Document No.: UG99999-000 REVE
Release Date: April 10, 1992

COPYRIGHT NOTICE

INTERPHASE CORPORATION
13800 Senlac
Dallas, Texas 75234
Phone: (214) 919-9000
FAX: (214) 919-9200

DISCLAIMER

Information in this user document supersedes any preliminary specification, data sheets, and/or any other documents that may have been made available. Every effort has been made to supply accurate and complete information. However, Interphase Corporation assumes no responsibility or liability for its use. In addition, Interphase Corporation reserves the right to make product improvement without prior notice.

ADEMARK ACKNOWLEDGMENTS

All terms used in this manual that are known to be trademarks or service marks are listed below. In addition, terms suspected of being trademarks have been appropriately capitalized. Use of a term in this manual should not be regarded as affecting the validity of any trademark or service mark.

- Interphase® is a registered trademark of Interphase Corporation.
- Unix® is a registered trademark of AT&T Bell Laboratories.
- Multibus I® is a registered trademark of Intel Corporation.

FOR ASSISTANCE

To place an order for an Interphase product,
or for technical assistance, call:

CUSTOMER SERVICE:

In the U.S.: (214) 919-9111
n the United Kingdom: (869) 321222

TABLE OF CONTENTS

4. RC COMMAND SET

LIST OF FIGURES

LIST OF TABLES

This page is intentionally left blank.

CHAPTER 1
INTRODUCTION

SCOPE

This manual is intended for people who are writing a device driver for any Interphase controller that uses the Common Boot and Report/Command (RC) Interface. It documents the generic aspects of the Common Boot and RC Interface. Those parts of the interface which are controller-specific (e.g. unique commands, error messages, etc.) are documented in the user's guide of the individual controller.

Readers are assumed to have extensive knowledge of the following:

- the C programming language
- development and installation of interface software (drivers)
- the operating system of the host computer

This manual's organization allows you to focus on your specific areas of interest, without giving you more information than needed.

Specifically, this manual contains guidelines on:

- Booting the controller (Common Boot)
- Issuing commands to and obtaining responses from the controller (RC Interface)

HOW TO USE THIS MANUAL

You will find it very useful to read this *Introduction* completely. It contains information that will clarify many of your questions later. The conventions section can be especially useful for later reference since it defines how certain topics will be presented to you.

Chapter 2 describes the *Common Boot Interface* and command set. This provides a set of tools for getting the RC Interface up and running.

An overview of the *RC Interface* appears in Chapter 3, followed by the *RC Command Set* in Chapter 4. Together, these chapters tell you all you need to know to issue commands to and read messages from the RC Interface.

Appendix A contains *Data Structures* for the Common Boot and RC Interface.

Interphase can supply you with an example driver[1]. If your system only requires minor modifications of this driver, then the source code provided gives a good base from which to start. If your system is radically different, then the example driver at least gives you ideas on what must functionally be contained in a driver.

CHANGES FROM PREVIOUS REVISION

This section applies to users who are familiar with the previous revision of this manual (UG99999-000,REVD). The following list summarizes how the current revision of the manual differs from the previous one. Each item in the list includes a reference to page(s) affected by the change.

1. Replace references to "Command Response Word" with "Command Status Word". Both terms refer to the same field (pages 14 - 15).

2. Mention download code feature, referring reader to controller manual for board-specific implementation (pages 17, 18, and 26).

3. Reword DIAG command to refer the reader more clearly to the board-specific diagnostic commands in the controller manual (p. 20).

4. Reword the following commands to make them bus-generic: Set DMA Burst, Set Bus Timeout, and Set Interrupt. Remove references to VMEbus in these commands, except when using VMEbus as an example. (p. 4-6, 57, and 59).

CONVENTIONS

This section details many of the writing conventions used throughout the manual. In addition, it gives

1. If you have not received an example driver, and you need one, call us. Page 4 gives phone numbers that will help you place an order.

many of the technical conventions.

- The term "channel" has a meaning which is specific to the Report/Command interface. It is defined at length in the RC chapter. Unless explicitly stated otherwise, the term "channel" refers to an RC channel.

- The term "short I/O" refers to a VMEbus-addressed block of memory in which only the lower 16 VMEbus address lines are used for transactions. The upper 16 VMEbus address lines are not used.

  - "Byte" represents 8 bits; "word" represents 16 bits (2 bytes); and "longword" represents 32 bits (2 words, 4 bytes).

NOTE: In AMD 29000 documentation, a "word" is defined to be 32 bits long. This definition is **not** used in this manual.

  - Binary (single bit) data is represented as either '1' or '0'.

  - When used in the context of a single bit of data, the term "set" means that the bit is a one ('1').

  - Similarly, the term "cleared" means that the bit is a zero ('0').
  - To represent hexadecimal numbers, the manual adopts the C language notation. Decimal numbers are shown as decimal digits. For example:

    0x29 = 29 hex
    41 = 41 decimal

  - Many commands contain bits, bytes, or words which are marked "RESERVED". Such reserved fields fall into three categories:

[1] Reserved for future use, must be cleared ('0', 0x00, or 0x0000).

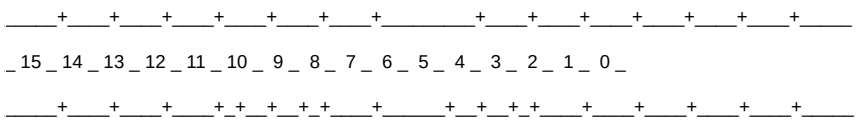
[2] Reserved for future use, do not write to this field.

[3] Reserved for a future implementation of the "pools" concept. Must be cleared ('0', 0x00, or 0x0000) under the current implementation.

In this manual, whenever a data structure contains reserved field(s) it will also contain a statement

about how the host should treat the reserved field(s). The statement will be placed immediately after the data structure.

· When showing binary representations of bytes or words, the diagrams may have many bits which do not have names. These are RESERVED. As an example:



Bits 10 and 11 are called Sample 1, bit 7 is called Sample 2, and bit 6 is called Sample 3. All other bits are RESERVED.

This page is intentionally left blank.

CHAPTER 2
COMMON BOOT

OVERVIEW

The Common Boot is a PROM-based monitor which begins to run when the controller is first powered up or reset. A general-purpose protocol, it provides a deterministic start-up sequence for resident or down-loadable host/board interfaces. It functions as a distinct entity in the controller's firmware architecture and can be ported to any controller which provides a shared memory region.

This chapter describes how to use the Common Boot interface to get the RC Interface up and running on the controller. It also details various commands that you can use when booting the controller.

POST POWER-UP AND RESET SEQUENCE

Immediately after the controller is powered up or reset, it executes its bootstrap code and performs a series of power-up diagnostics. As discussed in your controller manual, one or more on-board red/green LED(s) signal the execution of the diagnostics.

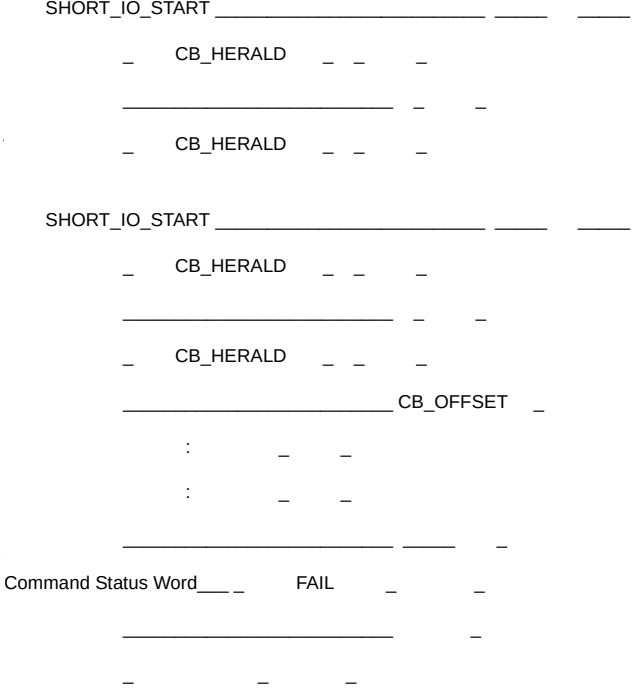
Next, the Common Boot interface fills the entire shared memory region with a 32-bit pattern known as the CB_HERALD. CB_HERALD contains information about the start of the Common Boot interface and the

|_____Byte 3_____|_____Byte 2_____|_____Byte 1_____|_____Byte 0_____|

_ 1100 1011 _ CB_OFFSET _ SHORT_IO_SIZE _

- Bytes 1 and 0 contain the size of your controller's shared memory space in bytes.
- Byte 2 specifies the offset (in longwords) at which the Common Boot interface starts in shared memory. The first longword of the Common Boot command area is referred to as the **Command Status Word**.
- Byte 3 contains the Common Boot signature, which is a hex CB (0xCB) or 203.

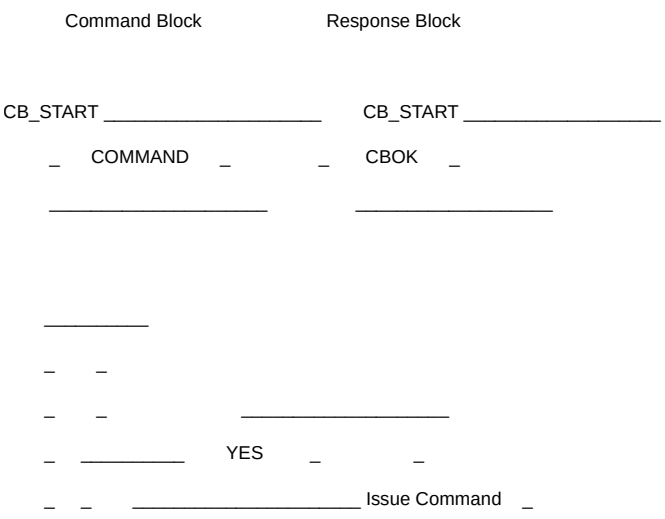
If the power-up diagnostics completed successfully, **CBOK** will be written to the Command Status Word.



If FAIL occurs, the host may issue DIAG (a Common Boot command) to identify the problem. For more information on DIAG, see p. 20 in this book. Also refer to the chapter on diagnostics in your controller manual.

NOTE: FAIL only occurs in conjunction with power-up diagnostics and the DIAG command. It is not returned in response to any other Common Boot command.

USING THE COMMON BOOT INTERFACE



Unless you need to perform diagnostics, the Common Boot is simply used to boot the RC Interface. With this scenario, the following occurs when the

controller is powered up or reset:

- The Common Boot starts up and fills shared memory with CB_HERALD.

- The host issues the BOOT command to boot the RC Interface.

The BOOT command allows you to create the initial host-to-board channel and board-to-host channel. These structures, which are discussed at length in Chapter 3, enable the host to interact with the controller using the RC Interface. Additional channels may be established once RC has booted up.

COMMON BOOT COMMANDS

Common Boot commands describe support the following functions:

_ Resident and downloadable interface boot
_ Controller LED commands
_ Controller memory peek and poke
_ Controller diagnostic commands

The following defines describe the Common Boot command set, along with the CBOK and FAIL responses:

```
#define MAKE_LONG(a,b,c,d)
        (((u32)a << 24) | ((u32)b << 16) | ((u32)c << 8) | (u32)d)
#define CB_BOOT     MAKE_LONG('B','O','O','T')
#define CB_DIAG     MAKE_LONG('D','I','A','G')
#define CB_FILL     MAKE_LONG('F','I','L','L')
#define CB_FLSH     MAKE_LONG('F','L','S','H')
#define CB_GRNL     MAKE_LONG('G','R','N','L')
#define CB_JUMP     MAKE_LONG('J','U','M','P')
#define CB_PEEK     MAKE_LONG('P','E','E','K')
#define CB_POKE     MAKE_LONG('P','O','K','E')
#define CB_REDL     MAKE_LONG('R','E','D','L')
#define CB_STUF     MAKE_LONG('S','T','U','F')
#define CB_CBOK     MAKE_LONG('C','B','O','K')
#define CB_FAIL     MAKE_LONG('F','A','I','L')
```

All of the above-listed defines are unique in both the upper and lower words. This enables the host to make both 16- and 32-bit accesses to shared memory when using Common Boot, provided that the controller supports both types of accesses.

NOTE: The V/FDDI 4211 Peregrine and Multibus® I 2211 controllers only support 16-bit accesses to shared memory. For details on accessing the shared memory on your specific board, consult the user's guide for your controller.

A description of the Common Boot command set appears at the end of this chapter, starting on p. 18.

DOWNLOADING CODE

A download mechanism is provided on Interphase controllers which support the Common Boot interface. This feature allows the host to download code to controller firmware for execution. Applications of the feature include:

- booting an interface other than the native RC Interface as documented in Chapter 3
- making firmware upgrades in the field

Interphase provides utilities which automate the download process. These utilities are tied to specific controller types (e.g. 4211, 4207, etc.) and hardware/firmware revision levels. They typically use the Common Boot commands "POKE" and "BOOT" to download and execute the code. To determine whether your controller supports this feature, consult the user's guide for your controller.

