is being put offline. At this writing, only CAPI_MODULE_TYPE_DM and CAPI_MODULE_TYPE_DG are supported.
moduleIndex identifies the specific module. This must be one of 0 through 3 for Data Gates. It must be CAPI_MODULE_A or CAPI_MODULE_B for Data Managers.
param3 is reserved for possible future use.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_PUT_ONLINE**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

Remarks:

This function requires capability bit CAPI_CAPABILITY_3_REPLACEABLE_MODULE to be set.

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_PutOffline()
CAPI_ForceOffline()
CAPI_ForceOnline()

Reboot Controller

Syntax:

```
CAPI_RC  CAPI_RebootController( CAPI_HANDLE          handle,
                                CAPI_CONTROLLER_ID   controllerId );
```

Description:

This command does the same thing as CAPI_ShutDownController and then reboots. It is also used to reboot a controller when it is in a shutdown state as a result of CAPI_ShutDownController.

handle is the handle of the controller that executes the command.
controllerId specifies which controller you want to reboot; one of: CAPI_CONTROLLER_A, CAPI_CONTROLLER_B, CAPI_CONTROLLER_BOTH.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_CONTROLLER_REBOOT_START**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_CONTROLLER_REBOOT_COMPLETE

Remarks :

Rebooting will flush the controller’s write back cache to disk.

See CAPI_ShutDownController for additional information.

	Lengthy Operation
	Need Current Configuration
✔	May Change Configuration
	See Capability Bits

See also:

CAPI_UpdateFirmware()
CAPI_ShutDownController()

Reconstruct Array

Syntax:

```
CAPI_RC  CAPI_ReconstructArray( CAPI_HANDLE  handle,
                                CAPI_U32     arrayIndex,
                                CAPI_U32     arrayDriveIndex,
                                CAPI_U32     priority );
```

Description:

This command is not supported and will be removed in future versions of CAPI. To reconstruct an array, just add a dedicated spare to the array, or add a pool spare and the reconstruct will start automatically.

handle is the handle of the controller that executes the command.
arrayIndex is the index of the target array on the specified controller.
arraydriveIndex is the array drive index in the array.
priority is unused.

Return Code:

This will return CAPI_ERROR_NOT_SUPPORTED.

Callback:

replyCode	
errorCode	
identifier	
param1	
param2	
dataPtr	

Events:

Remarks :

✔	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_AddDedicatedSpare()
CAPI_AddPoolSpare()

Register Callback

Syntax:

```
CAPI_RC  CAPI_RegisterCallback( CAPI_VFPTR  capiCallback );
```

> ***Note to CAPI 2.x users:*** Two additional parameters (*param3* and *param4*) were added to the callback function that you must provide. Make sure to change your source code if upgrading from CAPI2 or else you might meet with unpredictable results.

Description:

Provides a pointer to the application's callback routine to CAPI.

capiCallback is a pointer to the application callback function.

Return Code:

Always returns CAPI_STATUS_GOOD.

Callback:

replyCode	None
errorCode	
identifier	
param1	
param2	
dataPtr	

Events:

Remarks :

This command will not invoke the application’s callback function. The callback routine will be invoked by other CAPI commands to notify the application of asynchronous events and completion of asynchronous functions. The prototype for the callback routine is shown here:

```
void CallBack( CAPI_REPLY_CODE   replyCode,
               CAPI_ERROR_CODE   errorCode,
               CAPI_IDENTIFIER  *identifier,
               CAPI_U32          param1,
               CAPI_U32          param2,
               CAPI_U32          param3,
               CAPI_U32          param4,
               void             *pDataPtr )
```

See *Reply to Function Calls* on page 6 and *Initialization Details* on page 15.

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
	See Capability Bits

See also:

Remove Host

Syntax:

```
CAPI_RC  CAPI_RemoveHost( CAPI_HANDLE    handle,
                          CAPI_U32       channelIndex,
                          CAPI_U8       *partitionSerialNumber,
                          CAPI_U32       unitNum,
                          CAPI_FLEX_ID   hostId,
                          CAPI_BOOL      allHosts,
                          CAPI_BOOL      accessMode );
```

Description:

This function removes a host from the list of hosts that may communicate with a specified *unitNum* or *partitionSerialNumber*. The list is either a list of hosts that are included for access to the LUN or a list of hosts that are excluded from access. The *allHosts* flag may be used to override the list and have all hosts either included or excluded.

handle is the handle of the controller that executes the command.
channelIndex host channel index that the array or device is being presented on.
partitionSerialNumber is the serial number of the partition; if partitions are not supported (capability bit CAPI_CAPABILITY_2_ARRAY_PARTITIONS not set), then this is an array serial number. (Applies to RAID only; not routers.)
unitNum LUN that this array or device is being presented as.
hostId Fibre Channel or SCSI ID of the host.
allHosts setting to TRUE removes all hosts from this LUN's list of hosts that have access.
accessMode setting to TRUE designates a list of hosts that are to be included for access.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_REMOVE_HOST**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

Remarks:

Applications Errata for Router – The *accessMode* flag must match the current state of the list in order for the command to have the desired effect.

If *partitionSerialNumber* is not NULL, it will be used; if it is NULL, *channelIndex* and *unitNum* will be used.

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_GetHostTable()
CAPI_AddHost()
CAPI_ChangeInfoShieldType()

Rescan Bus

Syntax:

```
CAPI_RC  CAPI_RescanBus( CAPI_HANDLE  handle,
                         CAPI_U32     channelIndex );
```

Description:

Scans the drives on the back-end drive bus to detect new, moved, or deleted drives.

handle is the handle of the controller that executes the command.
channelIndex is the index of the channel to rescan. Pass CAPI_NULL_ID to rescan all channels.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_RESCAN_BUS_START**
errorCode	Completion status of the command.
identifier	controllerHandle and channelIndex are valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_RESCAN_BUS_START
CAPI_EVENT_RESCAN_BUS_COMPLETE

Remarks :

To avoid any performance degradation, the controller does not scan the SCSI buses for changes in configuration unless instructed to do so through CAPI or SAF-TE. This function should be called after new SCSI drives are added, if drives are moved to different IDs, or if unused or spare drives are removed. SAF-TE processors can do automatic rescans. Some controllers may do a rescan on a SCSI bus reset.

See *CAPI Versions*
Different Chaparral products support different versions of CAPI. If the major version (for example, the 3 in version 3.4) is the same for two products, you should be able to easily design a single CAPI app that manages both products, although the product with the lower minor version number (for example, the 4 in version 3.4) will not have all the features of the product with the higher minor version number. Specifically, there may be additional CAPI commands added for a higher version number and there may be additional members in some of the data structures passed to and from CAPI commands, but the members that existed in the lower version of CAPI are still in the same locations in the same structures.

When a new version of the CAPI SDK is released, typically the most recent Chaparral product(s) are released simultaneously with the SDK and the version of CAPI that is running in the controller corresponds to the SDK version. However, developers of new CAPI apps can use a more recent version of the CAPI SDK to talk to older products, provided the major version number matches, and, of course, the app cannot use the newest features that have been added to CAPI which are not supported on that product.

Conversely, if a CAPI app was developed using an older version of the CAPI SDK (for example, CAPI 3.2), it can still be used for managing a newer product that supports a newer version of CAPI (for example, CAPI3.4), but of course the app will not be able to take advantage of the newer features that have been added to CAPI 3.4 in the product but which are not in the CAPI 3.2 SDK.

As of this writing (September 2002), the following products support the corresponding version of CAPI:

- CAPI 2.x: G5312, G7313, all Kxxxx.
- CAPI 3.4: RIO, Stratis RAID S3300 (JFF224). (Thus, the features marked “NEW! in CAPI 3.3” and “NEW! in CAPI 3.4” in this document are supported by these products only.)
- CAPI 3.2: All other Chaparral products.

CAPI Capabilities on page 29 to determine if the controller supports rescanning of individual channels; if not, then *channelIndex* should be CAPI_NULL_ID. Requires CAPI_CAPABILITY_2_RESCAN_INDIVIDUAL_BUS set to rescan an individual bus.

✔	Lengthy Operation
	Need Current Configuration
	May Change Configuration
✔	See Capability Bits

See also:

Reset Array Statistics

Syntax:

```
CAPI_RC  CAPI_ResetArrayStatistics( CAPI_HANDLE  handle,
                                    CAPI_U32     arrayIndex );
```

Description:

Resets temporary array statistics.

handle is the handle of the controller that executes the command.
arrayIndex is the index of the target array on the specified controller.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_RESET_ARRAY_STATS**
errorCode	Completion status of the command.
identifier	controllerHandle and arrayIndex are valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_RESET_ARRAY_STATS

Remarks :

This function clears array statistics but those are not visible from the Disk Array Administrator or through a CAPI app. In earlier versions of Chaparral products we were only able to create 1 partition per array. Now we are able to create 1 or more partitions in an array so the array statistics are not used anymore but are replaced with array partition statistics.

	Lengthy Operation
✔	Need Current Configuration
	May Change Configuration
✔	See Capability Bits

See also:

CAPI_UpdateController()

Reset Array Partition Statistics

Syntax:

```
CAPI_RC   CAPI_ResetArrayPartitionStatistics( CAPI_HANDLE    handle,
                                              CAPI_U8       *partitionSerialNumber );
```

Description:

Resets temporary array partition statistics.

handle is the handle of the controller that executes the command.
partitionSerialNumber is a pointer to the serial number of an existing partition.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_RESET_ARRAY_PARTITION_STATS**
errorCode	Completion status of the command.
identifier	controllerHandle and arrayIndex are valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_RESET_ARRAY_PARTITION_STATS

Remarks :

	Lengthy Operation
✔	Need Current Configuration
	May Change Configuration
✔	See Capability Bits

See also:

CAPI_GetArrayPartitions()

Reset Drive Error Statistics NEW! in CAPI 3.3

Syntax:

```
CAPI_RC  CAPI_ResetDriveErrorStatistics( CAPI_HANDLE  handle,
                                         CAPI_U8     *driveNodeWWN,
                                         CAPI_U32     driveIndex );
```

Description:

This command allows a CAPI application to reset the drive error statistics for a designated disk drive. All values are set to 0.

handle is the handle of the controller that executes the command.
driveNodeWWN is a pointer to the drive node worldwide name, used for Fibre Channel-attached drives. It is represented as a string of 8 bytes.
driveIndex is the drive index, used for SCSI-attached drives.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_RESET_DRIVE_ERROR_STATS**
errorCode	Completion status of the command.
identifier	controllerHandle and driveIndex are valid.
param1	
param2	
dataPtr	

Events:

Remarks:

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
	See Capability Bits

See also:

CAPI_GetDriveErrorStatistics()

Reset Drive Statistics

Syntax:

```
CAPI_RC  CAPI_ResetDriveStatistics( CAPI_HANDLE  handle,
                                    CAPI_U32     channelIndex,
                                    CAPI_U32     driveIndex );
```

Description:

Resets the temporary drive statistics.
This command currently does nothing.

handle is the handle of the controller that executes the command.
channelIndex is index of the channel of the target drive for which you want to reset the temporary statistics. Pass CAPI_NULL_ID to reset all of the drive statistics on the controller.
driveIndex is the index of the target drive in the channel structure for which you want to reset the temporary statistics. Pass CAPI_NULL_ID to reset all of the drive statistics on the channel.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_RESET_DRIVE_STATS**
errorCode	Completion status of the command.
identifier	controllerHandle, channelIndex, and driveIndex are valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_RESET_DRIVE_STATS

Remarks :

Currently, the callback always contains errorCode = CAPI_ERROR_NOT_SUPPORTED.

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
✔	See Capability Bits

See also:

CAPI_ResetDriveErrorStatistics()

Reset LAN

Syntax:

```
CAPI_RC  CAPI_ResetLAN( CAPI_HANDLE  handle );
```

Description:

Resets the LAN Subsystem if one is present.

handle is the handle of the controller that executes the command.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_RESET_LAN**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_RESET_LAN

Remarks :

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
	See Capability Bits

See also:

Restore Controller Defaults

Syntax:

```
CAPI_RC  CAPI_RestoreControllerDefaults( CAPI_HANDLE  handle );
```

Description:

Restores the factory defaults of the controller.

handle is the handle of the controller that executes the command.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_RESTORE_CONTROLLER_DEFAULTS**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

Remarks :

A reboot is required for all defaults to take effect. See the controller's documentation to determine which defaults are restored immediately and which defaults take effect after the next reboot.

This command does *not* cause the following to be reset to defaults:
CAPI LUN (a.k.a. controller LUN or bridge LUN)
controller mode
drive channel speed
LAN Subsystem IP address
LAN Subsystem IP subnet mask
LAN Subsystem IP gateway

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
	See Capability Bits

See also:

SCSI Maintenance

Syntax:

```
CAPI_RC  CAPI_ScsiMaintenance( CAPI_HANDLE          handle,
                               CAPI_U32             bus,
                               CAPI_U32             target,
                               CAPI_U32             lun,
                               CAPI_MAINT_COMMAND   maintCommand,
                               CAPI_U32             param1,
                               CAPI_U32             param2,
                               CAPI_MAINT_CDB      *cdb,
                               CAPI_U32             cdbLength,
                               CAPI_U8             *dataBuffer,
                               CAPI_U32             dataBufferSize,
                               CAPI_DIRECTION       direction );
```

Description:

This command is used to send CAPI_MAINT_COMMANDs to the specified drive/device. However, when used with the RAID controller, this command cannot be sent to a drive that is part of a non-redundant array. See the warning statement below.

This command is sometimes referred to as "SCSI pass through."

handle is the handle of the controller that executes the command.
bus Bus number on the specified controller.
target SCSI ID of the device on the specified controller.
lun LUN of the device on the specified controller.
maintCommand possible values are shown on page 22. If CAPI_MAINT_USE_CDB is used, then *cdb* points to the CDB that will be passed to the designated drive. For all other values, *cdb* is ignored. Note: not all maintenance commands may be supported. Refer to your controller's documentation.
param1 for CAPI_MAINT_MODE_SENSE, this is the mode page and page control fields. This needs to follow the same format as byte 2 of a SCSI Mode Sense CDB. For CAPI_MAINT_MODE_SELECT, this is the SCSI mode page to write.
param2 contains any extra parameters needed for maintenance commands (currently unused).
cdb points to the CDB to be sent to the designated drive. This should be NULL for any command other than CAPI_MAINT_USE_CDB.
cdbLength is the length of the CDB (should be zero for any command other than CAPI_MAINT_USE_CDB).
dataBuffer points to the data buffer when this command transfers data to the drives. For CAPI_MAINT_MODE_SELECT, this *dataBuffer* contains the new mode page data. Data returned from the drive may be accessed via CAPI_ScsiMaintRetrieveData.
dataBufferSize is the number of bytes of data in *dataBuffer*.
direction is unused.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_SCSI_MAINT_START**
errorCode	Completion status of the command.
identifier	controllerHandle and driveIndex are valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_SCSI_MAINT_DONE
Other events (such as CAPI_EVENT_DISK_DETECTED_ERROR) are possible if the maintenance command causes an error.

Remarks:

After this event is received any data associated with the SCSI command can be retrieved from the controller using the CAPI_ScsiMaintRetrieveData command.

Note that all calls to CAPI_ScsiMaintenance and CAPI_U_DoScsiMaintenance make use of a single buffer. Thus, it is important that one SCSI maintenance operation be complete before the next one starts. The sequence of commands should be as follows:

- Call CAPI_ScsiMaintenance or CAPI_U_DoScsiMaintenance.
- Wait for an event to be posted to indicate that the operation is complete (normally CAPI_EVENT_SCSI_MAINT_DONE).
- Call CAPI_ScsiMaintRetrieveData or CAPI_U_GetScsiMaintenanceData to get any data.
- Repeat this sequence as desired.

> ***WARNING:*** *This command should not be used on a drive that is part of an array. Doing so can cause undesirable results.*

> **Note:** *You must issue a rescan (CAPI_RescanBus) after CAPI_MAINT_CLEAR_METADATA for the clear metadata function to take effect. However, if you need to clear metadata on more than one drive, only a single rescan is needed after all the clear metadata commands complete.*

See the controller’s documentation to determine which maintenance commands, if any, are supported and which commands might remove the drive from the array.

✔	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_U_DoScsiMaintenance() is the corresponding unified command.
CAPI_ScsiMaintRetrieveData()

SCSI Maintenance Retrieve Data

Syntax:

```
CAPI_RC  CAPI_ScsiMaintRetrieveData( CAPI_HANDLE  handle,
                                     CAPI_U32     packetNumber,
                                     CAPI_U32     commandId,
                                     CAPI_U32    *dataBuffer,
                                     CAPI_U32     dataBufferSize );
```

Description:

Retrieves the additional data returned from a maintenance command.

handle is the handle of the controller that executes the command.
packetNumber If the data to be transferred is greater than the size of a CAPI packet, then this number is incremented by the application to get the next chunk of data. (Not implemented.)
commandId is the number returned with the CAPI_EVENT_SCSI_MAINT_DONE event. (Not implemented.)
dataBuffer is unused.
dataBufferSize is the number of bytes of data that you want to be returned in the CAPI_MAINT_DATA_STRUCT. You can specify a value that is appropriate for the SCSI command sent by CAPI_ScsiMaintenance. The maximum size that can be specified is sizeof(CAPI_MAINT_DATA_STRUCT); if you specify more that that number of bytes, only that many bytes will be returned.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_SCSI_MAINT_DATA**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	The number of bytes of data that have been put in the data buffer pointed to by *dataPtr*. (Normally equal to *dataBufferSize*, but never greater than sizeof(CAPI_MAINT_DATA_STRUCT).)
param2	
dataPtr	A pointer to a **CAPI_MAINT_DATA_STRUCT**. See page 82.

Events:

Remarks:

If you need to get more than the amount of data that will fit in CAPI_MAINT_DATA_STRUCT, use the non-CAPI SCSI pass through capability described in Chapter 17.

Note that all calls to CAPI_ScsiMaintenance and CAPI_U_DoScsiMaintenance make use of a single buffer. Thus, it is important that one SCSI maintenance operation be complete before the next one starts. The sequence of commands should be as follows:

- Call CAPI_ScsiMaintenance or CAPI_U_DoScsiMaintenance.
- Wait for an event to be posted to indicate that the operation is complete (normally CAPI_EVENT_SCSI_MAINT_DONE).
- Call CAPI_ScsiMaintRetrieveData or CAPI_U_GetScsiMaintenanceData to get any data.
- Repeat this sequence as desired.

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
✔	See Capability Bits

See also:

CAPI_U_GetScsiMaintenanceData() is the corresponding unified command.
CAPI_ScsiMaintenance()

Set Advanced Network Interface NEW!

Syntax:

```
CAPI_RC   CAPI_SetAdvancedNetworkInterface( CAPI_HANDLE                         handle,
                                            CAPI_ADVANCED_NETWORK_INTERFACE    *advNet );
```

Description:

This function allows configuration parameters to be set for the LAN processor.

handle is the handle of the controller that executes the command.
advNet is a pointer to a data structure containing the configuration parameters.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_SET_ADV_NETWORK_INTF**
errorCode	Completion status of the command.
identifier	ControllerHandle is valid.
param1	
param2	
dataPtr	

Events:

Remarks:

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_GetAdvancedNetworkInterface()

Set Advanced Unit Mapping

Syntax:

```
CAPI_RC  CAPI_SetAdvancedUnitMapping( CAPI_HANDLE     handle,
                                      CAPI_UNIT_MAP  *mapping,
                                      CAPI_U16        numMapping );
```

Description:

This function sets the mapping of back-end devices or arrays to front-end LUNs.

handle is the handle of the controller that executes the command.
mapping pointer to an array of CAPI_UNIT_MAP data structures.
numMapping number of CAPI_UNIT_MAPs in the array.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_SET_ADVANCED_UNIT_MAPPING**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

Remarks:

This command is currently supported only on the Router. RAID controllers will return CAPI_ERROR_NOT_SUPPORTED when sent this command.

Applications Errata for Router - This function may be used to pass a variable number of CAPI_UNIT_MAP structures to the Router. A CAPI Client uses this function to mask or unmask a particular device or to map a particular device to a particular LUN slot. All devices, including the Router LUN, occupy a LUN slot in the table of CAPI_UNIT_MAP structures. In order to mask a device, the CAPI Client passes in CAPI_LUN_UNASSIGNED (255 or 0xFF) for the front-end LUN value (this indicates the device should be masked), and the back-end SCSI device coordinates <bus, target, lun> for the device that is to be masked. In order to unmask a device, the CAPI Client passes in the desired LUN value that the device should be mapped to. In order to re-map a device (manual mapping), the CAPI Client passes in the desired front-end LUN value for the specified device. If the new LUN slot for the device is different from the present LUN slot for the device (re-mapping), then the new LUN slot is occupied by the specified device. The device, if any, that previously occupied the new LUN slot is moved to the previous LUN slot for the device that was just re-mapped. A CAPI_UNIT_MAP structure may therefore be used to swap two entries in the internally-maintained table of CAPI_UNIT_MAP structures.

There are a few rules that must be known to a CAPI Client in order to successfully use the CAPI_SetAdvancedUnitMapping function in the Router:

1.) The Router mapping mode must be Fixed, not Auto.

2.) The number of mappings must not be greater than or equal to 64.

3.) Each CAPI_UNIT_MAP structure passed in by the CAPI Client must obey the following rules:

```
//  ************************************************************
//
//  For each CAPI_UNIT_MAP passed in:
//
//  hostId.type        - must be CAPI_FLEX_TYPE_LUN
//  hostChannelIndex   - must be less than 1
//  hostId.unitNum     - must be less than 64,
//                       or 255 to mask the specified SCSI device, and
//                       must be for a real back-end device
//                       (i.e. not for the Router LUN)
//
//  deviceId.type      - must be CAPI_FLEX_TYPE_SCSI | CAPI_FLEX_TYPE_LUN
//  deviceChannelIndex - must be less than 3
//  deviceId.deviceId  - must be less than 16
//  deviceId.unitNum   - must be less than 8
//
//  ************************************************************
```

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_GetAdvancedUnitMapping()

Set Array Partition Cache Params NEW! in CAPI 3.3

Syntax:

```
CAPI_RC   CAPI_SetArrayPartitionCacheParams( CAPI_HANDLE        handle,
                                             CAPI_U8           *partitionSerialNumber,
                                             CAPI_CACHE_PARAMS *cacheParams );
```

Description:

This command allows a CAPI application to set parameters that determine characteristics of the cache associated with the specified partition.

handle is the handle of the controller that executes the command.
partitionSerialNumber is a pointer to the serial number of an existing partition.
cacheParams is a pointer to a structure that contains the new values for the cache parameters.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_SET_ARRAY_PARTITION_CACHE_PARAMS**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

Remarks:

At this writing, the only parameters in CAPI_CACHE_PARAMS that are supported are *writeBackEnable* and *readAheadSize*.

When *writeBackEnable* is set to TRUE, the write back cache is enabled.

readAheadSize should be set to 0 or to a power of 2 between 64KB and 32MB or to the default. That is, use one of these values: 0 (which disables read ahead), 0x10000, 0x20000, 0x40000, 0x80000, 0x100000, 0x200000, 0x400000, 0x800000, 0x1000000, 0x2000000, or CAPI_DEFAULT_READ_AHEAD_SIZE (which tells the controller to use an algorithm that tries to optimize read ahead based on whether reads are sequential or random). More cache improves performance of sequential reads but will hurt performance of random reads.

To apply CAPI_CACHE_PARAMS to all partitions in an array via a single function call, you can use CAPI_SetCacheParams.

This function requires capability bit CAPI_CAPABILITY_2_ARRAY_PARTITIONS to be set.

> ***CAUTION:*** *The RAID controller's default cache parameters are preset to provide optimal performance for virtually all applications. Modification of these parameters may significantly decrease performance.*

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_SetCacheParams()

Set Battery Monitor

Syntax:

```
CAPI_RC  CAPI_SetBatteryMonitor( CAPI_HANDLE  handle,
                                 CAPI_BOOL    monitorOn,
                                 CAPI_U8      months );
```

Description:

This function sets the age of the battery and enables/disables end-of-life monitoring.

handle is the handle of the controller that executes the command.
monitorOn set to TRUE to enable battery life monitoring.
months set to the number of months the battery has been installed (set to zero if the controller is new).

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_SET_BATTERY_MONITOR**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

Remarks:

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
	See Capability Bits

CAPI_EVENT_BATTERY_END_OF_LIFE will occur at the end of the battery life.

See also:

Set Cache Params

Syntax:

```
CAPI_RC  CAPI_SetCacheParams( CAPI_HANDLE         handle,
                              CAPI_U32            arrayIndex,
                              CAPI_CACHE_PARAMS   *cacheParams );
```

Description:

This command allows a CAPI application to set parameters that determine characteristics of the cache associated with the specified array.

handle is the handle of the controller that executes the command.
arrayIndex is the index of the target array on the specified controller. Pass CAPI_NULL_ID to configure all arrays on the specified controller with these parameters.
cacheParams points to a CAPI_CACHE_PARAMS structure containing the new cache parameter settings.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_SET_CACHE_PARAMS**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_SET_CACHE_PARAMS

Remarks :

Some of these parameters can also be set through SCSI mode pages from the host.

At this writing, the only parameter in CAPI_CACHE_PARAMS that is supported is *writeBackEnable*. When *writeBackEnable* is set to TRUE, the write back cache is enabled.

For more recent products that support multiple partitions (from RIO onward), *readAheadSize* is also supported. See CAPI_SetArrayPartitionCacheParams for details on this parameter.

Note that for arrays containing multiple partitions, the cache parameters for all partitions in the array are updated when this command is issued.

> ***CAUTION:*** *The RAID controller's default cache parameters are preset to provide optimal performance for virtually all applications. Modification of these parameters may significantly decrease performance.*

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_SetArrayPartitionCacheParams()
CAPI_FlushCache()
CAPI_FreeCache()

Set Channel Params

Syntax:

```
CAPI_RC  CAPI_SetChannelParams( CAPI_HANDLE          handle,
                                CAPI_CHANNEL_TYPE    type,
                                CAPI_U32             channelIndex,
                                CAPI_CHANNEL_PARAMS *params );
```

Description:

Sets new channel parameters for front-end or back-end channels.

handle is the handle of the controller that executes the command.
type specifies if this is a front or back-end channel.
channelIndex is the index of the channel for which the parameters are being updated.
params a pointer to a CAPI_CHANNEL_PARAMS structure

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_SET_CHANNEL_PARAMS**
errorCode	Completion status of the command.
identifier	controllerHandle valid.
param1	
param2	
dataPtr	

Events:

Remarks:

Developers can read the current parameters using CAPI_UpdateController, modify the parameters, and update them with CAPI_SetChannelParams.

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
	See Capability Bits

See also:

Set Controller Params

Syntax:

```
CAPI_RC  CAPI_SetControllerParams( CAPI_HANDLE             handle,
                                   CAPI_CONTROLLER_PARAMS  *controllerParams );
```

Description:

Sets the controller's parameters such as SCSI bus termination and utility priority.

handle is the handle of the controller that executes the command.
controllerParams is a pointer to a CAPI_CONTROLLER_PARAMS structure with the new controller settings.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_SET_CONTROLLER_PARAMS**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_SET_CONTROLLER_PARAMS

Remarks :

Developers can read the current parameters using CAPI_UpdateController, modify the parameters, and update them with CAPI_SetControllerParams.

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
	See Capability Bits

See also:

CAPI_UpdateController()

Set Controller Time Date

Syntax:

```
CAPI_RC   CAPI_SetControllerTimeDate( CAPI_HANDLE  handle,
                                      CAPI_TIME    timeDate );
```

Description:

Sets the controller time and date settings.

handle is the handle of the controller that executes the command.
timeDate contains the number of seconds since January 1, 1970 (i.e., UNIX time).

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_SET_CONTROLLER_TIMEDATE**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_SET_CONTROLLER_TIMEDATE

Remarks :

The standard library provided with many 'C' compilers includes functions for manipulating CAPI_TIME (of type time_t, usually an unsigned long) and generating a standard 'tm' structure. See time, gmtime, localtime, mktime, and strftime in your compiler's documentation. Note that a *timeDate* value of zero is invalid.

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
	See Capability Bits

See also:

Set Preferred Owner

Syntax:

```
CAPI_RC   CAPI_SetPreferredOwner( CAPI_HANDLE  handle,
                                  CAPI_U32     arrayIndex );
```

Description:

Allows the application to change the owner of an array from one controller to another.

handle is the handle of the controller that executes the command.
arrayIndex is the index of the target array on the specified controller.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_PREFERRED_OWNER_SET**
errorCode	Completion status of the command.
identifier	controllerHandle and arrayIndex are valid.
param1	
param2	
dataPtr	

Events:

Remarks :

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_CreateArray()

Set Unit Mapping

Syntax:

```
CAPI_RC  CAPI_SetUnitMapping( CAPI_HANDLE  handle,
                              CAPI_U32     arrayIndex,
                              CAPI_U32     newUnitNum );
```

Description:

Allows the application to change the LUN that an array presents to the host.

handle is the handle of the controller that executes the command.
arrayIndex is the index of the target array on the specified controller.
newUnitNum is the desired LUN for the specified array.

Return Code:

Indicates if the request was sent to the RAID controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_UNIT_MAPPING**
errorCode	Completion status of the command. A LUN conflict will return an error.
identifier	controllerHandle and arrayIndex are valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_UNIT_MAPPING

Remarks :

A reboot may be necessary on some products for the new LUN mapping to take effect.

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
	See Capability Bits

See also:

CAPI_CreateArray()

Shut Down Controller

Syntax:

```
CAPI_RC  CAPI_ShutDownController( CAPI_HANDLE          handle,
                                  CAPI_CONTROLLER_ID   controllerId,
                                  CAPI_BOOL            fwUpdate );
```

Description:

Perform a graceful shutdown on the specified controller.

handle is the handle of the controller that executes the command.
controllerId specifies which controller you want to shut down (CAPI_CONTROLLER_A, CAPI_CONTROLLER_B, or CAPI_CONTROLLER_BOTH.)
fwUpdate is set to true if a firmware update is to follow, this lets the other controller know why we are shutting down. This parameter does not affect this operation; it just provides information to the on-line controller so it is accessible via the *failoverReason* structure member obtainable via CAPI_UpdateController or CAPI_U_GetControllerData.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_SHUTDOWN_CONTROLLER**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	controllerId
param2	
dataPtr	

Events:

CAPI_EVENT_SHUTDOWN_CONTROLLER

Remarks :

Shutting down will flush the controllers' write back cache to disk. The controller shuts down, then calls the callback function. The controller is then in a special state that responds only to a limited selection of CAPI commands, most notably:

- CAPI_UpdateFirmware
- CAPI_RebootController

(For a complete list of CAPI commands that are supported during shutdown, see column allowWhileShutdown in the table in file capicmdsup.c.)

Also, once a controller is shut down, its serial port will no longer respond to the CTRL-P then CTRL-Z character sequence (which is used to restore terminal mode after a serial CAPI application has run). The reason is that a CAPI_COMMAND_UPDATE_CONTROLLER_FIRMWARE request over the serial port could have the CTRL-P/CTRL-Z sequence embedded in its binary data, which if recognized would cause the serial port to unintentionally transition to terminal mode.

If both controllers are shut down at the same time via this function, both can receive firmware updates from the host in-band. If only one controller is shut down, the shut down controller cannot receive firmware downloads in-band since the one that is not shut down is "impersonating" the shut down controller to the host and so the shut down controller has no host interface. If both controllers are shut down one after the other, the second one to be shut down still has a host interface so it can receive firmware downloads in-band. No matter what sequence is used to shut a controller down, the RS-232 connection can be used to download firmware (except that RS-232 download of firmware is not supported on RIO).

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
	See Capability Bits

See also:

CAPI_UpdateFirmware()
CAPI_RebootController()
CAPI_PutOffline()

Silence Alarm

Syntax:

```
CAPI_RC  CAPI_SilenceAlarm( CAPI_HANDLE  handle );
```

Description:

This command temporarily silences the controller's on-board audible alarm. (Depending on the storage system design, it may or may not silence an enclosure alarm produced by an EMP.) As soon as the controller has another event that causes it to turn on the alarm, the alarm will sound. To permanently disable the alarm, set the *alarmMute* field in the CAPI_CONTROLLER_PARAMS structure and call CAPI_SetControllerParams.

handle is the handle of the controller that executes the command.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_SILENCE_ALARM**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

Remarks :

If the alarm is caused by unwritable cache data (see CAPI_EVENT_ORPHAN_DATA), the cache data is not purged. If the alarm is caused by A/D failure, the command is ignored and the alarm will stay on. If the alarm is not on, this command is accepted successfully, but ignored.

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
	See Capability Bits

See also:

CAPI_SetControllerParams

Test Drive

Syntax:

```
CAPI_RC  CAPI_TestDrive( CAPI_HANDLE  handle,
                         CAPI_U32     channelIndex,
                         CAPI_U32     driveIndex );
```

Description:

Performs simple tests on a drive.

handle is the handle of the controller that executes the command.
channelIndex is the index of the target array on the specified controller.
driveIndex is the index of the drive to perform the test on.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_TEST_DRIVE**
errorCode	Completion status of the command.
identifier	controllerHandle, channelIndex, and driveIndex are valid.
param1	
param2	
dataPtr	

Events:

Remarks :

Currently, this command only executes a command that causes an indicator lamp on the specified drive to blink. In the future this command may be implemented to do additional testing of the drive that is nondestructive to the drive and the drive's data. See the controller's documentation.

	Lengthy Operation
✔	Need Current Configuration
	May Change Configuration
✔	See Capability Bits

See also:

CAPI_BlinkDrive()
CAPI_UnblinkDrive()

Test Spares

Syntax:

```
CAPI_RC  CAPI_TestSpares( CAPI_HANDLE  handle,
                          CAPI_BOOL    testSpares );
```

Description:

Enable or disable the RAID core's testing of spare drives to verify that they are still available. Power up default is TRUE.

handle is the handle of the controller that executes the command.
testSpares can be set to TRUE to enable spare tests or to FALSE to disable spare tests.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_TEST_SPARES**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

Remarks :

This is a continuous background test on all spare drives (global spares and pool spares) until a subsequent call is made to disable the test. See the controller's documentation for specific implementation details. If a test fails, then a CAPI_EVENT_DOWN_DRIVE event is generated and the spare is removed from the spare list.

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
✔	See Capability Bits

See also:

Timer Tick

Syntax:

```
CAPI_RC   CAPI_TimerTick( void );
```

Description:

The application must call this function every ½ second to enable the LMX layer to timeout links that are not responding.

Return Code:

Always returns CAPI_STATUS_GOOD.

Callback:

replyCode	None
errorCode	
identifier	
param1	
param2	
dataPtr	

Events:

Remarks :

This is used by the LMX layer for timeout purpose on down links.

Note that this is not used by all LMXs; this is not required for the SCSI LMX (lmx_sc32.c), but is required for the two serial LMXs (lmx232.c and lmx232j.c).

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
	See Capability Bits

See also:

Trust Array

Syntax:

```
CAPI_RC  CAPI_TrustArray( CAPI_HANDLE  handle,
                          CAPI_U32     arrayIndex );
```

Description:

This function allows use of an array that is unusable because of failed drives. The data may be corrupt, and therefore this function should only be used for testing or data recovery purposes.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_TRUST_ARRAY**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_TRUST_ARRAY

Remarks :

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
	See Capability Bits

See also:

Unblink Drive

Syntax:

```
CAPI_RC  CAPI_UnblinkDrive( CAPI_HANDLE  handle,
                            CAPI_U32     channelIndex,
                            CAPI_U32     driveIndex );
```

Description:

This command stops blinking the drive's activity light.

handle is the handle of the controller that executes the command.
channelIndex is the index of the channel on the specified controller.
driveIndex is the index of the target drive on the specified channel.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_DRIVE_UNBLINK**
errorCode	Completion status of the command.
identifier	controllerHandle, channelIndex, and driveIndex are valid.
param1	
param2	
dataPtr	

Events:

Remarks :

Blinking a drive activity light is initiated by a call to CAPI_BlinkDrive. The controller continues blinking the drive light until this function is called.

