Loader Debugger Protocol

RFC-909

Christopher Welles

BBN Communications Corporation

Walter Milliken

BBN Laboratories

July 1984

Status of This Memo

This RFC specifies a proposed protocol for the ARPA Internet community, and requests discussion and suggestions for improvements. Distribution of this memo is unlimited.

Table of Contents

FIGURES

CHAPTER 1

Introduction

The Loader-Debugger Protocol (LDP) is an application layer protocol for loading, dumping and debugging target machines from hosts in a network environment. This protocol is designed to accommodate a variety of target cpu types. It provides a powerful set of debugging services. At the same time, it is structured so that a simple subset may be implemented in applications like boot loading where efficiency and space are at a premium.

The authors would like to thank Dan Franklin and Peter Cudhea for providing many of the ideas on which this protocol is based.

1.1 Purpose of This Document

This is a technical specification for the LDP protocol. It is intended to be comprehensive enough to be used by implementors of the protocol. It contains detailed descriptions of the formats and usage of over forty commands. Readers interested in an overview of LDP should read the Summary of Features, below, and skim Sections 2 through 3.1. Also see Appendix B, the Command Summary. The remainder of the document reads best when accompanied by strong coffee or tea.

1.2 Summary of Features

LDP has the following features:

- commands to perform loading, dumping and debugging
- support for multiple connections to a single target
- reliable performance in an internet environment
- a small protocol subset for target loaders
- addressing modes and commands to support multiple machine types
- breakpoints and watchpoints which run in the target machine.

CHAPTER 2

General Description

2.1 Motivation

LDP is an application protocol that provides a set of commands used by application programs for loading, dumping and debugging target machines across a network.

The goals of this protocol are shown in the following list:

- The protocol should support various processor types and operating systems. Overhead and complexity should be minimized for simpler cases.

- The protocol should provide support for applications in which more than one user can debug the same target machine. This implies an underlying transport mechanism that supports multiple connections between a host-target pair.

- LDP should have a minimal subset of commands for boot loading and dumping. Target machine implementations of these applications are often restricted in the amount of code-space they may take. The services needed for loading and dumping should be provided in a small, easily implemented set of commands.

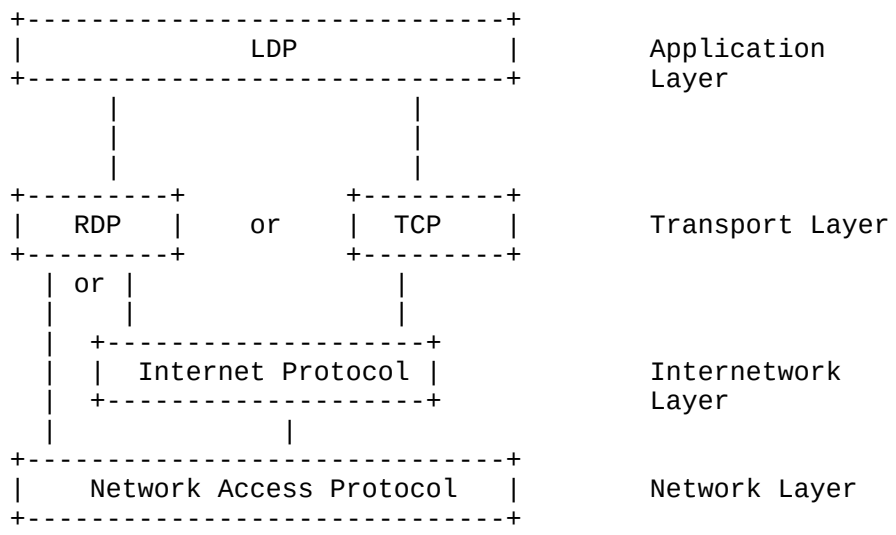
- There should be a means for communicating exceptions and errors from the target LDP process to the host process.

- LDP should allow the application to implement a full set of debugging functions without crippling the performance of the target's application (i.e., PSN, PAD, gateway). For example, a breakpoint mechanism that halts the target machine while breakpoint commands are sent from the host to the target is of limited usefulness, since the target will be unable to service the real-time

demands of its application.

2.2 Relation to Other Protocols

LDP is an application protocol that fits into the layered internet protocol environment. Figure 1 illustrates the place of LDP in the protocol hierarchy.



Relation to Other Protocols
Figure 1

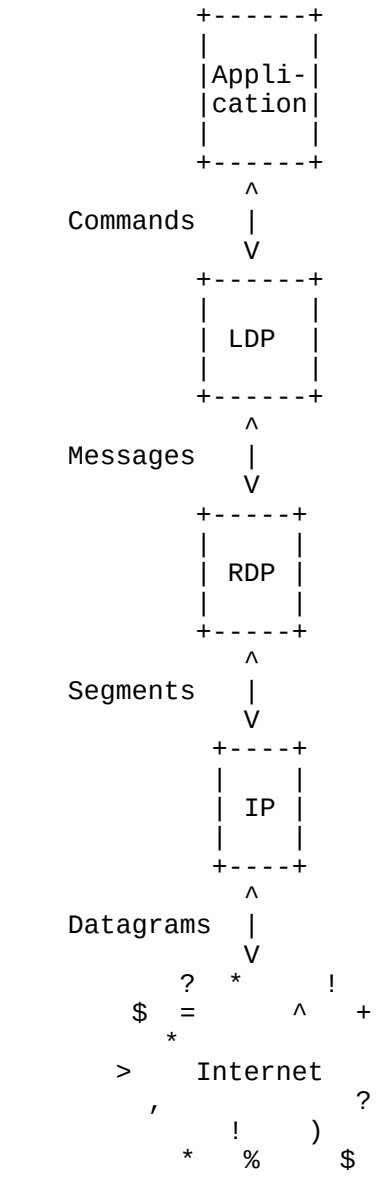
2.2.1 Transport Service Requirements

LDP requires that the underlying transport layer:

- allow connections to be opened by specifying a network (or internet) address. Support passive and active opens.
- for each connection, specify the maximum message size.
- provide a mechanism for sending and receiving messages over an open connection.
- deliver messages reliably and in sequence
- support multiple connections, and distinguish messages associated with different connections. This is only a requirement where LDP is expected to support several users at the same time.
- explictly return the outcome (success/failure) of each request (open, send, receive), and provide a means of querying the status of a connection (unacknowledged message count, etc.).

Data is passed from the application program to the LDP user process in the form of commands. In the case of an LDP server process, command responses originate in LDP itself. Below LDP is the transport protocol. The Reliable Data Protocol (RDP -- RFC 908) is the recommended transport procotol. Data is passed across the LDP/RDP interface in the form of messages. (TCP may be used in place of RDP, but it will be less efficient and it will require more resources to implement.) An internet layer (IP) normally comes between RDP and the network layer, but RDP may exchange data packets directly with the network layer.

Figure 2 shows the flow of data across the protocol interfaces:



Form of Data Exchange Between Layers
Figure 2

CHAPTER 3

Protocol Operation

3.1 Overview

An LDP session consists of an exchange of commands and responses between an LDP user process and an LDP server process. Normally, the user process resides on a host machine (a timesharing computer used for network monitoring and control), and the server process resides on a target machine (PSN, PAD, gateway, etc.). Throughout this document, host and target are used as synonyms for user process and server process, respectively, although in some implementations (the Butterfly, for example) this correspondence may be reversed. The host controls the session by sending commands to the target. Some commands elicit responses, and all commands may elicit an error reply.

The protocol contains five classes of commands: protocol, data transfer, management, control and breakpoint. Protocol commands are used to verify the command sequencing mechanism and to handle erroneous commands. Data transfer commands involve the transfer of data from one place to another, such as for memory examine/deposit, or loading. Management commands are used for creating and deleting objects (processes, breakpoints, watchpoints, etc.) in the target machine. Control commands are used to control the execution of target code and breakpoints. Breakpoint commands are used to control the execution of commands inside breakpoints and watchpoints.

3.2 Session Management

An LDP session consists of a series of commands sent from a host LDP to a target LDP, some of which may be followed by responses from the target. A session begins when a host opens a transport connection to a target listening on a well known port. LDP uses RDP port number zzz or TCP port number yyy. When the connection has been established, the host sends a HELLO command, and the target replies with a HELLO_REPLY. The HELLO_REPLY contains parameters that describe the target's implementation of LDP, including protocol version, implementation level, system

type, and address format. The session terminates when the host closes the underlying transport connection. When the target detects that the transport connection has been closed, it should deallocate any resources dedicated to the session.

The target process is the passive partner in an LDP session, and it waits for the host process to terminate the session. As an implementation consideration, either LDP or the underlying transport protocol in the target should have a method for detecting if the host process has died. Otherwise, an LDP target that supported only one connection could be rendered useless by a host that crashed in the middle of a session. The problem of detecting half-dead connections can be avoided by taking a different tack: the target could allow new connections to usurp inactive connections. A connection with no activity could be declared 'dead', but would not be usurped until the connection resource was needed. However, this would still require the transport layer to support two connection channels: one to receive connection requests, and another to use for an active connection.

3.3 Command Sequencing

Each command sent from the host to the target has a sequence number. The sequence number is used by the target to refer to the command in normal replies and error replies. To save space, these numbers are not actually included in host commands. Instead, each command sent from the host is assigned an implicit sequence number. The sequence number starts at zero at the beginning of the LDP session and increases by one for each command sent. The host and target each keep track of the current number. The SYNCH <sequence number> command may be used by the host to synchronize the sequence number.

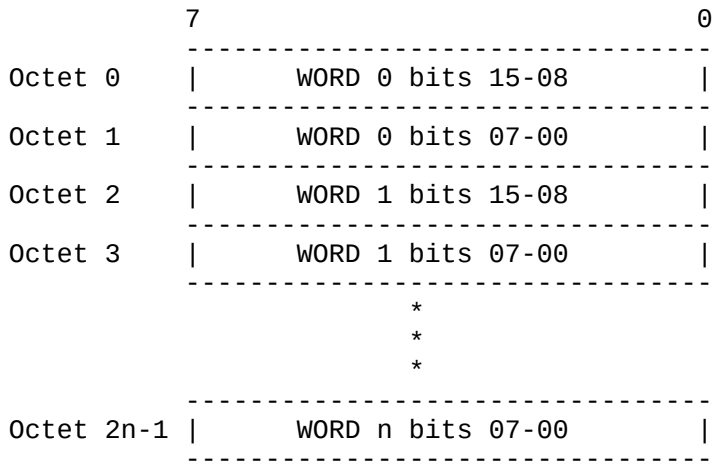
3.4 Data Packing and Transmission

The convention for the order of data packing was chosen for its simplicity: data are packed most significant bit first, in order of increasing target address, into eight-bit octets. The octets of packed data are transmitted in sequential order.

Data are always packed according to the address format of the target machine. For example, in an LDP session between a 20-bit host and a 16-bit target, 16-bit words (packed into octets) are transmitted in both directions. For ease of discussion, targets are treated here as if they have uniform address spaces. In practice, the size of address units may vary within a target -- 16-bit macromemory, 32-bit micromemory, 10-bit dispatch memory, etc. Data packing between host and target is tailored to the units of the current target address space.