16 + (Nx8) :	ALLOCATE BOARD-TO-HOST	: (N = # segments)
:	SEGMENT MEMORY (RC COMMAND)	:

Function:Boot a native or downloaded interface. This section describes the format of the BOOT command when booting the controller's native Report/Command (RC) Interface (i.e. the default interface stored in EPROM.) For information on booting a downloaded interface, see "Downloading Code", p. 17.

If you are booting the native RC Interface, the BOOT command includes three RC commands:

- Allocate Board-to-Host Segment Memory
- Create Board-to-Host Channel
- Create Host-to-Board Channel

These commands establish the initial pair of RC channels needed to issue commands to and obtain responses from the controller. See Chapter 3 for a detailed discussion of segments, channels, and related topics.

Code: 0x424F4F54 (ASCII "BOOT")

Parameters: INDEX

Boot Entry Point Index (0,1,2,...).
With a supplied 0, the native RC Interface will boot.

TOTAL LENGTH OF REMAINING PARAMETERS
Specifies the aggregate length, in bytes, of all RC commands which the controller may expect to find in shared memory following the Total Length field in this command

ALLOCATE BOARD-TO-HOST SEGMENT MEMORY
Allocates host-resident channel segment memory to the board. See p. 65 for a command description.

The command must be executed ***before*** the two channel creation commands. It must therefore precede these two commands in the BOOT command structure, as shown in . This is necessary

because the controller requires at least two board-to-host segments to create a board-to-host channel.

CREATE BOARD-TO-HOST CHANNEL

Establishes the initial board-to-host channel. The position of the descriptor for the created channel is written to a fixed location in shared memory.[1] This is an RC command; see p. 63 for a command description.

CREATE HOST-TO-BOARD CHANNEL

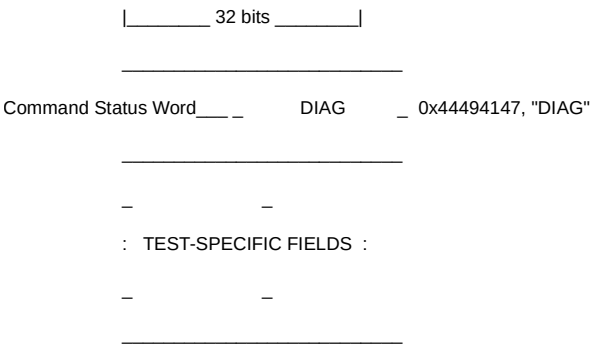
Establishes the initial host-to-board channel. The position of the descriptor for the created channel is written to a fixed location in shared memory. This is an RC command; see p. 61 for a command description.

Programming Sequence: If CBOK, supply the parameters, then supply the BOOT command. When CBOK occurs or the booted interface responds, the Common Boot Interface "goes away" until the host performs a hardware reset on the board.

As discussed in Chapter 4 (p. 61), the Create Channel directives normally return to the host the logical address of the first segment for the new channel. This is not possible at boot time since the channel for response is not yet established. Therefore, the controller will use the first segment in the list provided by the Allocate Board-to-Host Segment Memory command as the initial board-to-host channel segment.

1. The offset of the initial channel descriptors is shown in the shared memory map on p. 39.

Command Block:



Function: Execute Diagnostics. This command performs a set of tests whose definition is specific to your controller. For a description of how the fields are defined, consult the chapter on Diagnostics in the user's guide for your controller.

Code: 0x44494147 (ASCII "DIAG")

Parameters: TEST-SPECIFIC FIELDS One or more fields which specify such values as the test ID and data to be used in the test. For details, consult the chapter on Diagnostics in the user's guide for your controller.

Programming Sequence: If CBOK, supply the parameters, then supply the DIAG command. If CBOK occurs, the test completed successfully.

If FAIL occurs, the test failed. The RETURNED DATA field will contain information regarding the problem. Refer to the appropriate test description in your controller manual for details.

_ BOARD DESTINATION _ Board fill start address

3 _____

_ COUNT _ Number of units to fill

Function:Fill area with pattern

Code: 0x46494C4C (ASCII "FILL")

Parameters: PATTERN The 32-bit pattern used for filling memory.

BOARD DESTINATION The starting board address for pattern filling.

COUNT The number of units to fill with pattern.

UNIT SIZE The size of unit (1=8, 2=16, or 4=32 bits).

Programming Sequence:If CBOK, supply the parameters, then supply the FILL command. When CBOK occurs, the pattern has been successfully laid down.

Command Status Word___ _ FLSH _ 0x464C5348, "FLSH"

1 _____

_ CYCLE COUNT _ 1 - N; N an integer

Function:Flash the LED(s).

Code: 0x464C5348 (ASCII "FLSH")

Parameters: CYCLE COUNT The number of times an LED red -> green transition occurs.

Programming Sequence:If CBOK, then supply the flash cycle parameter, then supply the FLSH command.
When CBOK occurs, FLSH is finished.

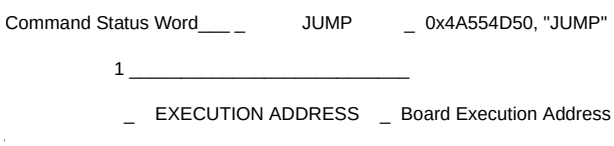
Command Status Word___ _ GRNL _ 0x47524E4C, "GRNL"

Function:Turn on the green LED (if present on controller)

Code: 0x47524E4C (ASCII "GRNL")

Parameters: None required

Programming Sequence:If CBOK, supply the GRNL command. When CBOK occurs or the LED turns green, GRNL is finished.



Function:Transfer execution to address.

Code: 0x4A554D50 (ASCII "JUMP")

Parameters: EXECUTION ADDRESS Board execution address.

Programming Sequence:If CBOK, supply the target execution address, then supply the JUMP command.
When CBOK occurs or an indication of execution transfer occurs,
JUMP is finished.

UNIT SIZE _ Size in bytes of unit (1, 2, or 4)

Function:Examine controller memory.

Code: 0x5045454B (ASCII "PEEK")

Parameters: SOURCE ADDRESS Board memory address to read.

UNIT COUNT Number of units to read.

UNIT SIZE Size in bytes of units (1, 2, or 4).

Programming Sequence:If CBOK, supply the parameters, then supply the PEEK command. When CBOK occurs, PEEK is finished and memory will be dumped at the location immediately following CB_START (i.e. CB_START + 4).

_ UNIT SIZE _ Size of data transfers to SRAM (1, 2, or 4)

4 _____

_ DATA [0] _ Data to write.

Function:Modify controller memory. For an application of this command, see "Downloading Code", p. 17.

Code: 0x504F4B45 (ASCII "POKE")

Parameters: DESTINATION ADDRESS Where data is to be written.

UNIT COUNT Number of bytes to write.

UNIT SIZE Width of data transfer from shared memory to static RAM.

1 = 8-bit transfers
2 = 16-bit transfers
4 = 32-bit transfers

Legal values in this field depend on the controller's memory organization. For example, the V/FDDI 4211 Peregrine only supports the values 2 or 4 in this field.

Programming Sequence:If CBOK, supply the parameters, then supply the POKE command. The data to be poked immediately follows the UNIT SIZE field. When CBOK occurs, POKE is finished and written memory exists on the controller at the destination address.

Command Status Word___ _ REDL _ 0x5245444C, "REDL"

Function:Turn on the red LED (if present on controller)

Code: 0x5245444C (ASCII "REDL")

Parameters: None

Programming Sequence:If CBOK, then supply the REDL command. When CBOK occurs or the LED turns red, REDL is finished.

_ UNIT SIZE _ Size in bytes of unit (1, 2, or 4)

4 _____

_ DATA [0] _ Data to write.

5 _____

Function:Modify controller memory by continually stuffing a non-incrementing destination address.

Code: 0x53545546 (ASCII "STUF")

Parameters: DESTINATION ADDRESS Where data is written. This address does not increment.

UNIT COUNT Number of units to write.

UNIT SIZE Size in bytes of units (1, 2, or 4).

Programming Sequence:If CBOK, supply the parameters and then supply the STUF command. The data to be stuffed immediately follows the UNIT SIZE field. When CBOK occurs, STUF is finished and the specified data exists on the controller at the destination address.

CHAPTER 3
USING THE REPORT/COMMAND INTERFACE

SYSTEM REQUIREMENTS

In order to drive a controller which incorporates the RC Interface, the host system must meet the following requirements:

_ The byte ordering of commands must be "big-endian" (i.e. data is read/written Most Significant Byte first)

_ The host must have at least 16-bit access into the controller's onboard shared memory ("short I/O") space. In addition, it must be able to read or write a 16-bit quantity in one operation without leaving an access window between bytes.

_ At least 512 bytes of VMEbus short I/O address space must be available on the system VMEbus for use by the RC Interface. (For the requirements of non-VME controllers which support the RC Interface, consult the user's guide for the controller.)

In addition, there may be other requirements which are specific to your controller. For example, the V/FDDI 4211 Peregrine only supports 16-bit accesses to shared memory. For details, consult the user's guide for your controller.

OVERVIEW

The Report/Command (RC) Interface is an asynchronous mechanism for queuing commands and responses between the host and controller. It is designed to minimize the overhead involved in issuing commands to and reading responses from the controller.

Most of the interaction between the host and controller takes place via **channels**. An RC channel is a data pathway through which message traffic can be moved from one subsystem to another. It is not a piece of hardware. It is, rather, a set of techniques or methods whereby host-based software and controller firmware can make efficient use of hardware devices for the purpose of communicating with each other.

A Note on RC-Specific Terminology

The term "channel" has a specific definition in the RC Interface. Other terms with an RC-specific meaning include: segment, channel descriptor, read and write pointer, and the four RC command types (request, directive, response, and indication). All of these terms are defined in this chapter, except for the four command types. The latter are defined in Chapter 4.

Terms which are RC-specific are printed in **bold** the first time they appear in this text. Later sections assume that the reader is familiar with these terms as defined in the text.

DEFINITION OF RC CHANNELS

In the context of the RC Interface, a channel consists of one or more blocks of memory used for host-to-board or board-to-host communication. Commands are written to the board through a structure referred to as a **host-to-board channel**. If the controller needs to report command completion or other statuses to the host, it does so through another structure called a **board-to-host channel**.

The host can create multiple host-to-board and board-to-host channels. It determines any associations between these channels. For example, the host can create a board-to-host channel and instruct the controller to use it only when responding to commands issued through a specific host-to-board channel. Or, if desired, multiple host-to-board channels can issue commands to which the controller responds via a single board-to-host channel.

GENERAL FORMAT OF RC CHANNELS

A channel is composed of two or more **segments**, where a segment is defined to be a block of contiguous memory up to 65,534 (64K - 1) bytes in length. These segments may be discontiguous with each other. A link command placed at or near the end of each segment provides information which links segments together (see the Link Segment directive and indication, pp. 64 and 74).

The linking of the segments is done by the owner ("writer") of the channel. The host owns host-to-board channels, and the controller owns board-to-host channels. The non-owner ("reader") of the channel follows the command list through a segment until it encounters the link command. It then spans to the next segment in the channel.

An RC channel may be thought of as being infinite in size, because the segments may be linked such that there is no return to the starting point. In practice, it is likely that segments will be reused and thus chained together to form a logical ring.

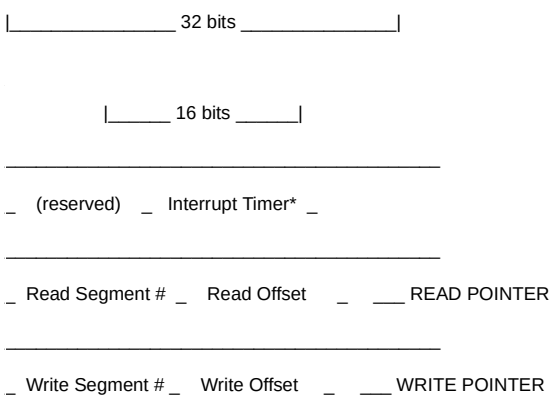
Theoretically, the simplest case of an RC channel would be a single segment which links to itself. In practice, however, this setup is impractical due to the overhead associated with bounds checking. Thus, the minimum configuration for a channel is two segments which link to each other.

The RC Interface requires at least two channels in order to function _ one for host-to-board communication and the other for the board-to-host path. As discussed later in the chapter (p. 38), the host establishes these two initial channels when it initializes the RC Interface. Once RC boots, the host may create additional channels.

The maximum number of channels is determined by how much onboard shared memory space can be dedicated to holding "channel descriptors". These structures are described in the next section.

CHANNEL DESCRIPTORS

For each RC channel the host creates, the controller will write a **channel descriptor** into its shared memory space. The channel descriptors begin at an offset of +16 bytes from the start of shared memory. The first and second channel descriptors respectively belong to the host-to-board and board-to-host channels



Each channel descriptor contains a read and write pointer. The owner of the channel uses the **write pointer** to keep track of its place in the channel's segments. The non-owner uses the **read pointer** to mark its location in the channel. Thus, the owner must update the write pointer, while the non-owner updates the read pointer.