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
	See Capability Bits

See also:

CAPI_BlinkDrive()

Unkill Other

Syntax:

```
CAPI_RC  CAPI_UnkillOther( CAPI_HANDLE  handle );
```

Description:

This command releases the kill mechanism from being asserted to the other controller allowing the other controller to boot.

handle is the handle of the controller that executes the command.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_UNKILL_OTHER**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

Remarks :

If CAPI_CAPABILITY_3_REPLACEABLE_MODULE is *not* set, this function is supported.

	Lengthy Operation
	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_KillOther()

Unpause Bus

Syntax:

```
CAPI_RC  CAPI_UnpauseBus( CAPI_HANDLE  handle,
                          CAPI_U32     channelIndex );
```

Description:

Resumes I/O to the specified back-end SCSI bus.

handle is the handle of the controller that executes the command.
channelIndex is the index of the disk channel on the specified controller. Pass CAPI_NULL_ID to unpause all disk channels.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_UNPAUSE_BUS**
errorCode	Completion status of the command.
identifier	controllerHandle and channelIndex are valid.
param1	
param2	
dataPtr	

Events:

Remarks :

I/O to a back-end SCSI bus is paused through a call to CAPI_PauseBus. This command may not be implemented on this controller or you may not be able to pause individual buses. See *CAPI Versions* Different Chaparral products support different versions of CAPI. If the major version (for example, the 3 in version 3.4) is the same for two products, you should be able to easily design a single CAPI app that manages both products, although the product with the lower minor version number (for example, the 4 in version 3.4) will not have all the features of the product with the higher minor version number. Specifically, there may be additional CAPI commands added for a higher version number and there may be additional members in some of the data structures passed to and from CAPI commands, but the members that existed in the lower version of CAPI are still in the same locations in the same structures.

When a new version of the CAPI SDK is released, typically the most recent Chaparral product(s) are released simultaneously with the SDK and the version of CAPI that is running in the controller corresponds to the SDK version. However, developers of new CAPI apps can use a more recent version of the CAPI SDK to talk to older products, provided the major version number matches, and, of course, the app cannot use the newest features that have been added to CAPI which are not supported on that product.

Conversely, if a CAPI app was developed using an older version of the CAPI SDK (for example, CAPI 3.2), it can still be used for managing a newer product that supports a newer version of CAPI (for example, CAPI3.4), but of course the app will not be able to take advantage of the newer features that have been added to CAPI 3.4 in the product but which are not in the CAPI 3.2 SDK.

As of this writing (September 2002), the following products support the corresponding version of CAPI:

- CAPI 2.x: G5312, G7313, all Kxxxx.
- CAPI 3.4: RIO, Stratis RAID S3300 (JFF224). (Thus, the features marked "NEW! in CAPI 3.3" and "NEW! in CAPI 3.4" in this document are supported by these products only.)

- CAPI 3.2: All other Chaparral products.

CAPI Capabilities on page 29.

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
✔	See Capability Bits

See also:

CAPI_PauseBus()

Update Controller

Syntax:

```
CAPI_RC  CAPI_UpdateController( CAPI_HANDLE  handle );
```

Description:

Returns a current copy of the CAPI_CONTROLLER structure for the specified controller through a subsequent callback. This updates your copy of the controller struct.

handle is the handle of the controller that executes the command.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_CONTROLLER_UPDATE**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	Configuration sequence number.
dataPtr	Contains a pointer to a **CAPI_CONTROLLER** structure.

Events:

Remarks :

This functions queries the controller for its current CAPI_CONTROLLER structure contents. A pointer to this structure is provided to the callback function in a temporary buffer that must be copied by the application into a permanent copy.

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
	See Capability Bits

See also:

CAPI_U_GetControllerData() is the corresponding unified command.

Update Firmware

Syntax:

```
CAPI_RC  CAPI_UpdateFirmware( CAPI_HANDLE   handle,
                              CAPI_U8      *firmwareImage,
                              CAPI_U32      size );
```

Description:

Loads new firmware into the controller.

handle is the handle of the controller that executes the command.
firmwareImage is a pointer to the new firmware image to be loaded.
size is the size of the image in bytes.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_UPDATE_FIRMWARE**
errorCode	**CAPI_NO_ERROR** indicates that the firmware image was received without errors.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_UPDATE_FIRMWARE_COMPLETE, posted after the controller reboots.

Remarks :

A call to CAPI_ShutdownController must precede this call.

Automatic reboot occurs if there are no errors updating the firmware.

Firmware updates are not permitted when orphan data is present in the controller.

> ***Note:*** *Since the firmware image is large, transfer of data from the host to the controller occurs as multiple messages, which are handled by code in the ReceivePacket function in capi2pak.c (part of the CAPI Client in the SDK). The callback function is not called until the entire firmware image has been transferred.*

✔	Lengthy Operation
	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_ShutdownController()
CAPI_FreeCache()

Use Key NEW!

Syntax:

```
CAPI_RC  CAPI_UseKey( CAPI_HANDLE  handle,
                      CAPI_U8     *key,
                      CAPI_BOOL    doit );
```

Description:

This function is not supported yet.

handle is the handle of the controller that executes the command.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

ReplyCode	**CAPI_REPLY_USE_KEY**
ErrorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	Features that were just enabled with this key.
param2	All enabled features.
DataPtr	

Events:

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

Verify Array

Syntax:

```
CAPI_RC   CAPI_VerifyArray( CAPI_HANDLE   handle,
                            CAPI_U32      arrayIndex,
                            CAPI_BOOL     disableAutoFix );
```

Description:

Verifies the state of a RAID 1, 3, 4, 5, 10, or 50 array.

handle is the handle of the controller that executes the command.
arrayIndex is the index of the target array on the specified controller.
disableAutoFix allows you to specify if the controller should correct any inconsistencies it may find. If the controller supports this function (see *CAPI Versions*
Different Chaparral products support different versions of CAPI. If the major version (for example, the 3 in version 3.4) is the same for two products, you should be able to easily design a single CAPI app that manages both products, although the product with the lower minor version number (for example, the 4 in version 3.4) will not have all the features of the product with the higher minor version number. Specifically, there may be additional CAPI commands added for a higher version number and there may be additional members in some of the data structures passed to and from CAPI commands, but the members that existed in the lower version of CAPI are still in the same locations in the same structures.

When a new version of the CAPI SDK is released, typically the most recent Chaparral product(s) are released simultaneously with the SDK and the version of CAPI that is running in the controller corresponds to the SDK version. However, developers of new CAPI apps can use a more recent version of the CAPI SDK to talk to older products, provided the major version number matches, and, of course, the app cannot use the newest features that have been added to CAPI which are not supported on that product.

Conversely, if a CAPI app was developed using an older version of the CAPI SDK (for example, CAPI 3.2), it can still be used for managing a newer product that supports a newer version of CAPI (for example, CAPI3.4), but of course the app will not be able to take advantage of the newer features that have been added to CAPI 3.4 in the product but which are not in the CAPI 3.2 SDK.

As of this writing (September 2002), the following products support the corresponding version of CAPI:

- CAPI 2.x: G5312, G7313, all Kxxxx.
- CAPI 3.4: RIO, Stratis RAID S3300 (JFF224). (Thus, the features marked "NEW! in CAPI 3.3" and "NEW! in CAPI 3.4" in this document are supported by these products only.)
- CAPI 3.2: All other Chaparral products.

CAPI Capabilities on page 29). **(Currently unused.)**

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_VERIFY_ARRAY_START**
errorCode	Completion status of the command.
identifier	controllerHandle and arrayIndex are valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_VERIFY_ARRAY_START
CAPI_EVENT_VERIFY_ARRAY_COMPLETE

Remarks :

The Verify function allows you to verify the data on the selected array (RAID 1, RAID 3, RAID 4, RAID 5, RAID 10, and RAID 50 only):

- RAID 3, RAID 4, RAID 5, and RAID 50: Verifies all parity blocks in the selected array and corrects any bad parity.
- RAID 1 and RAID 10: Compares the primary and secondary drives. If a mismatch occurs, the primary is copied to the secondary.

You may want to verify an array when you suspect there is a problem.

The number of fixes made is included with event CAPI_EVENT_VERIFY_ARRAY_COMPLETE.

✔	Lengthy Operation
✔	Need Current Configuration
	May Change Configuration
✔	See Capability Bits

See also:

Unified Abort Utility NEW! in CAPI 3.4

Syntax:

```
CAPI_RC  CAPI_U_AbortUtility( CAPI_HANDLE  handle,
                              CAPI_U8     *arraySerialNumber );
```

Description:

Aborts the configuration/management utility running on the specified array.

handle is the handle of the controller that executes the command.
arraySerialNumber is a pointer to the serial number of the target array for which the utility should be aborted.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_UTILITY_ABORT**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_UTILITY_ABORT

Remarks:

Each RAID array can have a maximum of one configuration or management utility running at a time. This function aborts the utility; however, not all utilities may be aborted. See *CAPI Versions*
Different Chaparral products support different versions of CAPI. If the major version (for example, the 3 in version 3.4) is the same for two products, you should be able to easily design a single CAPI app that manages both products, although the product with the lower minor version number (for example, the 4 in version 3.4) will not have all the features of the product with the higher minor version number. Specifically, there may be additional CAPI commands added for a higher version number and there may be additional members in some of the data structures passed to and from CAPI commands, but the members that existed in the lower version of CAPI are still in the same locations in the same structures.

When a new version of the CAPI SDK is released, typically the most recent Chaparral product(s) are released simultaneously with the SDK and the version of CAPI that is running in the controller corresponds to the SDK version. However, developers of new CAPI apps can use a more recent version of the CAPI SDK to talk to older products, provided the major version number matches, and, of course, the app cannot use the newest features that have been added to CAPI which are not supported on that product.

Conversely, if a CAPI app was developed using an older version of the CAPI SDK (for example, CAPI 3.2), it can still be used for managing a newer product that supports a newer version of CAPI (for example, CAPI3.4), but of course the app will not be able to take advantage of the newer features that have been added to CAPI 3.4 in the product but which are not in the CAPI 3.2 SDK.

As of this writing (September 2002), the following products support the corresponding version of CAPI:

- CAPI 2.x: G5312, G7313, all Kxxxx.
- CAPI 3.4: RIO, Stratis RAID S3300 (JFF224). (Thus, the features marked “NEW! in CAPI 3.3” and “NEW! in CAPI 3.4” in this document are supported by these products only.)

- CAPI 3.2: All other Chaparral products.

CAPI Capabilities on page 29.

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_AbortUtility()

Unified Add Array Partition NEW! in CAPI 3.4

Syntax:

```
CAPI_RC   CAPI_U_AddArrayPartition( CAPI_HANDLE               handle,
                                    CAPI_U8                   *arraySerialNumber,
                                    CAPI_PARTITION_REQUEST    *addPartition );
```

Description:

Adds (i.e., creates) a new partition to an existing array.

handle is the handle of the controller that executes the command.
arraySerialNumber is a pointer to the serial number of the array to which the partition will be added.
addPartition is a pointer to the CAPI_PARTITION_REQUEST structure which is used to specify the characteristics of the partition to be created. All the members of CAPI_PARTITION_REQUEST must be specified except *partitionSerialNumber* and *arraySerialNumber*. The *arraySerialNumber* member is filled in by the function; it copies the *arraySerialNumber* function param to the *arraySerialNumber* structure member.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_ADD_ARRAY_PARTITION**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_ADD_ARRAY_PARTITION_COMPLETE

Remarks :

This command is valid if the CAPI_CAPABILITY_2_ARRAY_PARTITIONS capability flag is set.

The maximum number of partitions supported by one array is given by CAPI_MAX_PARTITIONS_PER_ARRAY. The maximum number of partitions supported by a controller is given by CAPI_MAX_ARRAY_PARTITIONS_PER_CONTROLLER.

The partition serial number of the new partition is included with the event CAPI_EVENT_ADD_ARRAY_PARTITION_COMPLETE as u.serialNumbers.arraySerialNumber. This serial number can then be used as a parameter when calling other CAPI functions that require a partition serial number.

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_AddArrayPartition()
CAPI_U_ChangeArrayPartitionGeometry()

CAPI_U_ChangeArrayPartitionLun()
CAPI_U_ChangeArrayPartitionName()
CAPI_U_DeleteArrayPartition()
CAPI_U_GetArrayPartitions()
CAPI_U_GetFreeArrayPartitions()
CAPI_U_ResetArrayPartitionStatistics()

Unified Add Dedicated Spare NEW! in CAPI 3.4

Syntax:

```
CAPI_RC  CAPI_U_AddDedicatedSpare( CAPI_HANDLE  handle,
                                   CAPI_U8     *arraySerialNumber,
                                   CAPI_U8     *driveSerialNumber );
```

Description:

This function adds an unused or free drive as a dedicated spare to a redundant array.

handle is the handle of the controller that executes the command.
arraySerialNumber is a pointer to the serial number of the target array.
driveSerialNumber is a pointer to the serial number of the drive to be added.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_ADD_DEDICATED_SPARE**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_ADD_DEDICATED_SPARE

Remarks :

It is assumed that the calling routine has verified that the drive has sufficient capacity for the array. If the array has a down drive, a reconstruct utility immediately starts.

If a drive contains metadata from a previous array, you must clear the metadata using the CAPI_ScsiMaintenance or CAPI_U_DoScsiMaintenance command before adding the drive as a dedicated spare or pool spare. The controller will automatically rescan the bus when the metadata is cleared.

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_AddDedicatedSpare()
CAPI_U_AddPoolSpare()
CAPI_U_DeleteSpare()
CAPI_U_DoScsiMaintenance()

Unified Add Host NEW! in CAPI 3.4

Syntax:

```
CAPI_RC   CAPI_U_AddHost( CAPI_HANDLE    handle,
                          CAPI_U8       *partitionSerialNumber,
                          CAPI_FLEX_ID   hostId );
```

Description:

This function adds a host to the list of hosts that is allowed to communicate with a particular partition or is blocked from communication with a particular partition.

handle is the handle of the controller that executes the command.
partitionSerialNumber is the serial number of the partition; if partitions are not supported (capability bit CAPI_CAPABILITY_2_ARRAY_PARTITIONS not set), then this is an array serial number.
hostId is the Fibre Channel or SCSI ID of the host.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_ADD_HOST**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

Remarks:

The list of hosts associated with this partition is either allowed access or blocked from access by the value of the *include* parameter in CAPI_U_ChangeInfoShieldType.

Recommended CAPI application design approach:

When an app needs a list of hosts (typically for the AddHost and RemoveHost commands), it should get both the known hosts list (by calling CAPI_U_GetKnownHosts) and the host nicknames list (by calling CAPI_U_GetHostNicknames) and combine them, deleting duplicate entries. These can be presented to the user in any order the app designer wishes and could be sorted under user control in various ways: by nickname, by WWN, by order of timestamp (which is the way the app will get them), or whatever. (The timestamp is the *age* member of the struct that is received by these two commands and is the number of seconds since January 1, 1970.) We recommend that an app specify CAPI_CONTROLLER_BOTH when calling CAPI_U_GetKnownHosts so it can see the hosts known to both controllers. Suggested order of presentation to user: known host table at the top, with the most-recent entries at the top, followed by the host nicknames table, with the most-recent entries at the top. Or the known hosts table could be presented in this order: hosts known to both controllers, then hosts known to A only, then hosts known to B only. (This is the order that CAPI_U_GetKnownHosts will return them.)

The app should provide a mechanism for a user to specify a nickname for a WWN at the time the user is adding a host to the InfoShield host table for a partition, and the app would call both the CAPI_U_AddHost and CAPI_U_AddHostNickname functions with this data. (The app doesn't need to call CAPI_U_AddHostNickname if the user selects from a list of hosts and the selected host already has a nickname.)

If a user types in a WWN that is not in the list of hosts but does not specify a nickname for it, the user interface can either prompt to force the user to provide a nickname, or the UI can simply call AddHostNickname with a dummy nickname (for example, a single space character – not a null string or that will be interpreted as a request to delete an entry in the host nickname list). Thus, the WWN will be saved in the host nicknames list even if it doesn't have a real nickname, and will thereby be available for future calls to AddHost and RemoveHost.

The app should probably have a utility that allows adding, deleting, and changing host nicknames and their associated WWNs. This would be implemented in the app as calls to AddHostNickname.

As an alternative, if an application designer wishes to maintain a separate nickname scheme or not use nicknames, the app can do that and not make use of the CAPI_U_GetHostNicknames and CAPI_U_AddHostNicknames commands.

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_AddHost()
CAPI_U_RemoveHost()
CAPI_U_ChangeInfoShieldType()
CAPI_U_GetHostTable()
CAPI_U_AddHostNicknames()
CAPI_U_GetHostNicknames()

Unified Add Host Nickname NEW! in CAPI 3.4

Syntax:

```
CAPI_RC  CAPI_U_AddHostNickname( CAPI_HANDLE   handle,
                                 CAPI_FLEX_ID  hostId,
                                 CAPI_U8      *nickname );
```

Description:

This command allows a CAPI application to define a “nickname” that corresponds to the worldwide name for a host. This capability of CAPI is provided so a CAPI application can provide a mechanism for the user of that application to more conveniently refer to a host. The CAPI application can access these host nicknames via the CAPI_GetHostNicknames and CAPI_GetKnownHosts functions.

handle is the handle of the controller that executes the command.

hostId is the worldwide name of the host that this nickname applies to. In the CAPI_FLEX_ID struct, the CAPI_FLEX_TYPE may be set to either CAPI_FLEX_TYPE_FC_WWN_NODE or CAPI_FLEX_TYPE_FC_WWN_PORT and the corresponding field, *FCNodeWWN* or *FCPortWWN*, is then used.

nickname points to a null-terminated string provided by the CAPI application. Maximum number of characters allowed in this string is CAPI_MAX_HOST_NAME (15 characters plus NULL).

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_ADD_HOST_NICKNAME**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

Remarks:

See CAPI_U_AddHost for a discussion of how a typical application might best use this command.

This function can be used to change a nickname as well as add a new one.

Caution: This function performs no check that the nickname is unique. That is, it is possible for the same nickname to be used for two or more different worldwide names, with unpredictable results.

Note that nicknames can be added or changed via the Disk Array Administrator (MUI) or other user interfaces; there is a single table of nicknames. Thus, name changes and additions made via one user interface are visible via other user interfaces.

The list of nicknames is saved on both controllers in a dual-controller system. The list of nicknames is preserved through a reboot and through replacement of one of the two controller boards.

Nicknames can be deleted by using this function with the nickname defined as a null string (that is, first character in the string is 0).

This function requires capability bit CAPI_CAPABILITY_2_INFOSHIELD to be set.

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
✔	See Capability Bits

See also:

CAPI_AddHostNickname()
CAPI_U_GetHostNicknames()
CAPI_U_GetKnownHosts()
CAPI_U_AddHost()
CAPI_U_RemoveHost()

Unified Add Pool Spare NEW! in CAPI 3.4

Syntax:

```
CAPI_RC   CAPI_U_AddPoolSpare( CAPI_HANDLE  handle,
                               CAPI_U8     *driveSerialNumber );
```

Description:

This function adds an unused or free drive to the spare pool.

handle is the handle of the controller that executes the command.
driveSerialNumber is a pointer to the serial number of the drive to add as a pool spare.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_ADD_POOL_SPARE**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_ADD_POOL_SPARE

Remarks :

It is assumed that the calling routine has verified that the drive has sufficient capacity for the array. If the array has a down drive, a Reconstruct utility immediately starts.

If a drive contains metadata from a previous array, you must clear the metadata using the CAPI_ScsiMaintenance or CAPI_U_DoScsiMaintenance command before adding the drive as a dedicated spare or pool spare. The controller will automatically rescan the bus when the metadata is cleared.

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_AddPoolSpare()
CAPI_U_AddDedicatedSpare()
CAPI_U_DeleteSpare()
CAPI_U_DoScsiMaintenance()

Unified Blink Drive NEW! in CAPI 3.4

Syntax:

```
CAPI_RC  CAPI_U_BlinkDrive( CAPI_HANDLE  handle,
                            CAPI_U8     *driveSerialNumber );
```

Description:

Blinks the drive activity light. The light is blinked by issuing a non-destructive command, such as a single sector read or a SCSI Test Unit Ready, at regular intervals.

handle is the handle of the controller that executes the command.
driveSerialNumber is a pointer to the serial number of the drive to blink.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_BLINK_DRIVE**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

Remarks :

The controller continues blinking the drive light until a call to CAPI_U_Unblink_drive.

	Lengthy Operation
✔	Need Current Configuration
	May Change Configuration
	See Capability Bits

See also:

CAPI_BlinkDrive()
CAPI_U_UnblinkDrive()

Unified Change Array Name NEW! in CAPI 3.4

Syntax:

```
CAPI_RC   CAPI_U_ChangeArrayName( CAPI_HANDLE   handle,
                                  CAPI_U8      *arraySerialNumber,
                                  CAPI_CHAR    *arrayName );
```

Description:

This command changes the array name.

handle is the handle of the controller that executes the command.
arraySerialNumber is a pointer to the serial number of the target array.
arrayName is a pointer to a NULL-terminated string containing the new array name. Length must be less than or equal to CAPI_MAX_ARRAY_NAME.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_CHANGE_ARRAY_NAME**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_ARRAY_NAME_CHANGE

Remarks :

An error will occur if the string is too long.

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
	See Capability Bits

See also:

CAPI_ChangeArrayName()
CAPI_U_CreateArray()

Unified Change Array Partition Geometry NEW! in CAPI 3.4

Syntax:

```
CAPI_RC CAPI_U_ChangeArrayPartitionGeometry(CAPI_HANDLE           controllerHandle,
                                            CAPI_U8              *partitionSerialNumber,
                                            CAPI_PARTITION_REQUEST *changePartition );
```

Description:

Changes the size of an existing array partition. Currently, the size of a partition may only be *increased*, not decreased.

handle is the handle of the controller that executes the command.
partitionSerialNumber is a pointer to the serial number of an existing partition.
changePartition is a pointer to the structure that is used to specify the new size of the partition. The members of this struct that must be specified are: *startLba* (must be the same as that specified when the partition was added), *sizeLba* (specifies the new size), and *arraySerialNumber*. The *partitionSerialNumber* member is filled in by the function; it copies the *partitionSerialNumber* function param to the *partitionSerialNumber* structure member. The *name* and *unitNum* members of this struct are ignored.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_CHANGE_ARRAY_PARTITION_GEOMETRY**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_ARRAY_PARTITION_GEOMETRY_CHANGE

Remarks :

This command is valid if the CAPI_CAPABILITY_2_ARRAY_PARTITIONS capability flag is set. Note that the size of a partition may only be increased if the partition is immediately followed by a free partition area. If an array is expanded, this creates free space at the end of the array, allowing the last partition in an array to expand into this area.

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_ChangeArrayPartitionGeometry()
CAPI_U_AddArrayPartition()
CAPI_U_GetFreeArrayPartitions()

Unified Change Array Partition LUN NEW! in CAPI 3.4

Syntax:

```
CAPI_RC  CAPI_U_ChangeArrayPartitionLun( CAPI_HANDLE   controllerHandle,
                                         CAPI_U8      *partitionSerialNumber,
                                         CAPI_U8       lun );
```

Description:

Allows the application to change the logical unit number (LUN) that a partition presents to the host.

handle is the handle of the controller that executes the command.
partitionSerialNumber is a pointer to the serial number of an existing partition.
lun is the new LUN value of the partition (this must be a currently unused LUN value).

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_CHANGE_ARRAY_PARTITION_LUN**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_ARRAY_PARTITION_LUN_CHANGE

Remarks :

This command is valid if the CAPI_CAPABILITY_2_ARRAY_PARTITIONS capability flag is set. No reboot is required for this change to take effect.

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_ChangeArrayPartitionLun()
CAPI_U_AddArrayPartition()

Unified Change Array Partition Name NEW! in CAPI 3.4

Syntax:

```
CAPI_RC   CAPI_U_ChangeArrayPartitionName( CAPI_HANDLE   controllerHandle,
                                           CAPI_U8      *partitionSerialNumber,
                                           CAPI_CHAR    *name );
```

Description:

Changes the name value of an existing array partition.

handle is the handle of the controller that executes the command.
partitionSerialNumber is a pointer to the serial number of an existing partition.
name is a pointer to a NULL-terminated string containing the new partition name. Length must be less than or equal to CAPI_MAX_ARRAY_NAME.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_ARRAY_PARTITION_NAME_CHANGE**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_ARRAY_PARTITION_NAME_CHANGE

Remarks :

This command is valid if the CAPI_CAPABILITY_2_ARRAY_PARTITIONS capability flag is set.

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_ChangeArrayPartitionName()
CAPI_U_AddArrayPartition()

Unified Change InfoShield Type NEW! in CAPI 3.4

Syntax:

```
CAPI_RC   CAPI_U_ChangeInfoShieldType( CAPI_HANDLE   handle,
                                       CAPI_U8       *partitionSerialNumber,
                                       CAPI_BOOL     allHosts,
                                       CAPI_BOOL     include );
```

Description:

This function changes the type of access that a list of hosts has for the specified *partitionSerialNumber*.

handle is the handle of the controller that executes the command.
partitionSerialNumber is the serial number of the partition; if partitions are not supported (capability bit CAPI_CAPABILITY_2_ARRAY_PARTITIONS not set), then this is an array serial number.
allHosts setting to TRUE causes the *include* parameter to apply to all hosts; setting to FALSE causes the *include* parameter to apply to this partition's list of hosts.
include setting to TRUE designates a list of hosts that are to be *included* for access; setting to FALSE designates a list of hosts that are to be *excluded* for access.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_CHANGE_INFOSHIELD_TYPE**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

Remarks:

The list of partitions that applies when *allHosts* is FALSE is configured using the CAPI_U_AddHost and CAPI_U_RemoveHost commands.

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_ChangeInfoShieldType()
CAPI_U_AddHost()
CAPI_U_RemoveHost()
CAPI_U_GetHostTable()

Unified Clear Event Log NEW! in CAPI 3.4

Syntax:

```
CAPI_RC   CAPI_U_ClearEventLog( CAPI_HANDLE         handle,
                                CAPI_CONTROLLER_ID  controllerId );
```

Description:

This command clears the non-volatile event log memory on the controller and resets the Event Log sequenceNumber.

handle is the handle of the controller that executes the command.
controllerId specifies which controller to send this command to; one of: CAPI_CONTROLLER_A, CAPI_CONTROLLER_B.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_LOG_CLEAR**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_LOG_CLEAR

Remarks :

This command should **only** be used to reset a controller to an empty log state before shipping to a customer. An application can clear its event log without actually clearing the event log on the controller by disregarding the last logged sequenceNumber and anything prior.

> ***WARNING:*** *This can cause problems for other attached applications currently polling for events.*

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
	See Capability Bits

See also:

CAPI_ClearEventLog()
CAPI_U_GetEvent()
CAPI_U_GetFirstEvent()
CAPI_U_GetLastEvent()

Unified Create Array NEW! in CAPI 3.4

Syntax:

```
CAPI_RC   CAPI_U_CreateArray( CAPI_HANDLE                          handle,
                              CAPI_UNIFIED_CREATE_ARRAY_STRUCT *addArrayStruct );
```

Description:

Creates a RAID array from a list of single drives.

handle is the handle of the controller that executes the command.
addArrayStruct is a pointer to a structure that provides all the parameters necessary to create an array.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_CREATE_ARRAY_START**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	pointer to **CAPI_UNIFIED_CREATE_ARRAY_SERIAL_NUMBER_STRUCT**

Events:

CAPI_EVENT_CREATE_ARRAY_START
CAPI_EVENT_CREATE_ARRAY_COMPLETE

Remarks :

The callback contains the serial number of the new array. This serial number can then be used by CAPI applications in subsequent calls to CAPI functions that take an array serial number as a parameter. The progress of the Create Array utility can be monitored by calling CAPI_GetPercentComplete. Completion status is obtained via calls to CAPI_GetLastEvent. The array serial number of the new partition is included with both events CAPI_EVENT_CREATE_ARRAY_START and CAPI_EVENT_CREATE_ARRAY_COMPLETE as *u.serialNumbers.arraySerialNumber*. The array serial number can also be obtained from the CAPI_ARRAY struct returned by function CAPI_GetArrayList.

After event CAPI_EVENT_CREATE_ARRAY_START is logged, there is a slight delay (generally less than a second) before CAPI_U_GetArrayList will return the new array as part of its list of arrays. You can call other CAPI functions related to this array (such as CAPI_U_AddArrayPartition) once your app sees the new array in the list of arrays.

The array serial number is 12 bytes; 8 bytes is the controller serial number and 4 bytes is a timestamp. An example is shown here:

```
 0    1    2    3    4    5    6    7    8    9   10   11     byte #
--------------------------------------------------------------
00   50   13   B0   30   00   00   00   2A   0F   58   3C     value
----------serial num----------  --time stamp--
```

In a typical application, this could be displayed as 0x005013B0300000002A0F583C.

In the Chaparral Disk Array Administrator and in the RAIDar web browser interface, only bytes 3-5 and 8-11 are displayed since bytes 0-2 are always 005013 and bytes 6 and 7 are always zeroes. Thus, the array serial number would display as B030002A0F583C.

In the case of a RAID 1, 10, 3, 4, or 5 array, the utility writes zeros to each LBA on each drive. The final step writes controller-specific information to the reserved sectors of each drive.

✔	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_CreateArray()
CAPI_U_DeleteArray()
CAPI_U_AddArrayPartition()
CAPI_U_GetPercentComplete()
CAPI_U_GetLastEvent()

Unified Delete Array NEW! in CAPI 3.4

Syntax:

```
CAPI_RC  CAPI_U_DeleteArray( CAPI_HANDLE  handle,
                             CAPI_U8     *arraySerialNumber );
```

Description:

Removes information in the reserved sectors of an array's member drives so that they are no longer associated with a RAID array.

handle is the handle of the controller that executes the command.
arraySerialNumber is a pointer to the serial number of the target array.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_DELETE_ARRAY**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_ARRAY_DELETE

Remarks :

After completion of this utility, the array is no longer valid and is no longer visible to the host. The member drives become single, free drives that can be assigned for use in new arrays or as spare drives. The drives are not reformatted by this utility and are not visible to the host.

> ***Warning****: All partitions contained in the array are automatically deleted when the array is deleted.*

	Lengthy Operation
	Need Current Configuration
✔	May Change Configuration
	See Capability Bits

See also:

CAPI_DeleteArray()
CAPI_U_CreateArray()

Unified Delete Array Partition NEW! in CAPI 3.4

Syntax:

```
CAPI_RC  CAPI_U_DeleteArrayPartition( CAPI_HANDLE   controllerHandle,
                                      CAPI_U8      *partitionSerialNumber );
```

Description:

Permanently deletes an existing array partition. The area formerly occupied by the partition becomes a free partition area, which can be used for partition expansion or to add a new partition.

handle is the handle of the controller that executes the command.
partitionSerialNumber is a pointer to the serial number of an existing partition.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_DELETE_ARRAY_PARTITION**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_DELETE_ARRAY_PARTITION_COMPLETE

Remarks :

This command is valid if the CAPI_CAPABILITY_2_ARRAY_PARTITIONS capability flag is set. Note that once the partition is deleted, it *cannot* be recovered.

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_DeleteArrayPartition()
CAPI_U_AddArrayPartition()
CAPI_U_GetArrayPartitions()

Unified Delete Spare NEW! in CAPI 3.4

Syntax:

```
CAPI_RC   CAPI_U_DeleteSpare( CAPI_HANDLE  handle,
                              CAPI_U32     driveSerialNumber );
```

Description:

This function changes the drive from spare drive to unused.

handle is the handle of the controller that executes the command.
driveSerialNumber is a pointer to the serial number of the drive to delete.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_DELETE_SPARE**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_ SPARE_DELETE

Remarks :

The drive becomes an available drive, which can be assigned for use in new arrays or as another spare drive. This command can be used to delete both pool spares and dedicated spares.

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
	See Capability Bits

See also:

CAPI_DeleteSpare()
CAPI_U_AddDedicatedSpare()
CAPI_U_AddPoolSpare()

Unified Do SCSI Maintenance NEW! in CAPI 3.4

Syntax:

```
CAPI_RC  CAPI_U_DoScsiMaintenance( CAPI_HANDLE          handle,
                                   CAPI_U8             *driveSerialNumber,
                                   CAPI_U32             bus,
                                   CAPI_U32             target,
                                   CAPI_U32             lun,
                                   CAPI_MAINT_COMMAND   maintCommand,
                                   CAPI_U32             param1,
                                   CAPI_U32             param2,
                                   CAPI_MAINT_CDB      *cdb,
                                   CAPI_U32             cdbLength,
                                   CAPI_U8             *dataBuffer,
                                   CAPI_U32             dataBufferSize );
```

Description:

This command is used to send CAPI_MAINT_COMMANDs to the specified drive/device. However, when used with the RAID controller, this command cannot be sent to a drive that is part of a non-redundant array. See the warning statements below.

This command is sometimes referred to as “SCSI pass through.”

handle is the handle of the controller that executes the command.
driveSerialNumber is a pointer to the serial number of the drive/device.
bus Bus number on the specified controller.
target SCSI ID of the device on the specified controller.
lun LUN of the device on the specified controller.
maintCommand possible values are shown on page 22. If CAPI_MAINT_USE_CDB is used, then *cdb* points to the CDB that will be passed to the designated drive. For all other values, *cdb* is ignored. Note: not all maintenance commands may be supported. Refer to your controller’s documentation.
param1 for CAPI_MAINT_MODE_SENSE, this is the mode page and page control fields. This needs to follow the same format as byte 2 of a SCSI Mode Sense CDB. For CAPI_MAINT_MODE_SELECT, this is the SCSI mode page to write.
param2 contains any extra parameters needed for maintenance commands (currently unused).
cdb points to the CDB to be sent to the designated drive. This should be NULL for any command other than CAPI_MAINT_USE_CDB.
cdbLength is the length of the CDB (should be zero for any command other than CAPI_MAINT_USE_CDB).
dataBuffer points to the data buffer when this command transfers data to the drives. For CAPI_MAINT_MODE_SELECT, this *dataBuffer* contains the new mode page data. Data returned from the drive may be accessed via CAPI_ScsiMaintRetrieveData.
dataBufferSize is the number of bytes of data in *dataBuffer.*

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_DO_SCSI_MAINT_START**
errorCode	Completion status of the command.
identifier	controllerHandle and driveIndex are valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_SCSI_MAINT_DONE

Other events (such as CAPI_EVENT_DISK_DETECTED_ERROR) are possible if the maintenance command causes an error.

Remarks:

After this event is received any data associated with the SCSI command can be retrieved from the controller using the CAPI_ScsiMaintRetrieveData command.

Note that all calls to CAPI_ScsiMaintenance and CAPI_U_DoScsiMaintenance make use of a single buffer. Thus, it is important that one SCSI maintenance operation be complete before the next one starts. The sequence of commands should be as follows:

- Call CAPI_ScsiMaintenance or CAPI_U_DoScsiMaintenance.
- Wait for an event to be posted to indicate that the operation is complete (normally CAPI_EVENT_SCSI_MAINT_DONE).
- Call CAPI_ScsiMaintRetrieveData or CAPI_U_GetScsiMaintenanceData to get any data.
- Repeat this sequence as desired.