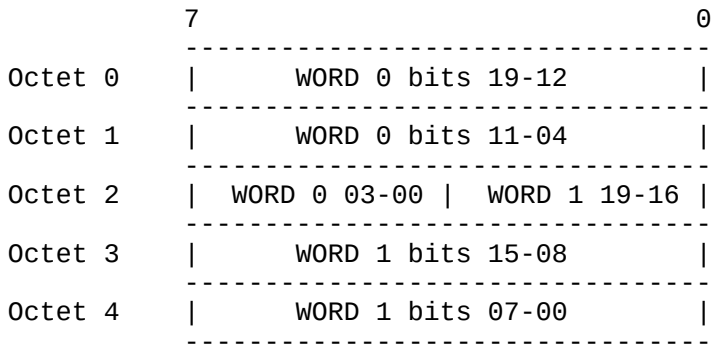
Figures showing the packing of data for targets with various address unit sizes are given below. The order of transmission with respect to the diagrams is top to bottom. Bit numbering in the following diagrams refers to significance in the octet: bit zero is the least significant bit in an octet. For an explanation of the bit numbering convention that applies in the rest of this document, please see Appendix A.

The packing of data for targets with word lengths that are multiples of 8 is straightforward. The following diagram illustrates 16-bit packing:



Packing of 16-bit Words
Figure 3

Packing for targets with peculiar word lengths is more complicated. For 20-bit machines, 2 words of data are packed into 5 octets. When an odd number of 20-bit words are transmitted, the partially used octet is included in the length of the command, and the octet is padded to the right with zeroes.



Packing of 20-bit Words
Figure 4

3.5 Implementations

A subset of LDP commands may be implemented in targets where machine resources are limited and the full capabilities of LDP are not needed. There are three basic levels of target implementations: LOADER_DUMPER, BASIC_DEBUGGER and FULL_DEBUGGER. The target communicates its LDP implementation level to the host during session initiation. The implementation levels are described below:

LOADER_DUMPER

Used for loading/dumping of the target machine. Includes all protocol class commands and replies; data transfer commands READ, WRITE, MOVE and their responses; control command START and control reply EXCEPTION. Understands at least PHYS_MACRO and HOST addressing modes; others if desired.

BASIC_DEBUGGER

Implements LOADER_DUMPER commands, all control commands, all addressing modes appropriate to the target machine, but does not have finite state machine (FSM) breakpoints or watchpoints. Default breakpoints are implemented. The target understands long addressing mode.

FULL_DEBUGGER

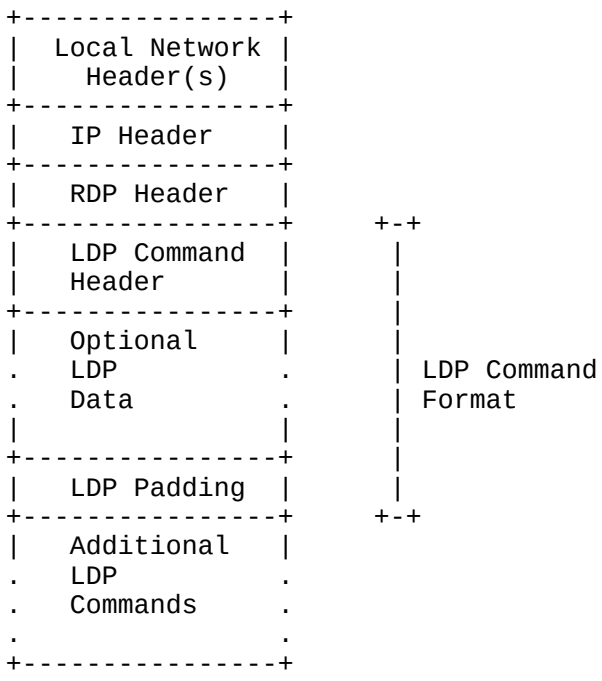
Implements all commands and addressing modes appropriate to the target machine, and includes breakpoint commands, conditional commands and BREAKPOINT_DATA. Watchpoints are optional.

CHAPTER 4

Commands and Formats

4.1 Packet Format

LDP commands are enclosed in RDP transport messages. An RDP message may contain more than one command, but each command must fit entirely within a single message. Network packets containing LDP commands have the format shown in Figure 5.



Network Packet Format
Figure 5

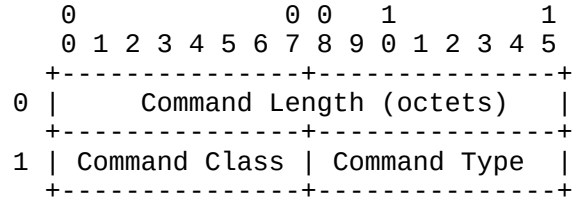
4.2 Command Format

LDP commands consist of a standard two-word header followed optionally by additional data. To facilitate parsing of multi-command messages, all commands contain an even number of octets. Commands that contain an odd number of data octets must be padded with a null octet.

The commands defined by the LDP specification are intended to be of universal application to provide a common basis for all implementations. Command class and type codes from 0 to 63. are reserved by the protocol. Codes above 63. are available for the implementation of target-specific commands.

4.2.1 Command Header

LDP commands begin with a fixed length header. The header specifies the type of command and its length in octets.



LDP Command Header Format
Figure 6

HEADER FIELDS:

Command Length

The command length gives the total number of octets in the command, including the length field and data, and excluding padding.

Command Class
Command Type

The command class and type together specify a particular command. The class selects one of six command categories, and the type gives the command within that category. All codes are decimal. The symbols given in Figures 7 and 8 for command classes and types are used in the remainder of this document for reference.

The command classes that have been defined are:

Command Class	Symbol
1	PROTOCOL
2	DATA_TRANSFER
3	CONTROL
4	MANAGEMENT
5	BREAKPOINT
6	CONDITION
7 - 63	<reserved>

Command Classes
Figure 7

Command type codes are assigned in order of expected frequency of use. Commands and their responses/replies are numbered sequentially. The command types, ordered by command class, are:

Command Class	Command Type	Symbol
PROTOCOL	1	HELLO
	2	HELLO_REPLY
	3	SYNCH
	4	SYNCH_REPLY
	5	ERROR
	6	ERRACK
	7	ABORT
	8	ABORT_DONE
	9 - 63	<reserved>
DATA_TRANSFER	1	WRITE
	2	READ
	3	READ_DONE
	4	READ_DATA
	5	MOVE
	6	MOVE_DONE
	7	MOVE_DATA
	8	REPEAT_DATA
	9	BREAKPOINT_DATA
	10	WRITE_MASK
	11 - 63	<reserved>
CONTROL	1	START
	2	STOP
	3	CONTINUE
	4	STEP
	5	REPORT
	6	STATUS
	7	EXCEPTION
	8 - 63	<reserved>
MANAGEMENT	1	CREATE
	2	CREATE_DONE
	3	DELETE
	4	DELETE_DONE
	5	LIST_ADDRESSES
	6	ADDRESS_LIST
	7	GET_PHYS_ADDRESS
	8	GOT_PHYS_ADDRESS
	9	GET_OBJECT
	10	GOT_OBJECT
	11	LIST_BREAKPOINTS
	12	BREAKPOINT_LIST

	13	LIST_NAMES
	14	NAME_LIST
	15	LIST_PROCESSES
	16	PROCESS_LIST
	17 - 63	<reserved>
BREAKPOINT	1	INCREMENT
	2	INC_COUNT
	3	OR
	4	SET_PTR
	5	SET_STATE
	6 - 63	<reserved>
CONDITION	1	CHANGED
	2	COMPARE
	3	COUNT_EQ
	4	COUNT_GT
	5	COUNT_LT
	6	TEST
	7 - 63	<reserved>

Command Types
Figure 8

4.3 Addressing

Addresses are used in LDP commands to refer to memory locations, processes, buffers, breakpoints and other entities. Many of these entities are machine-dependent; some machines have named objects, some machines have multiple address spaces, the size of address spaces varies, etc. The format for specifying addresses needs to be general enough to handle all of these cases. This speaks for a large, hierarchically structured address format. However, the disadvantage of a large format is that it imposes extra overhead on communication with targets that have simpler address schemes.

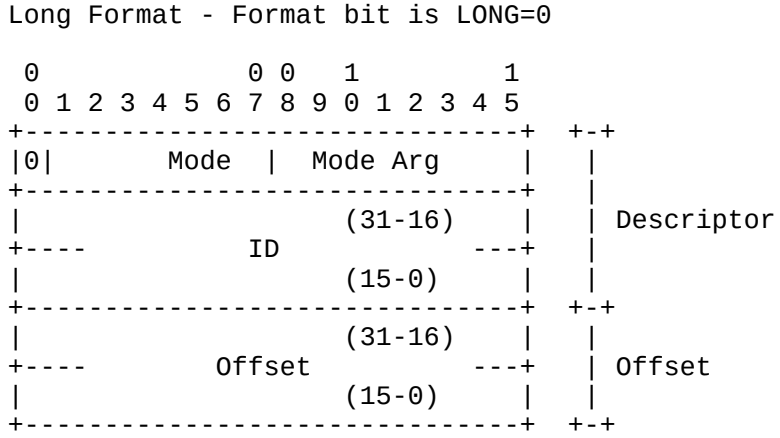
LDP resolves this conflict by employing two address formats: a short three-word format for addressing simpler targets, and a long five-word format for others. Each target LDP is required to implement at least one of these formats. At the start of an LDP session, the target specifies the address format(s) it uses in

the Flag field of the HELLO_REPLY message. In each address, the first bit of the mode octet is a format flag: 0 indicates LONG address format, and 1 indicates SHORT format.

4.3.1 Long Address Format

The long address format is five words long and consists of a three-word address descriptor and a two-word offset (see Figure 9). The descriptor specifies an address space to which the offset is applied. The descriptor is subdivided into several fields, as described below. The structuring of the descriptor is designed to support complex addressing modes. For example, on targets with multiple processes, descriptors may reference virtual addresses, registers, and other entities within a particular process.

The addressing modes defined below are intended as a base to which target-specific modes may be added. Modes up to 63. are reserved by the protocol. The range 64. to 127. may be used for target-specific address modes.



Long Address Format
Figure 9

LONG ADDRESS FIELDS:

Mode

The address mode identifies the type of address space being referenced. The mode is qualified by the mode argument and the ID field. Implementation of modes other than physical and host is machine-dependent. Currently defined modes and the address space they reference are shown in Figure 10.

Mode	Symbol	Address space
0	HOST	Host
1	PHYS_MACRO	Macromemory
2	PHYS_MICRO	Micromemory
3	PHYS_I/O	I/O space
4	PHYS_MACRO_PTR	Macro contains a pointer
5	PHYS_REG	Register
6	PHYS_REG_OFFSET	Register plus offset
7	PHYS_REG_INDIRECT	Register contains address of a pointer
8	PROCESS_CODE	Process code space
9	PROCESS_DATA	Process data space
10	PROCESS_DATA_PTR	Process data contains a ptr
11	PROCESS_REG	Process virtual register
12	PROCESS_REG_OFFSET	Process register plus offset
13	PROCESS_REG_INDIRECT	Process register contains address of a pointer
14	OBJECT_OFFSET	Memory object (queue, pool)
15	OBJECT_HEADER	System header for an object
16	BREAKPOINT	Breakpoint
17	WATCHPOINT	Watchpoint
18	BPT_PTR_OFFSET	Breakpoint ptr plus offset
19	BPT_PTR_INDIRECT	Breakpoint ptr plus offset gives address of a pointer
20 - 63	<reserved>	

Long Address Modes
Figure 10

Mode Argument

Provides a numeric argument to the mode field. Specifies the register in physical and process REG and REG_OFFSET modes.

ID Field

Identifies a particular process, buffer or object.

Offset

The offset into the linear address space defined by the mode. The size of the machine word determines the number of significant bits in the offset. Likewise, the addressing units of the target are the units of the offset.