For example, when the host issues one or more commands, it updates the write pointer in the channel descriptor of the host-to-board channel that will pass the command(s) to the controller. The controller updates the read pointer once it has extracted the command(s) from the channel and queued them for execution. This is discussed at length in the section "Issuing Commands to the Controller" (p. 40).

Read and write pointers are composed of two 16-bit quantities _ a segment number and an offset. The segment number is used to keep track of which segment is currently being accessed. The offset points to the next memory location in the segment that will be written/read.

The offset field of a pointer is set to zero at the beginning of each segment. As the owner and non-owner work their way through the segment (one providing information and the other consuming information), each updates its offset to point past the data it has just processed. When each encounters the end of the segment (set using the Link Segment directive and indication, pages 64 and 74), it links to the new segment. This is done by writing a special value (0xFFFF) to the offset field and then incrementing the segment number. For a detailed discussion of segment spanning, see p. 42.

UPDATING READ AND WRITE POINTERS

As discussed previously, a channel descriptor contains both a read and write pointer. These pointers in turn consist of a segment number and offset which identify where the next read or write operation is to take place.

This section discusses the procedure for updating the read and write pointers. It focusses on what the host needs to do to keep track of where it is in a channel. **If the driver software properly creates and maintains the channels, the controller will update its read and write pointers automatically.**

Reading a Pointer

Since pointers change as one segment links to another, the host must take steps to ensure that it is recording a valid pointer. Of particular interest is the write pointer in a board-to-host channel descriptor. When such a pointer contains a different value than the read pointer in the same descriptor, it means that the board has information to pass back to the host.

Any algorithm for reading a pointer must check to see if segment spanning is underway. The owner of a pointer writes 0xFFFF into its offset field when it is linking one segment to another. The following procedure may be used to read a pointer:

1. Read the segment number.
2. Read the offset.
3. If the offset is 0xFFFF, go to Step 1.
4. Read the segment number again.
5. If the segment numbers have changed, go to Step 1.

6. If the segment numbers are identical, record the valid pointer.

The above algorithm is depicted in .

Updating the Offset of a Pointer

No special handshaking or protocol is required to update an offset, except when the segment number of the pointer is also being updated. The entity responsible for maintaining the pointer (host or controller) may increment the offset at any time. Any offset from 0x0 to 0xFFFE may be written to the offset field.

The value 0xFFFF cannot be used as an offset. This is because it indicates that the pointer's segment number is about to be changed, as described in the next section.

Updating a Segment Number

A segment number is updated only when the entity responsible for maintaining the pointer (host or controller) needs to span from one segment to another. When this occurs, both the offset and segment number are updated. The procedure for doing so is as follows:

1. Write the value 0xFFFF into the offset field.

2. Increment the segment number field by 1.

3. Write a valid offset (0 or some other positive integer less then 0xFFFF) into the offset field.

The use of segment numbers and offsets is discussed at length throughout the remainder of this chapter. As noted previously, the controller will automatically update its read/write pointers if the driver software properly maintains the read/write pointers belonging to the host.

MANAGING SEGMENT MEMORY

Segments typically reside in host memory. The host must set up two segment "pools" _ one for host-to-board channels and the other for board-to-host channels. The maximum size of a segment is 65,534 bytes.[1] The host may choose to match segment sizes to system page sizes for efficient memory utilization.

1. This limitation is due to the read and write pointers which RC uses to represent physical addresses within a segment. Since these pointers are 16-bit fields, the maximum legal offset is 0xFFFE. (As discussed previously in this chapter, 0xFFFF cannot be used as an offset since it indicates that the segment number is being updated.)

A segment should not link back to itself. Therefore, the host must allow sufficient memory to support ***at least two*** segments for ***each*** channel to be created. The controller will enforce this rule for the board-to-host segments (but not the host-to-board segments). It does this when it processes requests to create new board-to-host channels. If there is insufficient segment memory available to the controller when it encounters such a request, it will respond with an error message.

Segment Ownership

Segments are owned and managed by the "writer" of the channel. This means that the host is responsible for tracking and reusing the host-to-board segments. Likewise, the board-to-host segments are tracked and reused only by the controller.

Host-Owned Segment Memory

The host, as it writes commands to a host-to-board channel and fills up segments, links to new or previously allocated segments. Allocation of these segments is completely under the control of the host. For example, different-sized segments can be used in the same channel.

The only restrictions on host-owned segment memory are as follows:

1) All segments must begin on a longword boundary.

2) The maximum segment size is 65,534 (0xFFFE).

3) All segments must be I/O mapped in a manner that permits the controller to perform DMA operations on them.

Controller-Owned Segment Memory

The host must dedicate memory for sole use by the controller as board-to-host segments. The controller independently manages its segments, reusing them when the host has extracted the responses they contain.

To inform the controller about what memory is assigned to it, the host issues an Allocate Board-to-Host Segment Memory directive (p. 65). This is done at RC boot time, so that the board-to-host segment memory is in place after RC initialization. The directive may

also be reissued after booting RC, if necessary. For more information, see "Initializing the RC Interface," p. 38.

Unlike segments in host-to-board channels, all board-to-host channel segments are same-sized. The host sets this size when it allocates memory to the controller.

Assigning Segment Numbers

The RC Interface has some important rules concerning the "Segment #" fields in the read and write pointer. These rules are:

- Segment numbers are assigned on a per-channel basis (that is, the segment numbers associated with one channel are not affected by those of another).

- Segment numbers start at 0 and are incremented by 1 until they exceed 0xFFFF. At that point, they roll over to 0 and start incrementing again.

- When the writer of the channel links to a new segment, it increments the "Write Segment #" field. When the reader of the channel links to the segment, it increments the "Read Segment #" field.

It is quite possible for the writer of a channel to be several segments ahead of the reader. Segment numbers provide a way for the writer to determine where the reader is in the channel. They thus enable the writer to reclaim segments after the reader has finished extracting information from them (see "Reusing Segments", p. 37).
The writer of a channel "owns" the segments of that channel. It must therefore keep track of what segment number (if any) is currently associated with a given segment in physical memory.

The reader, on the other hand, cannot use segment numbers to find a segment in memory. It only uses segment numbers to determine whether the writer is still pointing to the same segment which it (the reader) is accessing. This information affects how the reader will process any remaining information in the segment, as will be discussed later in the chapter.

Location of Segments

When looking for a segment in host memory, the reader of a channel is faced with one of three scenarios:

1) **At RC bootup time**, it must find the first segment in the initial channel that it is supposed to

read.

When the host boots the RC Interface, it creates an initial host-to-board and board-to-host channel. The location of the first segment in these initial channels is as follows:

- The first segment in the initial host-to-board channel is specified in the command used to create the channel.

- The first segment in the initial board-to-host channel is, by convention, the first item in the list of memory resources allocated to the controller by the host.

See "Initializing the RC Interface" (p. 38) for more information.

2) **When accessing a newly created channel**, the reader must be able to find the first segment in the channel.

This depends on whether the channel is a host-to-board or board-to-host channel. For host-to-board channels, the location of the first segment is specified in the channel creation command (see Create Host-to-Board Channel, p. 61). For board-to-host channels, is returned by the controller in response to the channel creation command (see Report New Board-to-Host Channel, p. 72).

3) **When linking to a new segment in a channel**, the reader must be able to find the next segment.

Each time the writer of the channel links to a new segment, it places a link command at the end of the old segment. The link command contains the location of the next segment in the channel.

Reusing Segments

Once the reader of a channel has linked from one segment to the next, the writer can reuse the old segment to pass new information to the reader. This section describes what the host needs to do to reuse its segments. The controller follows a similar procedure without host intervention.

Before the host can reuse a segment, it must make sure that the controller has extracted all the

commands from the segment. The controller indicates that it is finished with a segment when it updates the "Read Segment #" field in the channel descriptor. It updates this field after executing the Link Host-to-Board Segment directive placed by the host at the end of the segment.

The procedure for identifying such freed-up segments is relatively straightforward, provided that you adhere to the convention that segment numbers always increment. One method is for the host to keep a table of the segment numbers currently associated with memory being used to pass commands to the controller. Data for the table would extracted from the host-to-board channel descriptors in shared memory. For each host-to-board channel, the host can assume that a segment is reusable if its segment number is less then the current value of the channel's "Read Segment #" field.

Rollover of Segment Numbers to 0

Segment numbers are 16-bit quantities and eventually roll over to 0. In general, the reader of the channel need not concern itself with this event. It simply increments the "Read Segment #" field as necessary.

On the other hand, the writer of a channel needs to know when the "Read Segment #" and "Write Segment #" fields roll over in the channel descriptor. Otherwise, after the "Write Segment #" rolls over, there will be an unnecessary delay before some of the segments in the channel can be reused. This is because all segments which have not been processed when the Write Segment # rolls over will be associated with large segment numbers.

Since the controller manages its own segments, the host only needs to check for segment number rollover in host-to-board segments. One method is to allocate an array of segments whose total quantity divides evenly into 0xFFFF, and then use this array as a circular queue. To find the array index of the segment currently being read, apply the current segment number in the read pointer as a modulus against the total number of segments. This procedure would be repeated each time you link to another segment in the array. If the indices were not equal, the candidate link segment in the array would be assumed to be free.

INITIALIZING THE RC INTERFACE

The RC Interface is initialized via the Common Boot command "BOOT" (see p. 18). This command informs RC of the host virtual address of shared memory, thus enabling the controller to convert host-referenced addresses. BOOT requires the following three RC commands as parameters:

1. Allocate Board-to-Host Segment Memory (see p. 65 for command description)
2. Create Board-to-Host Channel (" p. 63"" ")
3. Create Host-to-Board Channel (" p. 61"" ")

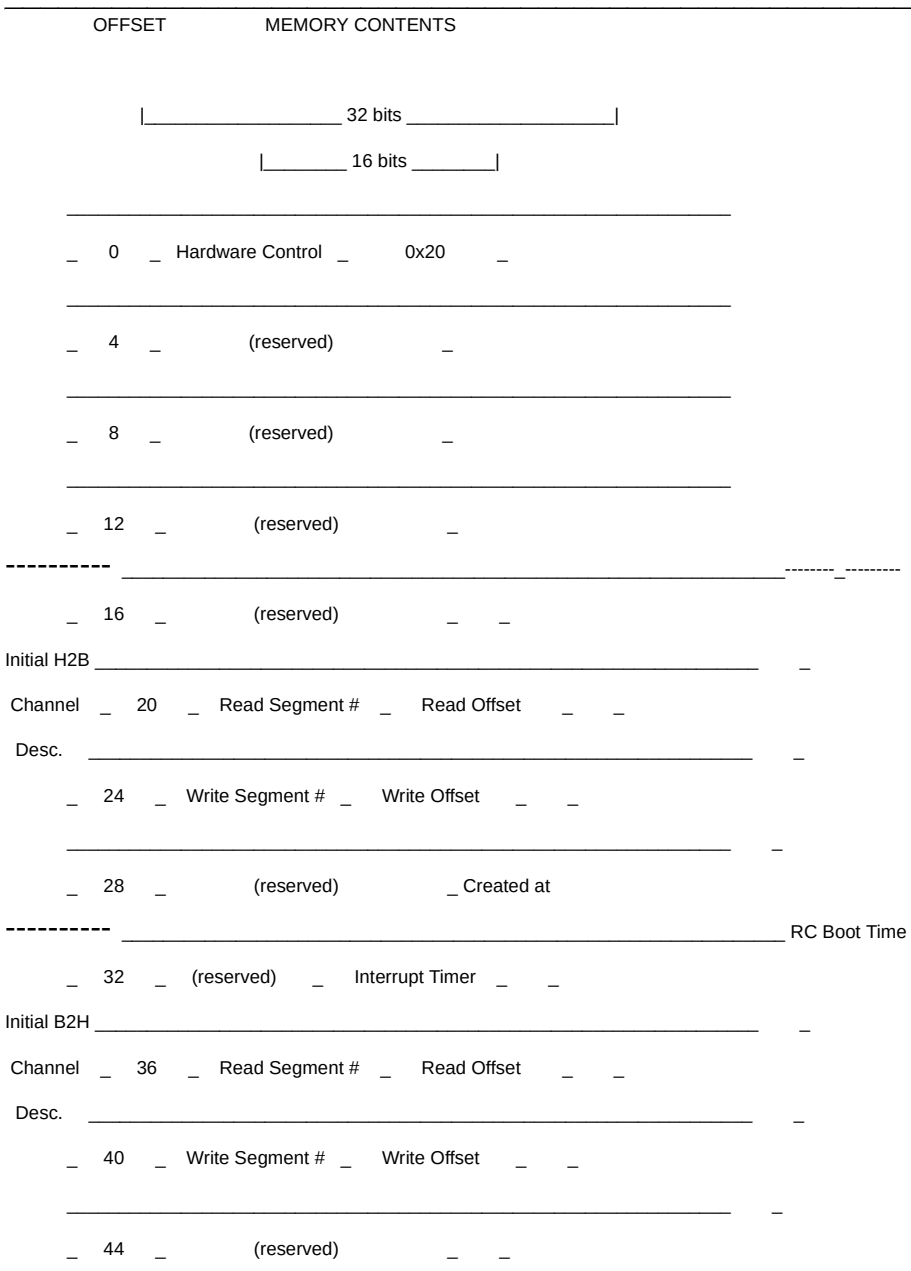
The first command provides the controller with a list of segments for use in board-to-host channels. The latter two commands submit the characteristics of two RC channels _ a host-to-board and a board-to-host channel. This initial pair of channels can then be used to create additional channels as necessary. Both channels must be in place when RC boots up, because the host must have a path to receive a response from any additional channel creation commands it may issue.