If *driveSerialNumber* is non-zero, then it will be used to identify the back-end device that the command will be sent to. If *driveSerialNumber* is zero, then the *bus*, *target*, and *lun* will be used to identify the back-end device that the command will be sent to.

> ***WARNING:*** *There is less checking of safety of carrying out a command if bus, target, lun are used. Specifically, if a destructive command (such as clear metadata) is sent to a drive that is in an array owned by the other controller, undesirable results can occur.*

> ***WARNING:*** *This command should not be used on a drive that is part of an array. Doing so can cause undesirable results.*

> ***Note:*** *You must issue a rescan (CAPI_RescanBus) after CAPI_MAINT_CLEAR_METADATA for the clear metadata function to take effect. However, if you need to clear metadata on more than one drive, only a single rescan is needed after all the clear metadata commands complete.*

See the controller's documentation to determine which maintenance commands, if any, are supported and which commands might remove the drive from the array.

✔	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_ScsiMaintenance() is the corresponding non-unified function.
CAPI_U_GetScsiMaintenanceData()

Unified Down Drive NEW! in CAPI 3.4

Syntax:

```
CAPI_RC   CAPI_U_DownDrive( CAPI_HANDLE   handle,
                            CAPI_U32      driveSerialNumber );
```

Description:

Disables a drive that is a member of an array and can cause the array to switch to degraded operation.

handle is the handle of the controller that executes the command.
driveSerialNumber is a pointer to the serial number of the drive to down.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_DOWN_DRIVE**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_DRIVE_DOWN

Remarks:

This command should only be used for system testing. It will degrade an array to a critical state if one of the member drives is downed. Remember, after downing a drive, to use it again you must clear the metadata on the drive (with CAPI_U_DoScsiMaintenance) and then rescan the bus.

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
	See Capability Bits

See also:

CAPI_DownDrive()
CAPI_U_RescanBus()
CAPI_U_ScsiMaintenance()

Unified Environ Read NEW! in CAPI 3.4

Syntax:

```
CAPI_RC  CAPI_U_EnvironRead( CAPI_HANDLE  handle,
                             CAPI_U32     environProcessorIndex,
                             CAPI_U32     environCommand );
```

Description:

Requests data from an environmental processor (for either the SAF-TE or SES standard) attached to a controller.

handle is the handle of the controller that executes the command.
environProcessorIndex is the index of the environmental processor you are issuing the command to. This is the same as the index used in the CAPI_FindNextEnvironProcessor function.
environCommand is the environmental command code. See list of valid commands below.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_READ_ENVIRON**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	The number of valid bytes of data in the data buffer pointed to by *dataPtr*.
param2	
dataPtr	Pointer to **CAPI_ENVIRON_PROCESSOR_DATA**.

Events:

Remarks :

param1 will be less than or equal to CAPI_ENVIRON_MAX_ENVIRON_DATA_LENGTH.

If errorCode is equal to CAPI_NO_ERROR, then the data buffer contains valid inquiry data. However, if it is equal to CAPI_ERROR_COMMAND_FAILED, then sense data is automatically returned; the first byte in the data buffer contains the SCSI status byte and the rest of the data buffer contains SCSI sense data.

In this document, the terms “environmental processor,” “environmental device,” “environmental unit,” “Enclosure Management Processor,” and “EMP” are used interchangeably.

Chaparral enclosure management is intended for disk array enclosures that comply with either of the following two standards for enclosure services:

- SAF-TE (SCSI Accessed Fault-Tolerant Enclosure) – commonly used in SCSI/SCSI RAID enclosures.
- SES (SCSI-3 Enclosure Services) – an ANSI standard used widely for Fibre/Fibre RAID controllers and for SCSI-ATA and Fibre-ATA RAID controllers.

Each of these two enclosure services use different terminology for the Enclosure Management Processors (EMPs) that provide the enclosure services:

- SEP (SAF-TE Enclosure Processor for SAF-TE)
- ESP (Enclosure Services Processor for SES)

The following SAF-TE commands are valid for the *environCommand* parameter above.

Command
SAFTE_READ_ENCLOSURE_CFG_CMD
SAFTE_READ_ENCLOSURE_STATUS_CMD
SAFTE_READ_USAGE_STATS_CMD
SAFTE_READ_DEV_INSERTIONS_CMD
SAFTE_READ_DEV_SLOT_STATUS_CMD
SAFTE_READ_GLOBAL_FLAGS_CMD

Read Enclosure Configuration should be issued first before issuing any other SAF-TE reads. Refer to the SAF-TE Specification for more details. Also note that some SEP vendors do not support all of the commands listed and may return error codes.

The following SES commands are valid for the *environCommand* parameter above:

Command
SES_RECV_SUPPORTED_DIAGS
SES_RECV_CONFIGURATION
SES_RECV_ENCLOSURE_STATUS
SES_RECV_HELP_TEXT
SES_RECV_STRING_IN
SES_RECV_THRESHOLD_IN
SES_RECV_ARRAY_STATUS
SES_RECV_ELEMENT_DESCRIPTOR
SES_RECV_SHORT_ENCLOSURE_STAT

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
✔	See Capability Bits

See also:

CAPI_EnvironRead()
CAPI_U_EnvironWrite()
CAPI_U_FindNextEnvironProcessor()

Unified Environ Write NEW! in CAPI 3.4

Syntax:

```
CAPI_RC   CAPI_U_EnvironWrite( CAPI_HANDLE                    handle,
                               CAPI_U32                       environProcessorIndex,
                               CAPI_U32                       environCommand,
                               CAPI_ENVIRON_PROCESSOR_DATA   *buffer,
                               CAPI_U32                       length );
```

Description:

Sends data to an environmental processor (for either the SAF-TE or SES standard) attached to a controller.

handle is the handle of the controller that executes the command.
environProcessorIndex is the index of the environmental processor you are issuing the command to. This is the same as the index used in the CAPI_FindNextEnvironProcessor function.
environCommand is the environmental command code. See list of valid commands below.
buffer is a pointer to buffer containing the CAPI_ENVIRON_PROCESSOR_DATA structure.
length is the number of bytes to send to the EMP from the *buffer.*

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

A return code of CAPI_STATUS_INVALID_PARAM will be returned if *length* is greater than sizeof(CAPI_ENVIRON_PROCESSOR_DATA).

Callback:

replyCode	**CAPI_REPLY_U_WRITE_ENVIRON**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Remarks :

In this document, the terms “environmental processor,” “environmental device,” “environmental unit,” “Enclosure Management Processor,” and “EMP” are used interchangeably.

The following SAF-TE commands are valid for the *environCommand* parameter above:

Command
SAFTE_WRITE_DEV_SSLOT_STATUS_CMD
SAFTE_SET_SCSI_ID_CMD
SAFTE_PERFORM_SLOT_OPERATION_CMD
SAFTE_SET_FAN_SPEED_CMD
SAFTE_ACTIVATE_POWER_SUPPLY_CMD
SAFTE_SEND_GLOBAL_FLAGS_CMD

Note: The *buffer* parameter points to the structure that contains the write buffer command data only. It does not contain the write buffer Operation Code in the first byte as described in the SAF-TE Interface Specification. The Operation Code is inserted by the controller before the actual command is sent to the SEP, using the *environCommand* parameter.

The following SES commands are valid for the *environCommand* parameter above:

Command
SES_SEND_ENCLOSURE_CONTROL
SES_SEND_STRING_OUT
SES_SEND_THRESHOLD_OUT
SES_SEND_ARRAY_CONTROL

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
✔	See Capability Bits

See also:

CAPI_EnvironWrite()
CAPI_U_EnvironRead()
CAPI_U_FindNextEnvironProcessor()

Unified Expand Array NEW! in CAPI 3.4

Syntax:

```
CAPI_RC   CAPI_U_ExpandArray( CAPI_HANDLE                         handle,
                              CAPI_UNIFIED_CREATE_ARRAY_STRUCT *expandArrayStruct );
```

Description:

This function adds a new drive to an existing array and begins online capacity expansion to increase the size of the array. The original array is indicated by the *arraySerialNumber* member of *expandArrayStruct.*

handle is the handle of the controller that executes the command.
expandArrayStruct is a pointer to a structure that provides all the parameters necessary to expand an array.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_EXPAND_ARRAY_START**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_EXPAND_ARRAY_START
CAPI_EVENT_EXPAND_ARRAY_COMPLETE

Remarks :

> ***Note:*** *The new drives must be at least as large as the smallest existing member drive in the array.*

✔	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_ExpandArray()

Unified Find Next Environ Processor NEW! in CAPI 3.4

Syntax:

```
CAPI_RC  CAPI_U_FindNextEnvironProcessor( CAPI_HANDLE  handle,
                                          CAPI_U32     environProcessorIndex );
```

Description:

Finds environmental devices (Enclosure Management Processors or EMPs) that may be attached to the controller. The information that is returned in the **CAPI_ENVIRON_PROCESSOR_INFO** structure is the standard SCSI inquiry data.

handle is the handle of the controller that executes the command.

environProcessorIndex is the index of the EMP you are trying to find. This is a zero-based sequential index, so on the first call to this function, set index to zero. For the next call, set index to one, and so on. When the callback returns a value of CAPI_ERROR_NO_SUCH_ENVIRON_PROCESSOR, you are finished.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_FIND_NEXT_ENVIRON_PROCESSOR**
errorCode	**CAPI_ERROR_NO_SUCH_ENVIRON_PROCESSOR** means no more EMPs.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	If an EMP is found (i.e., as long as error code is not CAPI_NO_SUCH_ENVIRON_PROCESSOR), this points to a **CAPI_ENVIRON_PROCESSOR_INFO** structure.

Events:

Remarks :

In this document, the terms “environmental processor,” “environmental device,” “environmental unit,” “Enclosure Management Processor,” and “EMP” are used interchangeably.

Call this function with an increasing index value, starting at 0, until you receive an error code of CAPI_ERROR_NO_SUCH_ENVIRON_PROCESSOR. Use the found index values in the CAPI_U_EnvironRead and CAPI_U_EnvironWrite function calls.

This command issues a SCSI Inquiry command to each EMP. If the Inquiry succeeds, the Callback contains errorCode = CAPI_NO_ERROR and *u.inquiry* in the CAPI_ENVIRON_PROCESSOR struct contains valid inquiry data. In the unlikely event that the Inquiry fails, the callback contains errorCode = CAPI_ERROR_COMMAND_FAILED and *u.e* in the CAPI_ENVIRON_PROCESSOR struct contains valid status and sense data. In either case, the *empId, busId, targetId* and *lun* members of CAPI_ENVIRON_PROCESSOR are valid.

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
	See Capability Bits

See also:

CAPI_FindNextEnvironProcessor()
CAPI_U_EnvironRead()
CAPI_U_EnvironWrite()

Unified Force Offline NEW! in CAPI 3.4

Syntax:

```
CAPI_RC  CAPI_U_ForceOffline( CAPI_HANDLE        handle,
                              CAPI_MODULE_TYPE   moduleType,
                              CAPI_MODULE_INDEX  moduleIndex,
                              CAPI_U8            param3 );
```

Description:

Forces the replaceable module (FRU) offline. The module will carry out this request even if it affects performance (for example, putting one Data Manager offline in an active-active RAID system) and even if it affects availability (for example, putting a Data Manager offline in a RAID system when the other Data Manager is already offline). If the request affects availability, this command returns an error code indicating the problem, but that error code will be returned in param1, not in errorCode.

handle is the handle of the controller that executes the command.
moduleType is the type of FRU that is being put offline. At this writing, only CAPI_MODULE_TYPE_DM and CAPI_MODULE_TYPE_DG are supported.
moduleIndex identifies the specific module. This must be one of 0 through 3 for Data Gates. It must be CAPI_MODULE_A or CAPI_MODULE_B for Data Managers.
param3 is reserved for possible future use.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_FORCE_OFFLINE**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	error code that would have been returned if this was a call to CAPI_PutOffline
param2	
dataPtr	

Events:

Remarks:

If the specified Data Manager (DM) that is to be forced offline is the other DM (not the one processing this command), this is accomplished by asserting the hardware reset line of that DM board to kill it.

If the specified DM that is to be forced offline is the one processing this command, this is accomplished by asking the other DM to kill this DM by asserting the hardware reset line.

But if the specified controller board that is to be forced offline is the one processing this command and the other controller board is offline, this is accomplished by gracefully shutting down the controller board via software (equivalent to CAPI_ShutDownController or CAPI_PutOffline).

This function requires capability bit CAPI_CAPABILITY_3_REPLACEABLE_MODULE to be set.

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_ForceOffline()
CAPI_U_PutOffline()
CAPI_U_PutOnline()
CAPI_U_ForceOnline()

Unified Force Online NEW! in CAPI 3.4

Syntax:

```
CAPI_RC  CAPI_U_ForceOnline( CAPI_HANDLE         handle,
                             CAPI_MODULE_TYPE    moduleType,
                             CAPI_MODULE_INDEX   moduleIndex,
                             CAPI_U8             param3 );
```

Description:

Forces the replaceable module (FRU) online ungracefully. Putting a module online ungracefully means not running full diagnostics and not running compatibility checks to see if the hardware and firmware of the FRU are compatible with the other FRUs. **This command is only for Chaparral internal use and it is available only in beta builds, not in customer builds.**

handle is the handle of the controller that executes the command.
moduleType is the type of FRU that is being put offline. At this writing, only CAPI_MODULE_TYPE_DM and CAPI_MODULE_TYPE_DG are supported.
moduleIndex identifies the specific module. This must be one of 0 through 3 for Data Gates. It must be CAPI_MODULE_A or CAPI_MODULE_B for Data Managers.
param3 is reserved for possible future use.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_FORCE_ONLINE**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

Remarks:

This function requires capability bit CAPI_CAPABILITY_3_REPLACEABLE_MODULE to be set.

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_ForceOnline()
CAPI_U_ForceOffline()
CAPI_U_PutOffline()
CAPI_U_PutOnline()

Unified Free Cache NEW! in CAPI 3.4

Syntax:

```
CAPI_RC   CAPI_U_FreeCache( CAPI_HANDLE    controllerHandle,
                            CAPI_U8       *arraySerialNumber );
```

Description:

Frees memory used by the write-back cache in the controller for a specific array. Discards any data that is not flushed to the drive.
Not implemented yet. Use CAPI_FreeCache.

handle is the handle of the controller that executes the command.
arraySerialNumber serial number of array with orphan data(from CAPI_EVENT_ORPHAN_DATA)

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback*:*

replyCode	**CAPI_REPLY_U_CACHE_FREE**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

Remarks :

In the event of a catastrophic array failure (such as a multiple drive failure under RAID 5), or if an array is moved from one controller to another, the controller is unable to flush cached write data to the array. To make this memory available to other arrays, free cache causes this memory to be made free for use to other arrays. The data is not written to the disks and is permanently lost. Use CAPI_EVENT_ORPHAN_DATA to trigger this command.

	Lengthy Operation
✔	Need Current Configuration
	May Change Configuration
	See Capability Bits

See also:

CAPI_FreeCache()
CAPI_U_FlushCache()
CAPI_U_SetCacheParams()

Unified Get Advanced Network Interface

Description:

There is no need for a unified version of this function because the members of the CAPI_ADVANCED_NETWORK_INTERFACE structure can be gotten and set with CAPI_U_GetControllerData and CAPI_U_SetControllerParams.

See also:

CAPI_GetAdvancedNetworkInterface()
CAPI_SetAdvancedNetworkInterface()
CAPI_U_GetControllerData()
CAPI_U_SetControllerParams()

Unified Get Array List NEW! in CAPI 3.4

Syntax:

```
CAPI_RC   CAPI_U_GetArrayList( CAPI_HANDLE          handle,
                               CAPI_CONTROLLER_ID   controllerId );
```

Description:

This returns an array of CAPI_ARRAY structures.

handle is the handle of the controller that executes the command.
controllerId specifies which controller to get the array list from; one of: CAPI_CONTROLLER_A, CAPI_CONTROLLER_B, CAPI_CONTROLLER_BOTH.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_GET_ARRAY_LIST**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	Number of CAPI_ARRAY structs returned.
param2	Configuration sequence number for controller A.
param3	Configuration sequence number for controller B.
dataPtr	Pointer to the first element of an array of **CAPI_ARRAY** structures; there are param1 elements in the array.

Events:

Remarks:

The application developer needs to make sure that the configuration sequence number on their copy of the array list (an array of CAPI_ARRAY structures retrieved with a call to CAPI_GetArrayList) matches the configuration sequence number on their copy of CAPI_CONTROLLER (retrieved with a call to CAPI_UpdateController). A CAPI_ERROR_FAILURE_DUE_TO_CONFIG_CHANGE will occur if a configuration change is attempted with incompatible structures.

The list of arrays is returned sorted by the creation timestamp. Note that this means that if *controllerId* is specified as CAPI_CONTROLLER_BOTH, arrays owned by both controllers will be sorted into one list.

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
✔	See Capability Bits

See also:

CAPI_GetArrayList()

Unified Get Array Partitions NEW! in CAPI 3.4

Syntax:

```
CAPI_RC  CAPI_U_GetArrayPartitions( CAPI_HANDLE   handle,
                                    CAPI_U8      *arraySerialNumber );
```

Description:

Gets a list of partitions contained in the specified array.

handle is the handle of the controller that executes the command.
arraySerialNumber is a pointer to the serial number of the array which contains the partitions.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_GET_ARRAY_PARTITIONS**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	Number of CAPI_ARRAY_PARTITION structs returned.
param2	Configuration sequence number of the controller that owns the array.
dataPtr	Pointer to the first element of an array of **CAPI_ARRAY_PARTITION** structures; there are param1 elements in the array.

Events:

Remarks :

This command is valid if the CAPI_CAPABILITY_2_ARRAY_PARTITIONS capability flag is set. The maximum number of partitions supported by one array is given by CAPI_MAX_PARTITIONS_PER_ARRAY.

	Lengthy Operation
✔	Need Current Configuration
	May Change Configuration
✔	See Capability Bits

See also:

CAPI_GetArrayPartitions()
CAPI_U_AddArrayPartition()

Unified Get Config Sequence Number NEW! in CAPI 3.4

Syntax:

```
CAPI_RC  CAPI_U_GetConfigSequenceNumber( CAPI_HANDLE  handle );
```

Description:

Replies with both controllers' current configuration sequence numbers, which can be used to determine if a controller structures update is required (CAPI_U_GetControllerData, CAPI_U_GetDriveList, CAPI_U_GetArrayList). (See commands in this document with a check next to "Need Current Configuration"; for example, CAPI_U_SetControllerParams.)

handle is the handle of the controller that executes the command.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_GET_CONFIG_SEQ_NUMBER**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	Configuration sequence number for controller A.
param3	Configuration sequence number for controller B.
dataPtr	

Events:

Remarks:

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
✔	See Capability Bits

See also:

CAPI_GetConfigSequenceNumber()

Unified Get Controller Data NEW! in CAPI 3.4

Syntax:

```
CAPI_RC  CAPI_U_GetControllerData( CAPI_HANDLE  handle );
```

Description:

Returns a current copy of the CAPI_UNIFIED_CONTROLLER structure through a subsequent callback. This is used to update your copy of the unified controller structure.

handle is the handle of the controller that executes the command.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_GET_CONTROLLER_DATA**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	Configuration sequence number for controller A.
param3	Configuration sequence number for controller B.
dataPtr	Contains a pointer to a **CAPI_UNIFIED_CONTROLLER** structure.

Events:

Remarks :

A pointer to the structure is provided to the callback function in a temporary buffer that must be copied by the application into a permanent copy.

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
	See Capability Bits

See also:

CAPI_UpdateController() is the corresponding non-unified command.

Unified Get Debug Data NEW! in CAPI 3.4

Syntax:

```
CAPI_RC  CAPI_U_GetDebugData( CAPI_HANDLE               handle,
                              CAPI_DEBUG_DATA_REGION    region,
                              CAPI_U32                  debugDataOffset,
                              CAPI_CONTROLLER_ID        controllerId );
```

Description:

This command allows a CAPI application to get the debug data that has been logged in the controller. Debug data is logged by many parts of the controller software. Data is in ASCII text format and consists of printable characters plus space, tab, and new-line characters. Many lines start with a time stamp.

handle is the handle of the controller that executes the command.
region is the portion of the debug data to get.
debugDataOffset is the offset (in bytes, 0-based) at which to start retrieving the debug data in the controller.
controllerId specifies which controller to send this command to; one of: CAPI_CONTROLLER_A, CAPI_CONTROLLER_B.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_GET_DEBUG_DATA**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	The number of characters that have been put in the data buffer pointed to by *dataPtr*.
param2	
dataPtr	CAPI_CHAR *

Events:

Remarks:

The data is not null-terminated; use param1 to determine how much data is available. There may be garbage characters in the data buffer after the valid data.

Notes on using *debugDataOffset*

The area on a controller that is dedicated to saving debug data is typically several hundred kilobytes. It is not possible to get all of this data in one call to this function, because of size limitations of the data buffer in the LMX. The maximum size of a block of data that will be returned by a call to this function is CAPI_MAX_DEBUG_DATA_PER_GET (defined as 32768 as of this writing). Your CAPI application should call this function repeatedly (with the region set to the same value) until it returns with param1 set to a value that is less than CAPI_MAX_DEBUG_DATA_PER_GET. Each time you call this function, you should increase the value of *debugDataOffset* by CAPI_MAX_DEBUG_DATA_PER_GET, starting with 0. For example, if a particular controller has debug data in the boot-up region that has a total size of 70000 bytes, the first time your app calls this command, *debugDataOffset* should be set to 0 and the callback will contain 32768 characters and param1 will be 32768. The second time the app calls this command, *debugDataOffset* should be set to 32768 and the callback will contain 32768 characters and param1 will be 32768. The third time the app calls this command, *debugDataOffset* should be set to 65536. This call will get the remaining 4464 characters (70000 – 65536 = 4464). The callback will contain 4464 characters and param1 will be 4464. Your app should concatenate these 3 blocks of data for display to a user.

If the *debugDataOffset* is beyond the end of valid debug data, 0 characters will be put in the data buffer and param1 will be 0.

When this function is called with offset = 0, a snapshot copy is made of the debug data in the specified region. Subsequent calls to this function with offset != 0 will retrieve data from that snapshot buffer.

WARNING: If more than one application is calling this function at the same time, there is the potential for interaction between the applications and the data that it retrieved may not be the desired data. (This is because the large buffer sizes involved require that all CAPI apps share a single, global snapshot buffer.)

Organization of the debug data into regions

The debug data is organized into 6 separate regions. They are:

- Boot-up prints (region = CAPI_DEBUG_DATA_REGION_BOOT_LOG)
- 4 crash-dump regions (region = CAPI_DEBUG_DATA_REGION_CRASH_LOG1 through 4)
- General debug prints (region = CAPI_DEBUG_DATA_REGION_PRINT_LOG)

Note that a CAPI application should not assume a region is any particular size, since this will vary from product to product and may vary with future releases of a product. Instead, the application should keep asking for data until param1 indicates all data has been retrieved, as discussed above. But to give you some idea as to the size, as of this writing the boot-up region is 20480 bytes; the other regions are each 102400 bytes.

Each region fills up from the lowest address. If the buffer has not filled up, param1 will indicate how many bytes of data you have received, and this number may even be 0. Once the buffer fills up, older data will be lost. The oldest line of debug data may be an incomplete line.

The 4 crash-dump regions wrap in this way: Crash-dump region 1 is used to save the first crash, then the second crash-dump region is used to save the second crash, and so on till all 4 are used, then the first crash-dump region is reused, then successive crash-dump regions are reused.

If a controller is gracefully shut down or put off line (for example, via CAPI_PutOffline, CAPI_RebootController, or CAPI_ShutDownController), all the debug data is cleared. If a controller is ungracefully shut down or forced off line (for example, killed by the other controller, or the power is shut off, or via CAPI_ForceOffline) then the debug data will be preserved in battery-backed RAM on the controller.

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
	See Capability Bits

See also:

CAPI_GetDebugData()

Unified Get Drive Error Statistics NEW! in CAPI 3.4

Syntax:

```
CAPI_RC  CAPI_U_GetDriveErrorStatistics( CAPI_HANDLE  handle,
                                         CAPI_U8     *driveSerialNumber );
```

Description:

This command gets drive error statistics for a specified disk drive.
Not implemented yet. Use CAPI_GetDriveErrorStatistics.

handle is the handle of the controller that executes the command.
driveSerialNumber is a pointer to a drive serial number.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_GET_DRIVE_ERROR_STATS**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	Pointer to a **CAPI_DRIVE_ERROR_STATS** structure.

Events:

Remarks:

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
	See Capability Bits

See also:

CAPI_GetDriveErrorStatistics()
CAPI_U_ResetDriveErrorStatistics()

Unified Get Drive List NEW! in CAPI 3.4

Syntax:

```
CAPI_RC   CAPI_U_GetDriveList( CAPI_HANDLE          handle,
                               CAPI_CONTROLLER_ID   controllerId );
```

Description:

This returns an array of CAPI_DRIVE structures.

handle is the handle of the controller that executes the command.
controllerId specifies which controller to get the drive list from; one of: CAPI_CONTROLLER_A, CAPI_CONTROLLER_B, CAPI_CONTROLLER_BOTH.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_GET_DRIVE_LIST**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	Number of CAPI_DRIVE structs returned.
param2	Configuration sequence number for controller A.
param3	Configuration sequence number for controller B.
dataPtr	Pointer to the first element of an array of **CAPI_DRIVE** structures; there are param1 elements in the array.

Events:

Remarks:

The application developer needs to make sure that the configuration sequence number on their copy of the drive list (an array of CAPI_DRIVE structures retrieved with a call to CAPI_U_GetDriveList) matches the configuration sequence number on their copy of CAPI_UNIFIED_CONTROLLER (retrieved with a call to CAPI_U_GetControllerData). A CAPI_ERROR_FAILURE_DUE_TO_CONFIG_CHANGE will occur if a configuration change is attempted with incompatible structures.

A controller does not have visibility to drives that are members of an array owned by the other controller nor to drives that are dedicated spares of an array owned by the other controller, and therefore does not return these drives in its list of drives. For example, if CAPI_CONTROLLER_A is specified for *controllerId,* drives that are members or dedicated spares of arrays owned by Controller B are not included in the list of drives.

If CAPI_CONTROLLER_BOTH is specified for *controllerId*, lists from both controllers are combined into a single list. The controller combines them by starting with Controller A's list, then compares the drive serial number of each element of Controller B's list against Controller A's list and adds it to the list after Controller A's list if its serial number differs from all elements of Controller A's list. If a developer of a CAPI app doesn't like this algorithm, the app can call CAPI_U_GetDriveList twice, once to get Controller A's list and then again to get Controller B's list, and deal with the 2 lists any way the developer chooses.

While it is handy to be able to specify CAPI_CONTROLER_BOTH for *controllerId* to get a combined list of drives, note the following caveats:

- The *dualPorted* member of CAPI_DRIVE may differ between Controller A's list and Controller B's list in the event of a hardware failure that causes one of the two controllers to be connected as single-ported; thus, when CAPI_CONTROLLER_BOTH is specified for *controllerId*, only Controller A's value for *dualPorted* is returned.

- The value of struct member *seeErrorStats* may also differ between the two controllers; a logical “or” of the values from the two controllers is returned in this case.
- The value of struct member *blinking* may also differ between the two controllers (since only the controller that executed the command to blink the drive will have this flag set in CAPI_DRIVE); a logical “or” of the values from the two controllers is returned in this case to accurately reflect the state of the drive.

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
✔	See Capability Bits

See also:

CAPI_GetDriveList()

Unified Get Event NEW! in CAPI 3.4

Syntax:

```
CAPI_RC   CAPI_U_GetEvent( CAPI_HANDLE          handle,
                           CAPI_U32             eventNum,
                           CAPI_CONTROLLER_ID   controllerId );
```

Description:

Get event information from the controller.

handle is the handle of the controller that executes the command.
eventNum is the sequential number of the event to retrieve (zero is an invalid event number).
controllerId specifies which controller to get the event from; one of: CAPI_CONTROLLER_A, CAPI_CONTROLLER_B.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_GET_EVENT**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	The requested event sequence number.
param2	The first event sequence number available on the controller.
param3	The last event sequence number available on the controller.
dataPtr	A pointer to a **CAPI_EVENT** structure.

Events:

Remarks:

Event numbers start at one. If the controller reports that the last event sequence number is zero, then this indicates an empty event log.

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
	See Capability Bits

See also:

CAPI_GetEvent()
CAPI_U_GetFirstEvent()
CAPI_U_GetLastEvent()

Unified Get First Event NEW! in CAPI 3.4

Syntax:

```
CAPI_RC   CAPI_U_GetFirstEvent( CAPI_HANDLE          handle,
                                CAPI_CONTROLLER_ID   controllerId );
```

Description:

Gets the first event information in the event queue from the controller.

handle is the handle of the controller that executes the command.
controllerId specifies which controller to get the event from; one of: CAPI_CONTROLLER_A, CAPI_CONTROLLER_B.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_GET_FIRST_EVENT**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	The first event sequence number available on the controller.
param2	The event sequence number of the last controller power up; that is, the most recent event that has an event code of CAPI_EVENT_POWER_UP.
param3	The last event sequence number available on the controller.
dataPtr	A pointer to a **CAPI_EVENT** structure.

Events:

Remarks:

Event numbers start at one. If the controller reports that the last event sequence number is zero, then this indicates an empty event log. As the controller runs, the sequence numbers increment and the event trace will wrap. The first and last event numbers allow the application to determine how many events are in the event log.

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
	See Capability Bits

See also:

CAPI_GetFirstEvent()
CAPI_U_GetEvent()
CAPI_U_GetLastEvent()

Unified Get Free Array Partitions NEW! in CAPI 3.4

Syntax:

```
CAPI_RC   CAPI_U_GetFreeArrayPartitions( CAPI_HANDLE   handle,
                                         CAPI_U8      *arraySerialNumber );
```

Description:

Gets the list of free array partitions contained in the specified array. These are essentially the unpartitioned or “free” areas on the array. Each of these free areas is a location where a new partition can be added or into which an adjacent (and physically lower) partition can be expanded.

handle is the handle of the controller that executes the command.
arraySerialNumber is a pointer to the serial number of the array that contains the free partitions.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_GET_FREE_ARRAY_PARTITIONS**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	Number of CAPI_ARRAY_PARTITION structs returned.
param2	Configuration sequence number of the controller that owns this array.
dataPtr	Pointer to the first element of an array of **CAPI_ARRAY_PARTITION** structures; there are param1 elements in the array.

Events:

Remarks :

This command is valid if the CAPI_CAPABILITY_2_ARRAY_PARTITIONS capability bit is set. The maximum number of free partitions supported by one array is given by CAPI_MAX_FREE_PARTITIONS_PER_ARRAY. Note that the only fields of interest in the returned CAPI_ARRAY_PARTITION structure are *startLba* and *sizeLba*.

	Lengthy Operation
✔	Need Current Configuration
	May Change Configuration
✔	See Capability Bits

See also:

CAPI_GetFreeArrayPartitions()
CAPI_U_AddArrayPartition()
CAPI_U_GetArrayPartitions()

Unified Get Host Nicknames NEW! in CAPI 3.4

Syntax:

```
CAPI_RC  CAPI_U_GetHostNicknames( CAPI_HANDLE  handle );
```

Description:

This command allows a CAPI application to get a structure containing a list of all hosts that have nicknames defined. This structure maps worldwide names to nicknames. This mapping can be used by a CAPI application to allow a user to use nicknames instead of worldwide names.

handle is the handle of the controller that executes the command.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_GET_HOST_NICKNAMES**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	Pointer to a **CAPI_HOST_NICKNAMES** structure.

Events:

Remarks:

See CAPI_U_AddHost for a discussion of how a typical application might best use this command.

This function requires capability bit CAPI_CAPABILITY_2_INFOSHIELD to be set.

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
✔	See Capability Bits

See also:

CAPI_GetHostNicknames()
CAPI_U_AddHostNickname()
CAPI_U_GetKnownHosts()
CAPI_U_AddHost()
CAPI_U_RemoveHost()

Unified Get Host Table NEW! in CAPI 3.4

Syntax:

```
CAPI_RC   CAPI_U_GetHostTable( CAPI_HANDLE  handle,
                               CAPI_U32     channelIndex,
                               CAPI_U32     unitNum,
                               CAPI_U8     *partitionSerialNumber );
```

Description:

This function returns the table of hosts that either do or do not have access to the specified *unitNum* or *partitionSerialNumber*.

handle is the handle of the controller that executes the command.
channelIndex host channel index that the array or device is being presented on.
unitNum LUN that this array or device is being presented as.
partitionSerialNumber is the serial number of the partition; if partitions are not supported (capability bit CAPI_CAPABILITY_2_ARRAY_PARTITIONS not set), then this is an array serial number. (Applies to RAID only; not routers.)

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_GET_HOST_TABLE**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	Pointer to a **CAPI_HOST_TABLE** structure.

Events:

Remarks:

If *partitionSerialNumber* is not NULL, it will be used; if it is NULL, *channelIndex* and *unitNum* will be used.

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_GetHostTable()
CAPI_U_AddHost()
CAPI_U_RemoveHost()
CAPI_U_ChangeInfoshieldType()

Unified Get Known Hosts NEW! in CAPI 3.4

Syntax:

```
CAPI_RC  CAPI_U_GetKnownHosts( CAPI_HANDLE         handle,
                               CAPI_CONTROLLER_ID  controllerId );
```

Description:

This function returns the table of hosts that are known to have communicated with the controller.

handle is the handle of the controller that executes the command.
controllerId specifies which controller to get the list of known hosts from; one of: CAPI_CONTROLLER_A, CAPI_CONTROLLER_B, CAPI_CONTROLLER_BOTH.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_GET_KNOWN_HOSTS**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	Pointer to a **CAPI_KNOWN_HOSTS** structure.

Events:

Remarks:

See CAPI_U_AddHost for a discussion of how a typical application might best use this command.

The list can contain up to 64 hosts; if more hosts contact the controller than 64, the oldest entries are dropped.

If controllerId is specified as CAPI_CONTROLLER_BOTH, then the CAPI_KNOWN_HOSTS struct will contain all hosts known to either or both controllers. Some hosts may be known to both controllers; other hosts may only be known to one controller. We start by getting a list of known hosts from controller A and a separate list of known hosts from controller B. We combine them by starting with Controller A's list, then we compare each element of Controller B's list against Controller A's list and add it to the list after Controller A's list if it differs from all elements of Controller A's list. We set the value of the controllerId field in the list as we go through this algorithm, marking each element of the list as known to A-only, B-only, or BOTH. Then we sort the list by the controllerId field so it is in order of BOTH, A, B. If a developer of a CAPI app doesn't like this algorithm, the app can call CAPI_U_GetKnownHosts twice, once to get Controller A's list and then again to get Controller B's list, and deal with them any way the developer chooses.

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_GetKnownHosts()
CAPI_U_GetHostTable()

Unified Get Last Event NEW! in CAPI 3.4

Syntax:

```
CAPI_RC  CAPI_U_GetLastEvent( CAPI_HANDLE          handle,
                              CAPI_CONTROLLER_ID   controllerId );
```

Description:

Gets the last event information in the event queue from the controller.

handle is the handle of the controller that executes the command.
controllerId specifies which controller to get the last event from; one of: CAPI_CONTROLLER_A, CAPI_CONTROLLER_B.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_GET_LAST_EVENT**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	The last event sequence number available on the controller.
param2	The first event sequence number available on the controller.
dataPtr	A pointer to a **CAPI_EVENT** structure.

Events:

Remarks:

Event numbers start at one. If the controller reports that the last event sequence number is zero, then this indicates an empty event log.