The interpretation of the mode argument, ID field and offset for each address mode is given below:

HOST

The ID and offset fields are numbers assigned arbitrarily by the host side of the debugger. These numbers are used in MOVE and MOVE_DATA messages. MOVE_DATA responses containing this mode as the destination are sent by the target to the host. This may occur in debugging when data is sent to the host from the target breakpoint.

PHYS_MACRO

The offset contains the 32-bit physical address of a location in macromemory. The mode argument and ID field are not used. For example, mode=PHYS_MACRO and offset=1000 specifies location 1000 in physical memory.

PHYS_MICRO

Like PHYS_MACRO, but the location is in micromemory.

PHYS_I/O

Like PHYS_MACRO, but the location is in I/O space.

PHYS_MACRO_PTR

The offset contains the address of a pointer in macromemory. The location pointed to (the effective address) is also in macromemory. The mode argument and ID field are unused.

PHYS_REG

The mode argument gives the physical register. If the register is used by the LDP target process, then the saved copy from the previous context is used. This comment applies to PHYS_REG_OFFSET mode as well. The ID field is not used.

PHYS_REG_OFFSET

The offset is added to the contents of a register given as the mode argument. The result is used as a physical address in macromemory. ID is unused.

PHYS_REG_INDIRECT

The register specified in the mode arg contains the address of a pointer in macromemory. The effective address is the macromemory location specified in the pointer, plus the offset. The ID field is unused.

PROCESS_CODE

The ID is a process ID, the offset is into the code space for this process. Mode argument is not used.

PROCESS_DATA

The ID is a process ID, the offset is into the data space for this process. Mode argument is not used. On systems that do not distinguish between code and data space, these two modes are equivalent, and reference the virtual address space of the process.

PROCESS_DATA_PTR

The offset contains the address of a pointer in the data space of the process specified by the ID. The location pointed to (the effective address) is also in the data space. The mode argument is not used.

PROCESS_REG

Accesses the registers (and other system data) of the process given by the ID field. Mode argument 0 starts the registers. After the registers, the mode argument is an offset into the system area for the process.

PROCESS_REG_OFFSET

The offset plus the contents of the register given in the mode argument specifies a location in the data space of the process specified by the ID.

PROCESS_REG_INDIRECT

The register specified in the mode arg contains the address of a pointer in the data space of the process given by the ID. The effective address is the location in process data space specified in the pointer, plus the offset.

OBJECT_OFFSET (optional)

The offset is into the memory space defined by the object ID in ID. Recommended for remote control of parameter segments.

OBJECT_HEADER (optional)

The offset is into the system header for the object specified by the ID. Intended for use with the Butterfly.

BREAKPOINT

The descriptor specifies a breakpoint. The offset is never used, this type is only used in descriptors referring to breakpoints. (See Breakpoints and Watchpoints, below, for an explanation of breakpoint descriptors.)

WATCHPOINT

The descriptor specifies a watchpoint. The offset is never used, this type is only used in descriptors referring to watchpoints. (See Breakpoints and Watchpoints, below, for an explanation of watchpoint descriptors).

BPT_PTR_OFFSET

For this mode and BPT_PTR_INDIRECT, the mode argument specifies one of two breakpoint pointer variables local to the breakpoint in which this address occurs. These pointers and the SET_PTR command which manipulates them provide for an arbitrary amount of address indirection. They are intended for use in traversing data structures: for example, chasing queues. In BPT_PTR_OFFSET, the offset is added to

the pointer variable to give the effective address. In targets which support multiple processes, the location is in the data space of the process given by the ID. Otherwise, the location is a physical address in macro-memory. BPT_PTR.* modes are valid only in breakpoints and watchpoints.

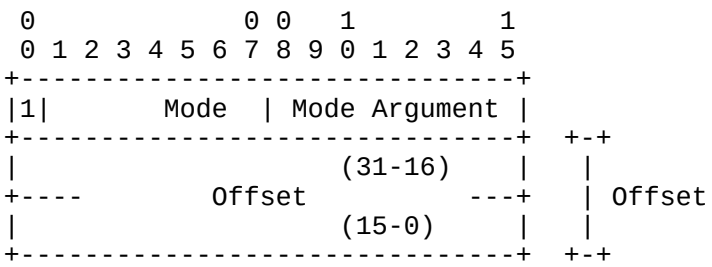
BPT_PTR_INDIRECT

Like BPT_PTR_OFFSET, except that it uses one more level of indirection. The pointer variable given by the mode argument plus the offset specify an address which points to the effective address. See the description of BPT_PTR_OFFSET for a discussion of usage, limitations and address space.

4.3.2 Short Address Format

The short address format is intended for use in implementations where protocol overhead must be minimized. This format is a subset of the long address format: it contains the same fields except for the ID field. Therefore, the short addressing format supports only HOST and PHYS_* address modes. Only the LOADER_DUMPER implementation level commands may be used with the short addressing format. The short address format is three words long, consisting of a 16-bit word describing the address space, and a 32-bit offset.

Short Format - Format bit is SHORT=1



Short Address Format
Figure 11

SHORT ADDRESS FIELDS:
Mode

The high-order bit is 1, indicating the short address format. A list of the address modes supported is given below. The interpretation of the remaining fields is as described above for the long addressing format.

Mode	Symbol	Address space
0	HOST	Host
1	PHYS_MACRO	Macro-memory
2	PHYS_MICRO	Micro-memory
3	PHYS_I/O	I/O space
4	PHYS_MACRO_PTR	Macro contains a pointer
5	PHYS_REG	Register
6	PHYS_REG_OFFSET	Register plus offset
7	PHYS_REG_INDIRECT	Register contains address of a pointer
8 - 32	<reserved>	

Short Address Modes
Figure 12

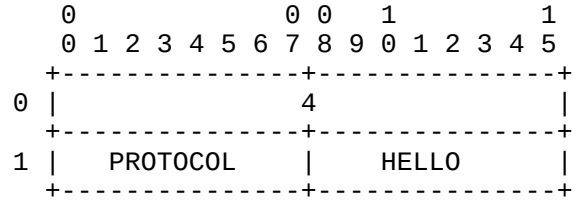
CHAPTER 5

Protocol Commands

Protocol commands are used for error handling, for synchronizing the command sequence number, and for communicating protocol implementation parameters. Every protocol command has a corresponding reply. All protocol commands are sent from the host to the target, with replies flowing in the opposite direction.

5.1 HELLO Command

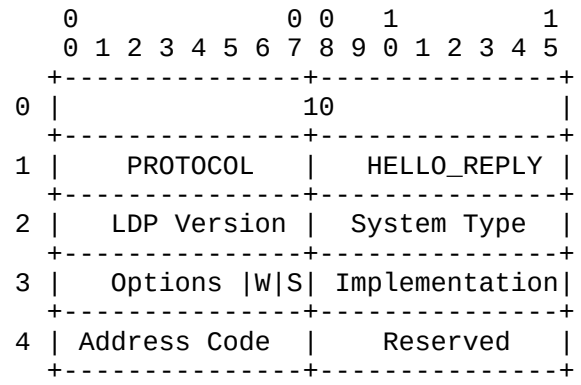
The HELLO command is sent by the host to signal the start of an LDP session. The target responds with HELLO_REPLY.



HELLO Command Format
Figure 13

5.2 HELLO_REPLY

A HELLO_REPLY is sent by the target in response to the HELLO command at the start of an LDP session. This reply is used to inform the host about the target’s implementation of LDP.



HELLO_REPLY Format
Figure 14

HELLO_REPLY FIELDS:

LDP Version

The target's LDP protocol version. If the current host protocol version does not agree with the target's protocol version, the host may terminate the session, or may continue it, at the discretion of the implementor. The current version number is 2.

System Type

The type of system running on the target. This is used as a check against what the host thinks the target is. The host is expected to have a table of target system types with information about target address spaces, target-specific commands and addressing modes, and so forth.

Currently defined system types are shown in Figure 15. This list includes some systems normally thought of as 'hosts' (e.g. C70, VAX), for implementations where targets actively initiate and direct a load of themselves.

Code	System	Description
1	C30_16_BIT	BBN 16-bit C30
2	C30_20_BIT	BBN 20-bit C30
3	H316	Honeywell-316
4	BUTTERFLY	BBN Butterfly
5	PDP-11	DEC PDP-11
6	C10	BBN C10
7	C50	BBN C50
8	PLURIBUS	BBN Pluribus
9	C70	BBN C70
10	VAX	DEC VAX
11	MACINTOSH	Apple MacIntosh

System Types
Figure 15

Address Code

The address code indicates which LDP address format(s) the target is prepared to use. Address codes are show in Figure 16.

Address Code	Symbol	Description
1	LONG_ADDRESS	Five word address format. Supports all address modes and commands.
2	SHORT_ADDRESS	Three word address format. Supports only physical and host address modes. Only the LOADER_DUMPER set of commands are supported.

Target Address Codes
Figure 16

Implementation

The implementation level specifies which features of the protocol are implemented in the target. There are three levels of protocol implementation. These levels are intended to correspond to the three most likely applications of LDP: simple loading and dumping, basic debugging, and full debugging. (Please see Implementations, above, for a detailed description of implementation levels.) There are are also several optional features that are not included in any particular level.

Implementation levels are cumulative, that is, each higher level includes the features of all previous levels. The levels are shown in Figure 17.

Feature Level	Symbol	Description
1	LOADER_DUMPER	Loader/dumper subset of LDP
2	BASIC_DEBUGGER	Control commands, CREATE
3	FULL_DEBUGGER	FSM breakpoints

Feature Levels
Figure 17

Options

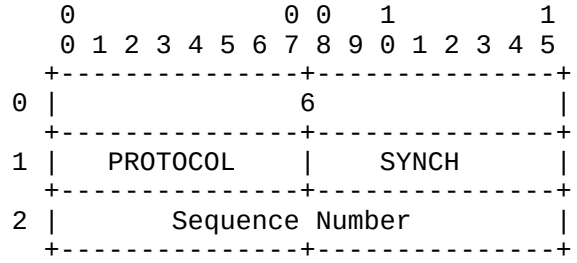
The options field (see Figure 18) is an eight-bit flag field. Bit flags are used to indicate if the target has implemented particular optional commands. Not all optional commands are referenced in this field. Commands whose implementation depends on target machine features are omitted. The LDP application is expected to 'know' about target features that are not intrinsic to the protocol. Examples of target-dependent commands are commands that refer to named objects (CREATE, LIST_NAMES).

Mask	Symbol	Description
1	STEP	The STEP command is implemented
2	WATCHPOINTS	Watchpoints are implemented

Options
Figure 18

5.3 SYNCH Command

The SYNCH command is sent by the host to the target. The target responds with a SYNCH_REPLY. The SYNCH - SYNCH_REPLY exchange serves two functions: it synchronizes the host-to-target implicit sequence number and acts as a cumulative acknowledgement of the receipt and execution of all host commands up to the SYNCH.



SYNCH Command Format
Figure 19

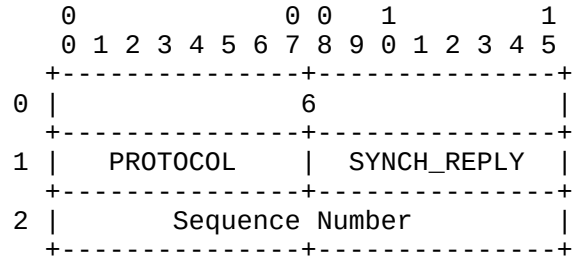
SYNCH FIELDS:

Sequence Number

The sequence number of this command. If this is not what the target is expecting, the target will reset to it and respond with an ERROR reply.