There is an important convention to note concerning the BOOT command. When the controller executes a Create Board-to-Host Channel directive, it normally returns to the host the logical address of the first segment for the new channel. This is not possible at boot time since the channel for response is not yet established. Therefore, after RC boots up, the controller examines the list of segments provided in the Allocate Board-to-Host Segment Memory directive. It then uses the first segment in the list as the **initial segment** of the **first board-to-host channel**.

STRUCTURE OF RC INTERFACE IN SHARED MEMORY

RC channel descriptors are located in the controller's shared memory space, starting at an offset of +16 bytes. The first word in shared memory is typically used for board reset purposes. (As noted previously, the RC Interface ***does not*** include a means for resetting the controller. The controller must have some reset mechanism of its own. All controllers from Interphase Corporation that use the RC Interface incorporate such a mechanism. Consult your controller manual for details.)

The RC control structure is aligned on 32-bit longwords. The shows how this space is organized.



COMMANDS AND RESPONSES

Once RC has booted up, the host can issue commands to the controller through the initial host-to-board channel and read controller-generated data from the board-to-host channel. See Chapter 4 for a description of the commands in the generic RC command set.

Issuing Commands to the Controller

To issue one or more commands to a host-to-board channel, the host should:

1. **Examine the appropriate channel descriptor**

This refers to the channel descriptor of the host-to-board channel that will be used to issue the command. Two fields in the channel descriptor are of particular interest: 1) the Write Segment # (which identifies the segment that is currently being used by this channel to pass commands to the board), and 2) the Write Offset (which specifies the offset into the segment at which the next command should be written).

2. **Make sure the current segment has enough available memory for the command(s).**

A command cannot be split between two segments. Moreover, there must always be enough space in the segment to issue the Link Host-to-Board Segment directive. Otherwise, the controller will not be able to find the next segment in the channel.

To determine whether the command(s) fit into this segment, compute the following values:

Required Space = [Total length of command(s) in bytes] + 20 bytes[1]

Available Space = [Total length of segment in bytes] __ [Current value of Write Offset field]

The command(s) can be written to this segment if the Available Space exceeds the Required Space. If not, the host must determine how many of the commands can fit into the current segment, including the 20-byte link directive. The "excess" commands should be written to the next segment.

3. **Issue the command(s).**

Write the command(s) into the segment, starting at the offset indicated by the write pointer. Link to a new segment if necessary.

NOTE: If linking to a new segment, update the Write Pointer *before* issuing the link directive. Refer to the next step and also to the section "Linking from One Segment to the Next" (p. 42).

4. **Update the Write Pointer.**

If not linking to a new segment, increment the Write Offset by the number of bytes in the issued command(s). Make sure that the Write Offset points **past** the last command to the place where the **next** command will be written.

1. The length of the Link Host-to-Board Segment directive.

If linking to a new segment, update the pointer and issue the link directive as follows:

a) Write the value 0xFFFF into the Write Offset field.

b) Write a Link Host-to-Board Segment directive into the space remaining in the current segment.

c) Increment the Write Segment # field by 1.

d) Write any additional commands into the new segment.

e) Update the Write Offset to point past the command(s) in the new segment.

In addition to queuing up commands in a single channel, the host can also issue multiple commands to the board at the same time. This is done by writing each command to a different host-to-board channel, and then updating write pointers in the corresponding channel descriptors.

The controller polls the host-to-board channel descriptors starting at the low end of shared memory. When the board detects that the read pointer in a host-to-board channel descriptor is not the same as the write pointer, it responds by:

1) Identifying the memory that contains the new command(s). If the Write Segment # is still the same as the Read Segment #, this is simply a matter of subtracting the Read Offset from the Write Offset.

2) DMA'ing the command(s) into on-board memory for processing. When processing the commands, the controller uses the "Command Length" field in each command to determine where one command ends and another begins.

   If the commands span between segments, the board will fetch all the remaining bytes in this segment (since segments are assumed to be discontiguous). It then uses the link directive at the end of the segment to locate the next segment. When linking to the new segment, it increments the Read Segment # and resets the Read Offset. It will continue to DMA entire segments onto the controller until the read and write segment numbers match.

3) Updating the Read Pointer. Once the controller has processed the command(s) in the channel, it updates the Read Offset to point past the last processed command.

shows an overview of what takes place in memory when the host issues a command.

_ _1020

_____

end ____ _1024

_____

Linking from One Segment to the Next

When issuing commands, the host should link to a new segment whenever the number of bytes between the current Write Offset and the end of the segment is less than:

(length of proposed next command in bytes) plus 20 bytes[1]

The procedure to link one host-to-board segment to another is as follows:

1. Obtain memory in the host system for the new segment. This is typically done by: 1) finding previously used segments to reuse, 2) searching preallocated memory tables in the driver, or 3) allocating new memory from the operating system.

2. Write 0xFFFF into the Write Offset field of the channel descriptor. This indicates that channel spanning is taking place.

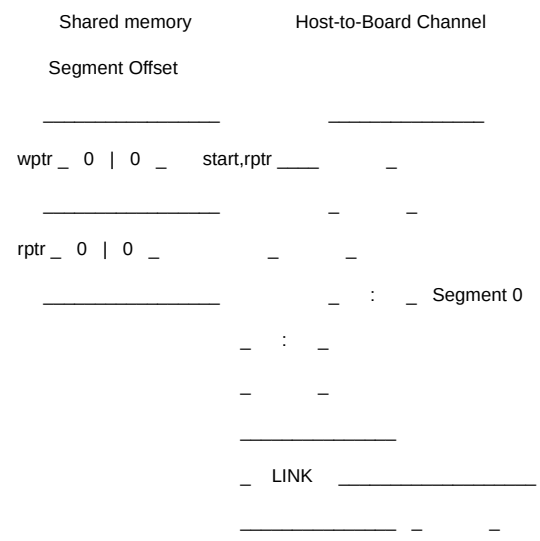
3. Write a Link Host-to-Board Segment directive into the space remaining in the current segment. This tells the controller where the new segment is located.

4. Increment the segment number.

5. Reset the Write Offset to 0 (or, if the host has already written commands in the new segment, some other valid positive number which reflects the host's write position in the new segment).

depicts linked segments.

1. The length of the Link Host-to-Board Segment directive.



Reading Command Responses

RC communicates the following types of information via the board-to-host channel(s):

- Responses to host-supplied RC commands

- Asynchronous indications concerning board status, transmitted/received data, etc.

Since there may be multiple board-to-host channels, there is a convention for determining which channel will be used to convey a particular response or indication.

- *Responses* are posted to the board-to-host channel identified in the command. If no channel is identified, the controller will use the default response channel. The default response channel is the initial board-to-host channel created at RC boot time.

- Most *Indications* are automatically posted to the default response channel. The exception is Report Link to Next Segment, which the controller places in a given channel to inform the host that it is time to span to another segment.

The controller passes messages to the host the same way that the host issues commands to the controller. That is, it writes its next message at the physical address indicated by the write pointer and then updates the write pointer to point just past the newly written message.

To locate pending responses in a board-to-host channel, the host compares its read pointer (which it man-

ages) to the write pointer (which the board manages). Whenever these are unequal, the channel contains unprocessed messages. <<lift text from section on issuing commands, rework from a host standpoint>>

When a segment becomes full, the controller spans to a new segment. It informs the host by:

1. Writing 0xFFFF into the Write Offset field of the channel descriptor. This tells the host that segment spanning is taking place.

3. Writing a Report Link to Next Segment indication into the space remaining in the current segment. This tells the host where the new segment is located.

4. Incrementing the segment number.

5. Resetting the Write Offset to 0 (or, if it has already written messages in the new segment, some other valid positive number which reflects its write position in the new segment).

When the segment and offset numbers of the read pointer match those of the write pointer, the host may assume that it has read all of the pending messages in that channel.

The controller writes to its board-to-host channel(s) asynchronously. New reports may come to the host from the controller at any time, and the host does not engage in per-message handshaking. It merely processes the messages between the read pointer and write pointer of a given channel, and then updates the read pointer. You can arrange for the host to be interrupted periodically by a given board-to-host channel. In addition, the host can poll the channel(s) for new messages. These topics are discussed in the section on interrupts, below.

Note that segment memory is allocated to the controller as a pool of buffers (using the Allocate Board-to-Host Segment Memory directive, p. 65). These segments are not assigned to any particular channel. If the controller runs out of available segments, it queues its messages internally until the host frees up some of the segments by reading the pending messages.

INTERRUPTS

Enabling Interrupts

After creating a board-to-host channel, the host can enable interrupts for the channel using the Set Interrupt directive (p. 59). Interrupts are enabled on a per-channel basis.

If interrupts are enabled for a given channel, it will cause the controller to interrupt the host when either of the following conditions is met:

1. The controller has just placed a response in a channel which previously had no unprocessed responses ***and*** whose Interrupt Timer is zero.

2. A channel's Interrupt Timer field decrements to zero (times out), and the controller determines that there are unprocessed responses in the channel.

The Interrupt Timer field is located in the channel descriptor of each board-to-host channel. As discussed in the following sections, it serves several purposes. These include suspending interrupts from the associated channel, re-enabling them, and fine-tuning how often the channel interrupts the host. For more information, see "Using the Interrupt Timer", p. 46.

Generating an Interrupt

To generate an interrupt, the controller:

1. Interrupts the host.

2. Writes the special value 0xFFFF into the Interrupt Timer field of the channel which caused the interrupt. This suspends further interrupts from the channel until the host resets the timer.

Servicing Interrupts

To service interrupts:

1. The host acknowledges each interrupt as it occurs on a hardware level.

2. As soon as convenient, the host polls the board-to-host channel descriptors to determine which Interrupt Timer field(s) contain 0xFFFF.

3. For each board-to-host channel containing 0xFFFF in its Interrupt Timer field, the host:

_ Processes all pending responses in the channel, updating the read pointer when done (or whenever it links to a new segment, if the responses are written to more than one segment). See "Reading Command Responses", p. 43, for details.

_ Resets the channel's Interrupt Timer field to any value from 0 and 0xFFFE. This re-enables the countdown-to-interrupt function for that channel.

4. If time allows, the host may check other board-to-host channels for pending responses. If there are, and the host chooses to service them, it should temporarily suspend interrupts from the channel by writing 0xFFFF to its Interrupt Timer field. For more information, see "Channel Polling (with Interrupts Suspended)" on p. 47.

NOTES: [1]The host should only write 0xFFFF to the Interrupt Timer field when it wishes to suspend interrupts from a channel whose timer has ***not*** expired. If the Interrupt Timer field already contains 0xFFFF, interrupts are suspended and the host should not rewrite this value.

[2] On VMEbus controllers, it may not be necessary to poll the channels to determine the source of the interrupt(s). If the host assigns separate interrupt vectors to each board-to-host channel (using the Set Interrupt directive, p. 59), it is possible for the interrupt service routine to already "know" the location of the channel descriptor associated with a given interrupt.

[3] Unlike VMEbus boards, the Multibus I 2211 controller uses autovectored interrupts which are not automatically cleared by servicing the interrupt. To clear an interrupt on the 2211, read the first word of its shared memory. The value contained in the word is meaningless; simply reading the word will clear the interrupt.

Using the Interrupt Timer

Each board-to-host channel has a 16-bit Interrupt Timer field as part of its channel descriptor. The host can dynamically modify this timer to meter its interrupt load on a per-channel basis.

The host sets the timer by writing any value from 0 through 0xFFFE into the field. The controller interprets this value as units of 100 microseconds. For example, a value of 7 corresponds to 700 µsec. of time. Once the specified amount of time expires, the controller will interrupt the host if (or as soon as) the channel contains pending messages.

Writing 0xFFFF to the Interrupt Timer field causes interrupts to be suspended from the associated channel. This value is used by both the controller and the host, as follows:

- After interrupting the host, the controller writes 0xFFFF to the timer of the channel which caused the interrupt. This suspends further interrupts from the channel until the interrupt service routine resets the timer.

- Before polling a board-to-host channel for pending messages, the host may write 0xFFFF to its

timer field to suspend interrupts from the channel. It then resets the timer once the polling is completed. See "Channel Polling", below, for more information.