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
	See Capability Bits

See also:

CAPI_GetLastEvent()
CAPI_U_GetEvent()
CAPI_U_GetFirstEvent()

Unified Get Percent Complete NEW! in CAPI 3.4

Syntax:

```
CAPI_RC  CAPI_U_GetPercentComplete( CAPI_HANDLE  handle,
                                    CAPI_U8     *arraySerialNumber );
```

Description:

Returns the percent complete of the currently running utility.

handle is the handle of the controller that executes the command.
arraySerialNumber is a pointer to the serial number of the target array.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_GET_PERCENT_COMPLETE**
errorCode	Completion status. If successful, param1 contains a valid percentage.
identifier	controllerHandle is valid.
param1	Contains the percent complete value as a 32-bit unsigned integer.
param2	Contains the **CAPI_UTILITY_RUNNING** type of utility running.
dataPtr	

Events:

Remarks :

If param2 equals CAPI_NO_UTILITY_RUNNING, then param1 is undefined.

	Lengthy Operation
✔	Need Current Configuration
	May Change Configuration
	See Capability Bits

See also:

CAPI_GetPercentComplete

Unified Get SCSI Maintenance Data NEW! in CAPI 3.4

Syntax:

```
CAPI_RC   CAPI_U_GetScsiMaintenanceData( CAPI_HANDLE  handle,
                                         CAPI_U32     dataBufferSize );
```

Description:

Retrieves the additional data returned from a maintenance command.

handle is the handle of the controller that executes the command.
dataBufferSize is the number of bytes of data that you want to be returned in the CAPI_MAINT_DATA_STRUCT. You can specify a value that is appropriate for the SCSI command sent by CAPI_ScsiMaintenance. The maximum size that can be specified is sizeof(CAPI_MAINT_DATA_STRUCT); if you specify more that that number of bytes, only that many bytes will be returned.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_GET_SCSI_MAINT_DATA**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	The number of bytes of data that have been put in the data buffer pointed to by *dataPtr*. (Normally equal to *dataBufferSize*, but never greater than sizeof(CAPI_MAINT_DATA_STRUCT).)
param2	
dataPtr	A pointer to a **CAPI_MAINT_DATA_STRUCT**. See page 82.

Events:

Remarks:

If you need to get more than the amount of data that will fit in CAPI_MAINT_DATA_STRUCT, use the non-CAPI SCSI pass through capability described in Chapter 17.

Note that all calls to CAPI_ScsiMaintenance and CAPI_U_DoScsiMaintenance make use of a single buffer. Thus, it is important that one SCSI maintenance operation be complete before the next one starts. The sequence of commands should be as follows:

- Call CAPI_ScsiMaintenance or CAPI_U_DoScsiMaintenance.
- Wait for an event to be posted to indicate that the operation is complete (normally CAPI_EVENT_SCSI_MAINT_DONE).
- Call CAPI_ScsiMaintRetrieveData or CAPI_U_GetScsiMaintenanceData to get any data.
- Repeat this sequence as desired.

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
✔	See Capability Bits

See also:

CAPI_ScsiMaintRetrieveData() is the corresponding non-unified command.
CAPI_U_DoScsiMaintenance()

Unified Log Event NEW! in CAPI 3.4

Syntax:

```
CAPI_RC   CAPI_U_LogEvent( CAPI_HANDLE          handle,
                           CAPI_EVENT          *event,
                           CAPI_CONTROLLER_ID   controllerId );
```

Description:

This command allows a CAPI application to make an entry in the event log that is maintained by and on a Chaparral controller board. **This command is for Chaparral internal use only.**

handle is the handle of the controller that executes the command.
event is a pointer to a structure containing the event data to be logged.
controllerId specifies which controller to get the last event from; one of: CAPI_CONTROLLER_A, CAPI_CONTROLLER_B.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_LOG_EVENT**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

Remarks:

This command is intended for use by Chaparral's software only (specifically, to allow the LAN Subsystem to log events in the event log maintained by the Storage Controller processor). This function should not be used by external CAPI applications to avoid using up the limited space available for events (400 events at this writing).

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
	See Capability Bits

See also:

CAPI_LogEvent()

Unified Pause Bus NEW! in CAPI 3.4

Syntax:

```
CAPI_RC  CAPI_U_PauseBus( CAPI_HANDLE  handle,
                          CAPI_U32     channelIndex );
```

Description:

Suspends I/O to all back-end SCSI buses.

handle is the handle of the controller that executes the command.
channelIndex is the index of the bus or channel on the specified controller. However, this parameter is not used at this time. By default, all buses will be paused.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_PAUSE_BUS**
errorCode	Status of the operation. If successful, the disk channels are paused.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

Remarks :

While some connectors are designed to allow hot-plugging SCSI drives, most are not. In all cases, the SCSI bus should be paused to prevent corrupted data. If a SCSI drive is inserted or removed from the bus, the pins may disrupt the signals. This function can be used to pause I/O on the bus while drives are added or removed.

After a call to CAPI_PauseBus, the bus remains paused until a call to CAPI_UnpauseBus. When the pause is issued, any SCSI commands currently in progress are allowed to complete. Any SCSI commands received after the pause is issued are queued by the RAID controller. If the queue becomes full, a status of queue full is returned to the host via the SCSI interface. Pass CAPI_NULL_ID in *channelIndex* to pause all buses.
This command may not be implemented on this controller or you may not be able to pause individual buses. See *CAPI Versions*
Different Chaparral products support different versions of CAPI. If the major version (for example, the 3 in version 3.4) is the same for two products, you should be able to easily design a single CAPI app that manages both products, although the product with the lower minor version number (for example, the 4 in version 3.4) will not have all the features of the product with the higher minor version number. Specifically, there may be additional CAPI commands added for a higher version number and there may be additional members in some of the data structures passed to and from CAPI commands, but the members that existed in the lower version of CAPI are still in the same locations in the same structures.

When a new version of the CAPI SDK is released, typically the most recent Chaparral product(s) are released simultaneously with the SDK and the version of CAPI that is running in the controller corresponds to the SDK version. However, developers of new CAPI apps can use a more recent version of the CAPI SDK to talk to older products, provided the major version number matches, and, of course, the app cannot use the newest features that have been added to CAPI which are not supported on that product.

Conversely, if a CAPI app was developed using an older version of the CAPI SDK (for example, CAPI 3.2), it can still be used for managing a newer product that supports a newer version of CAPI (for example, CAPI3.4), but of course the app will not be able to take advantage of the newer features that have been added to CAPI 3.4 in the product but which are not in the CAPI 3.2 SDK.

As of this writing (September 2002), the following products support the corresponding version of CAPI:

- CAPI 2.x: G5312, G7313, all Kxxxx.
- CAPI 3.4: RIO, Stratis RAID S3300 (JFF224). (Thus, the features marked "NEW! in CAPI 3.3" and "NEW! in CAPI 3.4" in this document are supported by these products only.)
- CAPI 3.2: All other Chaparral products.

CAPI Capabilities on page 29. Requires CAPI_CAPABILITY_2_PAUSE_INDIVIDUAL_BUS set to pause an individual bus.

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
✔	See Capability Bits

See also:

CAPI_PauseBus()
CAPI_U_UnpauseBus()

Unified Put Offline NEW! in CAPI 3.4

Syntax:

```
CAPI_RC  CAPI_U_PutOffline( CAPI_HANDLE        handle,
                            CAPI_MODULE_TYPE   moduleType,
                            CAPI_MODULE_INDEX  moduleIndex,
                            CAPI_U8            param3 );
```

Description:

Puts the replaceable module (FRU) offline gracefully. The controller will carry out this request even if it affects performance (for example, putting one Data Manager offline in an active-active RAID system), but will reject this request if it affects availability (for example, putting a Data Manager offline in a RAID system when the other Data Manager is already offline). If the request is rejected, this command returns an errorCode indicating the problem.

handle is the handle of the controller that executes the command.
moduleType is the type of FRU that is being put offline. At this writing, only CAPI_MODULE_TYPE_DM and CAPI_MODULE_TYPE_DG are supported.
moduleIndex identifies the specific module. This must be one of 0 through 3 for Data Gates. It must be CAPI_MODULE_A or CAPI_MODULE_B for Data Managers.
param3 is reserved for possible future use.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_PUT_OFFLINE**
errorCode	Completion status of the command.
identifier	ControllerHandle is valid.
param1	
param2	
dataPtr	

Events:

Remarks:

Putting a Data Manager (DM) offline is equivalent to shutting it down.

CAPI_ShutDownController provides similar functionality to this function. However, that function can only act on Data Managers and it can shut down both controller boards with a single function call.

Calling CAPI_PutOffline is equivalent to calling CAPI_ShutDownController for a single DM with *fwUpdate* set to FALSE except that CAPI_PutOffline does availability checking first.

This function requires capability bit CAPI_CAPABILITY_3_REPLACEABLE_MODULE to be set.

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_PutOffline()
CAPI_U_PutOnline()
CAPI_U_ForceOffline()
CAPI_U_ForceOnline()
CAPI_U_ShutDownController()

Unified Put Online NEW! in CAPI 3.4

Syntax:

```
CAPI_RC   CAPI_U_PutOnline( CAPI_HANDLE         handle,
                            CAPI_MODULE_TYPE    moduleType,
                            CAPI_MODULE_INDEX   moduleIndex,
                            CAPI_U8             param3 );
```

Description:

Puts the replaceable module (FRU) online gracefully. Putting a module online gracefully means running its diagnostics and running compatibility checks to see if the hardware and firmware of the FRU are compatible with the other FRUs. If these checks do not pass, this command returns an errorCode indicating the problem.

handle is the handle of the controller that executes the command.
moduleType is the type of FRU that is being put offline. At this writing, only CAPI_MODULE_TYPE_DM and CAPI_MODULE_TYPE_DG are supported.
moduleIndex identifies the specific module. This must be one of 0 through 3 for Data Gates. It must be CAPI_MODULE_A or CAPI_MODULE_B for Data Managers.
param3 is reserved for possible future use.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_PUT_ONLINE**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

Remarks:

This function requires capability bit CAPI_CAPABILITY_3_REPLACEABLE_MODULE to be set.

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_PutOnline()
CAPI_U_PutOffline()
CAPI_U_ForceOffline()
CAPI_U_ForceOnline()

Unified Reboot Controller NEW! in CAPI 3.4

Syntax:

```
CAPI_RC   CAPI_U_RebootController( CAPI_HANDLE          handle,
                                   CAPI_CONTROLLER_ID   controllerId );
```

Description:

This command does the same thing as CAPI_ShutDownController or CAPI_U_ShutDownController and then reboots. It is also used to reboot a controller when it is in a shutdown state as a result of CAPI_ShutDownController or CAPI_U_ShutDownController.

handle is the handle of the controller that executes the command.
controllerId specifies which controller you want to reboot; one of: CAPI_CONTROLLER_A, CAPI_CONTROLLER_B, CAPI_CONTROLLER_BOTH.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_REBOOT_CONTROLLER_START**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_CONTROLLER_REBOOT_COMPLETE

Remarks :

Rebooting will flush the controller's write back cache to disk.

See CAPI_U_ShutDownController for additional information.

✔	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_RebootController()
CAPI_U_UpdateFirmware()
CAPI_U_ShutDownController()

Unified Remove Host NEW! in CAPI 3.4

Syntax:

```
CAPI_RC  CAPI_U_RemoveHost( CAPI_HANDLE   handle,
                            CAPI_U8      *partitionSerialNumber,
                            CAPI_FLEX_ID  hostId );
```

Description:

This function removes a host from the list of hosts that is allowed to communicate with a particular partition or is blocked from communication with a particular partition.

handle is the handle of the controller that executes the command.
partitionSerialNumber is the serial number of the partition; if partitions are not supported (capability bit CAPI_CAPABILITY_2_ARRAY_PARTITIONS not set), then this is an array serial number.
hostId is the Fibre Channel or SCSI ID of the host.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_REMOVE_HOST**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

Remarks:

The list of hosts associated with this partition is either allowed access or blocked from access by the value of the *include* parameter in CAPI_U_ChangeInfoShieldType.

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_RemoveHost()
CAPI_U_GetHostTable()
CAPI_U_AddHost()
CAPI_U_ChangeInfoShieldType()

Unified Rescan Bus NEW! in CAPI 3.4

Syntax:

```
CAPI_RC  CAPI_U_RescanBus( CAPI_HANDLE  handle,
                           CAPI_U32     channelIndex );
```

Description:

Scans the drives on the back-end drive bus to detect new, moved, or deleted drives.

handle is the handle of the controller that executes the command.
channelIndex is the index of the channel to rescan. Pass CAPI_NULL_ID to rescan all channels.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_RESCAN_BUS_START**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_RESCAN_BUS_START
CAPI_EVENT_RESCAN_BUS_COMPLETE

Remarks :

To avoid any performance degradation, the controller does not scan the SCSI buses for changes in configuration unless instructed to do so through CAPI or SAF-TE. This function should be called after new SCSI drives are added, if drives are moved to different IDs, or if unused or spare drives are removed. SAF-TE processors can do automatic rescans. Some controllers may do a rescan on a SCSI bus reset.

See *CAPI Versions*
Different Chaparral products support different versions of CAPI. If the major version (for example, the 3 in version 3.4) is the same for two products, you should be able to easily design a single CAPI app that manages both products, although the product with the lower minor version number (for example, the 4 in version 3.4) will not have all the features of the product with the higher minor version number. Specifically, there may be additional CAPI commands added for a higher version number and there may be additional members in some of the data structures passed to and from CAPI commands, but the members that existed in the lower version of CAPI are still in the same locations in the same structures.

When a new version of the CAPI SDK is released, typically the most recent Chaparral product(s) are released simultaneously with the SDK and the version of CAPI that is running in the controller corresponds to the SDK version. However, developers of new CAPI apps can use a more recent version of the CAPI SDK to talk to older products, provided the major version number matches, and, of course, the app cannot use the newest features that have been added to CAPI which are not supported on that product.

Conversely, if a CAPI app was developed using an older version of the CAPI SDK (for example, CAPI 3.2), it can still be used for managing a newer product that supports a newer version of CAPI (for example, CAPI3.4), but of course the app will not be able to take advantage of the newer features that have been added to CAPI 3.4 in the product but which are not in the CAPI 3.2 SDK.

As of this writing (September 2002), the following products support the corresponding version of CAPI:

- CAPI 2.x: G5312, G7313, all Kxxxx.
- CAPI 3.4: RIO, Stratis RAID S3300 (JFF224). (Thus, the features marked "NEW! in CAPI 3.3" and "NEW! in CAPI 3.4" in this document are supported by these products only.)
- CAPI 3.2: All other Chaparral products.

CAPI Capabilities on page 29 to determine if the controller supports rescanning of individual channels; if not, then *channelIndex* should be CAPI_NULL_ID. Requires CAPI_CAPABILITY_2_RESCAN_INDIVIDUAL_BUS set to rescan an individual bus.

✔	Lengthy Operation
	Need Current Configuration
	May Change Configuration
✔	See Capability Bits

See also:

CAPI_RescanBus()

Unified Reset Array Statistics NEW! in CAPI 3.4

Syntax:

```
CAPI_RC   CAPI_U_ResetArrayStatistics( CAPI_HANDLE  handle,
                                       CAPI_U8     *arraySerialNumber );
```

Description:

Resets temporary array statistics.

handle is the handle of the controller that executes the command.
arraySerialNumber is a pointer to the serial number of the target array.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_RESET_ARRAY_STATS**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_RESET_ARRAY_STATS

Remarks :

This function clears array statistics but those are not visible from the Disk Array Administrator or through a CAPI app. In earlier versions of Chaparral products we were only able to create 1 partition per array. Now we are able to create 1 or more partitions in an array so the array statistics are not used anymore but are replaced with array partition statistics.

	Lengthy Operation
✔	Need Current Configuration
	May Change Configuration
✔	See Capability Bits

See also:

CAPI_ResetArrayStatistics()
CAPI_U_ResetArrayPartitionStatistics()

Unified Reset Array Partition Statistics NEW! in CAPI 3.4

Syntax:

```
CAPI_RC  CAPI_U_ResetArrayPartitionStatistics( CAPI_HANDLE  handle,
                                               CAPI_U8     *partitionSerialNumber );
```

Description:

Resets temporary array partition statistics.

handle is the handle of the controller that executes the command.
partitionSerialNumber is a pointer to the serial number of an existing partition.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_RESET_ARRAY_PARTITION_STATS**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_RESET_ARRAY_PARTITION_STATS

Remarks :

	Lengthy Operation
✔	Need Current Configuration
	May Change Configuration
✔	See Capability Bits

See also:

CAPI_ResetArrayPartitionStatistics()
CAPI_U_GetArrayPartitions()

Unified Reset Drive Error Statistics NEW! in CAPI 3.4

Syntax:

```
CAPI_RC   CAPI_U_ResetDriveErrorStatistics( CAPI_HANDLE  handle,
                                             CAPI_U8     *driveSerialNumber );
```

Description:

This command allows a CAPI application to reset the drive error statistics for a designated disk drive. All values are set to 0.
Not implemented yet. Use CAPI_ResetDriveErrorStatistics.

handle is the handle of the controller that executes the command.
driveSerialNumber is a pointer to a drive serial number.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_RESET_DRIVE_ERROR_STATS**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

Remarks:

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
	See Capability Bits

See also:

CAPI_ResetDriveErrorStatistics()
CAPI_U_GetDriveErrorStatistics()

Unified Reset LAN NEW! in CAPI 3.4

Syntax:

```
CAPI_RC   CAPI_U_ResetLAN( CAPI_HANDLE          handle,
                           CAPI_CONTROLLER_ID   controllerId );
```

Description:

Resets the LAN Subsystem if one is present.

handle is the handle of the controller that executes the command.
controllerId specifies which controller you want to reset its LAN processor; one of:
CAPI_CONTROLLER_A, CAPI_CONTROLLER_B, CAPI_CONTROLLER_BOTH.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_RESET_LAN**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_RESET_LAN

Remarks :

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
	See Capability Bits

See also:

CAPI_ResetLAN()

Unified Restore Controller Defaults NEW! in CAPI 3.4

Syntax:

```
CAPI_RC  CAPI_U_RestoreControllerDefaults( CAPI_HANDLE         handle,
                                           CAPI_CONTROLLER_ID  controllerId );
```

Description:

Restores the factory defaults of the controller.

handle is the handle of the controller that executes the command.
controllerId specifies which controller you want to restore defaults on; one of: CAPI_CONTROLLER_A, CAPI_CONTROLLER_B, CAPI_CONTROLLER_BOTH.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_RESTORE_CONTROLLER_DEFAULTS**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

Remarks :

A reboot is required for all defaults to take effect. See the controller's documentation to determine which defaults are restored immediately and which defaults take effect after the next reboot.

This command does *not* cause the following to be reset to defaults:
CAPI LUN (a.k.a. controller LUN or bridge LUN)
controller mode
drive channel speed
LAN Subsystem IP address
LAN Subsystem IP subnet mask
LAN Subsystem IP gateway

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
	See Capability Bits

See also:

CAPI_RestoreControllerDefaults()

Unified Set Advanced Network Interface

Description:

There is no need for a unified version of this function because the members of the CAPI_ADVANCED_NETWORK_INTERFACE structure can be gotten and set with CAPI_U_GetControllerData and CAPI_U_SetControllerParams.

See also:

CAPI_SetAdvancedNetworkInterface()
CAPI_GetAdvancedNetworkInterface()
CAPI_U_GetControllerData()
CAPI_U_SetControllerParams()

Unified Set Array Partition Cache Params NEW! in CAPI 3.4

Syntax:

```
CAPI_RC  CAPI_U_SetArrayPartitionCacheParams( CAPI_HANDLE        handle,
                                              CAPI_U8            *partitionSerialNumber,
                                              CAPI_CACHE_PARAMS  *cacheParams );
```

Description:

This command allows a CAPI application to set parameters that determine characteristics of the cache associated with the specified partition.

handle is the handle of the controller that executes the command.
partitionSerialNumber is a pointer to the serial number of an existing partition.
cacheParams is a pointer to a structure that contains the new values for the cache parameters.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_SET_ARRAY_PARTITION_CACHE_PARAMS**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

Remarks:

At this writing, the only parameters in CAPI_CACHE_PARAMS that are supported are *writeBackEnable* and *readAheadSize.*

When *writeBackEnable* is set to TRUE, the write back cache is enabled.

readAheadSize should be set to 0 or to a power of 2 between 64KB and 32MB or to the default. That is, use one of these values: 0 (which disables read ahead), 0x10000, 0x20000, 0x40000, 0x80000, 0x100000, 0x200000, 0x400000, 0x800000, 0x1000000, 0x2000000, or CAPI_DEFAULT_READ_AHEAD_SIZE (which tells the controller to use an algorithm that tries to optimize read ahead based on whether reads are sequential or random). More cache improves performance of sequential reads but will hurt performance of random reads.

To apply CAPI_CACHE_PARAMS to all partitions in an array via a single function call, you can use CAPI_SetCacheParams.

This function requires capability bit CAPI_CAPABILITY_2_ARRAY_PARTITIONS to be set.

> ***CAUTION:*** *The RAID controller's default cache parameters are preset to provide optimal performance for virtually all applications. Modification of these parameters may significantly decrease performance.*

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_SetArrayPartitionCacheParams()
CAPI_U_SetCacheParams()

Unified Set Battery Monitor NEW! in CAPI 3.4

Syntax:

```
CAPI_RC   CAPI_U_SetBatteryMonitor( CAPI_HANDLE         handle,
                                    CAPI_BOOL           monitorOn,
                                    CAPI_U8             months,
                                    CAPI_CONTROLLER_ID  controllerId );
```

Description:

This function sets the age of the battery and enables/disables end-of-life monitoring.

handle is the handle of the controller that executes the command.
monitorOn set to TRUE to enable battery life monitoring.
months set to the number of months the battery has been installed (set to zero if the controller is new).
controllerId specifies which controller you want to set the battery monitor on; one of:
CAPI_CONTROLLER_A, CAPI_CONTROLLER_B.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_SET_BATTERY_MONITOR**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

Remarks:

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
	See Capability Bits

CAPI_EVENT_BATTERY_END_OF_LIFE will occur at the end of the battery life.

See also:

CAPI_SetBatteryMonitor()

Unified Set Cache Params NEW! in CAPI 3.4

Syntax:

```
CAPI_RC   CAPI_U_SetCacheParams( CAPI_HANDLE          handle,
                                 CAPI_U8             *arraySerialNumber,
                                 CAPI_CACHE_PARAMS   *cacheParams );
```

Description:

This command allows a CAPI application to set parameters that determine characteristics of the cache associated with the specified array.

This function is not implemented yet. Use CAPI_SetCacheParams or CAPI_U_SetArrayPartitionCacheParams.

handle is the handle of the controller that executes the command.
arraySerialNumber is a pointer to the serial number of the target array. Pass a pointer to an array of 12 bytes of all zeros to configure all arrays with these parameters.
cacheParams points to a CAPI_CACHE_PARAMS structure containing the new cache parameter settings.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_SET_CACHE_PARAMS**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_SET_CACHE_PARAMS

Remarks :

Some of these parameters can also be set through SCSI mode pages from the host.

At this writing, the only parameter in CAPI_CACHE_PARAMS that is supported is *writeBackEnable*. When *writeBackEnable* is set to TRUE, the write back cache is enabled.

For more recent products that support multiple partitions (from RIO onward), *readAheadSize* is also supported. See CAPI_SetArrayPartitionCacheParams for details on this parameter.

Note that for arrays containing multiple partitions, the cache parameters for all partitions in the array are updated when this command is issued.

> ***CAUTION:*** *The RAID controller's default cache parameters are preset to provide optimal performance for virtually all applications. Modification of these parameters may significantly decrease performance.*

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_SetCacheParams()
CAPI_U_FlushCache()
CAPI_U_FreeCache()
CAPI_U_SetArrayPartitionCacheParams()

Unified Set Channel Params

Description:

There is no need for a unified version of this function because, as of CAPI 3.4, the channel parameters are now set as part of the data structure passed to the CAPI_U_SetControllerParams function.

See also:

CAPI_SetChannelParams()
CAPI_U_SetControllerParams()

Unified Set Controller Params NEW! in CAPI 3.4

Syntax:

```
CAPI_RC  CAPI_U_SetControllerParams( CAPI_HANDLE                        handle,
                                     CAPI_UNIFIED_CONTROLLER_PARAMS *controllerParams );
```

Description:

Sets the controller's parameters.

handle is the handle of the controller that executes the command.
controllerParams is a pointer to a CAPI_UNIFIED_CONTROLLER_PARAMS structure with the new controller settings.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_SET_CONTROLLER_PARAMS**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_SET_CONTROLLER_PARAMS

Remarks :

Developers can read the current parameters or pending parameters using CAPI_U_GetControllerData, modify the parameters, and update them with CAPI_U_SetControllerParams. Some parameters go into effect immediately, others require that the controller be restarted. Normally, an app should read and modify the pending parameters rather than the current parameters. This is because the pending parameters will reflect any changes that have been made by previous call(s) to CAPI_U_SetControllerParams but which have not gone into effect because a restart is required.

Note that parameters in CAPI_UNIFIED_CONTROLLER_PARAMS are divided into two classes: those parameters that are applied uniquely to each controller in a dual-controller system and those parameters that are applied to both controllers. A call to this function always sets the parameters on both controllers.

Note that this unified command sets the channel parameters, unlike CAPI_SetControllerParams which does not set channel parameters.

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
	See Capability Bits

See also:

CAPI_SetControllerParams() and CAPI_SetChannelParams() are the corresponding non-unified commands.
CAPI_U_GetControllerData()

Unified Set Controller Time Date NEW! in CAPI 3.4

Syntax:

```
CAPI_RC   CAPI_U_SetControllerTimeDate( CAPI_HANDLE   handle,
                                        CAPI_TIME     timeDate );
```

Description:

Sets the controller time and date settings on both controllers.

handle is the handle of the controller that executes the command.
timeDate contains the number of seconds since January 1, 1970 (i.e., UNIX time).

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_SET_CONTROLLER_TIMEDATE**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_SET_CONTROLLER_TIMEDATE

Remarks :

The standard library provided with many ‘C’ compilers includes functions for manipulating CAPI_TIME (of type time_t, usually an unsigned long) and generating a standard ‘tm’ structure. See time, gmtime, localtime, mktime, and strftime in your compiler’s documentation. Note that a *timeDate* value of zero is invalid.

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
	See Capability Bits

See also:

CAPI_SetControllerTimeDate()

Unified Set Preferred Owner NEW! in CAPI 3.4

Syntax:

```
CAPI_RC   CAPI_U_SetPreferredOwner( CAPI_HANDLE  handle,
                                    CAPI_U8     *arraySerialNumber );
```

Description:

Allows the application to change the owner of an array from one controller to another. A call to this function will result in a change from the current owner to the other controller, no matter which controller is the current owner.

handle is the handle of the controller that executes the command.
arraySerialNumber is a pointer to the serial number of the target array.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_SET_PREFERRED_OWNER**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

Remarks :

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_SetPreferredOwner()
CAPI_U_CreateArray()

Unified Set Unit Mapping NEW! in CAPI 3.4

Syntax:

```
CAPI_RC  CAPI_U_SetUnitMapping( CAPI_HANDLE  handle,
                                CAPI_U8     *arraySerialNumber,
                                CAPI_U32     newUnitNum );
```

Description:

Allows the application to change the LUN that an array presents to the host.

handle is the handle of the controller that executes the command.
arraySerialNumber is a pointer to the serial number of the target array.
newUnitNum is the desired LUN for the specified array.

Return Code:

Indicates if the request was sent to the RAID controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_SET_LUN_MAPPING**
errorCode	Completion status of the command. A LUN conflict will return an error.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_UNIT_MAPPING

Remarks :

A reboot may be necessary on some products for the new LUN mapping to take effect.

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
	See Capability Bits

See also:

CAPI_SetUnitMapping()
CAPI_U_CreateArray()

Unified Shut Down Controller NEW! in CAPI 3.4

Syntax:

```
CAPI_RC  CAPI_U_ShutDownController( CAPI_HANDLE          handle,
                                    CAPI_CONTROLLER_ID   controllerId,
                                    CAPI_BOOL            fwUpdate );
```

Description:

Perform a graceful shutdown on the specified controller.

handle is the handle of the controller that executes the command.
controllerId specifies which controller you want to shut down (CAPI_CONTROLLER_A, CAPI_CONTROLLER_B, or CAPI_CONTROLLER_BOTH.)
fwUpdate is set to true if a firmware update is to follow, this lets the other controller know why we are shutting down. This parameter does not affect this operation; it just provides information to the on-line controller so it is accessible via the *failoverReason* structure member obtainable via CAPI_UpdateController or CAPI_U_GetControllerData.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_SHUTDOWN_CONTROLLER**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	controllerId
param2	
dataPtr	

Events:

CAPI_EVENT_SHUTDOWN_CONTROLLER

Remarks :

Shutting down will flush the controllers' write back cache to disk. The controller shuts down, then calls the callback function. The controller is then in a special state where it does no data I/O and it responds only to a limited selection of CAPI commands, most notably:

- CAPI_UpdateFirmware/CAPI_U_UpdateFirmware
- CAPI_RebootController/CAPI_U_RebootController

(For a complete list of CAPI commands that are supported during shutdown, see column allowWhileShutdown in the table in file capicmdsup.c.)

Also, once a controller is shut down, its serial port will no longer respond to the CTRL-P then CTRL-Z character sequence (which is used to restore terminal mode after a serial CAPI application has run). The reason is that a CAPI_COMMAND_UPDATE_CONTROLLER_FIRMWARE request over the serial port could have the CTRL-P/CTRL-Z sequence embedded in its binary data, which if recognized would cause the serial port to unintentionally transition to terminal mode.

If both controllers are shut down at the same time via this function, both can receive firmware updates from the host in-band. If only one controller is shut down, the shut down controller cannot receive firmware downloads in-band since the one that is not shut down is "impersonating" the shut down controller to the host and so the shut down controller has no host interface. If both controllers are shut down one after the other, the second one to be shut down still has a host interface so it can receive firmware downloads in-band. No matter what sequence is used to shut a controller down, the RS-232 connection can be used to download firmware (except that RS-232 download of firmware via CAPI is not supported on RIO since the serial LMX is not supported on RIO).

	Lengthy Operation
	Need Current Configuration
✔	May Change Configuration
	See Capability Bits

See also:

CAPI_ShutDownController()
CAPI_U_UpdateFirmware()
CAPI_U_RebootController()
CAPI_U_PutOffline()

Unified Silence Alarm NEW! in CAPI 3.4

Syntax:

```
CAPI_RC  CAPI_U_SilenceAlarm( CAPI_HANDLE  handle );
```

Description:

This command temporarily silences both controllers' on-board audible alarm. (Depending on the storage system design, it may or may not silence an enclosure alarm produced by an EMP.) As soon as the controller has another event that causes it to turn on the alarm, the alarm will sound. To permanently disable the alarm, set the alarmMute field in the CAPI_UNIFIED_CONTROLLER_PARAMS structure and call CAPI_U_SetControllerParams.

handle is the handle of the controller that executes the command.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_SILENCE_ALARM**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

Remarks :

If the alarm is caused by unwritable cache data (see CAPI_EVENT_ORPHAN_DATA), the cache data is not purged. If the alarm is caused by A/D failure, the command is ignored and the alarm will stay on. If the alarm is not on, this command is accepted successfully, but ignored.

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
	See Capability Bits

See also:

CAPI_SilenceAlarm()
CAPI_U_SetControllerParams()

Unified Test Drive NEW! in CAPI 3.4

Syntax:

```
CAPI_RC   CAPI_U_TestDrive( CAPI_HANDLE  handle,
                            CAPI_U8     *driveSerialNumber );
```

Description:

Performs simple tests on a drive.

handle is the handle of the controller that executes the command.
driveSerialNumber is a pointer to the serial number of the drive to perform the test on.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_TEST_DRIVE**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

Remarks :

Currently, this command only executes a command that causes an indicator lamp on the specified drive to blink. In the future this command may be implemented to do additional testing of the drive that is nondestructive to the drive and the drive's data. See the controller's documentation.

	Lengthy Operation
✔	Need Current Configuration
	May Change Configuration
✔	See Capability Bits

See also:

CAPI_TestDrive()
CAPI_U_BlinkDrive()
CAPI_U_UnblinkDrive()

Unified Test Spares NEW! in CAPI 3.4

Syntax:

```
CAPI_RC  CAPI_U_TestSpares( CAPI_HANDLE  handle,
                            CAPI_BOOL    testSpares );
```

Description:

Enable or disable the RAID core's testing of spare drives to verify that they are still available. Applies to both controllers. Power up default is TRUE.

handle is the handle of the controller that executes the command.
testSpares can be set to TRUE to enable spare tests or to FALSE to disable spare tests.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_TEST_SPARES**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

Remarks :

This is a continuous background test on all spare drives (global spares and pool spares) until a subsequent call is made to disable the test. See the controller's documentation for specific implementation details. If a test fails, then a CAPI_EVENT_DOWN_DRIVE event is generated and the spare is removed from the spare list.

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
✔	See Capability Bits

See also:

CAPI_TestSpares()

Unified Trust Array NEW! in CAPI 3.4

Syntax:

```
CAPI_RC   CAPI_U_TrustArray( CAPI_HANDLE  handle,
                             CAPI_U8     *arraySerialNumber );
```

Description:

This function allows use of an array that is unusable because of failed drives. The data may be corrupt, and therefore this function should only be used for testing or data recovery purposes.

handle is the handle of the controller that executes the command.
arraySerialNumber is a pointer to the serial number of the target array.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_TRUST_ARRAY**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_TRUST_ARRAY

Remarks :

	Lengthy Operation
✔	Need Current Configuration
✔	May Change Configuration
	See Capability Bits

See also:

CAPI_TrustArray()

Unified Unblink Drive NEW! in CAPI 3.4

Syntax:

```
CAPI_RC  CAPI_U_UnblinkDrive( CAPI_HANDLE  handle,
                              CAPI_U8     *driveSerialNumber );
```

Description:

This command stops blinking of the drive's activity light.

handle is the handle of the controller that executes the command.
driveSerialNumber is a pointer to the serial number of the drive to unblink.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_UNBLINK_DRIVE**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

Remarks :

Blinking a drive activity light is initiated by a call to CAPI_U_BlinkDrive. The controller continues blinking the drive light until this function is called.

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
	See Capability Bits

See also:

CAPI_UnblinkDrive()
CAPI_U_BlinkDrive()

Unified Unpause Bus NEW! in CAPI 3.4

Syntax:

```
CAPI_RC  CAPI_U_UnpauseBus( CAPI_HANDLE  handle,
                            CAPI_U32     channelIndex );
```

Description:

Resumes I/O to the specified back-end SCSI bus.

handle is the handle of the controller that executes the command.
channelIndex is the index of the disk channel on the specified controller. Pass CAPI_NULL_ID to unpause all disk channels.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_UNPAUSE_BUS**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

Remarks :

I/O to a back-end SCSI bus is paused through a call to CAPI_PauseBus. This command may not be implemented on this controller or you may not be able to pause individual buses. See *CAPI Versions* Different Chaparral products support different versions of CAPI. If the major version (for example, the 3 in version 3.4) is the same for two products, you should be able to easily design a single CAPI app that manages both products, although the product with the lower minor version number (for example, the 4 in version 3.4) will not have all the features of the product with the higher minor version number. Specifically, there may be additional CAPI commands added for a higher version number and there may be additional members in some of the data structures passed to and from CAPI commands, but the members that existed in the lower version of CAPI are still in the same locations in the same structures.

When a new version of the CAPI SDK is released, typically the most recent Chaparral product(s) are released simultaneously with the SDK and the version of CAPI that is running in the controller corresponds to the SDK version. However, developers of new CAPI apps can use a more recent version of the CAPI SDK to talk to older products, provided the major version number matches, and, of course, the app cannot use the newest features that have been added to CAPI which are not supported on that product.