5.4 SYNCH_REPLY

A SYNCH_REPLY is sent by the target in reponse to a valid SYNCH command. A SYNCH command is valid if its sequence number agrees with the sequence number the target is expecting. Otherwise, the target will reset its sequence number to the SYNCH command and send an ERROR reply.



SYNCH_REPLY Format
Figure 20

SYNCH_REPLY FIELDS:

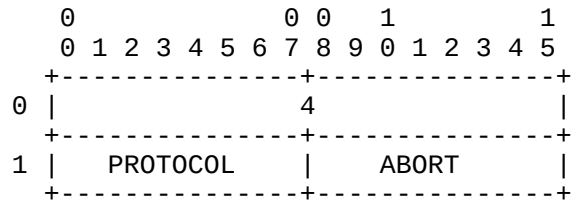
Sequence Number

The sequence number of the SYNCH command to which this SYNCH_REPLY is the response.

5.5 ABORT Command

The ABORT command is sent from the host to abort all pending operations at the target. The target responds with ABORT_DONE. This is primarily intended to stop large data transfers from the target. A likely application would be during a debugging session when the user types an interrupt to abort a large printout of data from the target. The ABORT command has no effect on any breakpoints or watchpoints that may be enabled in the target.

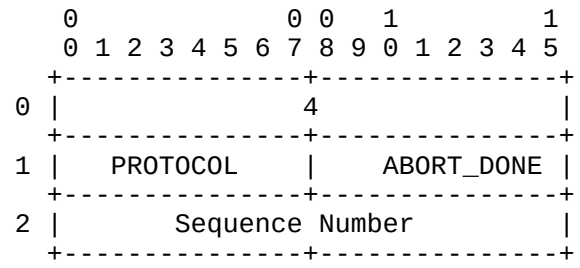
As a practical matter, the ABORT command may be difficult to implement on some targets. Its ability to interrupt command processing on the target depends on the target being able to look ahead at incoming commands and receive an out-of-band signal from the host. However, the effect of an ABORT may be achieved by simply closing and reopening the transport connection.



ABORT Command Format
Figure 21

5.6 ABORT_DONE Reply

The ABORT_DONE reply is sent from the target to the host in response to an ABORT command. This indicates that the target has terminated all operations that were pending when the ABORT command was received. The sequence number of the ABORT command is included in the reply.



ABORT_DONE Reply Format
Figure 22

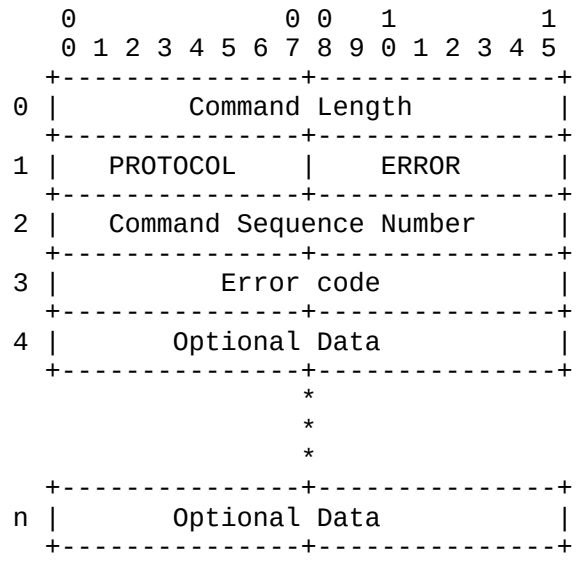
ABORT_DONE FIELDS:

Sequence Number

The sequence number of the ABORT command that elicited this reply. This enables the host to distinguish between replies to multiple aborts.

5.7 ERROR Reply

The ERROR reply is sent by the target in response to a bad command. The ERROR reply gives the sequence number of the offending command and a reason code. The target ignores further commands until an ERRACK command is received. The reason for ignoring commands is that the proper operation of outstanding commands may be predicated on the execution of the erroneous command.



ERROR Reply Format
Figure 23

ERROR Reply FIELDS:

Command Sequence Number

The implicit sequence number of the erroneous command.

Error Code

A code specifying what error has taken place. The currently defined codes are shown in Figure 24.

Error Code	Symbol
1	BAD_COMMAND
2	BAD_ADDRESS_MODE
3	BAD_ADDRESS_ID
4	BAD_ADDRESS_OFFSET
5	BAD_CREATE_TYPE
6	NO_RESOURCES
7	NO_OBJECT
8	OUT_OF_SYNCH
9	IN_BREAKPOINT

ERROR Codes
Figure 24

An explanation of each of these error codes follows:

BAD_COMMAND

The command was not meaningful to the target machine. This includes commands that are valid but unimplemented in this target. Also, the command was not valid in this context. For example, a command given by the host that is only legal in a breakpoint (e.g. IF, SET_STATE).

BAD_ADDRESS_MODE <offending-address>

The mode of an address given in the command is not meaningful to this target system. For example, a PROCESS address mode on a target that does not support multi-processing.

BAD_ADDRESS_ID <offending-address>

The ID field of an address didn't correspond to an appropriate thing. For example, for a PROCESS address mode, the ID of a non-existent process.

BAD_ADDRESS_OFFSET <offending-address>

The offset field of the address was outside the legal range for the thing addressed. For example, an offset of 200,000 in PHYS_MACRO mode on a target with 64K of

macro-memory.

BAD_CREATE_TYPE

The object type in a CREATE command was unknown.

NO_RESOURCES

A CREATE command failed due to lack of necessary resources.

NO_OBJECT

A GET_OBJECT command failed to find the named object.

OUT_OF_SYNCH

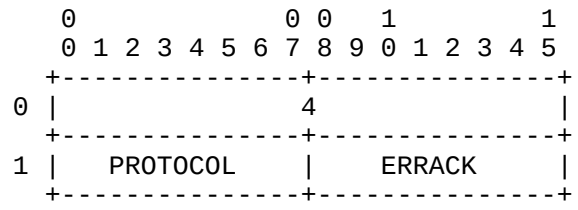
The sequence number of the SYNCH command was not expected by the target. The target has resynchronized to it.

IN_BREAKPOINT <breakpoint-descriptor> <breakpoint-sequence#> <reason-code> [<optional-info>]

An error occurred within a breakpoint command list. The given 16-bit sequence-number refers to the sequence number of the CREATE command that created the breakpoint, while breakpoint-sequence# refers to the sequence number of the command within the breakpoint given by <breakpoint-descriptor>.

5.8 ERRACK Acknowledgement

An ERRACK is sent by the host in response to an ERROR reply from the target. The ERRACK is used to acknowledge that the host has received the ERROR reply.



ERRACK Command Format
Figure 25

CHAPTER 6

Data Transfer Commands

Data transfer commands transfer data between the host and the target. These commands are used for loading and dumping the target, and examining and depositing locations on the target. The READ command reads data from the target, the MOVE command moves data within the target or from the target to another entity, and the WRITE command writes data to the target. REPEAT_DATA makes copies of a pattern to the target -- it is useful for zeroing memory. WRITE_MASK writes data with a mask, and is intended for modifying target parameter tables.

Data transmitted to and from the target always contains a target address. In writes to the target, this is used as the destination of the data. In reads from the target, the target address is used by the host to identify where in the target the data came from. In addition, the MOVE command may contain a 'host' address as its destination; this permits the host to further discriminate between possible sources of data from the target -- from different breakpoints, debugging windows, etc.

A read request to the target may generate one or more response messages. In particular, responses to requests for large amounts of data -- core dumps, for example -- must be broken up into multiple messages, if the block of data requested plus the LDP header exceeds the transport layer message size.

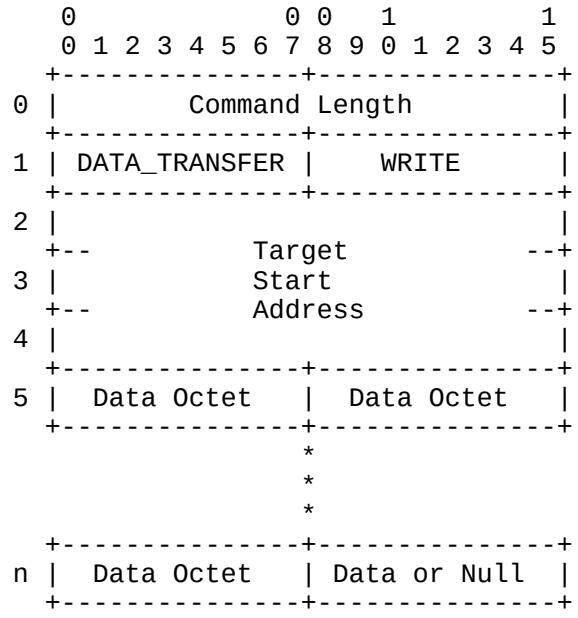
In commands which contain data (WRITE, READ_DATA, MOVE_DATA and REPEAT_DATA), if there are an odd number of data octets, then a null octet is appended. This is so that the next command in the message, if any, will begin on an even octet. The command length is the sum of the number of octets in the command header and the number of octets of data, excluding the null octet, if any.

The addressing formats which may be used with data transfer commands are specified for each LDP session at the start of the session by the target in the HELLO_REPLY response. See the section entitled 'Addressing', above, for a description of LDP addressing formats and modes. In the command diagrams given below, the short addressing format is illustrated. For LDP sessions using long addressing, addresses are five words long,

instead of three words, as shown here. In both addressing modes, descriptors are three words and offsets are two words.

6.1 WRITE Command

The WRITE command is used to send octets of data from the host to the target. This command specifies the address in the target where the data is to be stored, followed by a stream of data octets. If the data stream contains an odd number of octets, then a null octet is appended so that the next command, if any, will begin on an even octet. Since LDP must observe message size limitations imposed by the underlying transport layer, a single logical write may need to be broken up into multiple WRITEs in separate transport messages.



WRITE Command Format
Figure 26

WRITE FIELDS:

Command Length

The command length gives the number of octets in the command, including data octets, but excluding the padding octet, if any.

Target Start Address

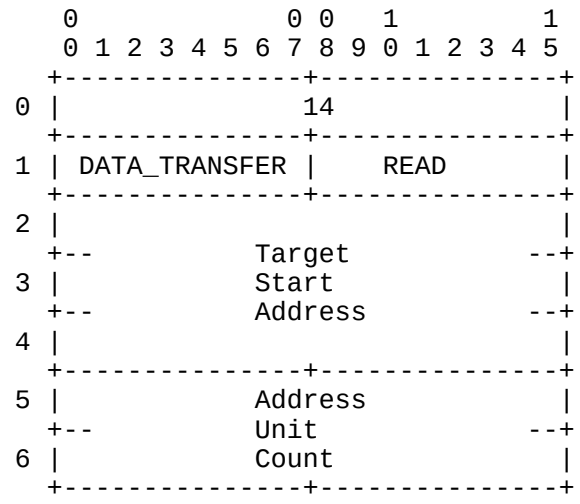
This is the address to begin storing data in the target. The length of the data to be stored may be inferred by the target from the command length. An illegal address or range will generate an ERROR reply.

Data Octets

Octets of data to be stored in the target. Data are packed according to the packing convention described above. Ends with a null octet if there are an odd number of data octets.

6.2 READ Command

The host uses the READ command to ask the target to send back a contiguous block of data. The data is specified by a target starting address and a count. The target returns the data in one or more READ_DATA commands, which give the starting address (in the target) of each segment of returned data. When the transfer is completed, the target sends a READ_DONE command to the host.