Note that the Interrupt Timer field is irrelevant in descriptors for host-to-board channels since they are polled instead of interrupt-driven. It is only meaningful in board-to-host channel descriptors.

Channel Polling

If the host can spare time to poll the board-to-host channels, it may be able to reduce the number of controller-generated interrupts. A board-to-host channel has pending messages whenever the write pointer in its channel descriptor differs from the read pointer. The host may extract these messages and update the read pointer as described in "Reading Command Responses," p. 43.

In general, the host suspends interrupts from a board-to-host channel before reading response(s) it has discovered in the channel. However, this is not required for all systems. Both cases are discussed in the following two subsections.

Channel Polling (with Interrupts Suspended)

The host may suspend interrupts from a board-to-host channel before processing messages that it has discovered by polling the channel. To do so, it writes 0xFFFF into the channel's Interrupt Timer field. Once the controller sees this value, it suspends interrupts from the channel until the host resets the timer to a value from 0x0 _ 0xFFFE.

NOTE: If the Interrupt Timing field already contains 0xFFFF, as is the case after the channel has produced an interrupt, interrupts are already suspended and the host should ***not*** rewrite the value.

Be aware that there is a small "timing window" during which the 0xFFFF can be inadvertently overwritten by the controller. The sequence of events might be: 1) The controller reads the Interrupt Timer field to decrement it and determine if it has reached zero. 2) The host writes the 0xFFFF value to the Interrupt Timer field. 3) The controller writes the decremented value to the Interrupt Timer field, overwriting the 0xFFFF value placed there by the host.

To rule out this possibility, the host should "debounce" the channel's Interrupt Timer field when writing the 0xFFFF value. This is done by:

1) writing 0xFFFF to the Interrupt Timer field
2) reading the Interrupt Timer field
3) comparing the value just read to 0xFFFF
4) rewriting 0xFFFF to the field if the values don’t match

The above sequence should be repeated until the 0xFFFF "sticks".

The benefit of explicitly suspending the interrupts associated with a channel is that the host does not need to mask controller interrupts during the entire interrupt service routine. Otherwise, masking might be necessary to insure that the board did not interrupt a routine which was already servicing the channel.

Channel Polling (with Interrupts Not Suspended)

In systems where masking interrupts from the controller is not a performance problem, the timer protocol described in the preceding section is not needed. If interrupts are masked in the host, the host may simply process the pending messages in the channel and write the new timer value afterwards.

This method causes "empty" interrupts to be generated if the timer expires while the host is processing messages. In such a situation, the controller generates an interrupt, but the host has interrupts masked and therefore does not "see" it until interrupts are unmasked. At that time, there are no more messages pending in the channel. Ordinarily, the timing window in which this can occur is small enough to make the frequency of empty interrupts insignificant. However, if the host uses very small values in the Interrupt Timer fields, the controller may generate an excessive number of empty interrupts. In this case, it is advisable to suspend interrupts before polling the channels.

ADDITIONAL CONSIDERATIONS

Creating Dedicated Channels

RC channels exist as independent entities and do not require any fixed relationship for correct operation. Most applications, however, will typically create separate channels dedicated to specific purposes. For example, you will probably want to set up board-to-host channels to respond to specific kinds of requests (such as requests to transmit data over a network). The "ID # of Response Channel" field in request-type commands allows you to select which board-to-host channel will be used by the controller to respond to a given request.

Off- vs. On-Board Segment Memory

Although it is possible to use the controller's shared memory space to store segments, this practice is discouraged. There is no performance benefit to be gained from using shared memory instead of host-resident memory.

0x0	(Not used; invalid)	--	--
0x1	Set DMA Burst	Directive	
0x2	Set Bus Timeout	Directive	
0x3	Request Statistics	Request	
0x4	Set Interrupt	Directive	
0x5	Create Host-to-Board Channel	Request	
0x6	Create Board-to-Host Channel	Request	
0x7	Link Host-to-Board Segment	Directive	
0x8	(reserved)	--	--
0x9	Allocate Board-to-Host Segment Memory	Directive	
RESPONSES AND INDICATIONS			
0x80	(Not used; invalid)	--	--
0x81	Report Printf Call	Indication	
0x82	Error Message	Indication	
0x83	Report Statistics	Response	

TYPES OF RC COMMANDS

The RC command set is divided into four groups _ requests, directives, responses, and indications.

Requests and **directives** are issued from the host to the controller. "Requests" have "responses" associated with them (i.e. a reply in the board-to-host direction). "Directives," on the other hand, have no associated reply.

Responses and **indications** are passed from the controller to the host. "Responses" are paired with requests that are host-to-board initiated. "Indications" are asynchronous events which have no pre-initi-

_____

_ _ _ TYPE OF _

_ TYPE _ DESCRIPTION _ RETURNED _

_ _ _ RC STRUCTURE _

+_____+_____+_____+

_ Request _ A host-to-board command that requires _ Response _

_ _ a response from the controller as a _ _

_ _ handshake. _ _

_ _ _ _

_ Directive _ A host-to-board command that does not _ _

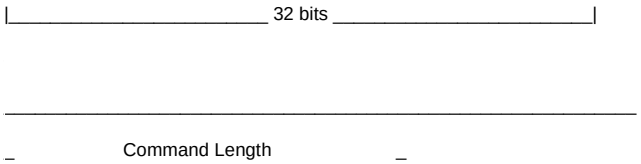
COMMONLY USED COMMAND FIELDS

With three exceptions, the fields in an RC command are explained in the section on that command. The three exceptions are:

- Command Control Block (CCB)
- Channel ID
- Transfer Options Word

Since the above fields occur in many RC commands (or all of them, in the case of the CCB), they are explained only once. A description of these fields appears in the next three sections.

Command Control Block Format



FIELDS

The fields in the Command Control Block are as follows:

· Command Length (4 bytes)

This field contains the number of bytes in the entire command, including the CCB, the parameters, and any extra padding. This value is used by the reader of a channel to find the next command in the channel.

· Command ID (4 bytes)

This field identifies the specific command (request, directive, response, or indication) to be processed. The command IDs of the generic RC command set are shown in the table on p. 50. Note that there is no relationship between a "command ID" and a "channel ID".

Channel ID Fields

Many RC commands include a field that identifies a specific channel. The name of the field varies slightly depending on its purpose. For example, the Set Interrupt directive has a "Target Channel ID" field that identifies the channel for which interrupts are being enabled. Likewise, request-type commands (such as Request Statistics) contain the field "ID # of Response Channel" to tell the controller where to write its response.

In all cases, the ID of a channel is the offset (in bytes) at which its channel descriptor starts in shared memory. Examples of channel IDs include:

- The ID of the initial board-to-host channel created at RC boot time. This value is always 0x20, since the channel descriptor of the initial board-to-host channel always begins at an offset of +0x20 (32 bytes) from the start of shared memory.

- The ID of the initial host-to-board channel created at RC boot time. This value is always 0x10, since the channel descriptor of the initial board-to-host channel always begins at an offset of +0x10 (16 bytes) from the start of shared memory.

- The ID of any channel created after RC boot time. The RC Interface assigns an ID to a channel upon successful completion of a Create Channel request (either host-to-board or board-to-host). It returns ID of the new channel via the appropriate response (Report New Host-to-Board Channel or Report New Board-to-Host Channel).

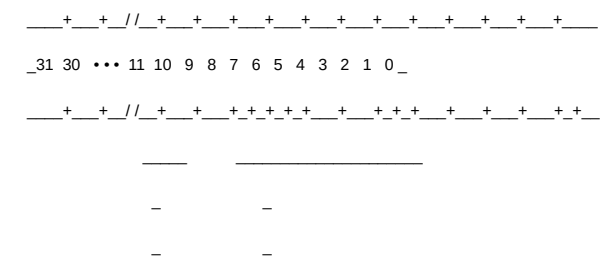
Transfer Options Word

The Transfer Options Word is used to specify what type of VMEbus transaction will be used for data transfers. The bits in the transfer options word indicate whether the transfer should be 16 or 32 bits wide and which address modifier should be used.

NOTE: The Transfer Options Word is defined differently for non-VME controllers such as the Multibus I 2211. For a description of the transfer options for such products, consult the user's guide accompanying the controller.

The Transfer Options Word is defined as a 32-bit unsigned integer, even though only the least significant word of the field (bits 0 - 15) is used to specify options.[1] It is recommended that the host treat the most significant word in the field (bits 16 - 31) as a reserved 16-bit field. The format of the Transfer Options Word is as follows:

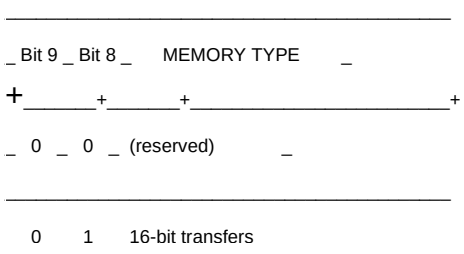
1. This is due to the way that static RAM is configured on Interphase controllers such as the V/FDDI 4211 Peregrine.



Bits 0-5 Address Modifier:

CODE	FUNCTION
0x09	Ext. Non-Privileged Data Access
0x0B	Ext. Non-Privileged Block Transfer
0x0D	Ext. Supervisory Data Access
0x0F	Ext. Supervisory Block Transfer

Bits 6-7 Reserved:



Bits 10-31 Reserved:

These bits are reserved and must be cleared to 0.

EXAMPLE:

For a DMA transfer in a SUN 3/260 4.1 operating system, the following transfer options word would be appropriate:

Bit #	Bits 10 - 31	9	8	7	6	5	4	3	2	1	0

This specifies 32-bit addressing and a VME address modifier of 0x3D.

$$\text{BYTES PER BURST} = \frac{\text{COUNT} * \text{TSIZE}}{8}$$

MNEMONICS:

COUNT = "Count Value" programmed using Set DMA Burst directive, or

For example, the V/FDDI 4211 Peregrine supports 16- and 32-bit DMA transfers and has an 8-bit transfer counter. Using the default transfer count (256), the DMA burst sizes are as follows:

Burst size (16-bit transfers) = (256 * 16) ÷ 8 = 512 bytes per burst

Burst size (32-bit transfers) = (256 * 32) ÷ 8 = 1024 bytes per burst

If the host does *not* issue this directive (or if it issues the directive with a Count Value of 0), the controller will use its default transfer count. See the Count Value field, p. 56, for more information.

After completing a DMA burst, the controller relinquishes the bus for the use of other devices on the bus. The controller's DMA engine will attempt to regain control of the bus as soon as possible in order to do another burst.

Size (in bytes)	DESCRIPTION
8	CCB

The RC Interface does not pass a reply message to the host for this command.

FIELDS

The fields in the Set DMA Burst directive are as follows:

- **Command Control Block (8 bytes)**

  This field contains the Command Control Block, as defined on p. 51. The command ID must be 0x1 for the Set DMA Burst directive.

- **Count Value (4 bytes)**

This field sets the maximum number of transfers that the transfer counter will perform per DMA burst. The set of legal values in this field = {0 .. 2^n-1}, where *n* equals the width of the transfer counter on your controller. For example, the V/FDDI 4211 Peregrine has an 8-bit transfer counter and therefore supports values of 0x0 _ 0xFF in this field.

Entering 0x0 in this field causes the controller to use its default transfer count. This is typically the maximum value supported by the counter (i.e. 2^n). On the 4211, for example, writing a 0 to this field causes the board to perform a maximum of 256 transfers per burst. As noted previously, you do not need to issue the Set DMA Burst directive to instruct the controller to use this default value.

For information on your controller's default transfer count, refer to "Default Report/Command Parameters" in the Specifications section of your controller manual.

The Set Bus Timeout directive sets the amount of time the controller will wait for a bus cycle to complete. If a bus cycle does not finish within this time, a DMA timeout message (error code 0x1) is sent to the host (see Error Message, p. 68).

The RC Interface does not pass a reply message to the host for this command.

Size (in bytes)	DESCRIPTION
8	CCB

FIELDS

The fields in the Set Bus Timeout directive are as follows:

· Command Control Block (8 bytes)

This field contains the Command Control Block, as defined on p. 51. The command ID must be set to 0x2 for the Set Bus Timeout directive.

· Number of Milliseconds (4 bytes)

This field sets the number of milliseconds the controller should wait for the bus cycle to complete. If this field is 0, the controller will use the default timeout value encoded in its firmware. This default value is stated in "Default Report/Command Parameters" in the Specifications chapter of your controller manual.

Note that the minimum possible timeout value is limited by the controller's timer resolution.

Request Statistics causes the controller to pass back various firmware parameters, including the firmware's version number, release date, and additional controller-specific parameters.

Size (in bytes)	DESCRIPTION
8	CCB
4	ID # of Response Channel

FIELDS

The fields in Request Statistics are as follows:

· Command Control Block (8 bytes)

This field contains the Command Control Block, as defined on p. 51. The command ID must be set to 0x3 for Request Statistics.