Conversely, if a CAPI app was developed using an older version of the CAPI SDK (for example, CAPI 3.2), it can still be used for managing a newer product that supports a newer version of CAPI (for example, CAPI3.4), but of course the app will not be able to take advantage of the newer features that have been added to CAPI 3.4 in the product but which are not in the CAPI 3.2 SDK.

As of this writing (September 2002), the following products support the corresponding version of CAPI:

- CAPI 2.x: G5312, G7313, all Kxxxx.
- CAPI 3.4: RIO, Stratis RAID S3300 (JFF224). (Thus, the features marked “NEW! in CAPI 3.3” and “NEW! in CAPI 3.4” in this document are supported by these products only.)

- CAPI 3.2: All other Chaparral products.

CAPI Capabilities on page 29. Requires CAPI_CAPABILITY_2_PAUSE_INDIVIDUAL_BUS set to unpause an individual bus.

	Lengthy Operation
	Need Current Configuration
	May Change Configuration
✔	See Capability Bits

See also:

CAPI_UnpauseBus()
CAPI_U_PauseBus()

Unified Update Firmware NEW! in CAPI 3.4

Syntax:

```
CAPI_RC  CAPI_U_UpdateFirmware( CAPI_HANDLE           handle,
                                CAPI_U8              *firmwareImage,
                                CAPI_U32              size,
                                CAPI_CONTROLLER_ID    controllerId );
```

Description:

Loads new firmware into the controller(s).

handle is the handle of the controller that executes the command.
firmwareImage is a pointer to the new firmware image to be loaded.
size is the size of the image in bytes.
controllerId specifies which controller you want to update the firmware on; one of:
CAPI_CONTROLLER_A, CAPI_CONTROLLER_B, CAPI_CONTROLLER_BOTH.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_UPDATE_CONTROLLER_FIRMWARE_START**
errorCode	**CAPI_NO_ERROR** indicates that the firmware image was received without errors.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_UPDATE_FIRMWARE_COMPLETE, posted after the controller reboots.

Remarks :

A call to CAPI_ShutdownController must precede this call.

Automatic reboot occurs if there are no errors updating the firmware.

Firmware updates are not permitted when orphan data is present in the controller.

> ***Note:*** *Since the firmware image is large, transfer of data from the host to the controller occurs as multiple messages, which are handled by code in the ReceivePacket function in capi2pak.c (part of the CAPI Client in the SDK). The callback function is not called until the entire firmware image has been transferred.*

✔	Lengthy Operation
	Need Current Configuration
✔	May Change Configuration
✔	See Capability Bits

See also:

CAPI_UpdateFirmware()
CAPI_U_ShutdownController()
CAPI_U_FreeCache()

Unified Verify Array NEW! in CAPI 3.4

Syntax:

```
CAPI_RC  CAPI_U_VerifyArray( CAPI_HANDLE  handle,
                             CAPI_U8     *arraySerialNumber );
```

Description:

Verifies the state of a RAID 1, 3, 4, 5, 10, or 50 array.

handle is the handle of the controller that executes the command.
arraySerialNumber is a pointer to the serial number of the target array.

Return Code:

Indicates if the request was sent to the controller and if not, provides an error status.

Callback:

replyCode	**CAPI_REPLY_U_VERIFY_ARRAY_START**
errorCode	Completion status of the command.
identifier	controllerHandle is valid.
param1	
param2	
dataPtr	

Events:

CAPI_EVENT_VERIFY_ARRAY_START
CAPI_EVENT_VERIFY_ARRAY_COMPLETE

Remarks :

The Verify function allows you to verify the data on the selected array (RAID 1, RAID 3, RAID 4, RAID 5, RAID 10, and RAID 50 only):

- RAID 3, RAID 4, RAID 5, and RAID 50: Verifies all parity blocks in the selected array and corrects any bad parity.
- RAID 1 and RAID 10: Compares the primary and secondary drives. If a mismatch occurs, the primary is copied to the secondary.

You may want to verify an array when you suspect there is a problem.

The number of fixes made is included with event CAPI_EVENT_VERIFY_ARRAY_COMPLETE.

✔	Lengthy Operation
✔	Need Current Configuration
	May Change Configuration
✔	See Capability Bits

See also:

CAPI_VerifyArray()

◇ ◇ ◇ 6

REPLY CODE REFERENCE

This chapter provides a reference for the replies that the configuration application may receive through the callback routine. Comments after each event specify which event fields are valid. The fields that are always valid are replyCode, errorCode, and identifier.controllerHandle. See Callback Function on page 6 for description of the call back parameters.

> ***NOTE:*** *This list is complete for products prior to RIO and for the non-unified CAPI commands supported by RIO. But because this table is redundant with data included in Chapter 5,* CAPI Function Reference*, it has not been updated to include the reply codes for the Unified CAPI commands introduced with RIO. See the* ***Callback*** *section of each function description in Chapter 5 for details of the reply code and the other parameters that are returned with the callback.*

The actual #define statements for each reply codes are in capi_event_reply.h (for non-unified commands) and in capu_v1.h (for unified commands). In most cases, the reply code (the name, not the number) is the same for corresponding unified and non-unified commands except that the unified reply codes all use the prefix “CAPI_REPLY_U_”. For Unified CAPI, the reply code value is always 1000 (decimal) greater than the corresponding command code that leads to that reply.

The data type CAPI_REPLY_CODE is used for reply codes and is typedef’d as a CAPI_U32.

Table 6-1. Reply Code Descriptions

Reply Code	Description
CAPI_REPLY_ADD_ARRAY_PARTITION	Array partition has been added (created)
CAPI_REPLY_ADD_DEDICATED_SPARE	A dedicated spare drive was added.
identifier	Describes arrayIndex, channelIndex, and driveIndex.
CAPI_REPLY_ADD_HOST	A host has been added to a host table
CAPI_REPLY_ADD_HOST_NICKNAME NEW! in CAPI 3.3	A host nickname has been added to the host nickname table
CAPI_REPLY_ADD_POOL_SPARE	A pool spare drive was added to the specified controller.
identifier	Describes channelIndex and driveIndex.
CAPI_REPLY_ARRAY_DELETE	An array was deleted.
identifier	Describes arrayIndex.
CAPI_REPLY_ARRAY_NAME_CHANGE	The name of an array changed.
identifier	Describes arrayIndex.
CAPI_REPLY_ARRAY_PARTITION_NAME_CHANGE	Array partition name change is complete
CAPI_REPLY_ARRAY_PARTITION_LUN_CHANGE	Array partition LUN change is complete
CAPI_REPLY_ARRAY_PARTITION_GEOMETRY_CHANGE	Array partition geometry change is complete
CAPI_REPLY_ARRAY_UTIL_PRIORITY_CHANGE	The utility priority changed.
identifier	Describes arrayIndex.
CAPI_REPLY_CACHE_FLUSH	The cache data was flushed. If arrayIndex is equal to CAPI_NULL_ID, then all of the arrays are flushed.
identifier	Describes arrayIndex.
CAPI_REPLY_CACHE_FREE	The dirty cache data was purged.
param1	SCSI LUN of the missing array.

CAPI_REPLY_CACHE_TEST	TBD.
CAPI_REPLY_CAPI_VERSION_MISMATCH	The controller is running an incompatible version of the CAPI interface.
CAPI_REPLY_CHANGE_INFOSHIELD_TYPE	Change InfoShield type is complete
CAPI_REPLY_COMMUNICATION_ERROR	A communication error occurred on the remote link.
CAPI_REPLY_COMMUNICATION_TIMEOUT	A communication time-out occurred on the remote link.
CAPI_REPLY_CONTROLLER_REBOOT_START	A controller reboot started.
CAPI_REPLY_CONTROLLER_UPDATE	A new controller structure was received.
dataPtr	A pointer to the new controller structure.
CAPI_REPLY_CREATE_ARRAY_START	Array creation has started.
identifier	Describes arrayIndex.
CAPI_REPLY_DELETE_ARRAY_PARTITION	Array partition has been deleted
CAPI_REPLY_DEBUG_LOOP_BACK_TEST	A loop back test reply package was received.
dataPtr	A pointer to CAPI_DEBUG_STRUCT structure.
CAPI_REPLY_DOWN_DRIVE	The specified drive was taken offline.
identifier	Describes arrayIndex, channelIndex, and driveIndex.
CAPI_REPLY_DRIVE_BLINK	The specified drive is blinking.
identifier	Describes channelIndex and driveIndex.
CAPI_REPLY_DRIVE_UNBLINK	The specified drive stopped blinking.
identifier	Describes channelIndex and driveIndex.
CAPI_REPLY_ENVIRON_READ	A CAPI_EnvironRead or CAPI_U_EnvironRead command was completed.
param1	The length of the read.
dataPtr	A pointer to CAPI_ENVIRON_PROCESSOR_DATA.
CAPI_REPLY_ENVIRON_WRITE	A CAPI_EnvironWrite or CAPI_U_EnvironWrite command was completed.
CAPI_REPLY_EVENT_LOG_CLEAR	The event log was cleared.
CAPI_REPLY_EXPAND_ARRAY_START	The expand array utility has begun.
identifier	Describes arrayIndex.
CAPI_REPLY_FIND_NEXT_ENVIRON_PROCESSOR	Identifies a possibly found environmental processor.
dataPtr	a pointer to CAPI_ENVIRON_PROCESSOR_INFO.
CAPI_REPLY_FORCE_CRITICAL_ERROR NEW! in CAPI 3.3	A critical error (i.e., controller dump) will be forced.
CAPI_REPLY_FORCE_OFFLINE NEW! in CAPI 3.3	The module has been forced offline.
param1 NEW! in CAPI 3.3	Error code that would have been returned if this was a reply to a CAPI_PutOffline.
CAPI_REPLY_FORCE_ONLINE NEW! in CAPI 3.3	The module has been forced online.
CAPI_REPLY_GET_ADV_ENVIRONMENTALS NEW! in CAPI 3.3	Advanced Controller Environmentals Structure was retrieved.
dataPtr NEW! in CAPI 3.3	A pointer to CAPI_ADVANCED_CONTROLLER_ENVIRONMENTALS.
CAPI_REPLY_GET_ADV_NETWORK_INTF NEW!	Advanced Network Interface Structure was retrieved.
dataPtr NEW!	A pointer to CAPI_ADVANCED_NETWORK_INTERFACE.
CAPI_REPLY_GET_ADVANCED_UNIT_MAPPING	Gets an array of CAPI_UNIT_MAP's
param1	number of CAPI_UNIT_MAP's returned
dataPtr	a pointer to the first CAPI_UNIT_MAP in an array [ ]
CAPI_REPLY_GET_ARRAY_LIST	A list of CAPI_ARRAY has been returned
param1	number of CAPI_ARRAY's returned as an array [ ]
param2	configuration sequence number
dataPtr	A pointer to the first CAPI_ARRAY If param1 is 0, then the pointer is not valid.
CAPI_REPLY_GET_ARRAY_PARTITIONS	Get list of array partitions is complete
CAPI_REPLY_GET_CONFIG_SEQ_NUMBER	Current configuration sequence number for the controller is returned
param1	configuration sequence number
CAPI_REPLY_GET_DEBUG_DATA	Debug data was returned.

NEW! in CAPI 3.3	
dataPtr NEW! in CAPI 3.3	A pointer to CAPI_CHAR (that is, an array of CAPI_CHARs containing the debug data).
CAPI_REPLY_GET_DRIVE_ERROR_STATS NEW! in CAPI 3.3	Drive error statistics structure was returned.
dataPtr NEW! in CAPI 3.3	A pointer to CAPI_DRIVE_ERROR_STATS.
CAPI_REPLY_GET_DRIVE_LIST	A list of CAPI_DRIVE has been returned
param1	number of CAPI_DRIVEs returned as an array [ ]
param2	configuration sequence number
dataPtr	A pointer to the first CAPI_DRIVE If param1 is 0, then the pointer is not valid.
CAPI_REPLY_GET_EVENT	The specified event was returned.
param1	Requested event number (a value of 0 indicates an empty log file).
param2	First event sequence number.
dataPtr	A pointer to an event structure. If param1 is 0, then the event is not valid.
CAPI_REPLY_GET_FIRST_EVENT	The first chronologically available event was returned.
param1	First event sequence number (a value of 0 indicates empty log file).
param2	Last controller power up sequence number. Zero if none are found.
dataPtr	A pointer to an event structure. If param1 is 0, then the event is not valid.
CAPI_REPLY_GET_FREE_ARRAY_PARTITIONS	Get list of free array partitions is complete
CAPI_REPLY_GET_KNOWN_HOSTS	Get known hosts is complete
CAPI_REPLY_GET_HOST_NICKNAME NEW! in CAPI 3.3	Host nickname struct was returned
dataPtr NEW! in CAPI 3.3	A pointer to CAPI_HOST_NICKNAMES.
CAPI_REPLY_GET_HOST_TABLE	A CAPI_HOST_TABLE has been returned
CAPI_REPLY_GET_LAST_EVENT	The last chronologically available event was returned.
param1	Last event sequence number (a value of 0 indicates an empty log file).
param2	First event sequence number.
dataPtr	A pointer to an event structure. If param1 is 0, then the event is not valid.
CAPI_REPLY_INITIALIZE_COMPLETE	CAPI initialization is complete. Note: The identifier controllerHandle is not valid.
CAPI_REPLY_KILL_OTHER	Holding the other controller in reset.
CAPI_REPLY_LOG_EVENT NEW! in CAPI 3.3	The event has been logged.
CAPI_REPLY_LOG_IN	Not currently used. TBD.
CAPI_REPLY_LOG_OUT	Not currently used. TBD.
CAPI_REPLY_MODEL_SPECIFIC	Not currently used. TBD.
CAPI_REPLY_PAUSE_BUS	The specified bus is paused. This means that I/O is not being performed on the drives until an unpause.
identifier	Describes channelIndex.
CAPI_REPLY_PREFERRED_OWNER_SET	An array preferred owner has been changed
CAPI_REPLY_PUT_OFFLINE NEW! in CAPI 3.3	The module has been put offline.
CAPI_REPLY_PUT_ONLINE NEW! in CAPI 3.3	The module has been put online.
CAPI_REPLY_REMOVE_HOST	A host has been removed from a host table
CAPI_REPLY_RESCAN_BUS_START	The bus rescan started.
identifier	Describes channelIndex.
CAPI_REPLY_RESET_ARRAY_PARTITION_STATS	The array partition statistics have been reset
CAPI_REPLY_RESET_ARRAY_STATS	The array statistics were reset.

identifier	Describes arrayIndex.
CAPI_REPLY_RESET_DRIVE_STATS	The drive statistics were reset.
identifier	Describes arrayIndex, channelIndex, and driveIndex.
CAPI_REPLY_RESET_LAN	Reset LAN subsystem command reply
CAPI_REPLY_RESTORE_CONTROLLER_DEFAULTS	The controller defaults were restored. See controller's documentation for default settings.
CAPI_REPLY_SCSI_MAINT_DATA	A SCSI maintenance command returned data.
identifier	Describes channelIndex and driveIndex.
dataPtr	A pointer to a CAPI_MAINT_DATA_STRUCT structure.
CAPI_REPLY_SCSI_MAINT_START	A SCSI maintenance command has started.
identifier	Describes channelIndex and driveIndex.
param1	comandId
CAPI_REPLY_SET_ADVANCED_NETWORK_INTF NEW!	Advanced Network Interface Structure was set
CAPI_REPLY_SET_ADVANCED_UNIT_MAPPING	Controller has received an array of CAPI_UNIT_MAP's
CAPI_REPLY_RESET_DRIVE_ERROR_STATS NEW! in CAPI 3.3	New cache parameters have been set for the specified array partition.
CAPI_REPLY_SET_ARRAY_PARTITION_CACHE_PARAMS NEW! in CAPI 3.3	New cache parameters have been set for the specified array partition.
CAPI_REPLY_SET_BATTERY_MONITOR	The Battery Life Monitor has been set.
CAPI_REPLY_SET_CACHE_PARAMS	The new cache parameters for the specified array have been set. If arrayIndex is CAPI_NULL_ID, then all arrays have been set.
identifier	Describes arrayIndex.
CAPI_REPLY_SET_CHANNEL_PARAMS	New channel parameters have been set.
CAPI_REPLY_SET_CONTROLLER_PARAMS	The new controller parameters have been set.
CAPI_REPLY_SET_CONTROLLER_TIMEDATE	The time and date for the specified controller were set.
CAPI_REPLY_SET_PREFERRED_OWNER	Sets the preferred owner of an array.
CAPI_REPLY_SHUTDOWN_CONTROLLER	The controller is in a shutdown state.
CAPI_REPLY_SILENCE_ALARM	Controller alarm has been silenced.
CAPI_REPLY_SPARE_DELETE	A spare drive was deleted.
identifier	Describes arrayIndex (CAPI_NULL_ID if pool spare), channelIndex, and driveIndex.
CAPI_REPLY_TEST_DRIVE	The drive test was completed.
identifier	Describes channelIndex and driveIndex.
CAPI_REPLY_TEST_SPARES	The test spares request was processed.
CAPI_REPLY_TRUST_ARRAY	Controller has finished clearing dead drives.
CAPI_REPLY_UNIT_MAPPING	The assigned LUN for the specified array changed.
identifier	Describes arrayIndex.
CAPI_REPLY_UNKILL_OTHER	Released the other controller from reset.
CAPI_REPLY_UNPAUSE_BUS	The bus was unpaused.
identifier	Describes channelIndex.
CAPI_REPLY_UPDATE_FIRMWARE	The controller has received a firmware update.
CAPI_REPLY_USE_KEY NEW!	Digital key has been used.
CAPI_REPLY_UTILITY_ABORT	The utility for the specified array was aborted.
identifier	Describes arrayIndex.
param1	Utility Running
CAPI_REPLY_UTILITY_PERCENT	The percent complete for the specified array was returned.
identifier	Describes arrayIndex.
param1	Percent complete.
param2	Utility Running (CAPI_UTILITY_RUNNING).
CAPI_REPLY_VERIFY_ARRAY_START	The verify operation for the specified array started.
identifier	Describes arrayIndex.
CAPI_REPLY_VERSION_MISMATCH	Major CAPI version on the controller does not match major CAPI version of the host application.

◊ ◊ ◊ 7

Event Code Reference

This chapter provides a reference for the event codes received by the configuration application through the call back routine. Comments after each event specify which fields in the CAPI_EVENT structure are valid. The fields that are always valid are sequenceNumber, timeStamp, eventCode, errorCode, criticality, and id.controllerHandle.

The serialNumbers.arraySerialNumber field is valid for any event having an arrayIndex; serialNumbers.driveSerialNumber is valid for any event having a driveIndex (not CAPI_NULL_ID). See the CAPI_EVENT structure for a full description of the fields.

The data type CAPI_EVENT_CODE is used for event codes and is typedef'd as a CAPI_U32.

> ***NOTE:*** *This list is complete for products prior to RIO. For additional event codes and their descriptions, please see file capi_event_reply.h in the SDK. File capi_event_reply.h also contains additional descriptive comments for some of these event codes.*

Table 7-1. Event Code Descriptions

Event Code	Description
CAPI_EVENT_3RD_PARTY_DISK_BUS_RESET	A disk channel was reset by a third-party device.
id	Describes the channelIndex.
CAPI_EVENT_AA_ENABLED	Active-active configuration is now enabled.
CAPI_EVENT_AD_FAILURE	An analog to digital converter failure occurred.
param1	See CAPI_AD_ALARM_SIGNAL in ◊ ◊ ◊ 3 Typedefs and Defines starting on page 18.
CAPI_EVENT_AD_OK	The analog to digital converter is now functional.
param1	See CAPI_AD_ALARM_SIGNAL in ◊ ◊ ◊ 3 Typedefs and Defines starting on page 18.
CAPI_EVENT_AD_WARNING	An analog to digital converter warning.
param1	See CAPI_AD_ALARM_SIGNAL in ◊ ◊ ◊ 3 Typedefs and Defines starting on page 18.
CAPI_EVENT_ADD_ARRAY_PARTITION_COMPLETE	Array partition has been added.
CAPI_EVENT_ADD_DEDICATED_SPARE	A dedicated spare drive was added.
id	Describes arrayIndex, channelIndex, and driveIndex.
CAPI_EVENT_ARRAY_CRITICAL	One drive in the specified array failed and the array is running in degraded mode.
id	Describes the arrayIndex.
param1	Number of suitable spare drives.
CAPI_EVENT_ARRAY_DELETE	The array was deleted.
id	Describes the arrayIndex.
CAPI_EVENT_ ARRAY_HOST_ID_CHANGED	
CAPI_EVENT_ ARRAY_LUN_CONFLICT	
CAPI_EVENT_ARRAY_NAME_CHANGE	The array name has been changed.
id	Describes the arrayIndex.
CAPI_EVENT_ARRAY_OFFLINE	The drives in an array without redundancy failed and the array is

	now off-line.
id	Describes arrayIndex.
CAPI_EVENT_ARRAY_PARTITION_GEOMETRY_CHANGE	Array partition geometry change is complete.
CAPI_EVENT_ARRAY_PARTITION_LUN_CHANGE	Array partition LUN change is complete.
CAPI_EVENT_ARRAY_PARTITION_NAME_CHANGE	Array partition name change is complete.
CAPI_EVENT_BACKEND_CHAN_LINK_DOWN	The link for this back-end port is down The id.channelIndex field is the port number
CAPI_EVENT_BACKEND_CHAN_LINK_UP	The link for this back-end port is up The id.channelIndex field is the port number
CAPI_EVENT_BATTERY_CHARGE_COMPLETE	The controller's battery used for cache backup is now charged.
CAPI_EVENT_BATTERY_FAILURE	A battery failure occurred on the controller.
param1	Battery failure code.
param4	State of the battery.
CAPI_EVENT_BATTERY_HW_FAILURE_INFO	Product-specific battery failure. Parameters are product-dependent.
CAPI_EVENT_BATTERY_TEMPERATURE_WARNING	Battery temperature is in the warning range.
CAPI_EVENT_BLOCK_REASSIGNED	A member of an array had an uncorrectable error and the controller reassigned the block.
id	Describes arrayIndex and driveLocation.
param1	Block number.
CAPI_EVENT_BOOT_SDRAM_UNCORR_ECC_ERR	An uncorrectable ECC error occurred on the SDRAM memory on bootup. The controller scrubbed the memory and continued.
CAPI_EVENT_CACHE_INIT	The cache was initialized as a result of power up
param1	0 initialized from clean shutdown 1 initialized with dirty (unwritten)
param2	0 memory region A 1 memory region B this parameter is valid only for products supporting failover
CAPI_EVENT_CONFIGURATION_DEFAULTS	The controller is using default configuration settings. This event will occur on the first power up, and may sometimes occur after a firmware update. If you have just performed a firmware update and your system requires special configuration settings, you must make those configuration changes before your system will operate as before.
CAPI_EVENT_CONFIGURATION_HAS_CHANGED	The array configuration changed on the controller. Applications should update their information.
CAPI_EVENT_CONTROLLER_REBOOT_COMPLETE	The controller rebooted. (Not implemented.)
CAPI_EVENT_CORRUPT_EVENT_ENTRY	This event entry is corrupt. This can happen when the power is lost while the controller is in the process of writing an event into the flash memory.
CAPI_EVENT_CREATE_ARRAY_COMPLETE	The array creation is complete.
id	Describes arrayIndex.
CAPI_EVENT_CREATE_ARRAY_START	An array creation started.
id	Describes arrayIndex.
CAPI_EVENT_CRITICAL_ERROR_ENCOUNTERED	A critical error has been encountered by the controller software. The severity of this error requires that the controller software be restarted -- this is done automatically, except in an Active-Active configuration, where the surviving controller will kill the controller that hit the critical error.
CAPI_EVENT_DELETE_ARRAY_PARTITION_COMPLETE	Array partition has been deleted.
CAPI_EVENT_DIAGNOSTIC_FAILURE	A controller diagnostic failed or returned a warning message.
param1	Diagnostic error code.
CAPI_EVENT_DISK_CHANNEL_ERROR	The controller's software observed an error while talking to a SCSI device on a disk channel. The error was detected by the controller, not the disk.
id	Describes the arrayIndex, channelIndex, and driveIndex.
deviceId	SCSI ID.
param1	Product dependent.

param2	CDB length
cdb	SCSI CDB related to this event
CAPI_EVENT_DISK_CHANNEL_FAILURE	A serious error was detected on one of the disk channels. This may indicate a hardware failure; however, the controller will attempt a recovery.
id	Describes the channelIndex.
CAPI_EVENT_DISK_CHANNEL_ID_CONFLICT	
CAPI_EVENT_DISK_DETECTED_ERROR	A disk drive or other SCSI device on the disk channel bus (such as a SAF-TE SEP device) reported a check condition and the following SCSI sense data was returned.
id	Describes arrayIndex, channelIndex, and driveIndex.
deviceId	SCSI ID.
param1	SCSI sense key.
param2	SCSI sense code.
param3	SCSI sense code qualifier.
param4	sense data information field, usually the LBA associated with the sense key. NEW! (Note, this param4 was CDB length in CAPI 3.1)
cdb	SCSI CDB related to this event.
CAPI_EVENT_DISKSET_OWNER_CHANGE	This is an information only event that is logged when the controller detects that new disks have been added that are from a different controller and have an existing array on them. The controller takes ownership of the disksets.
id	Describes arrayIndex.
CAPI_EVENT_DOMAIN_VALIDATION_FALLBACK	This event only applies to controllers with parallel SCSI disk channels. It indicates that Ultra 160 domain validation failed on one of the controllers disk channels. Parameters indicate the minimum and maximum negotiated rates on the disk channel, and which device ids were affected.
param1 (LSW)	Minimum negotiated rate, in MB/s
param1 (MSW)	width of minimum negotiated rate (8 or 16 bits)
param2 (LSW)	Maximum negotiated rate, in MB/s
param2 (MSW)	width of maximum negotiated rate (8 or 16 bits)
param3	16 bit bitmap of device ids that failed domain validation
CAPI_EVENT_DRIVE_DOWN	An array member failed and the array either changed to a critical or off-line state.
id	Describes arrayIndex and driveLocation.
param1	ArrayDriveIndex (index into array).
deviceId	SCSI ID.
CAPI_EVENT_EMP_EVENT	During Active/Active operations, an event (a potential error) has occurred while coordinating communications with the Enclosure Management Processor (used for SAF-TE or SES).
param1	Contains one of the event codes: EMP_SLAVE_REQ_FAILED EMP_BAD_EMP_ID
param2	If param1 == EMP_BAD_EMP_ID, this contains the EMP id.
CAPI_EVENT_EMP_FAILURE	A communications failure has occurred between the controller and the Enclosure Management Processor (used for SAF-TE or SES).
param1	If the error code is set to CAPI_ERROR_CAN'T_TALK_TO_EMP, then param1 contains the HIM Task Status codes: EMP_EVENT_UNDEFINED (0x00) EMP_RB_HIOB_NO_RESPONSE (0x01) EMP_RB_HIOB_UNKNOWN_ERROR (0x02) EMP_WB_HIOB_NO_RESPONSE (0x03)
param2	If param1 == EMP_RB_HIOB_UNKNOWN_ERROR, then this contains the actual HIOB task status value.
CAPI_EVENT_ENCLOSURE_FAILURE	The enclosure reported a general failure.
CAPI_EVENT_ENGLISH_STRING	Not implemented.
CAPI_EVENT_ENVIRON_COMMAND	This command is used internally by the controller to send commands to the environmental processor.
CAPI_EVENT_ENVIRON_FAILURE	Could not communicate with an environmental processor (EMP).
CAPI_EVENT_EXPAND_ARRAY_COMPLETE	The Expand Array utility was completed.

CAPI_EVENT_EXPAND_ARRAY_START	The Expand Array utility has begun.
id	Describes the arrayIndex.
CAPI_EVENT_FAILBACK	The controller has started failing over, or completed failing over.
param1	0 = initiated, 1 = completed
param2	failover set: 0=B, 1=A
CAPI_EVENT_FAILOVER	Description: the controller has started failing over, or completed failing over
param1	0 = initiated, 1 = completed
param2	failover set: 0=B, 1=A
CAPI_EVENT_GLOBAL_DISK_SETTING_CHANGE	The controller modified some mode parameters on one or more drives
param1	CAPI_DISK_SETTING (enable or disable only)
param2	1 write back cache 2 SMART support
CAPI_EVENT_HOST_CHANNEL_ERROR	The controller either generated or detected an error on one of its host channels.
param1	1 for controller detected errors, 2 for generated errors
param2	controller internal error code (when param1 == 1) or SCSI status byte (when param1 == 2)
param3	SCSI sense key (when param1 == 2 and param2 == 2)
param4	SCSI ASC/ASCQ (when param1 == 2 and param2 == 2)
CAPI_EVENT_HOST_CHAN_LINK_DOWN	The link for this host port is up The id.channelIndex field is the port number
CAPI_EVENT_HOST_CHAN_LINK_DOWN	The link for this host port is up The id.channelIndex field is the port number
CAPI_EVENT_HOST_TERMINATION_WARNING	The controller's termination may be bad.
CAPI_EVENT_KILL_OTHER_CONTROLLER	
	Reason for killing the other controller. See CAPI_FR_FAILOVER_REASON for a list of valid reason codes.
CAPI_EVENT_MODEL_SPECIFIC	This is a model-specific event.
CAPI_EVENT_NO_EVENT	Obsolete.
CAPI_EVENT_NON_NATIVE_WWN_BEING_USED	This replacement controller has assumed the World Wide Name (node and port) of the original controller. This is done to make the replacement of a controller in an Active-Active configuration transparent to the host. However, if both controllers lose power or are otherwise rebooted, then the original controller's WWN will be lost, and the current controller will generate a new WWN based on its own unique serial number. This means that a dual controller reboot will cause the controller's WWN to change from the host's perspective.
param1	First 4 bytes of current node WWN
param2	Last 4 bytes of current node WWN
param3	First 4 bytes of native node WWN
param4	Last 4 bytes of native node WWN
CAPI_EVENT_OEM_ENCLOSURE_STATUS	The OEM's enclosure has detected a change in the status of one of the items that it monitors.
param1	The device which has changed state: 1 = Enclosure Fan 2 Status 2 = Enclosure Fan 1 Status 3 = Fibre Channel Loop 2 GBIC Receiver Loss Of Signal 4 = Fibre Channel Loop 1 GBIC Receiver Loss Of Signal 5 = Enclosure power supply 1 status NEW! 6 = Enclosure power supply 2 status NEW! 7 = RS-232 configuration port switch NEW!
param2	Current state of the device: 0 = Device is operating correctly (for fans) or signal detected (for receivers) or external terminal mode (for RS-232 configuration port switch.) 1 = Device failed (for fans), or no receive signal detected (for receivers) or Internal LCD Mode (for configuration port switch)

	NEW!
CAPI_EVENT_ORPHAN_DATA	Dirty cache data exists on the controller without a corresponding array. Use CAPI_FreeCache to purge the data.
param1	LUN for which the data is associated with.
param2	Percentage of data occupied in the controller's memory.
CAPI_EVENT_OTHER_WRITE_BACK_DATA_LOST	
param1	new state: 10 = other shutting down to update firmware 11 = other shutting down 12 = other rebooting
CAPI_EVENT_OTHER_STATE_CHANGE	The other controller was in the process of mirroring write-back data to this controller after a failback, when the other controller was killed. This means that some writes to the storage LUNs owned by the other controller may have been lost.
CAPI_EVENT_POOL_SPARE_ADDED	A pool spare drive (available to all arrays) has been added.
id	Describes arrayIndex.
CAPI_EVENT_POWER_UP	The controller powered up.
CAPI_EVENT_REBOOT_TO_AVOID_OTHER_LOST_WRITE_DATA	The other controller was in the process of mirroring write-back data to this controller after a failback when the other controller was killed. We rebooted to avoid losing the data in the other controller's cache. If the other controller does not reboot successfully the data was lost.
CAPI_EVENT_RECONSTRUCT_ARRAY_COMPLETE	A reconstruct was completed on an array that is now fault-tolerant.
id	Describes arrayIndex.
CAPI_EVENT_RECONSTRUCT_ARRAY_START	A reconstruct operation was started.
id	Describes arrayIndex.
deviceId	SCSI ID of the drive being reconstructed.
CAPI_EVENT_RELEASE_OTHER_CONTROLLER	Release the other controller from reset.
CAPI_EVENT_RESCAN_BUS_COMPLETE	A bus scan was completed.
id	Describes channelIndex.
CAPI_EVENT_RESET_ARRAY_PARTITION_STATS	The statistics for the specified array have been reset.
CAPI_EVENT_RESET_ARRAY_STATS	The array statistics were cleared.
id	Describes arrayIndex.
CAPI_EVENT_RESET_DRIVE_STATS	The drive statistics were cleared.
id	Describes arrayIndex and driveLocation.
CAPI_EVENT_RESTORE_CONTROLLER_DEFAULT	The factory default settings were restored. Some controllers require a reboot for the default settings to be restored. See the controller's documentation.
id	Describes controllerHandle.
CAPI_EVENT_SCSI_MAINT_DONE	A SCSI maintenance command was completed. See errorCode for completion status.
id	Describes the channelIndex.
deviceId	SCSI ID.
param1	maintCommand.
param2	Sense key if failure.
param3	Sense ASC if failure.
CAPI_EVENT_SET_CHANNEL_PARAMS	Channel parameters have been changed.
CAPI_EVENT_SET_PREFERRED_OWNER	The array has been given to the other controller.
CAPI_EVENT_SDRAM_CORR_ECC_ERR	A correctable ECC error occurred on the SDRAM.
param1	address of memory with ECC error
CAPI_EVENT_SDRAM_UNCORR_ECC_ERR	An uncorrectable ECC error occurred on the SDRAM.
param1	address of memory with ECC error
CAPI_EVENT_SET_ARRAY_PARTITION_CACHE_PARAMS	The new cache parameters for an array partition were set.
CAPI_EVENT_SET_CACHE_PARAMS	The new cache parameters for an array were set.
id	Describes arrayIndex.

CAPI_EVENT_SET_CONTROLLER_PARAMS	The new controller parameters were set.
CAPI_EVENT_SET_CONTROLLER_TIMEDATE	The time and date were set on the controller.
CAPI_EVENT_SHUTDOWN_CONTROLLER	The controller is in a shutdown state.
CAPI_EVENT_SMART_EVENT A SMART	event occurred on a SCSI device.
id	Describes the arrayIndex, channelIndex, and driveIndex.
deviceId	SCSI ID.
param1	SCSI sense key.
param2	SCSI sense code.
param3	SCSI sense qualifier.
CAPI_EVENT_SPARE_DELETE	A spare drive was deleted.
id	Describes driveLocation.
deviceId	SCSI ID.
CAPI_EVENT_SPARE_DRIVE_FAILURE	A spare drive failed.
id	Describes the arrayIndex, channelIndex, and driveIndex.
deviceId	SCSI ID.
param1	SCSI sense key.
param2	SCSI sense code.
param3	SCSI sense qualifier.
CAPI_EVENT_SPARE_KICKED_IN	A spare drive automatically started to reconstruct due to a drive failure.
id	Describes arrayIndex and driveLocation.
param1	ArrayDriveIndex (index into array).
deviceId	SCSI ID.
CAPI_EVENT_SPARE_UNUSABLE	The controller could not use an assigned spare drive for an array because of a conflict in the spare's metadata. (The spare's metadata may indicate it was once part of the array that needs to be reconstructed, or it may have once been a member of another, no longer existent array. In either case, the metadata on the spare drive must be cleared before it can be used as a spare.)
CAPI_EVENT_TEST_DRIVE	A drive test was completed.
id	Describes driveLocation.
errorCode	Results of a successful drive test.
CAPI_EVENT_TRANSPORT_MODE_CHANGE	A disk channel changed from single-ended to LVD mode or Vice versa.
id	Describes the arrayIndex, channelIndex, driveIndex.
CAPI_EVENT_TRIGGER_EMP_UPDATE	Controller internal use only. CAPI applications should ignore this event.
CAPI_EVENT_TRUST_ARRAY	The controller cleared dead drives on an array
CAPI_EVENT_UNIT_MAPPING	The assigned LUN number for this array changed.
id	Describes arrayIndex.
param1	New unit number (the array is seen as a LUN).
deviceId	SCSI ID of mapped single drive (if arrayIndex is CAPI_NULL_ID).
CAPI_EVENT_UPDATE_FIRMWARE_COMPLETE	Firmware update operation is complete. A controller reboot is necessary for the new firmware to take effect. See errorCode for completion status of the operation.
CAPI_EVENT_UTILITY_ABORT	An array utility was aborted.
id	Describes arrayIndex.
param1	Type of utility aborted (CAPI_UTILITY_RUNNING).
CAPI_EVENT_VERIFY_ARRAY_COMPLETE	A verify operation was completed. See the errorCode in the CAPI_EVENT structure for completion status.
id	Describes arrayIndex.
param1	Number of fixes made.
CAPI_EVENT_VERIFY_ARRAY_START	A verify operation started.
id	Describes arrayIndex.
CAPI_EVENT_WWN_HAS_CHANGED	This controller was replaced at some time in the past and assumed the World Wide Names (node and port) of the original controller. However, a dual controller reboot has been done, and this controller is now using WWNs based on its own serial number.