READ Command Format
Figure 27

READ FIELDS:

Target Start Address

The starting address of the requested block of target data. The target sends an ERROR reply if the starting address is illegal, if the ending address computed from the sum of the start and the count is illegal, or if holes are encountered in the middle of the range.

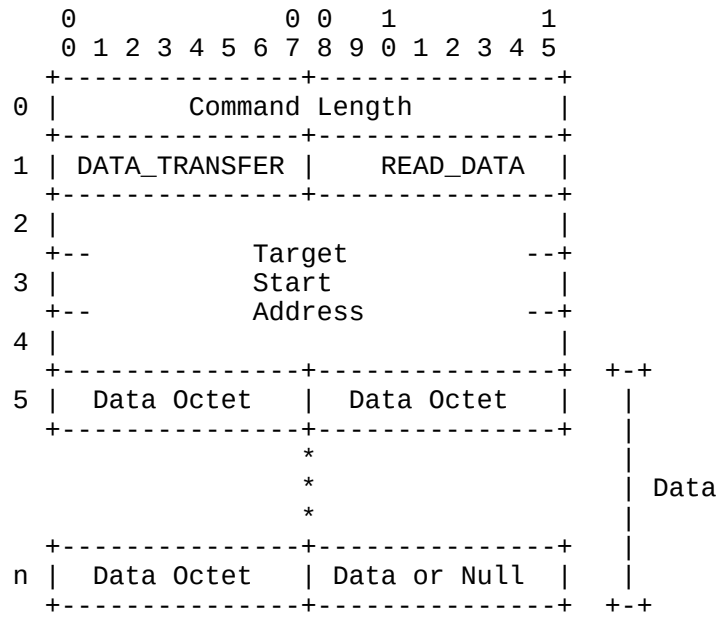
Address Unit Count

The count of the number of target indivisibly-addressable units to be transferred. For example, if the address space is PHYS_MACRO, a count of two and a start address of 1000 selects the contents of locations 1000 and 1001. 'Count' is used instead of 'length' to avoid the problem of determining units the length should be denominated in (octets, words, etc.). The size and type of the unit will vary depending on the address space selected by the target start address. The target should reply with an error (if it is able to

determine in advance of a transfer) if the inclusive range of addresses specified by the start address and the count contains an illegal or nonexistent address.

6.3 READ_DATA Response

The target uses the READ_DATA response to transmit data requested by a host READ command. One or more READ_DATA responses may be needed to fulfill a given READ command, depending on the size of the data block requested and the transport layer message size limits. Each READ_DATA response gives the target starting address of its segment of data. If the response contains an odd number of data octets, the target ends the response with a null octet.



DATA Response Format
Figure 28

READ_DATA FIELDS:

Command Length

The command length gives the number of octets in the command, including data octets, but excluding the padding octet, if any. The host can calculate the length of the data by subtracting the header length from the command length. Since the target address may be either three words (short format) or five words (long format), the address mode must be checked to determine which is being used.

Target Start Address

This is the starting address of the data segment in this message. The host may infer the length of the data from the command length. The address format (short or long) is the

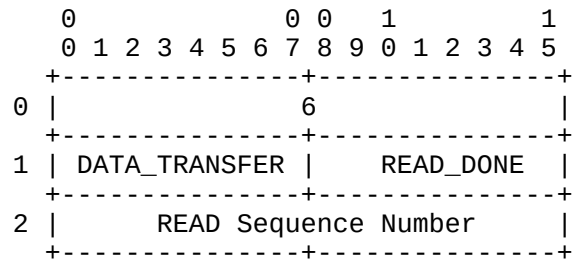
same as on the initial READ command.

Data Octets

Octets of data from the target. Data are packed according to the packing convention described above. Ends with a null octet if there are an odd number of data octets.

6.4 READ_DONE Reply

The target sends a READ_DONE reply to the host after it has finished transferring the data requested by a READ command. READ_DONE specifies the sequence number of the READ command.



READ_DONE Reply Format
Figure 29

READ_DONE FIELDS:

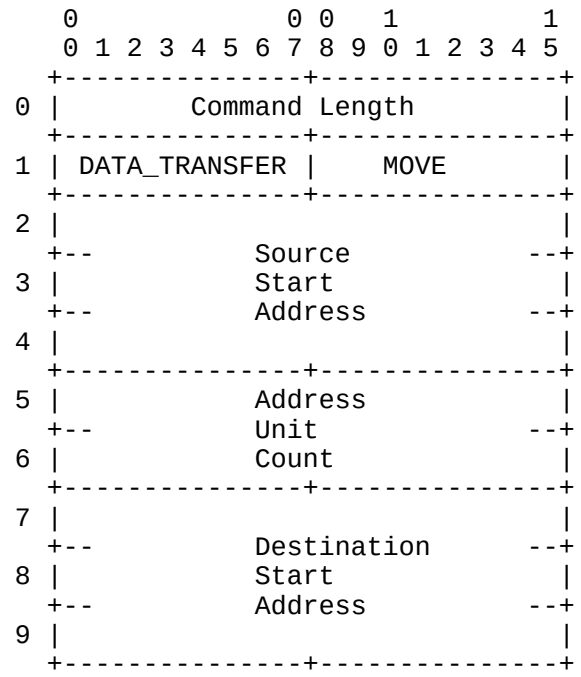
READ Sequence Number

The sequence number of the READ command this is a reply to.

6.5 MOVE Command

The MOVE command is sent by the host to move a block of data from the target to a specified destination. The destination address may specify a location in the target, in the host, or in another target (for loading one target from another). The data is specified by a target starting address and an address unit count. The target sends an ERROR reply if the starting address is illegal, if the ending address computed from the sum of the start and the count is illegal, or if holes are encountered in the middle of the range. If the MOVE destination is off-target, the target moves the data in one or MOVE_DATAs. Other commands arriving at the target during the transfer should be processed in a timely fashion, particularly the ABORT command. When the data has been moved, the target sends a MOVE_DONE to the host. However, a MOVE within a breakpoint will not generate a MOVE_DONE.

A MOVE with a host destination differs from a READ in that it contains a host address. This field is specified by the host in the MOVE command and copied by the target into the responding MOVE_DATA(s). The address may be used by the host to differentiate data returned from multiple MOVE requests. This information may be useful in breakpoints, in multi-window debugging and in communication with targets with multiple processors. For example, the host sends the MOVE command to the target to be executed during a breakpoint. The ID field in the host address might be an index into a host breakpoint table. When the breakpoint executes, the host would use the ID to associate the returning MOVE_DATA with this breakpoint.



MOVE Command Format
Figure 30

MOVE FIELDS:

Source Start Address

The starting address of the requested block of target data. An illegal address type will generate an error reply.

Address Unit Count

The count of the number of target indivisibly-addressable units to be transferred. For example, if the address space is PHYS_MACRO, a count of two and a start address of 1000 selects the contents of locations 1000 and 1001. 'Count' is used instead of 'length' to avoid the problem of determining units the length should be denominated in (octets, words,

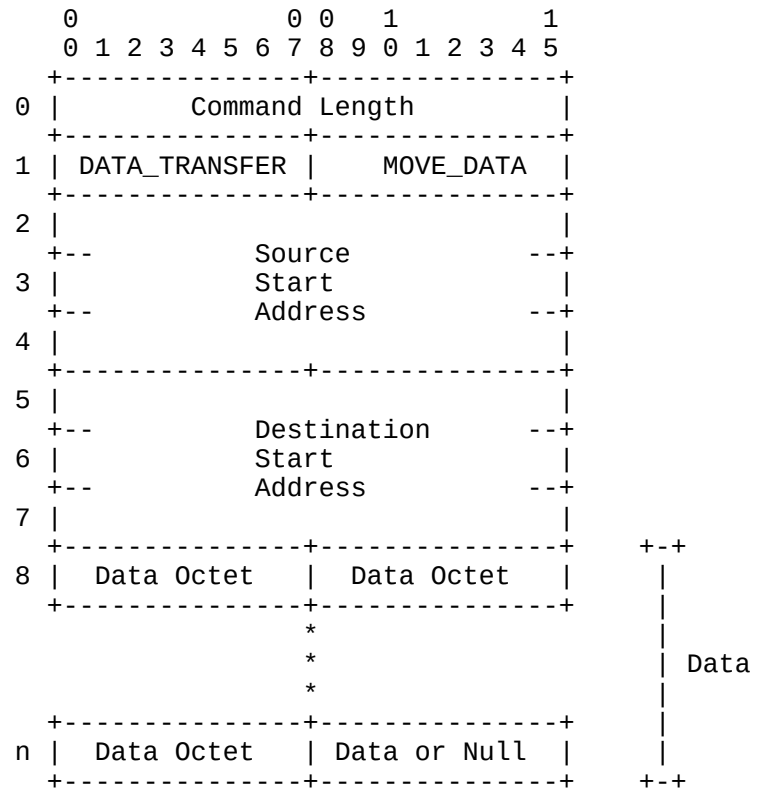
etc.). The size and type of the unit will vary depending on the address space selected by the target start address. The target should reply with an error (if it is able to determine in advance of a transfer) if the inclusive range of addresses specified by the start address and the count contains an illegal or nonexistent address.

Destination Address

The destination of the MOVE. If the address space is on the target, the address unit size should agree with that of the source address space. If the address mode is HOST, the values and interpretations of the remaining address fields are arbitrary, and are determined by the host implementation. For example, the mode argument might specify a table (breakpoint, debugging window, etc.) and the ID field an index into the table.

6.6 MOVE_DATA Response

The target uses the MOVE_DATA responses to transmit data requested by a host MOVE command. One or more MOVE_DATA responses may be needed to fulfill a given MOVE command, depending on the size of the data block requested and the transport layer message size limits. Each MOVE_DATA response gives the target starting address of its segment of data. If the response contains an odd number of data octets, the target should end the response with a null octet.



MOVE_DATA Response Format
Figure 31

MOVE_DATA FIELDS:

Command Length

The command length gives the number of octets in the command, including data octets, but excluding the padding octet, if any.

Source Start Address

This is the starting address of the data segment in this

message. The host may infer length of the data from the command length.

Destination Address

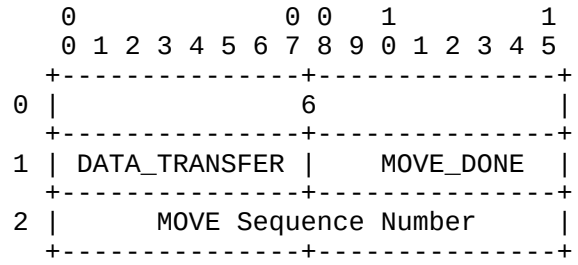
The destination address copied from the MOVE command that initiated this transfer. In the case of HOST MOVEs, this is used by the host to identify the source of the data.

Data Octets

Octets of data from the target. Data are packed according to the packing convention described above. Ends with a null octet if there are an odd number of data octets.

6.7 MOVE_DONE Reply

The target sends a MOVE_DONE reply to the host after it has finished transferring the data requested by a MOVE command. MOVE_DONE specifies the sequence number of the MOVE command.