· ID # of Response Channel (4 bytes)

This field contains the Channel ID of the board-to-host channel to which the controller should write the statistics. For a discussion of channel IDs, refer to p. 51.

· Command Tag (4 bytes)

This field may contain a host-generated command tag, if needed for the application. Command tags are typically generated by the host's device driver and can be used to notify host processes associated with a command. The RC Interface does not use the command tag in any way. It simply returns the value in the Report Statistics response.

As discussed in the section on interrupts (p. 44), the controller may be directed to interrupt the host for a particular channel when certain conditions are met.

The Set Interrupt directive is used to assign a bus interrupt level and vector to a specific board-to-host channel. Different board-to-host channels can have different interrupt vectors and/or levels. This provides a mechanism for separating different classes of responses.

The controller will not interrupt the host for any reason until this directive has been issued to the RC Interface.

Size (in bytes)	DESCRIPTION
8	CCB
4	Target Channel ID

FIELDS

The fields in the Set Interrupt directive are as follows:

- **Command Control Block (8 bytes)**

  This field contains the Command Control Block, as defined on p. 51. The command ID must be set to 0x4 for the Set Interrupt directive.

- **Target Channel ID (4 bytes)**

  This field identifies the board-to-host channel to which the interrupt level and vector are being assigned. For a discussion of channel IDs, refer to p. 51.

- **Interrupt Level (4 bytes)**

  This field sets the bus interrupt level used by the controller to notify the host that it has completed command(s) in a given board-to-host channel (where the board-to-host channel is specified in the "Target Channel ID" field). There is no default value for this field.

  Valid entries in the field depend on the bus environment in which the controller is operating.

EXAMPLES: On the V/FDDI 4211 Peregrine controller, valid entries in this field are 0x1 through 0x7 (respectively corresponding to IRQ1 _ IRQ7). On the Multibus I 2211 FDDI Node Controller, this field can contain any value from 0x0 _ 0x7.

· Interrupt Vector (4 bytes)

This field sets the bus interrupt vector that the controller will use to report command completion. There is no default value for the field.

Allowed values in this field depend on the bus environment in which the controller is operating. On VMEbus controllers such as the V/FDDI 4211 Peregrine, legal entries in this field are 0x0 through 0xFF.

This field is not utilized in autovectored bus environments such as the Multibus I 2211 controller. In this case, the controller ignores any value placed in the field.

Create Host-to-Board Channel allows the host to create a new host-to-board channel. This command is issued only by the host, and has the effect of determining who owns which read/write pointers in the channel descriptor.

In reply to this command, the RC Interface: 1) assigns an ID to the channel, and 2) returns the new ID via

Size (in bytes)	DESCRIPTION
8	CCB
4	ID # of Response Channel
4	Command Tag

FIELDS

The fields in the Create Host-to-Board Channel request are as follows:

· Command Control Block (8 bytes)

This field contains the Command Control Block, as defined on p. 51. The command ID must be set to 0x5 for Create Host-to-Board Channel.

· ID # of Response Channel (4 bytes)

This value is the ID of the board-to-host channel to which the controller should write the Report New Host-to-Board Channel response. For a discussion of channel IDs, refer to p. 51. For details on Report New Host-to-Board Channel, see p. 71.

· Command Tag (4 bytes)

This field may contain a host-generated command tag, if needed for the application. Command tags are typically generated by the host's device driver and can be used to notify host processes associated with a command. The RC Interface does not use the command tag in any way. It simply returns the value in the Report New Host-to-Board Channel response.

· Transfer Options of First H2B Segment (4 bytes)

This field contains the Transfer Options Word, as defined on p. 52. The controller uses this information to perform DMA accesses on the ***first*** segment in the channel. The transfer characteristics of subsequent segments are given when the host spans to each new segment. (This is specified using the Transfer Options Word in the Link Host-to-Board Segment directive, see p. 64.)

· Physical Address of First H2B Segment (4 bytes)

This field provides the controller with the address by which the onboard DMA mechanism can reach the first segment. This address may very well be different from the logical or virtual address by which host-resident software accesses the memory.

· Length in Bytes of First H2B Segment (4 bytes)

This field contains the number of bytes in the channel's first segment. The controller uses this value in cases where the "Write Segment #" field in the channel descriptor increments past the first segment before that segment has been processed by the controller.

The physical addresses and byte lengths of subsequent segments in the channel are provided to the controller using the Link Host-to-Board Segment directive.

Create Board-to-Host Channel allows the host to create a new board-to-host channel. It is issued only by the host, and has the effect of determining who owns which read/write pointers in the channel descriptor.

In reply to this command, the RC Interface: 1) assigns an ID to the channel, and 2) returns the new ID via a Report New Board-to-Host Channel response (see p. 72). The returned channel ID is the byte offset from the start of shared memory to the channel descriptor for the new channel. The controller will update

Size (in bytes)	DESCRIPTION
8	CCB

FIELDS

The fields in the Create Board-to-Host Channel request are as follows:

- **Command Control Block (8 bytes)**

  This field contains the Command Control Block, as defined on p. 51. The command ID must be set to 0x6 for Create Board-to-Host Channel.

- **ID # of Response Channel (4 bytes)**

  This value is the ID of the board-to-host channel to which the controller should write the Report New Board-to-Host Channel response. For a discussion of channel IDs, refer to p. 51. For details on Report New Board-to-Host Channel, see p. 72.

- **Command Tag (4 bytes)**

  This field may contain a host-generated command tag, if needed for the application. Command tags are typically generated by the host's device driver and can be used to notify host processes associated with a command. The RC Interface does not use the command tag in any way. It simply returns the value in the Report New Board-to-Host Channel response.

Size (in bytes)	DESCRIPTION
8	CCB
4	Transfer Options Word

FIELDS

The fields in the Link Host-to-Board Segment directive are as follows:

· Command Control Block (8 bytes)

This field contains the Command Control Block, as defined on p. 51. The command ID must be set to 0x7 for the Link Host-to-Board Segment directive.

· Transfer Options Word (4 bytes)

This field contains the Transfer Options Word, as defined on p. 52. It provides information required in order for the controller to have DMA access to the next segment in this host-to-board channel.

· Physical Address Of Next Segment (4 bytes)

This field contains the DMA-able byte address of the start of the next segment. The physical address of the segment may be different from the virtual address to which the host software writes.

· Length of Next Segment in Bytes (4 bytes)

This field gives the number of total bytes in the next segment. This information is needed by the controller in order to DMA the segment into onboard memory.

This directive supplies the controller with a list of host DMA buffer resources for board-to-host channel segments. The segments are not assigned by the host to any particular channel. Each segment is identified by a pair of addresses. The first is the physical address for board-to-host DMA purposes. The second address is the virtual, or "logical" address by which the host will access the segment. The controller uses this virtual address to tell the host the location of the next segment in a board-to-host channel. It does so by placing the address into a Report Link to Next Segment indication, which the host uses to span from one channel segment to another. For details on Report Link to Next Segment, see p. 74.

Allocate Board-to-Host Segment Memory must be issued as a parameter to "BOOT", which is the Common Boot command used to boot the RC Interface. This is necessary so that the board-to-host segment memory is in place after RC initialization. For details on BOOT, see p. 18.

The host must allocate ***at least two*** channel segments to the controller for ***each*** board-to-host channel to be created. The controller enforces this rule when it processes requests to create new board-to-host chan

Size (in bytes)	DESCRIPTION
8	CCB
4	Transfer Options Word (Same for all segments.)
4	Segment Length (Same for all segments.)
4	Physical Address of First Segment

FIELDS

The fields in the Allocate Board-to-Host Segment Memory directive are as follows:

· Command Control Block (8 bytes)

This field contains the Command Control Block, as defined on p. 51. The command ID must be set to 0x8 for the Allocate Board-to-Host Segment Memory directive.

· Transfer Options Word (4 bytes)

This field contains the Transfer Options Word, as defined on p. 52. It provides information

required in order for the controller to have DMA access to the board-to-host channel segments. All segments in a given board-to-host channel must have the same transfer characteristics.

· Segment Length (4 bytes)

This field contains the length of the segments in bytes. All segments must have the same size. The maximum legal value in this field is 65,534 bytes (0xFFFE).

· Physical Address of Segment n (4 bytes)

This field provides the controller with the DMA-able byte address of the start of segment **n**. The physical address of the segment may be different from the virtual address to which the host software writes.

· Command Tag or Virtual Address of Segment n (4 bytes)

This field contains a value which enables the host to identify segment **n** when it is used by the controller. The controller reports this value when it links to segment **n** (see Report Link to Next Segment, p. 74).

Report Printf Call is delivered to the host whenever an on-board software module calls the *printf()* routine. It is typically used to report boottime configuration information and non-fatal anomalies. It can also provide debugging information which is useful in the development of firmware or downloaded software. The body of this message is a null-terminated string that should be treated as a status message for the operator/

Size (in bytes)	DESCRIPTION
8	CCB
N	Variable-Length ASCII String

FIELDS

The fields in Report Printf Call are as follows:

· Command Control Block (8 bytes)

This field contains the Command Control Block, as defined on p. 51. The controller sets the Command ID field to 0x81 for Report Printf Call.

· Variable-Length ASCII String

This field contains a variable-length, null-terminated string that should be treated as a status message for the operator/user. It is followed by 0 to 3 bytes to pad the message to a longword boundary. Or, if the string ends on a longword boundary, it is followed by an extra word of 0's.

	address. The starting address of the transfer that failed is given in the error string.
0x2	An internal error has occurred on the controller.
0x3	The command identified in the "CCB in Error" field contains invalid parameter(s).
0x4	The controller made an unsuccessful malloc attempt.†
0x5	The command, though valid, is not currently implemented on this controller.
0x6	The host has specified an unknown command.
0x7	The controller's non-volatile RAM (NOVRAM) has failed.
0x8	A DMA bus error has occurred. The controller tried to DMA data to a location that is "not there" (from the controller's perspective). This indicates either a bad address, or failure by the host to map the memory properly for DMA.
0x9	The controller tried to allocate an mblk but was unable to do so. This means that the on-board static RAM is full.
0xB	A byte-alignment error has taken place. For information on how the controller expects data to be aligned, see "System Requirements" (p.) in this manual, as well as the Functional Overview and Specifications chapters in your controller manual.
0xD	The controller has run out of available board-to-host segments. To handle this situation, the host can take either (or both) of the following actions. 1) The host can free up "controller-owned" segments by extracting data from the board-to-host channel(s) more rapidly. 2) The host can give the board more segment memory (see Allocate Board-to-Host Segment Memory, p.).
0xE	The controller made an unsuccessful allocb attempt.†
0xF	The controller made an unsuccessful salloc attempt.†
0x10	The controller has run out of channel space. This means that there is no room left in shared memory to write new channel descriptors.

· CCB in Error (8 bytes)

This field contains the CCB of the command which caused the error.

· Variable-Length ASCII String

This field contains a variable-length, null-terminated string describing the error. It is followed by 0 to 3 bytes (if needed) to pad the message to a longword boundary. Or, if the string ends on a longword boundary, it is followed by an extra word of 0's.

Size (in bytes)	DESCRIPTION
8	CCB
4	Command Tag

FIELDS

The fields in the Report Statistics response are as follows:

- **Command Control Block (8 bytes)**

  This field contains the Command Control Block, as defined on p. 51. The controller sets the Command ID field to 0x83 for Report Statistics.

- **Command Tag (4 bytes)**

  This value is the host-supplied command tag (if any) contained in the command to which the controller is responding.

- **Firmware ID (16 bytes)**

  This field contains an ASCII string which identifies the ID number of the firmware on your controller.

- **Null-Terminated Release Date (16 bytes)**

  This field contains a null-terminated ASCII string stating the release date of the firmware on your controller.

- **Buffer RAM Size in Bytes (4 bytes)**

  This is the size of your controller's buffer RAM.

· **Static RAM Size in Bytes (4 bytes)**

This is the size of your controller’s static RAM.

Size (in bytes)	DESCRIPTION
8	CCB

FIELDS

The fields in the Report New Host-to-Board Channel response are as follows:

· Command Control Block (8 bytes)

This field contains the Command Control Block, as defined on p. 51. The controller sets the Command ID field to 0x84 for Report New Host-to-Board Channel.

· Command Tag (4 bytes)

This value is the host-supplied command tag (if any) contained in the command to which the controller is responding.

· ID # of Created Channel (4 bytes)

This field contains ID number of the newly created channel. For a discussion of channel IDs, refer to p. 51.

Size (in bytes)	DESCRIPTION
8	CCB
4	Command Tag

FIELDS

The fields in the Report New Board-to-Host Channel response are as follows:

· Command Control Block (8 bytes)

This field contains the Command Control Block, as defined on p. 51. The controller sets the Command ID field to 0x85 for Report New Board-to-Host Channel.

· Command Tag (4 bytes)

This value is the host-supplied command tag (if any) contained in the command to which the controller is responding.

· ID # of Created Channel (4 bytes)

This field contains ID number of the newly created channel. For a discussion of channel IDs, refer to p. 51.