	This transition takes place on a dual controller reboot because it is not advisable to assume another controller's WWNs indefinitely (in case that controller is repaired and plugged back into the same fabric), and because host operations have already been disrupted by the dual reboot. If you see this event, then you need to verify the WWN information for this controller on all hosts that access it.
param1	First 4 bytes of current node WWN
param2	Last 4 bytes of current node WWN
param3	First 4 bytes of previous node WWN
param4	Last 4 bytes of previous node WWN

◇ ◇ ◇ 8

RETURN CODE REFERENCE

This chapter provides a reference for the return codes that are returned by CAPI functions.

The data type CAPI_RETURN_CODE is used for return codes and is typedef'd as a CAPI_U32.

> ***NOTE:*** *In the CAPI Function Reference starting on page 115, the return codes are shown as CAPI_RC – this is just an abbreviation. The actual define for return code is listed in capi3.h as CAPI_RETURN_CODE.*

Table 8-1. Return Code Descriptions

Return Code	Description
CAPI_STATUS_COMMUNICATION_ERROR	A communication error occurred.
CAPI_STATUS_FIRMWARE_DOWNLOAD_ERROR	An error occurred during the controller's firmware download.
CAPI_STATUS_GOOD	The status of the command is good.
CAPI_STATUS_INVALID_PARAM	An invalid param was used.
CAPI_STATUS_LINK_BUSY	The Remote Link is busy. Retry in a few seconds.
CAPI_STATUS_NOT_IMPLEMENTED	The command is not implemented.
CAPI_STATUS_NOT_SUPPORTED	The command that was issued is not supported.
CAPI_STATUS_NULL_POINTER	A bad pointer was used in a call to CAPI

◇ ◇ ◇ 9

ERROR CODE REFERENCE

This chapter provides a reference for the errors that a CAPI application can receive through the callback routine.

The data type CAPI_ERROR_CODE is used for error codes and is typedef'd as a CAPI_U32.

> ***NOTE:*** *This list is complete for products prior to RIO. For additional error codes and their descriptions, please see file capi3.h in the SDK.*

Table 9-1. Return Code Descriptions

Error Code	Description
CAPI_NO_ERROR	No error was received.
CAPI_ERROR_AA_INCOMPAT_FIRMWARE_IMAGE	Firmware is incompatible with other controller in Active-Active cfg.
CAPI_ERROR_ARRAY_DOWN	Not allowed to modify the array when it is down.
CAPI_ERROR_ARRAY_INIT	This error occurs during a create array.
CAPI_ERROR_ARRAY_PARTITION_OVERLAP	The array partition that was added overlaps an existing partition.
CAPI_ERROR_ARRAY_PARTITION_TOO_SMALL	The array partition that was added is too small.
CAPI_ERROR_ARRAY_TOO_LARGE	Controller may not allow creation of an array that is bigger than 2 TB.
CAPI_ERROR_BACKOFF_PERCENT_TOO_LARGE	The backoff percent value given is 100.0% (i.e. 1000) or greater. This is illegal.
CAPI_ERROR_BAD_CONTROLLER_MODE	An invalid combination of dualPort and standAlone bits or an invalid controllerMode was selected in CAPI_CONTROLLER_PARAMS. (e.g. Some products only support standalone/single port, and some support dual port, but only in standalone mode.) NEW! Note: this event was called CAPI_ERROR_BAD_DUAL_SA_OPTION in CAPI3.1
CAPI_ERROR_BAD_IDENTIFIER	An invalid device was specified.
CAPI_ERROR_BAD_PASSWORD	An incorrect password was received.
CAPI_ERROR_BAD_PRIORITY	A bad utility priority was specified.
CAPI_ERROR_BUS_SPEED_OUT_OF_RANGE	The bus speed specified is beyond the maximum capable speed for this channel.
CAPI_ERROR_CANNOT_CHANGE_FAILOVER_ARRAY_LUN	The LUN value of an array belonging to a failed controller cannot be changed from the non-native controller. Repair the failed controller and change the LUN value from the native controller. Changing an array LUN value when failed over can cause LUN conflicts when the array fails back.
CAPI_ERROR_CANNOT_CHANGE_FAILOVER_CHAN_PARAMS	Host channel parameters for a failed over host channel cannot be changed from the non-native controller. Repair the failed controller and change the channel parameters from the native controller. Changing host channel information while failed over can cause problems when control of the channel fails back.
CAPI_ERROR_CANT_ADD_ARRAY_MAXED_OUT	Cannot add an array because there are already the maximum number of arrays.
CAPI_ERROR_CANT_ADD_ARRAY_MAXED_OWNER	This error is returned when a controller is already the preferred owner of its maximum number of arrays and an add array

	command is attempted.
CAPI_ERROR_CANT_ADD_SPARE_DURING_INIT	Cannot add spare drive while the array initialization is running.
CAPI_ERROR_CANT_ADD_SPARE_MAXED_OUT	Cannot add anymore spare drives.
CAPI_ERROR_CANT_RESCAN_DURING_ZERO_INIT	Cannot rescan while an array utility is running.
CAPI_ERROR_CANT_START_UNIT	SCSI failure.
CAPI_ERROR_CANT_TALK_TO_SEP	Cannot communicate with the SAF-TE device.
CAPI_ERROR_CANT_VERIFY_WHEN_CRITICAL	Cannot run verify utility because the array is not fault-tolerant.
CAPI_ERROR_CDB_DATA_TOO_LARGE	The amount of data on a maintenance use-CDB command is too large.
CAPI_ERROR_CHANNEL_NUM_OUT_OF_RANGE	Request for channel number that does not exist.
CAPI_ERROR_CHECK_CONDITION	A SCSI check condition occurred while communicating with the device.
CAPI_ERROR_COMMAND_FAILED	The command failed for non-specified reasons.
CAPI_ERROR_CONTROLLER_SHUTDOWN	The command cannot be completed because the controller is in a special shutdown state.
CAPI_ERROR_DMEP_BUFFER_SIZE_TOO_LARGE	The SCSI DMEP (Device Memory Export Protocol) memory buffer size specified in the controller parameters is too large. The "maxDmepMemoryBufferSize" field in the controller structure indicates the maximum buffer size.
CAPI_ERROR_DRIVE_NOT_ONLINE	The specified drive is not online.
CAPI_ERROR_DRIVE_TOO_SMALL	Proposed drive is too small to use.
CAPI_ERROR_FAILURE_DUE_TO_CONFIG_CHANGE	The command failed because the requesting application has outdated configuration information.
CAPI_ERROR_GET_PARAMS	SCSI failure. Could not get drive parameters.
CAPI_ERROR_INQUIRY	SCSI inquiry failure.
CAPI_ERROR_INVALID_ARRAY_FORMAT_TYPE	The create array command had an invalid formatType field.
CAPI_ERROR_INVALID_CHANNEL_ID	The SCSI or Fibre Channel ID specified is invalid.
CAPI_ERROR_INVALID_CHANNEL_TYPE	The channel type specified must be either a host or drive channel. The value was neither.
CAPI_ERROR_INVALID_CMD_IN_THIS_MODE	The controller is running in a mode (see controllerMode) that does not allow the requested command. The command may work if controllerMode is set differently.)
CAPI_ERROR_INVALID_CRITICAL_ERROR_PARAMETER	An invalid "magic number" value or error type parameter was supplied with the "force critical error command.
CAPI_ERROR_INVALID_DATA_CHUNK_SIZE	Invalid/bad data chunk sizes was specified.
CAPI_ERROR_INVALID_ENCLOSURE_FEATURE_FLAG	A bad enclosure feature flag was submitted to the controller. This enclosure may not support the specified feature flag. occurred
CAPI_ERROR_INVALID_FC_LINK_SPEED	The Fibre Channel link speed must be set to "1GB" or "2GB" or "AUTO". It was set to none of these.
CAPI_ERROR_INVALID_FC_TOPOLOGY	The Fibre Channel topology must be set to "loop" or "point-to-point". It was set to neither.
CAPI_ERROR_INVALID_FIRMWARE_CRC	Firmware is invalid because of CRC
CAPI_ERROR_INVALID_FIRMWARE_HEADER	Firmware is invalid because of header information
CAPI_ERROR_INVALID_FIRMWARE_MACHINE_TYPE	Firmware is invalid because of machine type
CAPI_ERROR_INVALID_FIRMWARE_SIZE	Firmware is invalid because of size of image
CAPI_ERROR_INVALID_KEY	An invalid digital key was used
CAPI_ERROR_INVALID_KEY_MAXIMUM_RETRIES_EXCEEDED	An invalid digital key was used more than the maximum number of times allowed. You must reboot the controller before you will be allowed to turn on features using a digital key
CAPI_ERROR_INVALID_NUM_OF_LOW_LEVEL_DRIVES	An invalid number of low level drives has been specified when creating a RAID50 array. The number of drives is too large, too small, or not evenly divisible into the number of drives specified for the array.
CAPI_ERROR_INVALID_NUMBER_OF_DRIVES	Invalid number of drives was specified.
CAPI_ERROR_INVALID_NUMBER_OF_SPARES	Invalid number of spare drives was specified.
CAPI_ERROR_INVALID_RAID_TYPE	Invalid RAID type given.
CAPI_ERROR_INVALID_TIME_DATE	The time and date parameter submitted to the controller was bad. The time/date parameter is the number of seconds since January 1, 1970. A date after December 31st, 2037 is not currently

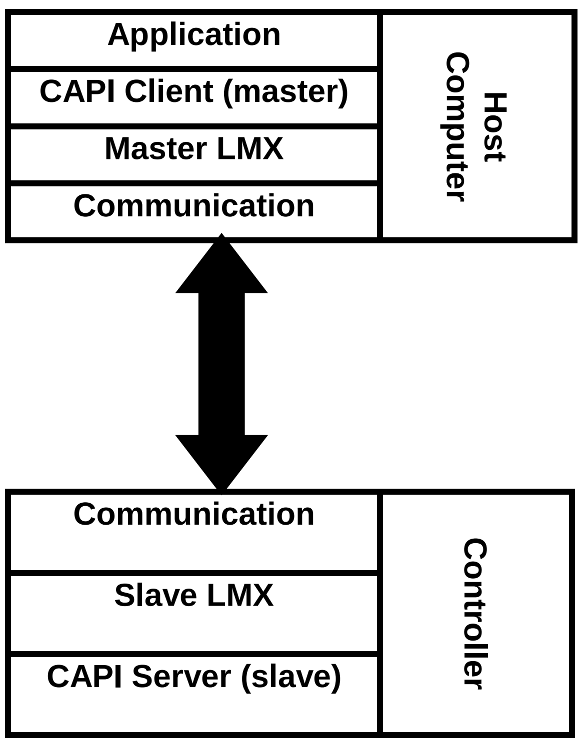
	accepted. NEW!
CAPI_ERROR_INVALID_UNIT_NUM	Invalid SCSI LUN was specified.
CAPI_ERROR_LUN_AUTO_SETTING_NOT_SUPPORTED	LUN auto-setting cannot be enabled for this product.
CAPI_ERROR_MAX_ONE_OCE	Can only have 1 OCE per controller/controller pair.
CAPI_ERROR_MODE_SENSE	SCSI failure.
CAPI_ERROR_NEW_ARRAY_CONFIG	Create array failure.
CAPI_ERROR_NO_ORPHAN_DATA	Could not find orphan data for serial number.
CAPI_ERROR_NO_RESOURCES	No resources are available to complete the request.
CAPI_ERROR_NO_SUCH_DRIVE	Invalid drive was specified.
CAPI_ERROR_NO_SUCH_EVENT	No such event exists on the controller.
CAPI_ERROR_NO_SUCH_ENVIRON_PROCESSOR	The specified SAF-TE or SES processor (EMP) does not exist.
CAPI_ERROR_NO_UTILITY_RUNNING	There is no utility running to abort.
CAPI_ERROR_NO_UTILITY_TO_ABORT	There is no utility to abort.
CAPI_ERROR_NOT_A_VALID_DRIVE_TO_RECONSTRUCT	Invalid drive was specified.
CAPI_ERROR_NOT_SUPPORTED	The command is not supported.
CAPI_ERROR_OCE_INTERNAL_ERROR	This is an OCE (Online Capacity Expansion) software error.
CAPI_ERROR_ORPHAN_DATA_PRESENT	Cannot complete the operation due to dirty cache that is present on the non-existent array. Use CAPI_FreeCache to purge the data. param1 on the reply contains the LUN number to purge.
CAPI_ERROR_OTHER_NOT_UP	The command cannot complete because the other controller in a dual-controller system is not running.
CAPI_ERROR_PARITY_NOT_VALID	This error is returned in the array offline event. It indicates the array is offline because parity is not known to be good. If the array is missing a member drive, then data has been lost. This situation can arise if a controller with a critical array is not shut down cleanly, and is replaced with a different controller. The parity information in the first controller's NVRAM is not available, and the disk parity may be inconsistent. If the array is not missing any drives, a verify will restore parity and make the array usable again
CAPI_ERROR_READ_CAPACITY	SCSI failure.
CAPI_ERROR_RECONSTRUCT_NOT_NEEDED	A reconstruction is not needed on the array.
CAPI_ERROR_SPARE_UNUSABLE	Refer to the comment for CAPI_EVENT_SPARE_UNUSABLE
CAPI_ERROR_SPARE_USED	Cannot add spare because the drive is already being used.
CAPI_ERROR_START_UNIT	SCSI failure.
CAPI_ERROR_TEST_UNIT_READY	SCSI failure.
CAPI_ERROR_TOO_MANY_ARRAY_PARTITIONS	The array partitions that was added overlaps an existing array partition.
CAPI_ERROR_UNIT_NUM_IN_USE	Invalid SCSI LUN number.
CAPI_ERROR_UTILITY_ABORTED_BY_USER	The user aborted the utility.
CAPI_ERROR_UTILITY_ALREADY_RUNNING	A utility is already running.
CAPI_ERROR_VERIFY_FAILED	Obsolete.
CAPI_ERROR_WRITE_RESERVED_SECTOR	Could not write data to array members.
CAPI_ERROR_WRONG_TOPOLOGY_FOR_PRIVATE_LOOP	The Fibre Channel topology must be set to "loop" in order to set "force private loop".
CAPI_ERROR_WWN_NOT_FOUND	The controller can't find the requested world wide name.
CAPI_ERROR_WWN_TABLE_FULL	The controller can't perform the requested operation because its world wide name table is already full.
CAPI_ERROR_ZERO_DRIVES	Could not write data to array members.

10

LINK MANAGER EXCHANGE (LMX)

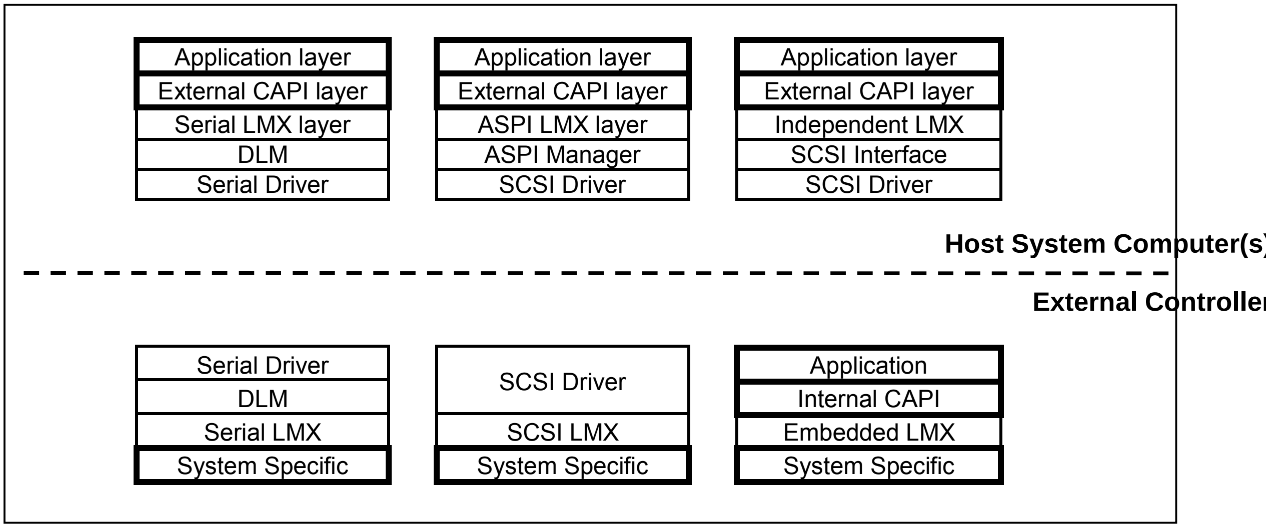
The Link Manager Exchange (LMX) is the layer between the CAPI Client and the data exchange interface and resides on the computer that is running the CAPI Client. (An LMX is also used within the controller.) The basic model of the CAPI stack is shown in Figure 10-1. This model allows the interface from application to CAPI Client to remain constant over a wide variety of environments.

Figure 10-1. General CAPI Architecture



The layers with bold borders remain the same, regardless of the data exchange layer. The LMX is used to match CAPI messages to the appropriate data exchange layer. The Data Link Manager (DLM) provides a reliable protocol over a serial link. The diagram also illustrates how an application may be used either from a host or from within the external controller.

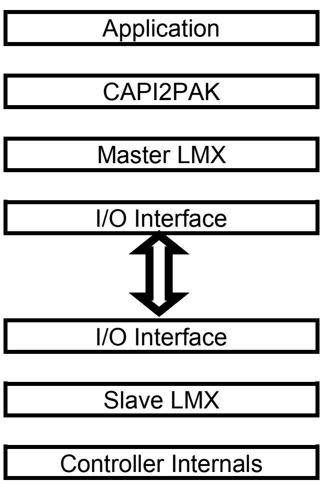
Figure 10-2. Example LMX Protocol Stacks



An LMX is used by both the host and controller sides; however, the actual code may be different. Often, they are referred to as the host LMX and the controller LMX. They are also referred to as a Master LMX and a Slave LMX.

Figure 10-3 shows a diagram of the LMX software.

Figure 10-3. LMX Software Diagram



Include Files

LMX.H

lmx.h is a general file that contains information common to all LMXs. It also contains ‘#includes’ for all LMXs. The #includes are activated by the use of one or more of the USE_xxx_LMX defines. For example, to use SCSI, the compile line should contain /DUSE_SCSI_LMX. This causes the inclusion of the file LMXSCSI.H. Multiple LMXs may be used.

LMX_IOB

The LMX I/O Block (LMX_IOB) is used to control I/O to the LMX. It is defined in lmx.h and contains the following:

```
typedef struct  _LMX_IOB
{
    void          *pControllerContext;
    LMX_CONTEXT   *pLmxContext;
    LMX_ENTRIES   *pLmxEntries;
    void         (*receivePacketCallback)( struct _LMX_IOB *plmxIob );
    unsigned char *sendBuf;
    unsigned long  sendLength;
    unsigned char *recBuf;
    unsigned long  recLength;
    unsigned long  maxRecLength;
    LMX_STATUS     status;
    unsigned char  linkType;
} LMX_IOB;
```

Table 10-4. LMX_IOB fields.

Field	Description
pControllerContext	Pointer to a context used by CAPI. It is not to be used by the LMX.
pLmxContext	Pointer to a context which the LMX may use to keep I/O relevant information. The size of the context area is the size of the context structure in this LMX’s .h file. See LMX Context on page 347.
pLmxEntries	Pointer to a structure containing pointers to each of the LMX’s exported routines. See LMX_ENTRIES on page 348.
receivePacketCallback	Pointer to a call back routine that must be called upon completion of an I/O operation. The LMX must pass a pointer to this LMX_IOB to this routine. Before calling this routine, be sure to set the status field.
sendBuf	Pointer to the buffer that contains data to be sent.
sendLength	Length of the data in sendBuf.
recBuf	Pointer to the buffer in which to place received data. The maxRecLength field gives the maximum size of this buffer.
recLength	Set by the LMX to the number of bytes received into recBuf. It must be set prior to calling receivePacketCallback.
maxRecLength	Size of the recBuf buffer. Do not to overflow the size of the recBef buffer.
status	Set by the LMX to the status of the completed operation. It must be set prior to calling receivePacketCallback.
linkType	This field contains the link type, as defined in the **lmx.h** file.

Values for receivePacketCallback status

Values for the *status* field of the LMX_IOB structure are as follows:

```
#define  LMX_STATUS_NO_STATUS             0
#define  LMX_STATUS_GOOD                  1
#define  LMX_STATUS_LINK_BUSY             2
#define  LMX_STATUS_COMMUNICATION_ERROR   3
#define  LMX_STATUS_COMMUNICATION_TIMEOUT 4
#define  LMX_STATUS_READY_FOR_PROCESSING  5
```

Table 10-5. LMX_STATUS_* typedef descriptions.

Status	Description
LMX_STATUS_NO_STATUS	Set by CAPI before calling an I/O routine. This value must not be returned in status when receivePacketCallback is called.
LMX_STATUS_GOOD	Set by the LMX if the operation is good.
LMX_STATUS_LINK_BUSY	Set by the LMX if, when called for an I/O operation, the previous I/O operation is not complete. It is not required that the LMX check for busy because CAPI does not make any overlapping calls. The LMX may optionally provide this status as a cross check.
LMX_STATUS_COMMUNICATION_ERROR	Set by the LMX if an unrecoverable communications error exists. For example, if SCSI gets a selection timeout, it may return this error.
LMX_STATUS_COMMUNICATION_TIMEOUT	Set by the LMX if an operation takes too long. Most operations should finish well within one second. A timeout of at least 5 seconds is recommended. Calls to the timerTick routine occur every ½ second. You can use that to time your I/O.
LMX_STATUS_READY_FOR_PROCESSING	Used internally by CAPI and must not be returned by the LMX.

lmxXXX.h

There is a file called lmxXXX.h for each type of exchange used. The XXX letters are replaced with the respective LMX name. There must be a #include for this file in file lmx.h. Some are already included in lmx.h and are enabled with a compiler define, as explained above in the *lmx.h* section. If you are creating a new LMX, you will need to edit lmx.h. See lmx.h for details.

Defining the LMX's initialization routine

The entry name of the LMX is defined in the lmxXXX.h file. The #define called xxx_yyy_LMX_INITIALIZE_NAME defines the name of the entry point, where xxx is replaced with SLAVE or MASTER, and yyy is replaced with the interface type (SERIAL, SCSI, or whatever). These names are picked up by capi2pak.c and are used to initialize each LMX. For example, lmxscsi.h may contain:

```
#define  MASTER_SCSI_LMX_INITIALIZE_NAME  SCSILMX_Initialize.
```

Defining a Master or Slave

As stated previously, there are two types of LMXs: master and slave. A master LMX is used to interface between an application CAPI and the link. A slave LMX is used within the controller to interface between the CAPI layer and the interface. This master/slave relationship is given in LMXxxx.H and is usually (but not always) determined by the ifdef called REALHW.

For example, on the host side, the SCSI LMX is a master, so the name used there would be MASTER_SCSI_LMX_INITIALIZE_NAME. On the controller side, the name would be SLAVE_SCSI_LMX_INITIALIZE_NAME.

(Master LMXs are also used within the controller. For example, the LAN Subsystem communicates with the Storage Controller processor via CAPI; the LAN Subsystem is the master and the Storage Controller processor is the slave. Also, for Unified CAPI, communications between the two controller boards is via CAPI operated in a master/slave relationship.)

LMX Context

The LMX context is an area of memory passed to the LMX by CAPI. The LMX may do whatever it wants with this area of memory. The LMX supplies its memory via the following typedef (which is placed in lmxXXX.h):

```
typedef struct _LMXxxx_CONTEXT
{
    int whateverYouNeedHere;
} LMXxxx_CONTEXT;
```

xxx is replaced with SCSI, 232, or whatever.

Routines

Initialization Routine

The initialization routine is named by xxx_yyy_LMX_INITIALIZE_NAME as discussed in *Defining the LMX's initialization routine* on page 347. Its prototype is:

```
void xxx_yyy_initialization_name( void                         *pContext,
                                  LMX_INIT_CALLBACK_FUNCTION *pInitCompleteCallback,
                                  struct _LMX_ENTRIES          *pLmxEntries );
```

For example, if the name of your initialization routine is SCSILMX_Initialize, your lmxscsi.h file may contain:

```
#define  MASTER_SCSI_INITIALIZE_NAME SCSILMX_Initialize
void SCSILMX_Initialize( void                         *pContext,
                         LMX_INIT_CALLBACK_FUNCTION *pInitCompleteCallback,
                         struct _LMX_ENTRIES          *pLmxEntries );
```

Table 10-6. LMX_IOB fields:

Field	Description
pContext	Points to an area of memory used by CAPI to keep track of the initialization progress. Although it has a similar name, it is not related to the typedef LMXxxx_CONTEXT. This pointer must be passed as the first argument when calling pInitCompleteCallback.
pInitCompleteCallback	Points to a function which the LMX must be called when the initialization is complete. The pContext argument is passed back to CAPI via this callback. The callback also gives the status of the initialization. The status argument uses the same values as the LMX_IOB status. See Values for receivePacketCallback status on page 346. (must be LMX_STATUS_GOOD)
pLmxEntries	points to a structure which is to be filled in by the initialization routine. See structure definition below.

LMX_ENTRIES Structure Definition:

```
typedef struct _LMX_ENTRIES
{
    void     (*slaveReceive  )( struct _LMX_IOB *pLmxIob );
    void     (*sendAndReceive)( struct _LMX_IOB *pLmxIob );
    void     (*timerTick     )( void );
    CAPI_S32 (*findNextController)( CAPI_S32 firstTime, CAPI_S32 *lastTime,
                                    struct _LMX_CONTEXT *pLmxContext );
} LMX_ENTRIES;
```

Table 10-7. LMX_ENTRIES field descriptions:

Field	Description
slaveReceive	Pointer to the (SLAVE LMX, applicable to the LMX on the controller only) receive routine.
sendAndReceive	Pointer to the send and receive routine.
timerTick	Pointer to the timer tick routine. This routine is called by CAPI every ½ second. This gives an O/S independent LMX timing that can be used to time I/Os.
findNextController	Pointer to the find next routine. Note that when this routine is used in SLAVE context, it finds a connection not a controller.

Find Next Controller

This routine (not to be confused with CAPI_FindNextController), finds the next controller on a master system or finds the next connection on a slave system and returns TRUE if a controller/connection is found; otherwise it returns FALSE. It finds as many controllers/connections that exist for the LMX it supports. CAPI calls this routine until it returns with either NOT FOUND or *lastTime* equals TRUE.

```
int findNextController( int                   firstTime,
                        int                  *lastTime,
                        struct _LMX_CONTEXT *pLmxContext );
```

Table 10-8. findNextController parameter descriptions:

Parameter	Description
firstTime	Set to TRUE to start the list at the beginning. Set to FALSE to get the remaining controllers..
lastController	Set by this function routine when it does not want to be called again.
pLmxContext	Passed in with a pointer to an area of memory which the LMX can use and will have a size at least as large as the LMXxxx_CONTEXT structure given in LMXxxx.H. Each LMX_IOB contains a pointer to this same context.

Send And Receive

```
void sendAndReceive( struct _LMX_IOB *pLmxIob );
```

This routine is used by both master and slave LMXs. It is used to send a block of information and then receive a resulting block.

The LMX must set the status field of the LMX_IOB prior to returning from this function. If the I/O can be started, the status field must be set to LMX_STATUS_GOOD and there must be an accompanying call back. If the I/O cannot start, the status field must be set to some other value and a call back must not occur. See *LMX_IOB* on page 345 and *Values for receivePacketCallback status* on page 346.

After the receive operation is complete or if an unrecoverable error occurs after the I/O is started, this routine must call pLmxIob->receivePacketCallback(pLmxIob). The IOB pointer passed (pLmxIob) must be used when calling the receivePacketCallback routine. Also, pLmxIob->recLength and pLmxIob->status must be set. See LMX_IOB on page 345 and Values for receivePacketCallback status on page 346.

Slave Receive

This routine is called by the SLAVE interface only (the controller code). It is used to place the slave LMX in a receive mode.

```
void slaveReceive( struct _LMX_IOB *pLmxIob );
```

The status field of the LMX_IOB must be set prior to returning from this function. If the I/O can be started, the status field must be set to LMX_STATUS_GOOD and there must be an accompanying call back. If the I/O cannot start, the status field must be set to some other value and a call back must not occur.

After the receive operation is complete or if an unrecoverable error occurs after the I/O is started, this routine must call pLmxIob->receivePacketCallback(pLmxIob). The IOB pointer passed (pLmxIob) must be used when calling the receivePacketCallback routine. Also, pLmxIob->recLength and pLmxIob->status must be set. See LMX_IOB on page 345 and *Values for receivePacketCallback status* on page 346.

Timer Tick

This is called every LMX_TIME_FREQ microseconds. LMX_TIME_FREQ has been fixed at ½ second to allow CAPI to be in a separate DLL. This define is found in lmx.h.

```
void timerTick( void );
```

timerTick is called only once per tick even if more than one controller/connection exists for the LMX. (For example, if this LMX supports multiple connections, a call is not made for each connection.)
Not all LMXs need to have this function called; see this function in the LMX you are using to determine if it really does anything.

Adding a New Type of LMX

The LMX types defined thus far are for SCSI and RS-232 (plus ones defined for internal use within and between controllers). To add a new type, modify the following files:

makefile

The makefile defines which LMX(s) are to be used in the system. For example, USE_SERIAL_LMX specifies the use the RS-232 LMX. Define a new USE_xxx_LMX for the new interface.

lmxXXX.h

This contains the name of the initialization routine. Create a new MASTER_xxx_LMX_INITIALIZE_NAME and/or SLAVE_ xxx _LMX_INITIALIZE_NAME. Define if the name should be used as a MASTER or a SLAVE by defining USE_xxx_LMX_MASTER and/or USE_xxx_LMX_SLAVE. Define the LMXxxx_CONTEXT.

capi2pak.c

There is a table called LmxMasterTable. This contains pointers to each MASTER initialization routine. The pointers are obtained from lmxXXX.h. You will see an ifdef around each pointer. Note that the ifdef has the word _MASTER appended to the normal USE_xxx_LMX (for example, USE_SERIAL_LMX_MASTER). This is defined in the lmxXXX.h file. Add a new pointer using the new define.

lmx.h

Modify this to include the new lmxXXX.h file name.

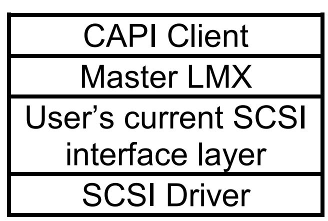
Specific Cases

The I/O hardware interface is not provided by this architecture. The application must interact with the hardware interface to set up its transmission characteristics such as data rates. For example, with a SCSI LMX, the LMX does not deal with fast/ultra, narrow/ wide, initiator IDs and so on; and with a Serial LMX, the LMX does not deal with data rates and methods used for polling.

Independent LMX

An Independent LMX is one that interfaces to user-supplied code. This can be used, for example, for an application that already has internal links to SCSI. This type of application only needs to supply a CAPI packet to the interface and receive a resulting packet back from the interface. One such implementation:

Figure 10-9. Independent LMX



In this example, the LMX only supplies a block of data to the user's current SCSI Interface, receives a block from the interface, calls the CAPI Client module back, and returns. This LMX may loop or block until the SCSI driver returns the CAPI result allowing the sequence of events to be synchronous rather than asynchronous.

Assume the user 's current SCSI interface sends data via a function int PutData(char *buffer, cdb) and receives data via a function int GetData(char *buffer, cdb). These functions return 0 if there was a SCSI timeout and GetData() returns the number of bytes received. The LMX would be written as follows:

```
/****************************************************************************/
void Lmx_Initialize( void *pContext, LMX_INIT_CALLBACK_FUNCTION
         *initCompleteCallback, struct _LMX_ENTRIES *pLmxEntries )
/****************************************************************************/
{
    pLmxEntries->findNextController = Lmx_FindNextController;
    pLmxEntries->sendAndReceive     = Lmx_SendAndReceivePacket;
    pLmxEntries->slaveReceive       = NULL;
    pLmxEntries->timerTick          = NULL;

    initCompleteCallback( pContext, LMX_STATUS_GOOD );
}

/****************************************************************************/
int Lmx_FindNextController( int firstTime, int *lastController,
                               struct _LMX_CONTEXT *pLmxContext )
/****************************************************************************/
{
    /* For completeness, you may want to scan the bus here and fill in
       LMX_CONTEXT with nexus information to allow for multiple controllers.
       A pointer to the LMX_CONTEXT is passed in the LMX_IOB. */

    *lastController = TRUE;
}

/****************************************************************************/
void Lmx_SendAndReceivePacket( struct _LMX_IOB *pLmxIob )
/****************************************************************************/
{
    static CAPI_U8 writeCdb[10] = { 0x3B,1,0,0,0,0,0,0,0,0 };
    static CAPI_U8 readCdb[10]  = { 0x3C,1,0,0,0,0,0,0,0,0 };
    *(CAPI_U16*)(&writeCdb[7])  = BigEndian16( pLmxIob->sendLength );

    if( PutData( pLmxIob->sendBuf, writeCdb ) )
    {
        *(CAPI_U16*)(&readCdb[7]) = BigEndian16( pLmxIob->maxRecLength );

        if( pLmxIob->recLength == GetData( pLmxIob->recBuf, readCdb ) )
            pLmxIob->status = LMX_STATUS_GOOD;
        else
            pLmxIob->status = LMX_STATUS_COMMUNICATIONS_ERROR;
    }
    else
        pLmxIob->status = LMX_STATUS_COMMUNICATIONS_ERROR;

    pLmxIob->receivePacketCallback( pLmxIob );
}
```

Notes:

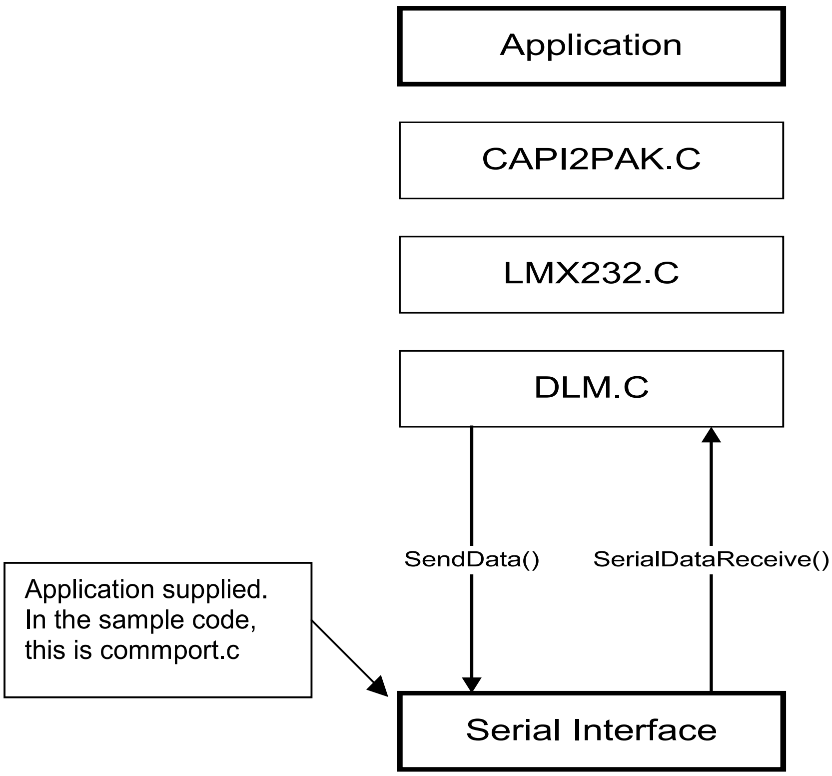
- A timerTick() is not needed because the example does not time the SCSI I/O at this level.
- Since this is a master, there is no need for the slaveReceive routine.
- The lmxXXX.h file appropriately reflects the initialization routine name.
- The function BigEndian16 is hypothetical and is not supplied by the CAPI SDK.