MOVE_DONE Reply Format
Figure 32

MOVE_DONE FIELDS:

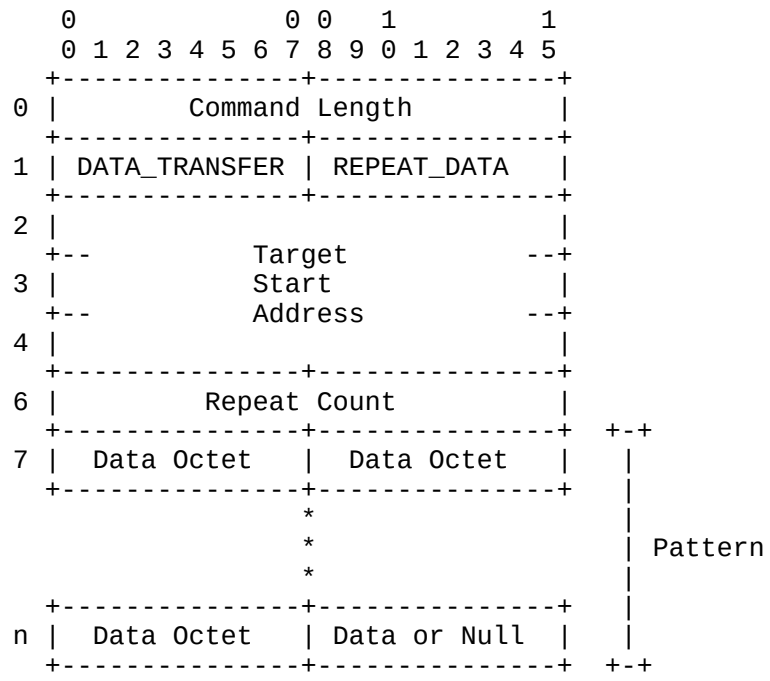
MOVE Sequence Number

The sequence number of the MOVE command this is a reply to.

6.8 REPEAT_DATA

The REPEAT_DATA command is sent by the host to write copies of a specified pattern into the target. This provides an efficient way of zeroing target memory and initializing target data structures. The command specifies the target starting address, the number of copies of the pattern to be made, and a stream of octets that constitutes the pattern.

This command differs from the other data transfer commands in that the effect of a REPEAT_DATA with a large pattern cannot be duplicated by sending the data in smaller chunks over several commands. Therefore, the maximum size of a pattern that can be copied with REPEAT_DATA will depend on the message size limits of the transport layer.



REPEAT_DATA Command Format
Figure 33

REPEAT_DATA FIELDS:

Command Length

The command length gives the number of octets in the command, including data octets in the pattern, but excluding the padding octet, if any.

Target Start Address

This is the starting address where the first copy of the pattern should be written in the target. Successive copies of the pattern are made contiguously starting at this address.

Repeat Count

The repeat count specifies the number of copies of the pattern that should be made in the target. The repeat count should be greater than zero.

Pattern

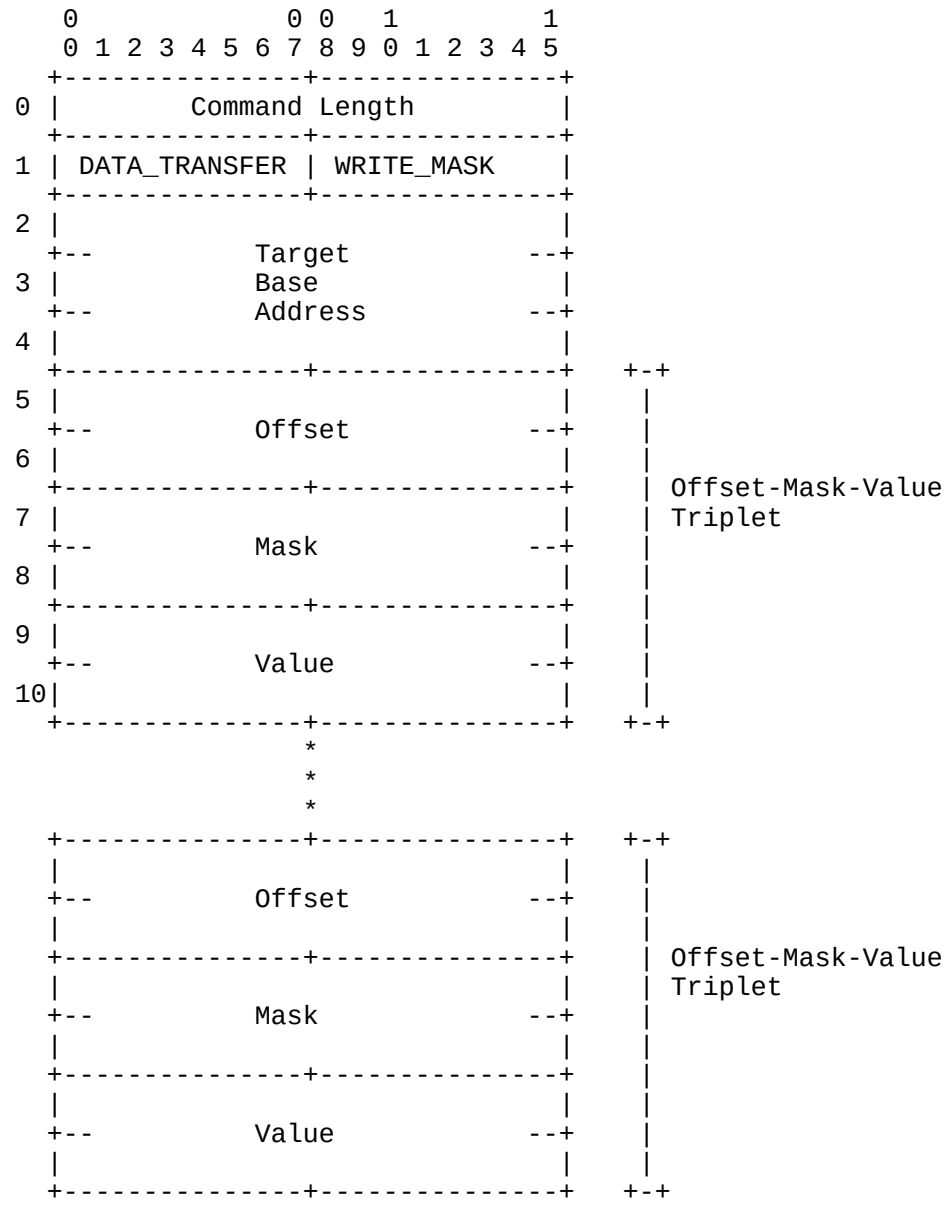
The pattern to be copied into the target, packed into a stream of octets. Data are packed according to the packing convention described above. Ends with a null octet if there are an odd number of data octets.

6.9 WRITE_MASK Command (Optional)

The host sends a WRITE_MASK command to the target to write one or more masked values. The command uses an address to specify a target base location, followed by one or more offset-mask-value triplets. Each triplet gives an offset from the base, a value, and a mask indicating which bits in the location at the offset are to be changed.

This optional command is intended for use in controlling the target by changing locations in a table. For example, it may be used to change entries in a target parameter table. The operation of modifying a specified location with a masked value is intended to be atomic. In other words, another target process should not be able to access the location to be modified between

the start and the end of the modification.



WRITE_MASK Format
Figure 34

WRITE_MASK FIELDS:

Command Length

The command length gives the number of octets in the command. The number of offset-value pairs may be calculated from this, since the command header is either 10 or 12 octets long (short or long address format), and each offset-mask-value triplet is 12 octets long.

Target Base Address

Specifies the target location to which the offset is added to yield the location to be modified.

Offset

An offset to be added to the base to select a location to be modified.

Mask

Specifies which bits in the value are to be copied into the location.

Value

A value to be stored at the specified offset from the base. The set bits in the mask determine which bits in the value are applied to the location. The following algorithm will achieve the intended result: take the one’s complement of the mask and AND it with the location, leaving the result in the location. Then AND the mask and the value, and OR the result into the location.

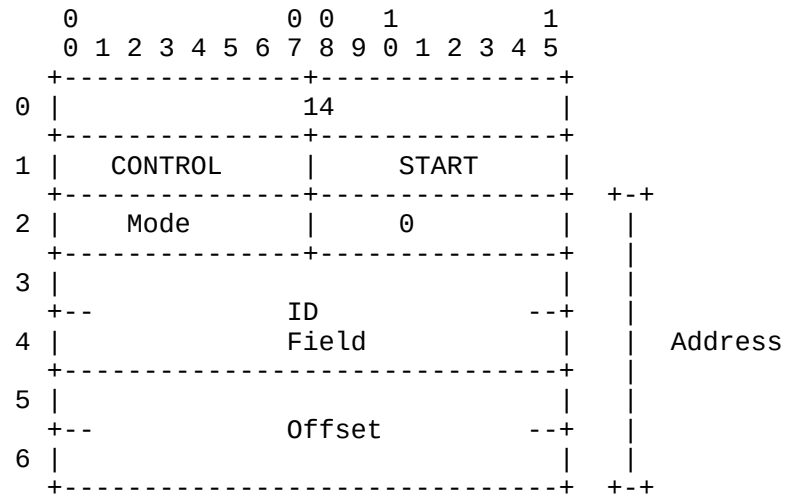
CHAPTER 7

Control Commands

Control commands are used to control the execution of target code, breakpoints and watchpoints. They are also used to read and report the state of these objects. The object to be controlled or reported on is specified with a descriptor. Valid descriptor modes include PHYS_* (for some commands) PROCESS_CODE, BREAKPOINT and WATCHPOINT. Control commands which change the state of the target are START, STOP, CONTINUE and STEP. REPORT requests a STATUS report on a target object. EXCEPTION is a spontaneous report on an object, used to report asynchronous events such as hardware traps. The host may verify the action of a START, STOP, STEP or CONTINUE command by following it with a REPORT command.

7.1 START Command

The START command is sent by the host to start execution of a specified object in the target. For targets which support multiple processes, a PROCESS_CODE address specifies the process to be started. Otherwise, one of the PHYS_* modes may specify a location in macro-memory where execution is to continue. Applied to a breakpoint or watchpoint, START sets the value of the object’s state variable, and activates the breakpoint. The breakpoint counter and pointer variables are initialized to zero.



START Command Format
Figure 35

START FIELDS:

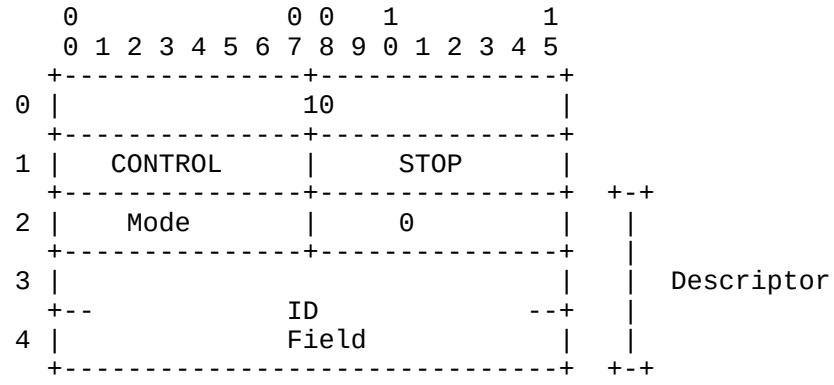
Address

The descriptor specifies the object to be started. If the mode is PROCESS_CODE, ID specifies the process to be started, and offset gives the process virtual address to start at. If the mode is PHYS_*, execution of the target is continued at the specified address.

For modes of BREAKPOINT and WATCHPOINT, the offset specifies the new value of the FSM state variable. This is for FSM breakpoints and watchpoints.

7.2 STOP Command

The STOP command is sent by the host to stop execution of a specified object in the target. A descriptor specifies the object. Applied to a breakpoint or watchpoint, STOP deactivates it. The breakpoint/watchpoint may be re-activated by issuing a START or a CONTINUE command for it.