· Virtual Address Of Initial Segment (4 bytes)

This field contains the logical address at which the initial segment in the channel begins. This address was originally generated by the host and passed to the controller via the Allocate Board-to-Host Segment Memory directive.

· Length Of Initial Segment in Bytes (4 bytes)

This field gives the number of total bytes in the initial segment. This information is provided to the host as a convenience _ it may not be needed other than for sanity checking of segment bound-

aries. In most applications, the host simply reads RC commands from the beginning of the next segment until: 1) encountering another Link Board-to-Host directive, or 2) determining that the read pointer matches the write pointer.

Report Link to Next Segment is issued by the controller when it needs to span to the next segment in a board-to-host channel. This indication tells the host to start drawing controller-generated responses/indications for this channel from a different block of memory.

Size (in bytes)	DESCRIPTION
8	CCB

FIELDS

The fields in Report Link to Next Segment are as follows:

· Command Control Block (8 bytes)

This field contains the Command Control Block, as defined on p. 51. The controller sets the Command ID field to 0x86 for Report Link to Next Segment.

· Command Tag or Virtual Address Of Next Segment (4 bytes)

This field identifies the segment to which the controller is linking. The value in this field was originally generated by the host and passed to the controller via the Allocate Board-to-Host Segment Memory directive (p. 65).

· Length of Next Segment in Bytes (4 bytes)

This field gives the number of total bytes in the next segment. This information is provided to the host as a convenience _ it may not be needed other than for sanity checking of segment boundaries. In most applications, the host simply reads RC commands from the beginning of the next segment until: 1) encountering another Link Board-to-Host directive, or 2) determining that the read pointer matches the write pointer.