Serial LMX

The serial LMX is called lmx232.c and lmx232.h. It uses a Data Link Manager called dlm.c and dlm.h that includes the file mt_call.h.

This LMX is the same code for both the host and controller. The define called REALHW tells the LMX which side it is running on. If defined, the code is running in the controller and if it is not defined, it is running on the host.

Figure 10-10. Serial LMX



The application opens the serial interface, sets the baud rate and port numbering before initializing CAPI. If the serial line is polled, the application must do this on a timely basis. In the Chaparral sample code, these are done via InitComPort(), CheckSerialPort(), and CloseSerialPort().

Functions

SerialDataReceived

The LMX exports one entry point to be called by the user each time data is received. Its prototype is:

```
void SerialDataReceived( CAPI_U32 portNum, CAPI_U8 *buf, CAPI_U32 length );
```

Field	Description
portNum	Port number in which to send the data. This is a 0 relative number used between the DLM and the serial interface and may not represent the physical serial port number.
buf	Pointer to the buffer that contains the data received.
length	Length, in bytes, of the data received.

If the communications is to run using polling, the application must call the serial interface often enough to keep the data flowing. In the software example in commport.c, this is done by calling CheckSerialPort(). The prototype is:

```
void CheckSerialPort(void);
```

If communications runs on interrupts or if commport.c is not used, this function is not called.

SendData

The LMX imports one entry point which it calls each time data is to be sent. Its prototype is:

```
CAPI_U32 SendData( CAPI_U32 portNum, CAPI_U8 *data, CAPI_U32 length);
```

Field	Description
portNum	The port number in which to send the data. This is a 0 relative number used between the DLM and the Serial Interface and may not represent the physical serial port number.
data	Pointer to the data that is to be sent.
length	Length of the data.

The function returns CAPI_STATUS_GOOD if the transmit was successful (this means that the data started OK, not that it continued to transmit OK). CAPI_STATUS_COMMUNICATIONS error is returned if it could not start.

The *portNum* is a number passed between the serial interface and the DLM. It is 0 relative and does not necessarily mean a physical port number. It can be thought of more as a *serial controller number*. The serial interface must route these to the physical port that is connected to a controller. For example, a "0" could go out COM2 and a "1" could go out COM3.

Serial Line Characteristics

The serial line must be set for 8 data bits, no parity, and one stop bit. The data rate must agree with the rate set in the controller.

◇◇◇ 11

SCSI LMX

Introduction

CAPI command packets may be sent via the SCSI protocol by using the SCSI Link Manager Exchange (LMX). Currently supported transports for SCSI are parallel SCSI and Fibre Channel. This is frequently referred to as “in-band CAPI.”

Terminology note: The LUN used to communicate with the CAPI code on a controller (router or RAID controller board) is referred to in this chapter as the “controller LUN.” In the code and in some Chaparral documentation, this is often referred to as the “bridge LUN” and sometimes as the “CAPI LUN”; these are all the same thing. This same LUN is used for both CAPI and for the non-CAPI pass through feature (described in Chapter 17).

Read Buffer and Write Buffer Command Usage

CAPI requests to an LMX in our sample CAPI app consist of a single send/receive action. This allows the caller to send a request and receive a confirmation with a single call. Since most SCSI interfaces do not support back-to-back Data In and Data Out phases, two CDBs are used for each CAPI packet. The first CDB performs a Write Buffer command that sends the CAPI packet during the Data Out phase. The second CDB performs a Read Buffer command that receives the confirmation or result in the Data In phase.

Read Buffer and Write Buffer Error Handling

The controller can handle more than one SCSI initiator, and will gracefully handle this at a low level by returning Queue Full or Busy status (discussed more below). At a higher level, you may want to have the initiators communicate to ensure they do not interfere with each other; for example, you may not want to allow two users to simultaneously engage in configuration activities.

Each SCSI initiator must maintain its own Write Buffer/Read Buffer sequence. That is, it must successfully complete a Write Buffer command before it sends a Read Buffer command, and then successfully complete the Read Buffer command before sending another Write Buffer command.

Check condition with sense key 0x05 (ILLEGAL REQUEST), additional sense code 0x3b (PAPER JAM), and qualifier 0x05 is returned if a single initiator:

- sends a read buffer command without a prior successful write buffer command, or
- sends two write buffer commands in a row, or
- sends two read buffer commands in a row.

Other than the cases listed just above, sense data values follow standard SCSI practices. See the Request Sense section below for a list of other sense data that a CAPI app may encounter.

The controller will return Queue Full status (0x28) when it is out of resources due to too many commands being queued to any and all LUNs. Busy status (0x8) will only be returned by the controller LUN for the following reasons:

- An application attempts to send a command to the controller LUN before the previous command has completed. (You have to wait for each command to finish before you send another one. You can still use tagged commands, however.)
- A command is received when the controller has sense data pending (i.e., contingent allegiance) for a different initiator. (The SCSI spec permits a target to respond with busy status while waiting for an initiator to request sense data, and we make use of this in our controllers.) This only applies to parallel SCSI, since FC returns sense data immediately (i.e., autosense) and contingent allegiance is effectively cleared immediately.
- The initiator sending a CAPI Read Buffer command is different from the one that sent the previous CAPI write buffer command. This means that more than one initiator is doing CAPI work; the write/read pair has to be done without interruption from another initiator.
- An initiator sends a CAPI Write Buffer command after a different initiator has sent a CAPI Write Buffer, but before that different initiator has sent a CAPI Read Buffer command. This means that more than one initiator is doing CAPI work; the write/read pair has to be done without interruption from another initiator.

Host LMX

LMX code on the host system uses the SCSI Inquiry command to find targets that are CAPI-capable. CAPI commands are sent to the controller LUN, which is a processor device type LUN on the controller. All LUNs on a target should be checked since the controller LUN may have been assigned to any LUN. It is also possible for there to be LUN gaps. For example, the controller LUN may be LUN 2, but LUNs 0 and 1 may be unassigned. However, some operating systems may only recognize contiguous LUNs starting at 0.

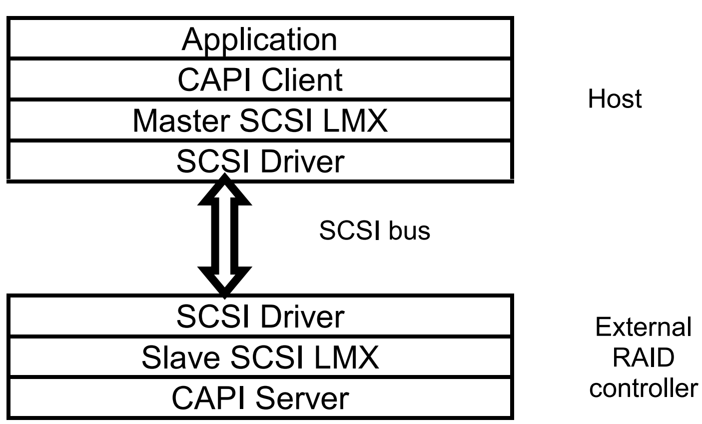
Send and Receive

For a send and receive LMX call, the sample code for an LMX performs the following actions:

1. Sends the WRITE BUFFER CDB and waits for completion.
2. Sends the READ BUFFER CDB and waits for completion.
3. Calls the CAPI layer's callback routine to signal completion.

These CDBs should be very quick and so spinning may be OK; however, if your operating system does not tolerate this, some form of blocking must be implemented in your CAPI application.

Figure 11-1. Example CAPI Protocol Stack



Hints

Here are some hints and caveats, mostly pulled from articles on the Microsoft Developers Network (MSDN) CDs:

- Windows NT by default applies the scanner device driver to all SCSI processor device type (0x03) devices. Thus, it sees CAPI as a scanner. But there is a bug in the scanner device driver that prevents more than one device from being seen by the OS. This limits CAPI. The workaround for this is to disable the scanner device driver on the PC running your CAPI application. The sample code in lmx_sc32.c assumes that this device driver has been disabled and the lmx_sc32.c code can find multiple controllers running CAPI. For Windows 2000, this is not an issue since the OS does not have a scanner device driver. To disable the device driver on an NT PC, select: Start|Settings|Control Panel|Devices|Scsiscan|Startup|Disabled|OK|Yes
- For Windows NT 4.0 and Windows 2000, SCSI pass through (SPT) requests are always synchronous, even if the caller to DeviceIoControl() has specified overlapped I/O (FILE_FLAG_OVERLAPPED). The sample code in lmx_sc32.c does not specify overlapped I/O and all commands are sent synchronously.
- For Windows NT 4.0 and Windows 2000, a SCSI command can be sent to the SCSI device as either untagged or tagged, but the SPT always uses untagged queuing while sending commands to the device. This should be a non-issue for CAPI applications; CAPI ignores whether commands are tagged or untagged. All commands to CAPI must be synchronous, as discussed at the beginning of this chapter and as implemented in the sample code in lmx_sc32.c.
- Starting with Windows NT 4.0 Service Pack 4 and beyond (including Windows 2000), there are new access requirements for SCSI pass through requests. For SCSI pass through requests, both GENERIC_READ and GENERIC_WRITE access must be specified in the dwDesiredAccess parameter of the CreateFile() call. If both read and write access are not specified, the DeviceIoControl() call will fail with ERROR_ACCESS_DENIED (5L). The sample code in lmx_sc32.c implements this requirement.
- For Win NT and Win 2000, only members of the administrator's group have the correct authority to send SCSI pass through requests. Users without administrator authority typically fail either CreateFile() or DeviceIoControl() with ERROR_ACCESS_DENIED (5L).
- For Win NT 3.5, when transferring data via the SCSI pass through (IOCTL_SCSI_PASS_THROUGH and IOCTL_SCSI_PASS_THROUGH_DIRECT), a transfer larger than the targeted SCSI host bus adapter (HBA) can support may crash the system.
- The sample code in lmx_sc32.c makes use of a call to DeviceIoControl() with a command of IOCTL_SCSI_GET_INQUIRY_DATA to find attached Chaparral controllers. This command returns the data that the OS found when it was booted up and may not reflect the current state of the SCSI devices connected to the PC. For example, if the Chaparral controller is not powered up at the time that the PC is booted, the code will not be able to find the CAPI LUN, so no CAPI management will be possible.
- Older versions of Solaris do not have a device driver that can see SCSI devices with a processor device type. A third-party device driver must be installed. One such driver is “sg” (generic SCSI device driver) available from Uniq Software Services. Solaris 8 apparently includes a driver called “sgen” that performs this function.
- Big-endian/little-endian issues are not addressed with the sample code in lmx_sc32.c. The embedded CAPI code runs on a processor that is compatible with Intel processors.

Controller SCSI Commands for CAPI

Inquiry

The controller responds to Inquiry requests with data identifying it as a CAPI device. The controller LUN and all SEP LUNs return with the peripheral device type set to SCSI processor device. The response data for the Inquiry command is of the standard form with some vendor-specific fields.

Table 11-1: INQUIRY Data

Bit / Byte	7	6	5	4	3	2	1	0
0	Peripheral Qualifier (0h)				Peripheral Device Type (03h)			
1	RMB (0)	*Reserved (00h)*						
2	0	0	0	0	0	ANSI-approved Version (03h)		
3	AERC (0)	Obsolete (0)	NormACA (0)	HiSup (1)	Response Data Format (02h)			
4	Additional Length (n-4) (9Bh)							
5	SCCS (0)	*Reserved (00h)*						
6	BQue (0)	EncServ (0)	VS (0)	MultiP (1)	MChngr (0)	Obsolete (0)	Obsolete (0)	Addr16 (0)
7	Rel Addr (0)	Wbus32 (0)	Wbus16 (0)	Sync (0)	Linked (0)	Obsolete (0)	CmdQue (1)	SoftRst (0)
8 - - - 15	(MSB) Vendor Identification (“CNSi “) (LSB)							
16 - - - 31	(MSB) Product Identification (LSB)							
32 - - - 35	(MSB) Product Revision Level (LSB)							
36 - - - 43	(MSB) *Unused* (LSB)							
44 - - - 49	(MSB) CAPI / SAF-TE Interface Identification String (“CAPI “) (LSB)							
50 - - - 95	(MSB) *Unused* (LSB)							
96 - - - 110	(MSB) Controller Identification String (“Chaptec Bridge “) (LSB)							

111 - - - 119	(MSB) Controller Firmware Version (LSB)
120 - - - 130	*Unused*
131 - - - 138	(MSB) SEP Vendor Identification (LSB)
139 - - - 154	(MSB) SEP Product Identification (LSB)
155 - - - 158	(MSB) SEP Product Revision Level (LSB)

Table 11-2: Inquiry Data Descriptions

CDB Field	Description
Peripheral Qualifier	Indicates if the selected LUN is a valid SCSI device. This field will be 000b.
Peripheral Device Type	Indicates the type of SCSI device. This field will be 03h (SCSI Processor Device).
ANSI-Approved Version	This field is 03h to indicate compliance with the ANSI SCSI-3 specifications.
Response Data Format	This field is 02h to indicate that the format of the INQUIRY response data is as defined in the ANSI SCSI-2 specification.
Additional Length	This field indicates the number of bytes of additional INQUIRY command parameters available for transfer, beginning with byte 05h. This value is not adjusted if the Allocation Length in the CDB is too large or too small to accommodate the entire response.
Rel Addr	Indicates the device supports relative addressing. Always 0.
WBus32	Indicates the device supports 32-bit wide data transfers. Always 0.
WBus16	Indicates the device supports 16-bit wide data transfers. (0 if Fibre Channel)
Sync	Indicates the device supports synchronous transfers. (0 if Fibre Channel)
Linked	Indicates the device supports linked commands. Always 0.
Reserved	Always 0.
CmdQue	Indicates the device supports tagged command queuing. Always 1 to be consistent with RAID LUNs, but not really supported for CAPI. (See explanation above in the section titled “Read Buffer and Write Buffer Error Handling.”)
SoftRst	Indicates the device supports soft resets. Always 0.
Vendor Identification	8-byte ASCII string that identifies the product vendor. It contains the same string used for data LUN INQUIRYs. This is “`CNSi    `“ or, in older firmware, “`ChapTec`”.
Product Identification	16-byte ASCII string that specifies the product ID. It contains the same string used for data LUN INQUIRYs.
Product Revision Level	4-byte ASCII string that specifies the product revision level (firmware level). It contains the same string used for data LUN INQUIRYs.
CAPI / SAF-TE Interface Identification String	6-byte ASCII string: It contains either the text string “CAPI ”, left aligned for the controller LUN and all data LUNs, or the string “SAF-TE” for any SEP

	LUNs. **Note**: The host CAPI LMX should check this string for “CAPI ”.
Controller Identification String	14-byte ASCII string that contains the key phrase “Chaptec Bridge” with a 15th pad character containing a blank (20h) before the next string. This text string is used by host-based CAPI applications to identify this as a CAPI controller.
Controller Firmware Version	8-byte ASCII string that contains the firmware version number with a 9th pad character containing a blank (20h) before the next string.
Note: The following definitions in bytes 131 through 158 are valid only if this is a SEP LUN **and** the *Insert Bridge Temperature* option is enabled. In this case, we save the original SEP inquiry vendor and product data in the vendor-specific parameters area below and insert the bridge’s vendor identification and product identification data into the standard inquiry positions from the bridge’s flash data.	
SEP Vendor Identification	8-byte product vendor identification string reported by the SEP.
SEP Product Identification	16-byte product identification string reported by the SEP.
SEP Product Revision Level	4-byte product revision level string reported by the SEP.

> ***Note:*** *The following Read Buffer and Write Buffer commands are used to implement the CAPI interface over SCSI and should not be confused with SAF-TE Read and Write Buffer commands.*

Write Buffer

The Write Buffer command is used to send a CAPI request during the Data Out phase. The contents of the data packet are described by the CAPI_PACKET structure in the file capipak.h. The CDB is as follows:

Table 11-3: Write Buffer Command CDB

Bit / Byte	7	6	5	4	3	2	1	0
0	Operation Code (**3B**h)							
1	Logical Unit Number			*Reserved*		Mode (01h)		
2	Buffer Id (00h)							
3	00h							
4	00h							
5	00h							
6	(MSB)							
7	Transfer Length							
8								(LSB)
9	00h							

Table 11-4: Write Buffer CDB field Descriptions

CDB Field	Description
Operation Code	3Bh is the Write Buffer command code.
Logical Unit Number	This field is ignored (LUN is specified via identify message.)
Mode	Should be set to 01h to indicate vendor specific mode.

Buffer Id	Set to 00h to indicate the CAPI command format.
Transfer Length	The number of bytes of data to be sent to the target = sizeof(CAPI_PACKET) + sizeof(any extra data).

Read Buffer

The Read Buffer command is used to receive data from CAPI in a Data In phase. The contents of the data packet is described by the CAPI_PACKET structure in the file capipak.h. The CDB is as follows:

Table 11-5: Read Buffer Command CDB

Bit Byte	7	6	5	4	3	2	1	0
0	Operation Code (**3C**h)							
1	Logical Unit Number			*Reserved*		Mode (01h)		
2	Buffer Id (00h)							
3	00h							
4	00h							
5	00h							
6	(MSB)							
7	Transfer Length							
8								(LSB)
9	00h							

Table 11-6: Read Buffer CDB field Descriptions

CDB Field	Description
Operation Code	3Bh is the Write Buffer command code.
Logical Unit Number	This field is ignored (LUN is specified via identify message.)
Mode	Should be set to 01h to indicate vendor specific mode.
Buffer Id	Set to 00h to indicate the CAPI command format.
Transfer Length	The maximum number of bytes of data to be returned from the target = sizeof(CAPI_PACKET) + sizeof(CAPI_EXTRA_DATA), which is slightly less than 2^{16} (65536) at this writing.

Test Unit Ready

This is the standard Test Unit Ready command, which returns a good status when the controller has completed its self-tests on power up.

Request Sense

The Request Sense command returns normal (standard SCSI) sense data. See above in the section titled “Read Buffer and Write Buffer Error Handling” for some notes on specific sense codes.

The following is a list of sense codes used in Chaparral controllers. Note that not all of these sense codes are used for the commands that you will use for CAPI applications, as described above. These codes are listed in the form that is used internally in Chaparral controllers. You may wish to use a different form in your CAPI application. For example, we have combined the ASC and ASCQ into a single, 16-bit number for our convenience, but you may wish to handle this as two, separate, 8-bit numbers.

Following the list of codes is information on which sense codes can occur for the commands described above and what events cause those codes to occur. Although we believe that this list of events is complete, we recommend that your CAPI app be designed to gracefully handle other codes as well.

```
// Sense keys.
const U8 SCSI_KEY_NO_SENSE          = 0x00;      // no sense data
const U8 SCSI_KEY_RECOVERED_ERROR   = 0x01;      // recovered error
const U8 SCSI_KEY_NOT_READY         = 0x02;      // not ready
const U8 SCSI_KEY_MEDIUM_ERROR      = 0x03;      // medium error
const U8 SCSI_KEY_HARDWARE          = 0x04;      // hardware error
const U8 SCSI_KEY_ILLEGAL_REQUEST   = 0x05;      // illegal request
const U8 SCSI_KEY_UNIT_ATTENTION    = 0x06;      // unit attention
const U8 SCSI_KEY_DATA_PROTECT      = 0x07;      // write/read protect
const U8 SCSI_KEY_BLANK_CHECK       = 0x08;      // blank medium or end of data
const U8 SCSI_KEY_VENDOR_SPECIFIC   = 0x09;      // vendor specific errors
const U8 SCSI_KEY_COPY_ABORTED      = 0x0a;      // copy aborted due to error
const U8 SCSI_KEY_ABORTED_COMMAND   = 0x0b;      // command aborted
const U8 SCSI_KEY_MISCOMPARE        = 0x0e;      // miscompare

// Additional sense codes and additional sense code qualifiers.
// Note:  These are kept in a 16 bit word with the ASCQ in the high byte and
//        the ASC in the low byte.  This is so on our little endian (x86)
//        processor, we can jam them in the sense data without byte reversing them.
//
const U16 SCSI_ASC_BUS_DEV_RESET    = 0x0329;    // bus device reset occurred
const U16 SCSI_ASC_CMD_PHASE        = 0x004a;    // command phase error
const U16 SCSI_ASC_CMD_SEQUENCE     = 0x004a;    // command sequence error
const U16 SCSI_ASC_CMDS_CLEARED     = 0x002f;    // commands cleared by another initiator
const U16 SCSI_ASC_DATA_PHASE       = 0x004b;    // data phase error
const U16 SCSI_ASC_DEFECT_LIST      = 0x0019;    // defect list error
const U16 SCSI_ASC_DEFECT_LIST_GRN  = 0x0319;    // defect list error in grown list
const U16 SCSI_ASC_DEFECT_LIST_PRI  = 0x0219;    // defect list error in primary list
const U16 SCSI_ASC_DEFECT_LIST_NA   = 0x0119;    // defect list not available
const U16 SCSI_ASC_DEFECT_LIST_NF   = 0x001c;    // defect list not found
const U16 SCSI_ASC_DEFECT_LIST_UPD  = 0x0132;    // defect list update failure
const U16 SCSI_ASC_DIAG_FAILURE     = 0x0040;    // diagnostic failure on component nn
const U16 SCSI_ASC_DME_NOT_ENABLED  = 0x0a04;    // DME segment not enabled
const U16 SCSI_ASC_DME_NOT_LOADED   = 0x1026;    // DME buffer id not loaded
const U16 SCSI_ASC_DME_BUFFER_ERROR = 0x0f26;    // DME physical buffer number miscompare
const U16 SCSI_ASC_DME_SEQ_ERROR    = 0x0e26;    // DME sequence number miscompare
const U16 SCSI_ASC_ERROR_LOG_OVF    = 0x000a;    // error log overflow
const U16 SCSI_ASC_IO_TERMINATED    = 0x0600;    // I/O process terminated
const U16 SCSI_ASC_INIT_DET_ERROR   = 0x0048;    // initiator detected error received
const U16 SCSI_ASC_INTERNAL_RESET   = 0x0429;    // device internal reset
const U16 SCSI_ASC_INVALID_IDENTIFY = 0x003d;    // invalid bits in identify message
const U16 SCSI_ASC_INVALID_CMD_CODE = 0x0020;    // invalid command operation code
const U16 SCSI_ASC_INVALID_CDB      = 0x0024;    // invalid field in CDB
const U16 SCSI_ASC_INVALID_PARM     = 0x0026;    // invalid field in parameter list
const U16 SCSI_ASC_INVALID_MESSAGE  = 0x0049;    // invalid message
const U16 SCSI_ASC_LAMP_FAILURE     = 0x0060;    // lamp failure
const U16 SCSI_ASC_LOG_COUNTER_MAX  = 0x025b;    // log counter at maximum
const U16 SCSI_ASC_LBA_TOO_BIG      = 0x0021;    // lba out of range
const U16 SCSI_ASC_LUN_FAILED_CFG   = 0x004c;    // LUN failed self configuration
const U16 SCSI_ASC_LUN_NOT_CFG_YET  = 0x003e;    // LUN not set configured yet
const U16 SCSI_ASC_LUN_GETTING_RDY  = 0x0104;    // LUN in process of becoming ready
const U16 SCSI_ASC_LUN_FORMATTING   = 0x0404;    // LUN not ready, format in progress
const U16 SCSI_ASC_LUN_MAN_INTERV   = 0x0304;    // LUN not ready, manual intervention needed
const U16 SCSI_ASC_LUN_NEEDS_INIT   = 0x0204;    // LUN not ready, init needed
const U16 SCSI_ASC_LUN_NOT_SUPP     = 0x0025;    // LUN not supported
const U16 SCSI_ASC_MESSAGE_ERROR    = 0x0043;    // message error
const U16 SCSI_ASC_NEW_MICROCODE    = 0x013f;    // microcode has changed
const U16 SCSI_ASC_VERIFY_MISCOMP   = 0x001d;    // miscompare during verify
const U16 SCSI_ASC_MODE_PARM_CHG    = 0x012a;    // mode parameters changed
const U16 SCSI_ASC_NO_SENSE         = 0x0000;    // no additional sense
const U16 SCSI_ASC_NO_SPARE         = 0x0032;    // no defect spare available
const U16 SCSI_ASC_OVERLAPPED_CMDS  = 0x004e;    // overlapped commands
```

```
const U16 SCSI_ASC_PAPER_JAM        = 0x053b;    // paper jam (for CAPI: 2 Read Buffer commands in
a row or 2 Write Buffer commands in a row)
const U16 SCSI_ASC_PARM_LIST_LENGTH = 0x001a;    // parameter list length error
const U16 SCSI_ASC_POWER_ON_RESET   = 0x0029;    // power on, reset or BDR occurred
const U16 SCSI_ASC_POWER_ON         = 0x0129;    // power on occurred
const U16 SCSI_ASC_PRIMARY_LIST_NF  = 0x0042;    // primary defect list not found
const U16 SCSI_ASC_RPT_LUN_CHANGE   = 0x0e3f;    // reported LUN’s data has changed
const U16 SCSI_ASC_SCSI_PARITY      = 0x0047;    // SCSI parity error
const U16 SCSI_ASC_SCSI_BUS_RESET   = 0x0229;    // SCSI bus reset occurred
const U16 SCSI_ASC_SDTR_ERROR       = 0x001b;    // SDTR error
const U16 SCSI_ASC_SYSTEM_RSRC      = 0x0055;    // system resource failure
const U16 SCSI_ASC_TARGET_CONDITION = 0x003f;    // target conditions changed
const U16 SCSI_ASC_XCVR_CHG_TO_LVD  = 0x0629;    // transceiver mode changed to LVD
const U16 SCSI_ASC_XCVR_CHG_TO_SE   = 0x0529;    // transceiver mode changed to SE
const U16 SCSI_ASC_WRITE_PROTECTED  = 0x0027;    // write protect error
const U16 SCSI_ASC_UNREC_READ_ERROR = 0x0011;    // unrecovered read error
const U16 SCSI_ASC_WRITE_ERROR      = 0x000c;    // write error
const U16 SCSI_ASC_NOTRDY_BUSY      = 0x0704;    // Logical Unit Not Ready, Operation in Progress

//
// Vendor-unique additional sense codes and qualifiers.
// These are used for the non-CAPI pass-through feature.
//
const U16 SCSI_ASC_VU_PT_NO_MEMORY  = 0x0080;    // pass through cmd couldn't allocate enough
memory
const U16 SCSI_ASC_VU_PT_DEVINARRAY = 0x0180;    // pass through cmd device in an array (not safe
to pass through)
const U16 SCSI_ASC_VU_PT_INVALIDBUS = 0x0280;    // pass through cmd sent to invalid bus
const U16 SCSI_ASC_VU_PT_SELTO      = 0x0380;    // pass through cmd selection timeout
const U16 SCSI_ASC_VU_PT_GENERROR   = 0x0480;    // pass through cmd general error
```

At initialization:

  SCSI_KEY_UNIT_ATTENTION SCSI_ASC_POWER_ON

General errors:

If invalid SCSI command code received:
  SCSI_KEY_ILLEGAL_REQUEST SCSI_ASC_INVALID_CMD_CODE

If parity error or (for Fibre Channel host connections) frame error:
  SCSI_KEY_ABORTED_COMMAND SCSI_ASC_SCSI_PARITY

If illegal request to non-zero LUN in LUA mode. (Should never happen unless CAPI app sends command to undefined LUN in our Target ID space.):
  SCSI_KEY_ILLEGAL_REQUEST SCSI_ASC_LUN_NOT_SUPP

For Inquiry:

If reserved fields in the message are non-zero, or the page code is not one of the ones that Chaparral supports (0x00 to retrieve supported pages, 0x80 for serial number, or 0x83 to retrieve device ids):
  SCSI_KEY_ILLEGAL_REQUEST SCSI_ASC_INVALID_CDB

For Test Unit Ready:

If reserved fields in the message are non-zero:
  SCSI_KEY_ILLEGAL_REQUEST SCSI_ASC_INVALID_CDB

For Request Sense:

If reserved fields in the message are non-zero:
  SCSI_KEY_ILLEGAL_REQUEST SCSI_ASC_INVALID_CDB

For Read Buffer:

If initiator sent a Read Buffer command without first sending a Write Buffer command:
  SCSI_KEY_ILLEGAL_REQUEST SCSI_ASC_PAPER_JAM

If initiator set the Mode to something other than 0x01 (vendor specific mode), or set the Buffer Id to an illegal value:
  SCSI_KEY_ILLEGAL_REQUEST SCSI_ASC_INVALID_CDB

For Write Buffer:

If parameter list length is an odd number of bytes:
  SCSI_KEY_ILLEGAL_REQUEST SCSI_ASC_PARM_LIST_LENGTH

If the Transfer Length is too long for our buffer:
  SCSI_KEY_ILLEGAL_REQUEST SCSI_ASC_INVALID_PARM

If initiator sent two Write Buffer commands in a row:
  SCSI_KEY_ILLEGAL_REQUEST SCSI_ASC_PAPER_JAM

If initiator set the Mode to something other than 0x01 (vendor specific mode), or set the Buffer Id to an illegal value:
  SCSI_KEY_ILLEGAL_REQUEST SCSI_ASC_INVALID_CDB

For pass-through requests to devices on the disk channels. (This is the non-CAPI pass-through feature.):

If can't get memory:
  SCSI_KEY_VENDOR_SPECIFIC SCSI_ASC_VU_PT_NO_MEMORY;

If not configured to allow passthrough:
  SCSI_KEY_VENDOR_SPECIFIC SCSI_ASC_VU_PT_DEVINARRAY

If user specified timeout >255:
  SCSI_KEY_ILLEGAL_REQUEST SCSI_ASC_INVALID_CDB

◇ ◇ ◇ 12

RS-232 LMX

Introduction

This chapter describes the protocol that implements a reliable, asynchronous, serial, RS-232 link between the host system and the controller. This is sometimes referred to as "out-of-band CAPI." The Data Link Manager (DLM) is defined as the code and the modules developed to implement the asynchronous, reliable protocol described in this chapter. The Link Manager Exchange (LMX) is defined as a higher-level module which uses the DLM.

Protocol Elements Description

A modified version of the BISYNC protocol for the Data Link Layer was used for the following reasons:

- Error checking and recovery is required.
- A sliding window of outstanding data frames is not required. At most, there is only one outstanding data frame open on both the master and slave sides.
- Absolute high performance is not required. This means the inefficiencies associated with the double transmission of the DLE-DLE character is acceptable.
- Binary byte data of any value is allowed in the data block portion of the frame.
- No XON/XOFF flow control characters are required, though their use is not prohibited.
- No flow control using the RTS-DTR-CTS … RS-232 leads is required. The RS-232 link is established on three wires: XMT Data, RCV Data, and ground.
- The Data Link can be full duplex; however, the actual exchange of data frames is done primarily in a half-duplex manner with a defined master-slave relationship.
- Large data transactions, up to 64 KBytes, can be handled. The largest data frame that can be sent is 256 bytes. For large data transactions, multiple data frames are coalesced into one final data buffer.
- Either the slave or the master can reset or initialize the link by sending the sequence DLE-BEL. However, the slave must respond with the frame DLE-SI to bring the link up. The master can detect that the link has gone down either by having a receive function time out when the link is turned or by no response to a polling status message periodically sent from the LMX for the master.

Framing

All I/O transmissions begin with DLE (Data Link Escape). Each DLE is followed by a unique character. When a DLE-DLE is received, it is treated as a single DLE. A typical block is framed as follows:

DLE-STX	payload	DLE-ETX	BCC-BCC

where DLE-STX indicates a start of frame, payload is up to 256 bytes, DLE-ETX is the end of frame marker, and BCC-BCC is a 16-bit Block Check Character (BCC).

Timeouts

```
RECOVERY_COUNT    = 5
LINE_BID_TIMEOUT = 1/2 second
TRANSMIT_TIMEOUT = 1   second
RECEIVE_TIMEOUT  = 2   seconds
```

TRANSMIT_TIMEOUT must exceed the maximum time to transmit a block. If an unrecognizable sequence is received when a block is expected, bytes are discarded for this amount of time.

RECEIVE_TIMEOUT is the maximum amount of time to receive an ACK and must be longer than TRANSMIT_TIMEOUT. If a block is transmitted but no ACK is received for a period of RECEIVE_TIMEOUT seconds, the transmitter waits TRANSMIT_TIMEOUT seconds, discards all incoming bytes, and sends DLE-ENQ. This solicits the last response sent by the receiver. The transmitter uses this to determine if the last block should be re-transmitted or if it was received correctly.

By waiting this length of time, a race condition is eliminated between sending the late ACK and the DLE-ENQ frame. Thus, RECEIVE_TIMEOUT is longer than TRANSMIT_TIMEOUT. It allows the receiver of garbage to re-sync and be ready to receive the DLE-ENQ sent RECEIVE_TIMEOUT seconds later.

If the connection is lost, the transmitter determines that there is a loss of connection after **RECOVERY_COUNT** unsuccessful timeout recovery attempts. The master attempts to establish a new connection by sending DLE-BEL every **LINE_BID_TIMEOUT** seconds. After the connection is re-established, the slave re-initializes its ACK counter as described above and responds appropriately.

BCC Calculation

BCC (Block Check Character) is a 16-bit CRC (CRC-16) and is calculated over all preceding characters except for the DLE-xxx lead-in characters and the first DLE of a DLE-xxx sequence. The CRC is not calculated over itself. The BCC accumulation consists of 2 to 4 bytes when it is transmitted on the line, but functionally is one sequence. For example, in the following sequence:

DLE-STX	some-data	DLE-DLE	more-data	DLE-ETX

the BCC is calculated over some-data, one DLE, more-data, and the ETX.

Embedded XON (0x11) and XOFF (0x13) characters may be within the BCC sequence. These cannot appear in the CRC because they are used for flow control. If the values 0x11 or 0x13 are part of the 16-bit CRC, they are encoded as DLE-DC2 (0x10-0x12) and DLE-DC4 (0x10-0x14). Therefore, the actual number of BCC bytes transmitted can be two, three, or four bytes depending on if any encoding is required for the BCC bytes.