STOP Command Format
Figure 36

STOP FIELDS:

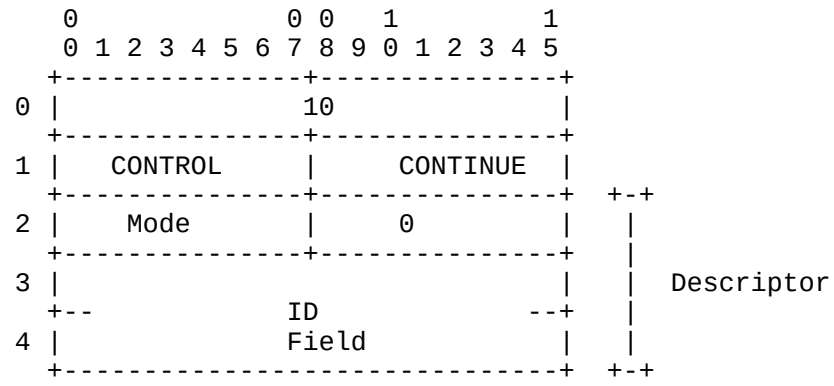
Descriptor

The descriptor specifies the object to be stopped or disarmed. If the mode is PROCESS_CODE, the ID specifies the process to be stopped.

For modes of BREAKPOINT and WATCHPOINT, the specified breakpoint or watchpoint is deactivated. It may be re-activated by a CONTINUE or START command.

7.3 CONTINUE Command

The CONTINUE command is sent by the host to resume execution of a specified object in the target. A descriptor specifies the object. Applied to a breakpoint or watchpoint, CONTINUE activates it.



CONTINUE Command Format
Figure 37

CONTINUE FIELDS:

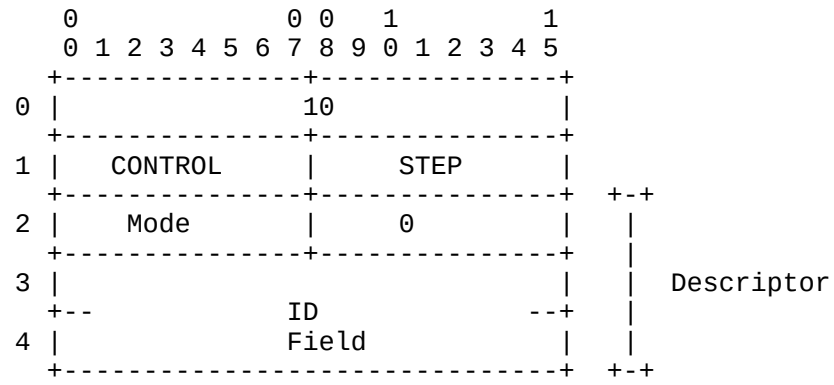
Descriptor

The descriptor specifies the object to be resumed or armed. If the mode is PROCESS_CODE, the ID specifies the process to be resumed.

For modes of BREAKPOINT and WATCHPOINT, the specified breakpoint or watchpoint is armed.

7.4 STEP Command

The STEP command is sent by the host to the target. It requests the execution of one instruction (or appropriate operation) in the object specified by the descriptor.



STEP Command Format
Figure 38

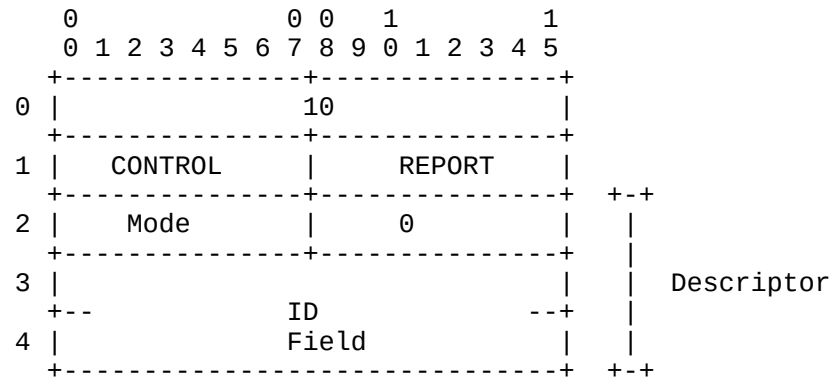
STEP FIELDS:

Descriptor

The descriptor specifies the object to be stepped. If the mode is PROCESS_CODE, the ID specifies a process.

7.5 REPORT Command

The REPORT command is sent by the host to request a status report on a specified target object. The status is returned in a STATUS reply.



REPORT Command Format
Figure 39

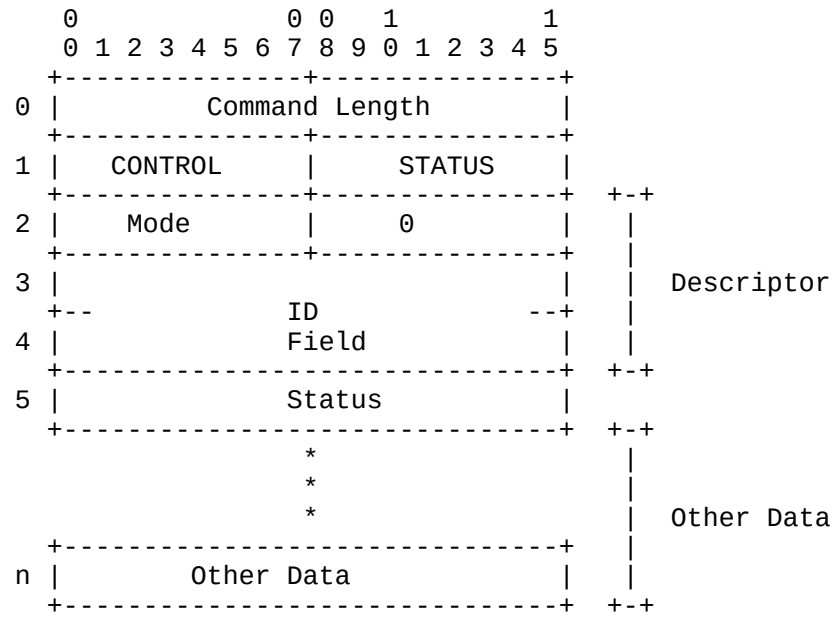
REPORT FIELDS:

Descriptor

The descriptor specifies the object for which a STATUS report is requested. For a mode of PROCESS_CODE, the ID specifies a process. Other valid modes are PHYS_MACRO, to query the status of the target application, and BREAKPOINT and WATCHPOINT, to get the status of a breakpoint or watchpoint.

7.6 STATUS Reply

The target sends a STATUS reply in response to a REPORT command from the host. STATUS gives the state of a specified object. For example, it may tell whether a particular target process is running or stopped.



STATUS Reply Format
Figure 40

STATUS FIELDS:

Descriptor

The descriptor specifies the object whose status is being given. If the mode is PROCESS_CODE, then the ID specifies a process. If the mode is PHYS_MACRO, then the status is that of the target application.

Status

The status code describes the status of the object. Status codes are 0=STOPPED and 1=RUNNING. For breakpoints and watchpoints, STOPPED means disarmed and RUNNING means armed.

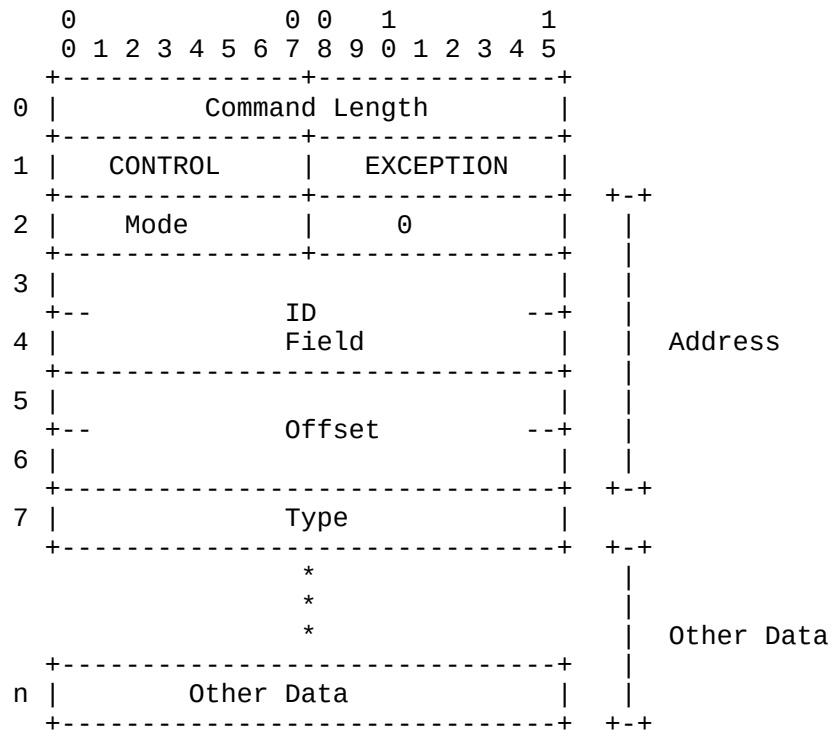
Other Data

For breakpoints and watchpoints, Other Data consists of a

16-bit word giving the current value of the FSM state variable.

7.7 EXCEPTION Trap

An EXCEPTION is a spontaneous message sent from the target indicating a target-machine exception associated with a particular object. The object is specified by an address.



EXCEPTION Format
Figure 41

EXCEPTION FIELDS:

Address

The address specifies the object the exception is for.

Type

The type of exception. Values are target-dependent.

Other Data

Values are target-dependent.

CHAPTER 8

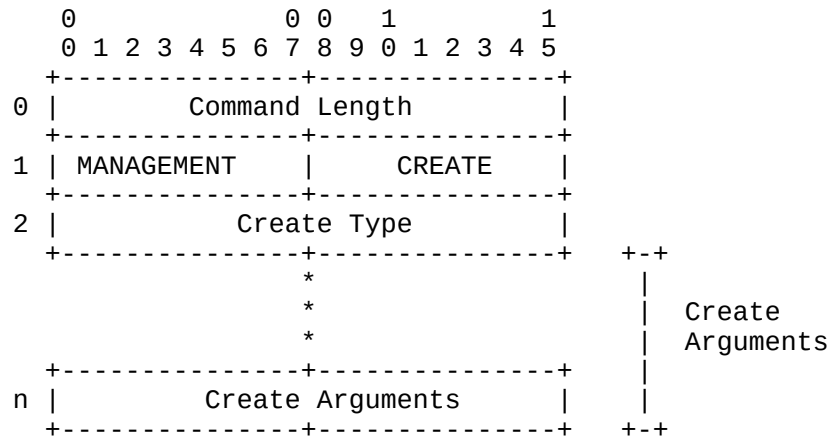
Management Commands

Management commands are used to control resources in the target machine. There are two kinds of commands: those that interrogate the remote machine about resources, and those that allocate and free resources. There are management commands to create, list and delete breakpoints. All commands have corresponding replies which include the sequence number of the request command. Failing requests produce ERROR replies.

There are two resource allocation commands, CREATE and DELETE, which create and delete objects in the remote machine. There are a number of listing commands for listing a variety of target objects -- breakpoints, watchpoints, processes, and names. The amount of data returned by listing commands may vary in length, depending on the state of the target. If a list is too large to fit in a single message, the target will send it in several list replies. A flag in each reply specifies whether more messages are to follow.

8.1 CREATE Command

The CREATE command is sent from the host to the target to create a target object. If the CREATE is successful, the target returns a CREATE_DONE reply, which contains a descriptor associated with the CREATEd object. The types of objects that may be specified in a CREATE include breakpoints, processes, memory objects and descriptors. All are optional except for breakpoints.