APPENDIX A
DATA STRUCTURES

This appendix contains data structures which are useful in understanding Common Boot and the RC Interface. These structures were copied from the header file of a device driver for an RC-based controller. For a more complete listing of the header file, see the Data Structures section of your controller manual. To get a sample driver from Interphase, calling Customer Service at the number shown on p. 4.

```
/*
    External view of RC channel control structure
*/
typedef volatile struct ex_rc_s {
    u16  reserved1;         /* Reserved word              */
    u16  intr_timer;        /* Interrupt/timer word       */
    u16  r_segment;         /* Read Segment number        */
    u16  r_offset;      /* Read offset value          */
    u16  w_segment;         /* Write Segment number       */
    u16  w_offset;      /* Write offset value         */
    u16  reserved2;         /* Reserved word              */
    u16  reserved3;         /* Reserved word              */
} EX_RC, * EX_RCP;

/*
    The format of short io space as seen by the host.
*/
typedef volatile struct erc_s {
    u16     cint;           /* Hardware control field (controller-specific)     */
    u16     icid;           /* Controller writes 0x20 here at RC initialization */
    u32     reserved1;
    u32     reserved2;
    u32     reserved3;
    EX_RC   rc[RC_MAX_NUMBER_OF_CHANNELS];
} E_SHIO, *E_SHIOP;
/*
    Reading and writing channel pointers must be done carefully due to
    the fact that the board may be updating them.  The following macros
    should be used to read and write these pointers.
*/
```

```
/*
    This macro updates the write pointer for a channel.  It is
    passed the address of the segment number field, the new segment
    number, the address of the offset field and the new offset value.
    This macro should be used when updating a b2h channel read pointer.
*/
#define UPDATE_CHANNEL_PTR(seg,newseg,offset,newoffset) { \
    if (*(u16 *)seg != (u16)newseg) { \
        *(u16 *)offset = (u16)RC_UPDATE_IN_PROGRESS;\
        *(u16 *)seg = (u16)newseg;\
        *(u16 *)offset = (u16)newoffset;\
    } \
    else \
        *(u16 *)offset = (u16)newoffset;\
}



/*
    This macro reads an RC channel pointer.  We must be careful when
    reading such a pointer because the pointer may be in the process
    of being updated by the board.  It is passed the address of the
    the segment number to read, the address of the offset to read and
    it returns the segment number and the offset value.  This macro should
    be used when reading a b2h write pointer or h2b read pointer.
*/
#define READ_CHANNEL_PTR(saddr,oaddr,seg,offset){ \
    u16 stmp; \
    do{ \
        do{ \
            (u16)seg = *(u16 *)saddr;\
            (u16)offset = *(u16 *)oaddr;\
        } while (offset == RC_UPDATE_IN_PROGRESS); \
        stmp = *(u16 *)saddr; /* insures that new seg number is read in */ \
    } while (stmp != seg); \
}
/*
    Update host-resident image of the segment offset.  The segment
    number either did not change or was updated in the do_h2b_link_dir()
    function.
```

```
        Update shortI/O version of the segment offset.  This actually
        notifies the board that the command is there, otherwise it
        will not be processed.
*/
#define    update_h2b_ptr(len, pi, cn) { \
            pi->prc[cn].w_offset += len; pi->shio->rc[cn].w_offset += len; \
}
/*
        For each segment of a channel a segment structure is kept.
*/
typedef struct  prg_seg_desc {
        caddr_t addr;               /* Virtual address of segment.       */
            caddr_t    paddr;                   /* Physical address of segment.     */
        ushort  segno;              /* Segment number                    */
} HSD, *HSDP;

/*
            Debug macros and defines.
*/
/*
        Layout of a 32 bit word in short IO space after the controller has
        been reset.
*/
typedef struct cb_herald {
        u8          hdr;
        u8          offset;
        ushort      shio_len;
} CBH,*CBHP;
/*
        Values for CB commands.
*/
#define CB_HERALD               0xcb
#define CB_MAX_DNLD_SIZE   0xd0                 /* bsd/sun only allow 256 data in ioctl   */
#define CB_BOARD_LOCATION       0x80002000
#define CB_BOOT_OFFSET          1

/*
        Format of a a CB boot command.
*/
typedef struct cbboot {
```

```
        u32     cmd;
        u32     index;          /* Index of image to boot. */
} CBBTCMD,*CBBTCMDP;

/*
    Format of a a CB poke command.
*/
typedef struct cbpoke {
        u32     cmd;
        u32     address;            /* Destination address (wrtie to here). */
        u32     count;                  /* Number of units to write.       */
        u32     unit_size;          /* Size in bytes of unit (1,2 or 4)    */
        u8      data[CB_MAX_DNLD_SIZE];     /* data.                       */
} POKECMD,*POKECMDP;

/*
    Format of a CB fill command.
*/
typedef struct cbfill {
        u32     cmd;            /* Command value                                    */
        u32     pattern;    /* Pattern to fill board memory with.      */
        u32     brdsa;          /* Board starting address to write to      */
        u32     count;          /* Number of units of pattern to write     */
        u32     unit_size;  /* Size in bits of a unit                  */
} CBFILLCMD;

#define MAKE_LONG(a,b,c,d) (((u32)a << 24) | ((u32)b << 16) | ((u32)c << 8) | \
                            (u32)d)

/*
    Values for the cmd field of a cb command.
*/
#define CB_BOOT     MAKE_LONG('B','O','O','T')
#define CB_FILL     MAKE_LONG('F','I','L','L')

/* Define some data types for the CB_POKE command */
#define CB_BYTE     1
#define CB_WORD     2
#define CB_LONG     4
```

```
/*
    CB responds to a command with the following value.
*/
#define CB_OK           MAKE_LONG('C','B','O','K')
#define CB_POKEC        MAKE_LONG('P','O','K','E')
#define CB_PEEKC        MAKE_LONG('P','E','E','K')
/*
    Structure for peeking and poking with cb.
*/
typedef struct cbpk {
    u32  cmd;
    u32  addr;
    u32  len;
    u32  unit;
    u32  data[1];
}CBPKS,*CBPKSP;

typedef struct smtmsg {
    u32  length;            /* Length of message in bytes      */
    char msg[128];          /* Actual message                  */
}SMTMSG,*SMTMSGP;

static unsigned char bitswaptbl[256] = {
    0x00, 0x80, 0x40, 0xc0, 0x20, 0xa0, 0x60, 0xe0,
    0x10, 0x90, 0x50, 0xd0, 0x30, 0xb0, 0x70, 0xf0,
    0x08, 0x88, 0x48, 0xc8, 0x28, 0xa8, 0x68, 0xe8,
    0x18, 0x98, 0x58, 0xd8, 0x38, 0xb8, 0x78, 0xf8,
    0x04, 0x84, 0x44, 0xc4, 0x24, 0xa4, 0x64, 0xe4,
    0x14, 0x94, 0x54, 0xd4, 0x34, 0xb4, 0x74, 0xf4,
    0x0c, 0x8c, 0x4c, 0xcc, 0x2c, 0xac, 0x6c, 0xec,
    0x1c, 0x9c, 0x5c, 0xdc, 0x3c, 0xbc, 0x7c, 0xfc,
    0x02, 0x82, 0x42, 0xc2, 0x22, 0xa2, 0x62, 0xe2,
    0x12, 0x92, 0x52, 0xd2, 0x32, 0xb2, 0x72, 0xf2,
    0x0a, 0x8a, 0x4a, 0xca, 0x2a, 0xaa, 0x6a, 0xea,
    0x1a, 0x9a, 0x5a, 0xda, 0x3a, 0xba, 0x7a, 0xfa,
    0x06, 0x86, 0x46, 0xc6, 0x26, 0xa6, 0x66, 0xe6,
    0x16, 0x96, 0x56, 0xd6, 0x36, 0xb6, 0x76, 0xf6,
    0x0e, 0x8e, 0x4e, 0xce, 0x2e, 0xae, 0x6e, 0xee,
    0x1e, 0x9e, 0x5e, 0xde, 0x3e, 0xbe, 0x7e, 0xfe,
    0x01, 0x81, 0x41, 0xc1, 0x21, 0xa1, 0x61, 0xe1,
```

```
        0x11, 0x91, 0x51, 0xd1, 0x31, 0xb1, 0x71, 0xf1,
        0x09, 0x89, 0x49, 0xc9, 0x29, 0xa9, 0x69, 0xe9,
        0x19, 0x99, 0x59, 0xd9, 0x39, 0xb9, 0x79, 0xf9,
        0x05, 0x85, 0x45, 0xc5, 0x25, 0xa5, 0x65, 0xe5,
        0x15, 0x95, 0x55, 0xd5, 0x35, 0xb5, 0x75, 0xf5,
        0x0d, 0x8d, 0x4d, 0xcd, 0x2d, 0xad, 0x6d, 0xed,
        0x1d, 0x9d, 0x5d, 0xdd, 0x3d, 0xbd, 0x7d, 0xfd,
        0x03, 0x83, 0x43, 0xc3, 0x23, 0xa3, 0x63, 0xe3,
        0x13, 0x93, 0x53, 0xd3, 0x33, 0xb3, 0x73, 0xf3,
        0x0b, 0x8b, 0x4b, 0xcb, 0x2b, 0xab, 0x6b, 0xeb,
        0x1b, 0x9b, 0x5b, 0xdb, 0x3b, 0xbb, 0x7b, 0xfb,
        0x07, 0x87, 0x47, 0xc7, 0x27, 0xa7, 0x67, 0xe7,
        0x17, 0x97, 0x57, 0xd7, 0x37, 0xb7, 0x77, 0xf7,
        0x0f, 0x8f, 0x4f, 0xcf, 0x2f, 0xaf, 0x6f, 0xef,
        0x1f, 0x9f, 0x5f, 0xdf, 0x3f, 0xbf, 0x7f, 0xff,
};

/*
        This struct is used when peeking or poking in short io space.
*/
typedef struct shio_pk {
        uint  offset;                   /* Offset within short io to peek or poke       */
        ushort      data;                       /* Data to poke, or data which was peeked    */
} SHPK,*SHPKP;

typedef struct cb_pk {
    uint    offset; /* Offset within short io to peek or poke   */
    uint    len;    /* Number of units to write.                */
    uint    unit;   /* Size of unit to write                    */
    uint    data;   /* Data to poke, or data which was peeked   */
} CBPK, *CBPKP;

typedef struct get_peek_poke {
        uint  address;          /* Offset within short io to peek or poke       */
        uint  data;                     /* Data to poke, or data which was peeked    */
} GET_PP,*GET_PPP;

/* tmb typedef struct data_s {              */
/* tmb      caddr_t dp;                     */
/* tmb      int     len;                    */
```

```
/* tmb      struct      sockaddr sa;    */
/* tmb } PASSARG, *PASSARGP;              */

typedef struct raw_data {
        caddr_t dp;
        int          len;
} RAW_DATA, * RAW_DATAP;

typedef struct get_stuff {
        ushort  addr[32];
} GET_STUFF, * GET_STUFFP;

typedef struct cbcmd {
        u32          actual_length;
        POKECMD     actual_command;
} CB_CMD, * CB_CMDP;

typedef struct maddr {
        u16          mac_addr[3];
        u16          type;
} MADDR, * MADDRP;

typedef struct mib_struct {
        u32          attribute_name;
        u32          attribute_length;
        u32          attribute_index;
        char *data_struct;
} MIB_STRUCT, * MIB_STRUCTP;

/*
        Defines for debugger.
*/
#define DBG_SIG                     0x0000db29
#define DBG_CMD_OFFSET              0x1dc
#define DBG_ARG0_OFFSET             0x1e0
#define DBG_ARG1_OFFSET             0x1e4
#define DBG_ARG2_OFFSET             0x1e8
#define DBG_BP_ADDR                 0x1ec
#define DBG_REG_PEEK_CMD            0x00001011
#define DBG_REG_POKE_CMD            0x00001012
```

```
#define DBG_MEM_PEEK_CMD        0x00001013
#define DBG_MEM_POKE_CMD        0x00001014
#define DBG_STEP_ADDR_CMD       0x00001015
#define DBG_CONT_CMD            0x00001016
#define DBG_SETVECTOR_CMD       0x00001017
#define DBG_BP_FLAG             0x7070

#define SHIO_POKE               _IOWR(i,1,SHPK)         /* Poke data into short io      */
#define SHIO_PEEK               _IOWR(i,2,SHPK)         /* Read data from short io      */
#define BOOT_RC                 _IOWR(i,3,SHPK)         /* Boot RC                      */
#define IOTEST                  _IOWR(i,4,struct ifreq) /* access test                  */
#define IORESET                 _IOWR(i,5,struct ifreq) /* reset board                  */
#define XMIT_RAW                _IOWR(i,6,RAW_DATA)             /* Transmit RAW data
*/
#define POST_RECV_RESOURCE      _IOWR(i,7,SHPK)         /* Post receive resources       */
#define POLL_B2H_RINGS          _IOWR(i,8,SHPK)         /* Poll the b2h rings           */

#define IODEBUG                 _IOWR(i,12,struct ifreq)        /* toggle diagnostic messages   */
#define IODMEM                  _IOWR(i,14,struct ifreq)        /* dump memory                  */

#define MEM_POKE                _IOWR(i,20,SHPK)        /* Poke data into memory        */
#define MEM_PEEK                _IOWR(i,21,SHPK)        /* Read data from memory        */
#define GET_MAC_ADDR            _IOWR(i,22,GET_STUFF)           /* Read station address         */
#define TRANSMIT_FRAME          _IOWR(i,23,GET_STUFF)           /* transmit a frame             */
#define SET_MAC_ADDR            _IOWR(i,25,MADDR)               /* Set station address          */

#define INTR_SET                _IOWR(i,29,GET_STUFF)           /* interrupt set directive      */
#define BURST_SET               _IOWR(i,30,GET_STUFF)           /* DMA burst set                */
#define BTIMEOUT_SET            _IOWR(i,31,GET_STUFF)           /* bus timeout set              */
#define DO_ECHO         _IOWR(i,32,GET_STUFF)           /* RC LOOP BACK                 */

#define CB_POKE                 _IOWR(i,33,CBPK)        /* Poke data into 4211 mem      */
#define CB_PEEK                 _IOWR(i,34,CBPK)        /* Read data from 4211 mem      */
#define REG_PBG                 _IOWR(i,35,int)         /* Register the debugger with the driver */
#define CB_PASS                 _IOWR(i,36,CB_CMD)              /* Pass thru CB commands to board       */

#define DBG_POKE                _IOWR(i,37,CBPK)        /* Poke data into mem using dbg */
#define DBG_PEEK                _IOWR(i,38,CBPK)        /* Read data from mem using dbg */
#define DBG_SET_VEC             _IOWR(i,39,CBPK)        /* Read data from mem using dbg */
#define DBG_STEP                _IOWR(i,40,int)         /* Do single step               */
```

```
#define DBG_CONT            _IOWR(i,41,int)                 /* Continue from breakpoint      */
#define REG_FBM             _IOWR(i,42,int)                 /* Register FBM with the driver. */
#define SEND_SMT_MSG        _IOWR(i,43,SMTMSG)              /* Register FBM with the driver. */
#define READ_SMT_MSG        _IOWR(i,44,SMTMSG)              /* Register FBM with the driver. */
#define GET_SMT_MAC_ADDR    _IOWR(i,45,SMT_MAC_ADDRESS)     /* Get the MAC address
*/
#define SET_SMT_MAC_ADDR    _IOWR(i,46,SMT_MAC_ADDRESS)     /* Set the MAC address
*/

#define SET_DBFLAGS         _IOWR(i,50,SHPK)                /* Set a driver debug flag       */
#define GET_DBFLAGS         _IOWR(i,51,SHPK)                /* Get driver debug flags        */
#define GET_RBLIST          _IOWR(i,52,SHPK)                /* print the receive buffer list */
#define PRINT_COUNTERS      _IOWR(i,53,SHPK)                /* print driver counters         */

#define BRD_PEEK            _IOWR(i,56,GET_PP)              /* look at the board             */
#define BRD_POKE            _IOWR(i,57,GET_PP)              /* change location on the board  */

/*
 * Common RC support definitions and structures
 */

#define RCC_MAX_COMMANDS    256  /* Max commands per command table */
#define RCC_MAX_CMD_MASK    (RCC_MAX_COMMANDS - 1)

#define RC_TABLE_INDEX      0
#define FDDI_TABLE_INDEX    1
#define CSP_TABLE_INDEX     2
#define RCC_COMMAND_COUNT   3


/* The index zero is invalid for a command index. */
#define RCC_RC_CMD_BASE         ((RCC_MAX_COMMANDS * RC_TABLE_INDEX) + 1)
#define RCC_FDDI_CMD_BASE   ((RCC_MAX_COMMANDS * FDDI_TABLE_INDEX) + 1)
#define RCC_CSPSMT_CMD_BASE    ((RCC_MAX_COMMANDS * CSP_TABLE_INDEX) + 1)

/*
 *  Common RC commands
 */
#define RC_CMD_BASE(x)              ((x) + RCC_RC_CMD_BASE)
#define RC_CMD_RBASE(x)             ((x) + RC_CMD_BASE(128))
```

```
#define RC_BUS_BURST_SET_DIRECTIVE          RC_CMD_BASE(0)
#define RC_BUS_TIMEOUT_SET_DIRECTIVE            RC_CMD_BASE(1)
#define RC_INFO_REQUEST                         RC_CMD_BASE(2)
#define RC_INTERRUPT_SET_DIRECTIVE          RC_CMD_BASE(3)
#define RC_H2B_CHANNEL_CREATE_REQUEST  RC_CMD_BASE(4)
#define RC_B2H_CHANNEL_CREATE_REQUEST  RC_CMD_BASE(5)
#define RC_H2B_LINK_DIRECTIVE               RC_CMD_BASE(6)
#define RC_B2H_SET_DIRECTIVE                RC_CMD_BASE(7)
#define RC_SEG_MEM_DIRECTIVE                RC_CMD_BASE(8)

#define RC_OPERATOR_MSG_INDICATION      RC_CMD_RBASE(0)
#define RC_ERROR_MSG_INDICATION             RC_CMD_RBASE(1)
#define RC_INFO_RESPONSE                        RC_CMD_RBASE(2)
#define RC_H2B_CHANNEL_CREATE_RESPONSE      RC_CMD_RBASE(3)
#define RC_B2H_CHANNEL_CREATE_RESPONSE      RC_CMD_RBASE(4)
#define RC_B2H_LINK_INDICATION              RC_CMD_RBASE(5)


/*
 * This is the header for all RC commands and responses.  For a description of
 * the CCB, see p. 51 in the text.
 */
typedef struct ccb_s {
      u32  len;       /* length of command including this header       */
      u32  cmd;            /* Command code                                  */
} CCB, * CCBP;

/************************** COMMANDS (GENERIC RC) ****************************/

/*
 * RC_BUS_BURST_SET_DIRECTIVE   See Set DMA Burst directive, p. 55,
 * for command description
*/
typedef struct rc_bbs_s {
      CCBccb;
      u32  burst_value;                 /* number of transfers per burst          */
} RC_BBS, * RC_BBSP;

/*
```

```
 * RC_BUS_TIMEOUT_SET   See Set Bus Timeout directive, p. 57,
 * for command description
*/
typedef struct rc_bts_s {
        CCBccb;
        u32  bus_timeout;               /* number of milliseconds               */
} RC_BTS, * RC_BTSP;

/*
 * RC_INTERRUPT_SET_DIRECTIVE   See Set Interrupt directive, p. 59,
 * for command description
*/
typedef struct rc_ris_s {
        CCBccb;
        u32  channel_id;                /* Id of channel to set level/vector    */
        u32  level;                     /* Interrupt level                      */
        u32  vector;                    /* Interrupt vector                     */
} RC_RIS, * RC_RISP;

/*
 * RC_INFO_REQUEST   See Request Statistics, p. 58,
 * for command description
*/
typedef struct rc_info_s {
        CCBccb;
        u32  response_id;               /* Channel id for response              */
        u32  tag;
} RC_INFO, * RC_INFOP;

/*
 * RC_H2B_CHANNEL_CREATE_REQUEST   See Create Host-to-Board Channel, p. 61,
 * for command description
*/
typedef struct rc_h2b_channelc_s {
        CCBccb;
        u32  response_id;               /* Channel id for response              */
        u32  tag;                       /* tag                                  */
        u32  xferopts;                  /* transfer options word                */
        u32  phyaddr;                   /* physical address of the seg          */
        u32  length;                        /* length of segment                */
```

```
} RC_H2B_CHANNELC, * RC_H2B_CHANNELCP;

/*
 * RC_B2H_CHANNEL_CREATE_REQUEST   See Create Board-to-Host Channel, p. 63,
 * for command description
*/
typedef struct rc_b2h_channelc_s {
        CCBccb;
        u32  response_id;               /* Channel id for response              */
        u32  tag;                       /* tag                                  */
} RC_B2H_CHANNELC, * RC_B2H_CHANNELCP;

/*
 * RC_H2B_LINK_DIRECTIVE   See Link Host-to-Board Segment directive, p. 64,
 * for command description
*/
typedef struct rc_h2b_link_s {
        CCBccb;
        u32  xferopts;                  /* transfer options lower 16 bits       */
        u32  phyaddr;                   /* Physical VME Address                 */
        u32  length;                        /* Length of new segment            */
} RC_H2B_LINK, * RC_H2B_LINKP;

/*
 * RC_SEG_MEM_DIRECTIVE   See Allocate Board-to-Host Segment Memory, p. 65,
 * for command description
*/
typedef struct rc_seg_mem_s {
        CCBccb;
        u32  segment_xferopts;          /* Transfer type for segments           */
        u32  segment_length;                /* length of each segment           */
        RC_SEG seg[1];                      /* one of more Physical addresses   */
} RC_SEGMEM, * RC_SEGMEMP;

/************************** RESPONSES (GENERIC RC) ****************************/

/*
 * RC_OPERATOR_MSG_INDICATION   See Report Printf Call indication, p. 67,
 * for command description
*/
```

```
typedef struct rc_mfo_s {
    CCBccb;
    u8    msg[4];                    /* Actually variable length          */
} RC_MFO, * RC_MFOP;

/*
 * RC_ERROR_MSG_INDICATION   See Error Message indication, p. 68,
 * for command description
*/
typedef struct rc_em_s {
    CCBccb;
    u32  code;
    CCBerr_ccb;
    u8    msg[4];                    /* Actually variable length          */
} RC_EM, * RC_EMP;

/*
 * RC_INFO_RESPONSE   See Report Statistics response, p. 69,
 * for command description
*/
typedef struct rc_infor_s {
    CCBccb;
    u32  tag;
    u8    firmware_id[16];
    u8    release_date[16];
    u32  buffer_ram_size;
    u32  static_ram_size;
} RC_INFOR, * RC_INFORP;

/*
 * RC_H2B_CHANNEL_CREATE_RESPONSE   See Report New Host-to-Board Channel, p. 71,
 * for command description
*/
typedef struct rc_h2b_channelcr_s {
    CCBccb;
    u32  tag;
    u32  channel_id;                 /* offset into shortio of new channel     */
} RC_H2B_CHANNELCR, * RC_H2B_CHANNELCRP;

/*
```

```
 * RC_B2H_CHANNEL_CREATE_RESPONSE   See Report New Board-to-Host Channel, p. 72,
 * for command description
*/
typedef struct rc_b2h_channelcr_s {
        CCBccb;
        u32  tag;
        u32  channel_id;                /* offset into shortio of new channel     */
        u32  vtag;                      /* virtual address of segment                */
        u32  length;                    /* length of segment                         */
} RC_B2H_CHANNELCR, * RC_B2H_CHANNELCRP;

/*
 *  RC_B2H_LINK_INDICATION   See Report Link to Next Segment, p. 74,
 * for command description
*/
typedef struct rc_b2h_link_s {
        CCBccb;
        u32  vtag;                      /* virtual tag from host                  */
        u32  length;                    /* Length of new segment                     */
} RC_B2H_LINK, * RC_B2H_LINKP;
```

This page is intentionally left blank.

INDEX