Responses

After transmitting a block, the receiver replies with an ACK0, ACK1, or a DLE-NAK. ACK0 and ACK1 are shorthand for the following sequences: ACK0 = DLE-‘p’, ACK1 = DLE-‘a’. Each time a block is acknowledged, the receiver of the block advances to the next ACK. Each time an acknowledgment is correctly received, the transmitter advances to the next ACK.

If the block is not received well enough to determine if it is even a block, no reply is given and characters are ignored for a period of RECEIVE_TIMEOUT seconds. If an acknowledgment is not received well enough to determine if it is valid, characters are ignored for a period of TRANSMIT_TIMEOUT seconds.

Out-of-Sequence

An out-of-sequence condition occurs when the sender of a block asks for a repeat of the last response and the receiver repeatedly responds with the wrong ACK.

When an ACK frame is received that is out of sequence, the DLM responds with a DLE-ENQ response. If the response to the DLE-ENQ is still out of sequence after RECOVERY_COUNT attempts, then the DLM returns to an uninitialized state.

Establishing a Connection

A new connection starts with DLE-BEL. That causes the receiver to initialize its next reply to ACK0. The receiver then responds with DLE-SI. DLE-BEL can also be used to synchronize the receiver in the event that a hopeless out-of-sequence condition exists.

Master/Slave and Line Turn

One device is designated to be a master and the other is a slave. During initialization, only one device may be a master and the other must be a slave. The master is always initialized as a transmitter and a slave is initialized as a receiver. When this relationship must be exchanged, the master turns the line around by issuing a Line Turn Sequence. This sequence is DLE-ESC and replaces the normal DLE-ETX of a data frame. It is acceptable to send a zero-length frame.

When the slave initializes, it must ignore all characters until a DLE-BEL is received.

Jabber Frames

For frames longer than 512 bytes, all data collected is thrown out. A jabber frame can occur when mismatching baud rates are present on the line. When a jabber frame is found, the DLM waits RECEIVE_TIMEOUT seconds and then begins scanning for the valid start of the frame. By waiting this time, a valid start of frame sequence embedded in a jabber frame is not misinterpreted.

Stalled Frames

Whenever a valid start of frame sequence is received, a receive timer starts which limits the time to wait to receive a valid end of frame. If the RECEIVE_TIMOUT occurs, then the DLM throws out any data received up to that point and searches for a valid start up data sequence.

Link Up Status Checking

The software layers above the DLM are responsible for sending data frames at regular time intervals that check the link status and the status of the other device. This protocol is not responsible for periodic polling to determine link status.

Data-Link Control

Control of the data link is maintained through the use of the following control characters and sequences:

Table 12-1. Control Character Sequences

DLE-STX	Start of Text Sequence. This indicates a start of a frame.
DLE-ETX	End of Text Sequence. This indicates the end of a frame as well as the last block of data within a data transaction. This also indicates that the link has not turned and our side is still the only one which can transmit data.
DLE-ETB	End of Block Sequence. This indicates the end of a frame as well as indicating that additional data blocks will follow to complete a data transaction transfer.
DLE-ESC	End of Text Sequence with Line Turn. This indicates the end of a frame as well as the last block of data within a data transmission. This also indicates that the link has turned and the other side can now transmit data. This is the same as DLE-ETX except with a line turn indication.
DLE-BEL	Line Bid. This is used to initialize the system.
DLE-SI	Response from slave to a line bid frame. This is used to bring the link up.
DLE-ENQ	Enquiry. This is used to recover from a lost ACK.
DLE-‘p’	ACK0 affirmative acknowledgment to an even block.
DLE-‘a’	ACK1 affirmative acknowledgment to an odd block.
DLE-NAK	Negative acknowledgment.
DLE-DLE	A single DLE within the payload.
DLE-DC2	A single DC1 (XON) byte within the payload or BCC.
DLE-DC4	A single DC3 (XOFF) byte within the payload or BCC.
BCC-BCC	A CRC-16 Block Check Character (BCC) sequence.

Line Encodings

One of the basic characteristics of the modified BISYNC protocol developed at Chaparral is to allow the transmission of binary byte data of any value. For some async terminal connections, this may cause problems with the XON (DC1 - Hex Value 0x11) and XOFF (DC3 - Hex Value 0x13) bytes. Therefore, these byte values are always encoded with a two-byte sequence where:

```
XON (DC1)  = DLE-DC2
XOFF (DC3) = DLE-DC4
```

Also, the DLE byte is encoded as DLE-DLE.

These three line encodings are in effect for payload data. The line encodings for XON/XOFF are valid for the BCC bytes as well.

Example Data Exchanges

The following tables are examples of different data exchanges.

Table 12-2. Initialize system to perform simple data exchange

Master	Slave
DLE-BEL —>	
	<—— DLE-SI
DLE-STX DATA DLE-ESC BCC —>	
	<—— ACK0
	<—— DLE-STX DATA DLE-ESC BCC
ACK0 —>	

Table 12-3. Perform data transaction of 513 bytes

Master	Slave
DLE-STX 0x00 0x01 … 0xff DLE-ETB BCC ——>	
	<—— ACK0
DLE-STX 0x00 0x01 … 0xff DLE-ETB BCC ——>	
	<—— ACK1
DLE-STX 0x00 DLE-ETX BCC ——>	
	<—— ACK0

Table 12-4. Out-of-sequence ACK received

Master	Slave
DLE-BEL ——>	
	<—— DLE-SI
DLE-STX DATA DLE-ETX BCC ——>	
	<—— ACK1
DLE-ENQ ——>	
	<—— ACK1
DLE-ENQ ——>	
	<—— ACK1
DLE-BEL ——>	
	<—— DLE-SI

Table 12-5. ACK timeout occurs

Master	Slave
DLE-BEL ——>	
	<—— DLE-SI
DLE-STX DATA DLE-ETX BCC ——>	
	<—— ACK1 (lost at Master)
(TRANSMIT_TIMEOUT Seconds Later:) DLE-ENQ ——>	
	<—— ACK1

Table 12-6. BCC error occurs on the data transmission

Master	Slave
DLE-BE ——>	
	<—— DLE-SI
DLE-STX DATA DLE-ETX BAD BCC ——>	
	<—— DLE-NAK
DLE-STX DATA DLE-ETX GOOD BCC ——>	
	<—— ACK0

The examples listed above do not represent all possible error conditions that can occur in data exchanges. The intent of the examples is to provide an understanding of how each Data Link Control sequences is used within the protocol.

Error Handling

The following cases describe how error conditions are handled on the line. References to LMX responses are dependent on the actual implementation.

Case 1: A Data Frame is sent with a Bad BCC

A DLE-NAK frame is sent in response. Up to RECOVERY_COUNT DLE-NAKs can be received by the data transmitter before a failure code is sent back to the LMX. After this, the DLM returns to an uninitialized state.

Case 2: ACK not received within TRANSMIT_TIMEOUT seconds

After a data frame is sent and does not get an ACK back within TRANSMIT_TIMEOUT seconds, the transmitter sends a DLE-ENQ frame up to RECOVERY_COUNT times. If a successful ACK is never received, a failure code is sent back to the LMX and the DLM returns to an uninitialized state. The DLE-ENQ frame is sent every TRANSMIT_TIMEOUT seconds.

Case 3: A Jabber Frame is received

The DLM discards the data frame and begins searching for a valid start of frame to respond to after RECEIVE_TIMEOUT seconds.

Case 4: A DLE-BEL reset request is received during data transaction

The data transaction is terminated and an error message is passed up to the LMX. The DLM is then placed into the uninitialized state and responds back with a DLE-BEL frame to bring up the link again.

Case 5: An out-of-sequence ACK frame is received

At the receipt of the out-of-sequence frame, a DLE-ENQ is sent. If the response is still out of sequence after RECOVERY_COUNT attempts, then the DLM returns to the uninitialized state. If the DLM is in the middle of a data transaction, then an error is reported back to the LMX.

Case 6: A receive overrun occurs

The BCC, RECEIVE_TIMEOUT, and TRANSMIT_TIMEOUT protocols give the proper error recovery. If desired, the DLM can return to an uninitialized state and an error can be reported to the LMX.

Case 7: Receipt of unexpected frames

Any receipt of frames that do not match one of the defined Data Link Control formats is ignored. Again, all legal frames start with one of the following:

- DLE-STX
- DLE-BEL
- DLE-SI
- DLE-ENQ
- DLE-‘a’
- DLE-‘p’
- DLE-NAK

The DLM constantly searches for these sequences.

Case 8: Receipt of unexpected escape sequence

Within the data block of a frame, the only legal escape sequences include:

- DLE-ETX—End of text marker.
- DLE-ESC—End of text and line turn marker.
- DLE-ETB—End of frame marker.
- DLE-DLE—DLE char within the data.
- DLE-DC2—DC1 (XON) char within the data or BCC.
- DLE-DC4—DC3 (XOFF) char within the data or BCC.

An invalid sequence results in a BCC error and the previously mentioned error handling takes care of this

How to Get Serial Port Back to Disk Array Administrator

Since the Disk Array Administrator (also known as the Menu User Interface or MUI) uses the same serial port that the serial LMXs use, it is necessary to have a mechanism to tell the controller which interface it should present. By default, the serial port comes up in MUI mode. The serial LMXs send a Ctrl-P character to the serial port to tell it that it should switch from MUI mode to CAPI mode. Your application should not need to be concerned with this if you use either of our serial LMXs.

If you are running a serial CAPI application and wish to switch back to MUI mode, you should send the character sequence **Ctrl-P Ctrl-Z**. Typically, this is accomplished by connecting a terminal emulator such as HyperTerminal to the serial port and then typing **Ctrl-P Ctrl-Z**.

◊◊◊ 13

SIMPLIFIED RS-232 LMX

Introduction

A simplified RS-232 protocol has been implemented on CAPI3 controllers to better support some embedded systems that have difficulties running the standard RS-232 protocol described in Chapter 12. This simplified protocol is a "non-guaranteed delivery" protocol and requires the user application to retry any failed commands. The only way to detect failed commands is to time out on not receiving a response from the external controller. This protocol can be used by compiling in DLMJ.C, DLMJ.H, LMX232J.C and defining DLMJ and not defining DLM. The standard protocol and not this protocol should be used in most cases, as it provides much error handling and correction. This protocol sends the entire data payload at once (which could be up to 64KB) without using any flow-control mechanism (XON/XOFF characters are supported by the protocol). CAPI3 controllers will respond to either protocol and cannot switch protocols without a reboot.

Note: *This LMX is not supported by controllers that have a LAN processor.*

Protocol Elements Description

The protocol is extremely simple. There is a MASTER/SLAVE relationship. The application is the MASTER and the external controller is the SLAVE. All transfers are initiated by the MASTER. If the SLAVE receives a good packet, it will reply with a packet. If the packet is bad or incomplete, the SLAVE will ignore it. In this case, the application should timeout waiting for the reply and retry the command. The protocol will not re-send the packet; it is up to the application to timeout and retry.

The entire message is in one packet. The message is not broken up into smaller blocks of data. The packet has this format:

PACKET FORMAT:

DLE-BS DLE-STX PAYLOAD HIGHBYTE_CRC16 LOWBYTE_CRC16 DLE-ETX

CONTROL CHARACTER ENCODING:

If a DLE is in the payload, it will be encoded as DLE – DLE
If a XON is in the payload, it will be encoded as DLE – DC2
if a XOFF is in the payload, it will be encoded as DLE – DC4
if a SPACE is in the payload, it will be encoded DLE – ENQ
if a CTRL-C is in the payload, it will be encoded DLE – TAB

If a DLE is in the CRC, it will be encoded as DLE - ETB

CHARACTER DEFINES:

```
#define SPACE_CHAR                    0x20
#define CTRLC_CHAR                    0x03
#define TAB_CHAR                      0x09
#define DLE_CHAR                      0x10
#define STX_CHAR                      0x02
#define ETX_CHAR                      0x03
#define ETB_CHAR                      0x17
#define ENQ_CHAR                      0x05
#define DC2_CHAR                      0x12
#define DC4_CHAR                      0x14
#define XON_CHAR                      0x11
#define XOFF_CHAR                     0x13
#define BS_CHAR                       0x08
```

How to Get Serial Port Back to Disk Array Administrator

Since the Disk Array Administrator (also known as the Menu User Interface or MUI) uses the same serial port that the serial LMXs use, it is necessary to have a mechanism to tell the controller which interface it should present. By default, the serial port comes up in MUI mode. The serial LMXs send a Ctrl-P character to the serial port to tell it that it should switch from MUI mode to CAPI mode. Your application should not need to be concerned with this if you use either of our serial LMXs.

If you are running a serial CAPI application and wish to switch back to MUI mode, you should send the character sequence **Ctrl-P Ctrl-Z**. Typically, this is accomplished by connecting a terminal emulator such as HyperTerminal to the serial port and then typing **Ctrl-P Ctrl-Z**.

◇ ◇ ◇ 14

CHANGES BETWEEN CAPI 2.X AND CAPI 3.X

Major changes

CAPI was upgraded significantly in CAPI 3.x. Here are some highlights of what changed:

> *NOTE:* ***CONTROLLER MAY NOT SUPPORT ALL NEW FEATURES; CAPABILITY BITS IN CAPI_CONTROLLER SHOULD BE CONSULTED.***

- CAPI_ARRAY_PARTITIONs added, allowing arrays to be carved up into multiple host visible LUNs.
- more front-end channels
- more back-end channels
- more drives (250 per controller maximum, 125 per channel maximum)
  - CAPI_DRIVE structures are retrieved via a call to CAPI_GetDrives.
- more arrays (32 initially, up to 32 * 8 using future bank switching)
  - CAPI_ARRAY structures are retrieved via a call to CAPI_ArrayDrives, CAPI_CONTROLLER has a reference to the index of this array.
- There is an added level of indirection to associate CAPI_MEMBER drive (logical array drive) to CAPI_DRIVE (physical drive)
- support for Fibre Channel devices
- support for Router product
- more logical unit numbers
- changed the word “SAFTE” to “ENVIRON” for inclusion of other environmental processors such as SES
- password capability
- advanced LUN mapping (Router products only)
- InfoShield (using GetHostTable, Add/RemoveHost)
- bus speed changed from a #define to actual speed in MB/s
- CAPI_FLEX_ID is a flexible ID that is used for both Fibre Channel and SCSI in the InfoShield functions
- some controller parameters have been moved to the channel parameter structure because of multiple front-end channel capability
- CAPI_CAPABILITY_2_SMART_SUPPORT has been split into HOST and DISK SMART_SUPPORT.
- Multiple Controller Modes, including some of which support a dual controller system (i.e. “Active-Active” controllers).

◇ ◇ ◇ 15

CAPABILITIES

JSS122 (G6322) L410 / JFS224 (G8324) L411 Implementation

This section shows the CAPI CAPABILITIES bits for the Chaparral JSS122 (G6322) and JFS224 (G8324) controllers with L410 and L411 firmware. For the most accurate information, the application developer should always consult the capability bits returned from a particular controller. &&&&

- CAPABILITIES
  - CAPI_CAPABILITY_SPARE_POOL
  - CAPI_CAPABILITY_DEDICATED_SPARE
  - CAPI_CAPABILITY_READ_AHEAD_CACHE
  - CAPI_CAPABILITY_WRITE_BACK_CACHE
  - CAPI_CAPABILITY_SAFTE
  - CAPI_CAPABILITY_ARRAY_STATS
  - CAPI_CAPABILITY_FORMAT_AT_CREATION
  - CAPI_CAPABILITY_AUTO_VERIFY_FIX
  - CAPI_CAPABILITY_ONLINE_CAPACITY_EXPAND
  - CAPI_CAPABILITY_ARRAY_NAME
  - CAPI_CAPABILITY_RAID0
  - CAPI_CAPABILITY_RAID1
  - CAPI_CAPABILITY_RAID3
  - CAPI_CAPABILITY_RAID4
  - CAPI_CAPABILITY_RAID5
  - CAPI_CAPABILITY_RAID10
  - CAPI_CAPABILITY_RAID50
  - CAPI_CAPABILITY_RAID_VOLUME_SET
  - CAPI_CAPABILITY_2_ABORT_CREATE_ARRAY
  - CAPI_CAPABILITY_2_SCSI_MAINT_COMMANDS
  - CAPI_CAPABILITY_2_TEST_SPARES
  - CAPI_CAPABILITY_2_FIRMWARE_DOWNLOAD
  - CAPI_CAPABILITY_2_DRIVE_SERIAL_NUMBERS
  - CAPI_CAPABILITY_2_DISK_SMART_SUPPORT
  - CAPI_CAPABILITY_2_MULTIPLE_HOST_CHANNELS
  - CAPI_CAPABILITY_2_FAILOVER_ACTIVE_ACTIVE
  - CAPI_CAPABILITY_2_INFOSHIELD (G7324/G8324 only)
  - CAPI_CAPABILITY_2_ARRAY_PARTITIONS
  - CAPI_CAPABILITY_2_DYNAMIC_POOL_SPARES
  - CAPI_CAPABILITY_2_2GB_FC_SPEED_SUPPORT (G8324 only)

- CAPI_CAPABILITY_2_AUTO_FC_TOPOLOGY_SUPPORT (G8324 only)
- CAPI_CAPABILITY_2_ONLINE_ARRAY_INIT

- FEATURES
  - controller->raid.maxChunkSize = 64
  - controller->raid.minChunkSize = 16

JFS226 (A8526) A400 Implementation

This section shows the CAPI CAPABILITIES bits for the Chaparral JFS226 (A8526) controller with A400 firmware. For the most accurate information, the application developer should always consult the capability bits returned from a particular controller.

- CAPABILITIES
  - CAPI_CAPABILITY_SPARE_POOL
  - CAPI_CAPABILITY_DEDICATED_SPARE
  - CAPI_CAPABILITY_READ_AHEAD_CACHE
  - CAPI_CAPABILITY_WRITE_BACK_CACHE
  - CAPI_CAPABILITY_SAFTE
  - CAPI_CAPABILITY_ARRAY_STATS
  - CAPI_CAPABILITY_FORMAT_AT_CREATION
  - CAPI_CAPABILITY_AUTO_VERIFY_FIX
  - CAPI_CAPABILITY_ONLINE_CAPACITY_EXPAND
  - CAPI_CAPABILITY_ARRAY_NAME
  - CAPI_CAPABILITY_RAID0
  - CAPI_CAPABILITY_RAID1
  - CAPI_CAPABILITY_RAID3
  - CAPI_CAPABILITY_RAID4
  - CAPI_CAPABILITY_RAID5
  - CAPI_CAPABILITY_RAID10
  - CAPI_CAPABILITY_RAID50
  - CAPI_CAPABILITY_RAID_VOLUME_SET
  - CAPI_CAPABILITY_2_ABORT_CREATE_ARRAY
  - CAPI_CAPABILITY_2_SCSI_MAINT_COMMANDS
  - CAPI_CAPABILITY_2_TEST_SPARES
  - CAPI_CAPABILITY_2_FIRMWARE_DOWNLOAD
  - CAPI_CAPABILITY_2_MULTIPLE_HOST_CHANNELS
  - CAPI_CAPABILITY_2_DRIVE_SERIAL_NUMBERS
  - CAPI_CAPABILITY_2_INFOSHIELD
  - CAPI_CAPABILITY_2_ARRAY_PARTITIONS
  - CAPI_CAPABILITY_2_DEV_MEM_EXPORT_PROTOCOL
  - CAPI_CAPABILITY_2_DYNAMIC_POOL_SPARES
  - CAPI_CAPABILITY_2_DISK_SMART_SUPPORT
  - CAPI_CAPABILITY_2_2GB_FC_SPEED_SUPPORT
  - CAPI_CAPABILITY_2_AUTO_FC_TOPOLOGY_SUPPORT
  - CAPI_CAPABILITY_2_AUTO_FC_SPEED_SUPPORT

- FEATURES
  - controller->raid.maxChunkSize = 64
  - controller->raid.minChunkSize = 16

◇ ◇ ◇ 16

FAILOVER NOTES

Placeholder LUN

The CAPI placeholder LUN is used automatically by the controller in an active-active configuration if failover occurs when information on the other controller's controller LUN is not available. This can happen if a single controller boots. In that case, the placeholder LUN will be enabled automatically if there is a LUN gap in the other controller's LUNs. For example, if the controller B has a single array LUN at LUN 1, and controller A boots when B is not plugged in, then A will present B's array LUN at LUN 1 and it will present a placeholder LUN at LUN 0 to fill in the LUN gap. The intent is that some host OS's can't handle gaps in the LUN sequence and will stop scanning if they see one. The placeholder allows them to see all LUNs if there is a single LUN gap.

◇ ◇ ◇ 17

NON-CAPI PASS THROUGH FEATURE

Introduction

Chaparral RAID controllers provide the ability to directly access SCSI devices on the back end (disk) channels with *pass through* commands. There are two mechanisms for pass through commands provided by Chaparral controllers:

- Via CAPI commands, blocks of data up to 32 KBytes can be passed through (defined as CAPI_MAX_MAINT_DATA_SIZE in capipak.h). This pass through feature is accessed via CAPI_ScsiMaintenance and CAPI_ScsiMaintRetrieveData.
- Blocks up to 1 MByte can be transferred via a different mechanism that bypasses CAPI. This feature is documented in this chapter. When this chapter refers to “pass through” it is this non-CAPI pass through mechanism that is being referred to.

Both of these pass through mechanisms use the same LUN, referred to as the “controller LUN” or sometimes as the “bridge LUN” or “CAPI LUN.” Messages sent to this LUN are routed internally to the CAPI or non-CAPI pass through mechanism within the Chaparral controller based on the SCSI Operation Code in Byte 0 of each message.

These pass through mechanisms are supported for Chaparral RAID controllers, but not Chaparral routers.

Pass Through Command

The non-SCSI pass through command uses a 16-byte CDB. Embedded in the 16-byte CDB is a 6- or 10-byte CDB that is sent directly to the back-end device, plus some routing information so the controller knows which device to send it to.

<table>
<tr><th></th><th>Bit7</th><th>Bit6</th><th>Bit5</th><th>Bit4</th><th>Bit3</th><th>Bit2</th><th>Bit1</th><th>Bit0</th></tr>
<tr><td>Byte 0</td><td colspan="8">Operation Code (D7h)</td></tr>
<tr><td>Byte 1</td><td colspan="8">TargetID</td></tr>
<tr><td>Byte 2</td><td>DOut</td><td colspan="3">Channel</td><td colspan="4">CDBLength</td></tr>
<tr><td>Byte 3</td><td colspan="4">TargetLUN</td><td colspan="4">(MSB)</td></tr>
<tr><td>Byte 4</td><td colspan="8">Data Transfer Length</td></tr>
<tr><td>Byte 5</td><td colspan="8">(LSB)</td></tr>
<tr><td>Byte 6</td><td colspan="8" rowspan="10">10-Byte CDB</td></tr>
<tr><td>Byte 7</td></tr>
<tr><td>Byte 8</td></tr>
<tr><td>Byte 9</td></tr>
<tr><td>Byte10</td></tr>
<tr><td>Byte11</td></tr>
<tr><td>Byte12</td></tr>
<tr><td>Byte13</td></tr>
<tr><td>Byte14</td></tr>
<tr><td>Byte15</td></tr>
</table>

Table 1 Pass Through CDB

The routing information consists of disk channel number (Channel), TargetID, and TargetLUN.

CDBLength is the length of the embedded CDB (in bytes 6-15).

DOut=1 indicates that the command requires data out phase. DOut=0 indicates data in phase, or no data transfer if Data Transfer Length is zero.

Data Transfer Length is the number of bytes to transfer. This is a 20-bit field, providing for maximum of 1MB of data transferred.

Pass Through To Array Members

Although any SCSI CDB may be passed through to any back end SCSI device, the controller attempts to protect array members' user data. Commands may be sent to non-array-member devices with no restrictions. Commands sent to an array member disk are permitted only if:

- the array is fault tolerant, or
- the command is one of: Inquiry, Mode Sense (6 or 10), Read (6 or 10), Read Capacity, Request Sense, or Test Unit Ready. (These are the "safe" commands.)

If the array is fault tolerant and the pass through CDB is not one of those listed above, a Down Drive command will be internally issued to the target device of the pass through operation**. This will cause the array to go to Critical (Non Fault Tolerant) state**. If the array was not originally fault tolerant and an unsafe pass through command is attempted, the controller will not pass the command through.

Pass Through Errors

Errors returned from the target device are indicated by request sense Error Code 7Fh (vendor specific), so that host software issuing them can distinguish them from errors detected by the controller. Sense data reported by the target device will be reported via the normal check condition/request sense method (or autosense for Fibre Channel). Pass through command errors detected by the controller (as opposed to the target device) are reported by request sense Error Code 70h, Sense Key 9 (vendor specific). The following vendor specific pass through errors are reported:

Error Code	Sense Key	Additional Sense Code	Additional Sense Code Qualifier	Meaning
70h	9	80h	0	Not enough memory for requested operation. The command may work if retried. A write back cache full of dirty data can cause this error.
70h	9	80h	1	Target device is a member of a non-fault-tolerant array and the command issued was not one of the “safe” commands. Command was not sent to target.
70h	9	80h	2	Invalid channel number specified.
70h	9	80h	3	No response from target (selection timeout in parallel SCSI).
70h	9	80h	4	General error (none of above).

Table 2 Pass Through Errors

Pass Through Timeout Command

Chaparral RAID controllers provide the ability to change the pass through command timeout. This is a useful feature for doing time-intensive pass through operations; for example, loading firmware onto a drive through the controller. The timeout pass through command is 16 bytes as described in Table 3.

	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
Byte 0	Operation Code (D8h)							
Byte 1	Timeout (MSB) (not used)							
Byte 2	Timeout (LSB)							
Byte 3	Reserved[0]							
Byte 4	Reserved[1]							
Byte 5	Reserved[2]							
Byte 6	Reserved[3]							
Byte 7	Reserved[4]							
Byte 8	Reserved[5]							
Byte 9	Reserved[6]							
Byte10	Reserved[7]							
Byte11	Reserved[8]							
Byte12	Reserved[9]							
Byte13	Reserved[10]							
Byte14	Reserved[11]							
Byte15	Reserved[12]							

Table 3 Pass Through Timeout CDB

Although there are 2 bytes, bytes 1 and 2, allocated for the timeout setting, the controller only evaluates the value at byte 2. Since the timeout value units are in seconds, the maximum timeout that can be set is 255 seconds.

If this command is not used to set the timeout, a default timeout of 60 seconds is used for pass through commands.

Pass Through Timeout Errors

Errors returned by the controller are listed in Table 4.

Error Code	Sense Key	Additional Sense Code	Additional Sense Code Qualifier	Meaning
70h	5	24h	0	Illegal request, invalid field in CDB, indicating timeout value is > 255. Note: since the controller doesn't evaluate the value in byte 1 in the current version, it won't return this error.

Table 4 Pass Through Timeout Errors

18

CAPI Interface Without Using the SDK

Some developers prefer to not use the Chaparral CAPI Software Developer's Kit (SDK), but instead choose to develop their own interface to Chaparral controllers. We generally recommend against this, especially if you are developing a CAPI app that will run on Windows NT or 2000 (since the sample code has been tested on Windows NT) or if you are developing a complex application that will use many of the commands defined in the Function Reference in Chapter 5. But if you prefer to design your own interface, this chapter provides some information that will be useful to you.

In the example below, we assume that all you want to do is monitor the health of the controller and so you just want to send the commands related to getting events. You can do this either using in-band communications (that is, using SCSI commands over parallel SCSI or Fibre Channel) or out-of-band communications (that is, RS-232 communications, also known as a serial communications).

In-band (SCSI/Fibre Channel) Communications

Even though you will not be using the Chaparral SCSI LMX, you should read Chapter 11 to understand about how to interface to Chaparral controllers via in-band communications.

The CAPI commands are passed to the controller using the SCSI Write Buffer command and replies are received with the SCSI Read Buffer command. The data that is passed with these commands always consists of at least the structure CAPI_PACKET. For some commands, extra data accompanies CAPI_PACKET. Referring to the *Write Buffer* section of Chapter 11, note that the Transfer Length that is in CDB bytes 6 through 8 is sizeof(CAPI_PACKET) (which is 80 decimal) if you are sending a command to get an event, as in the example below. This number must also be put in the CAPI_PACKET struct as member *packetLength*. If you are sending one of the other CAPI commands that requires passing data to the controller, then the size of Transfer Length = sizeof(CAPI_PACKET) + sizeof(the extra data you are passing to the controller), and that extra data must immediately follow the CAPI_PACKET struct.

Out-of-band (RS-232) Communications

Even though you will not be using either of the two Chaparral serial LMXs, you should read Chapters 12 and 13 to understand the serial communications implementations that you will have to interface with. The Simplified RS-232 LMX (Chapter 13) is much simpler, but is not supported on controllers that have a LAN processor.

The data format is the same as for in-band communications. That is, there is a CAPI_PACKET struct which, for some commands, is followed immediately by extra data.

Note that you must send a Ctrl-P before you send the first CAPI command, to switch the serial interface from MUI mode to CAPI mode, as noted at the end of Chapters 12 and 13.

Example CAPI_PACKET Usage

To determine which members of the CAPI_PACKET struct must contain data and what that data is, see the code in capi2pak.c for each of the commands that you want to use. Note especially the parameters passed to function BuildAndSendPacket, which in turn calls function BuildPacket.

For example, the values that need to go in CAPI_PACKET for the event commands are shown in the following code sample. We have pulled the following code out of the BuildPacket function in file capi2lmx.c in our SDK, then edited it to be specifically for the get-event commands. pPak is a pointer to a CAPI_PACKET struct.

```
pPak->control = 0;
pPak->byteOrder = 0;
pPak->capiVersionMajor = CAPI_VERSION_MAJOR;  /* Always 3, as of this writing */
pPak->capiVersionMinor = CAPI_VERSION_MINOR;  /* 2 for pre-RIO products,
                                                 4 for RIO and later products,
                                                 as of this writing */
pPak->requestCompressionType = CAPI_COMPRESSION_TYPE_NONE;  /* = 0 */
pPak->packetCompressionType = CAPI_COMPRESSION_TYPE_NONE;  /* = 0 */
pPak->eventOrCommand = CAPI_PACKET_TYPE_COMMAND;  /* = 0 */
pPak->signatureString[0]='C';
pPak->signatureString[1]='A';
pPak->signatureString[2]='P';
pPak->signatureString[3]='I';
pPak->includeStructType = INCLUDE_NO_STRUCTURE;  /* = 0 */
pPak->commandCode = commandCode;
pPak->identifier.controllerHandle = 0
pPak->identifier.arrayIndex = 0
pPak->identifier.channelIndex = 0
pPak->identifier.driveIndex = 0
pPak->configSequenceNumber = 0;
pPak->errorCode = 0;
pPak->param1 = param1;
pPak->param2 = 0;
pPak->param3 = 0;
pPak->param4 = 0;
pPak->packetLength = sizeof(CAPI_PACKET);  /* = 80 */
pPak->arrayListConfigSequenceNumber = 0;
pPak->uniqueId = 0;
pPak->driveListConfigSequenceNumber = 0;
```

Struct members *capiVersionMajor* and *capiVersionMinor* can be determined for the controller model(s) you are interfacing to by using the Disk Array Administrator (also known as MUI). Use Ctrl-E to get into the "CFG Info" screen, then scroll and look for "CAPI Version = ".

Struct member *requestCompressionType* should be set as shown in this example if you are doing a very simple app that only gets events. But if you are implementing a more complex management app and are using serial communications, you should set requestCompressionType to CAPI_COMPRESSION_TYPE_SIMPLE_RLE to speed up getting large structures and you will need to implement the uncompression algorithm. You can copy the uncompression algorithm from function ReceivePacket in capi2pak.c.

Struct member *commandCode* should be set to one of these for this example:
CAPI_COMMAND_GET_FIRST_EVENT = 36 (or 0x24000000 endian reversed if this is needed)
CAPI_COMMAND_GET_LAST_EVENT = 37 (or 0x25000000 endian reversed if this is needed)
CAPI_COMMAND_GET_EVENT = 38 (or 0x26000000 endian reversed if this is needed)

Struct member *param1* is the event number that you want to fetch when *commandCode* is CAPI_COMMAND_GET_EVENT and can be set to 0 for the other two *commandCode*s that get events. For information on how *param1* through *param4* are used for other commands, see the code in capi2pak.c for the commands that you are interested in implementing.

Struct member *includeStructType* is INCLUDE_NO_STRUCTURE for commands that pass no extra data to the controller, which is the case for the commands used to get events. For commands that pass data, this member must be set appropriately. For example, if *commandCode* = CAPI_COMMAND_SET_CONTROLLER_PARAMS, then you need to set this member to INCLUDE_CONTROLLER_PARAM_STRUCT and you need to include struct CAPI_CONTROLLER_PARAMS in the message, placed immediately after CAPI_PACKET. Again, see the code in capi2pak.c for the commands you are interested in implementing.

Struct member *packetLength* is sizeof(CAPI_PACKET) + sizeof(extra data), where "extra data" depends on the command. For the commands to get events, there is no extra data. But continuing the example of setting controller params from the previous paragraph, *packetLength* would be set to sizeof(CAPI_PACKET) + sizeof(CAPI_CONTROLLER_PARAMS).

If you are calling a command that is changing the configuration of the controller (typically a command with "Set" in the name), then you need to pass a valid value for the three struct members *configSequenceNumber*, *arrayListConfigSequenceNumber*, and *driveListConfigSequenceNumber*. See the discussion of configuration sequence numbers in Chapter 2 in the section titled *Controller Structure Updates* on page 9 and the two sections that follow that. Typically, your application would be designed to get all three of the key structures just before sending a set command, then it would copy each *configSequenceNumber* member of those three structures into the corresponding *configSequenceNumber*, *arrayListConfigSequenceNumber*, and *driveListConfigSequenceNumber* members of the CAPI_PACKET struct for the set command.

Struct member *errorCode* is unused when sending commands, but when you receive a reply, you should check to see if this is something other than CAPI_NO_ERROR and look at what the code means. (See Chapter 9, Error Code Reference.)

If the processor that your CAPI app is running on does not match the endian convention of an Intel processor, then you will also need to do endian reversal for the non-zero members of this struct that are larger than 8 bits: *commandCode*, *param1*, and *packetLength* in the above example. For example, *packetLength* is 80 bytes, or 0x50, so the endian reversal would give 0x50000000, so you can code it this way:

```
    pPak->packetLength = 0x50000000;  /* = 80 bytes */
```

See the table in the **Callback** section of each command in Chapter 5 to understand which members of CAPI_PACKET are important for you to look at when the reply comes back. For the commands that get events, the key items to look at in the callback table are the definitions for param1 through param3. (Note that you may not need CAPI_COMMAND_GET_FIRST_EVENT, since both the first event number and the last event number are returned when you do CAPI_COMMAND_GET_LAST_EVENT.) Since the event data comes back in a CAPI_EVENT struct, you may need to do some endian reversal again on the members of this struct, depending on the endian convention of your processor. This struct immediately follows the CAPI_PACKET struct in the received message data. The replyCode listed in the callback table is returned in the *commandCode* member of CAPI_PACKET and should always correspond to the *commandCode* sent to the controller; you can ignore this if you wish—it just provides a sanity check that the reply you are receiving is for the command that you sent. Some of the members of the *identifier* substructure are valid for some commands, as noted in the callback table; you can ignore the *controllerHandle* member of *identifier* for simple CAPI applications. The dataPtr row in the callback table should be used to determine the type of struct that contains the returned data; you can ignore the fact that it is referred to as a pointer in the callback table—the actual data always immediately follows the CAPI_PACKET struct.