CREATE Command Format
Figure 42

CREATE FIELDS:

Create Type

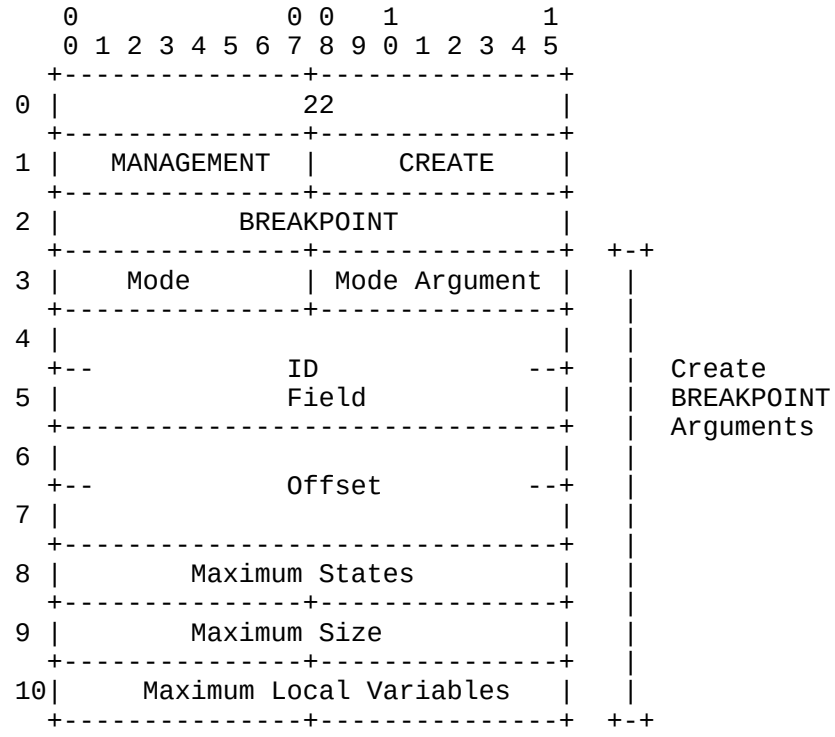
The type of object to be created. Arguments vary with the type. Currently defined types are shown in Figure 43. All are optional except for BREAKPOINT.

Create Type	Symbol
0	BREAKPOINT
1	WATCHPOINT
2	PROCESS
3	MEMORY_OBJECT
4	DESCRIPTOR

Create Types
Figure 43

Create Arguments

Create arguments depend on the type of object being created. The formats for each type of object are described below.



CREATE BREAKPOINT Format
Figure 44

BREAKPOINT and WATCHPOINT

The format is the same for CREATE BREAKPOINT and CREATE WATCHPOINT. In the following discussion, 'breakpoint' may be taken to mean either breakpoint or watchpoint.

The address is the location where the breakpoint is to be set. In the case of watchpoints it is the location to be

watched. Valid modes are any PHYS_* mode that addresses macro-memory, PROCESS_CODE for breakpoints and PROCESS_DATA for watchpoints.

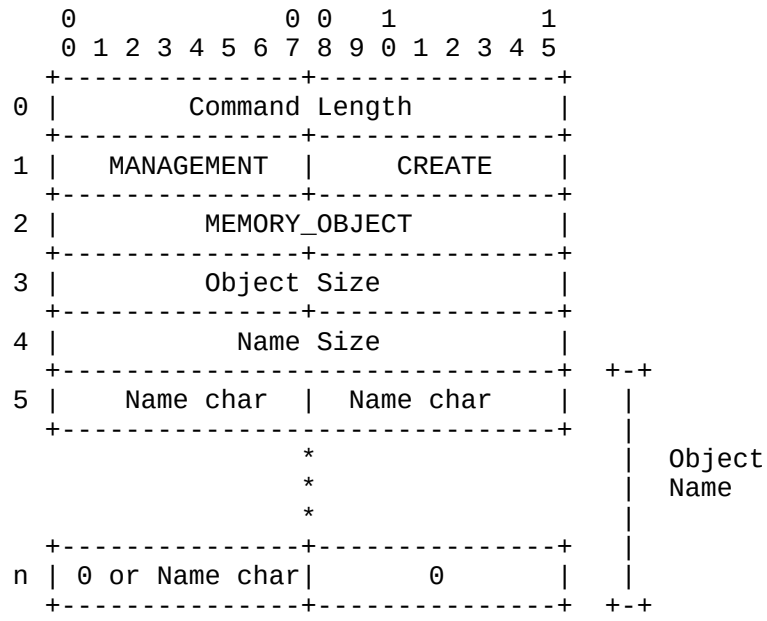
'Maximum states' is the number of states the finite state machine for this breakpoint will have. A value of zero indicates a default breakpoint, for targets which do not implement finite state machine (FSM) breakpoints. A default breakpoint is the same as an FSM with one state consisting of a STOP and a REPORT command for the process containing the breakpoint.

'Maximum size' is the total size, in octets, of the breakpoint data to be sent via subsequent BREAKPOINT_DATA commands. This is the size of the data only, and does not include the LDP command headers and breakpoint descriptors.

'Maximum local variables' is the number of 32-bit longs to reserve for local variables for this breakpoint. Normally this value will be zero.

PROCESS

Creates a new process. Arguments are target-dependent.



CREATE MEMORY_OBJECT Format
Figure 45

MEMORY_OBJECT

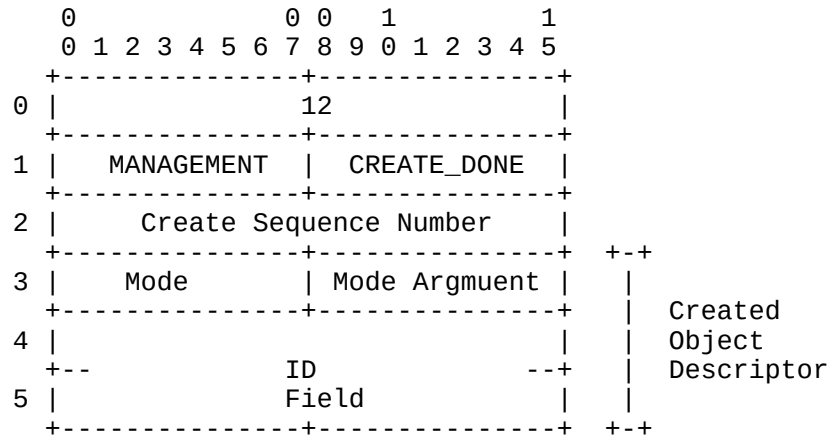
Creates an object of size Object Size, with the given name. Object Size is in target dependent units. The name may be the null string for unnamed objects. Name Size gives the number of characters in Object Name, and must be even. Always ends with a null octect.

DESCRIPTOR

Used for obtaining descriptors from IDs on target systems where IDs are longer than 32 bits. There is a single argument, Long ID, whose length is target dependent.

8.2 CREATE_DONE Reply

The target sends a CREATE_DONE reply to the host in response to a successful CREATE command. The reply contains the sequence number of the CREATE request, and a descriptor for the object created. This descriptor is used by the host to specify the object in subsequent commands referring to it. Commands which refer to created objects include LIST_* commands, DELETE and BREAKPOINT_DATA. For example, to delete a CREATEd object, the host sends a DELETE command that specifies the descriptor returned by the CREATE_DONE reply.



CREATE_DONE Reply Format
Figure 46

CREATE_DONE FIELDS:

Create Sequence Number

The sequence number of the CREATE command to which this is the reply.

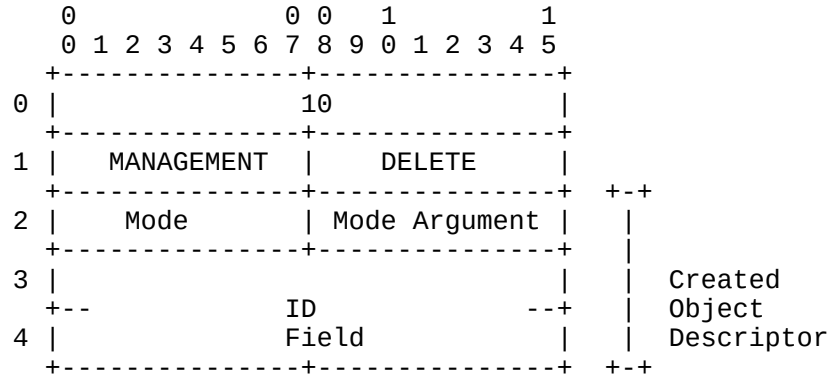
Created Object Descriptor

A descriptor assigned by the target to the created object. The contents of the descriptor fields are arbitrarily

assigned by the target at its convenience. The host treats the descriptor as a unitary object, used for referring to the created object in subsequent commands.

8.3 DELETE Command

The host sends a DELETE command to remove an object created by an earlier CREATE command. The object to be deleted is specified with a descriptor. The descriptor is from the CREATE_DONE reply to the original CREATE command.



DELETE Command Format
Figure 47

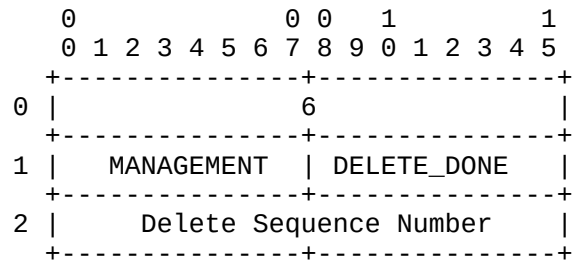
DELETE FIELDS:

Created Object Descriptor

Specifies the object to be deleted. This is the descriptor that was returned by the target in the CREATE_DONE reply to the original CREATE command.

8.4 DELETE_DONE Reply

The target sends a DELETE_DONE reply to the host in response to a successful DELETE command. The reply contains the sequence number of the DELETE request.



DELETE_DONE Reply Format
Figure 48

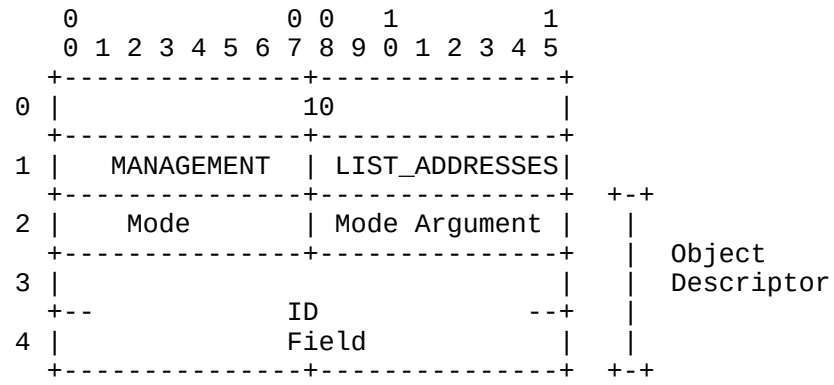
DELETE_DONE FIELDS:

Request Sequence Number

The sequence number of the DELETE command to which this is the reply.

8.5 LIST_ADDRESSES Command

The host sends a LIST_ADDRESSES command to request a list of valid address ranges for a specified object. The object is given by a descriptor. Typical objects are a target process, or the target physical machine. The target responds with an ADDRESS_LIST reply. This command is used for obtaining the size of dynamic address spaces and for determining dump ranges.



LIST_ADDRESSES Command Format
Figure 49

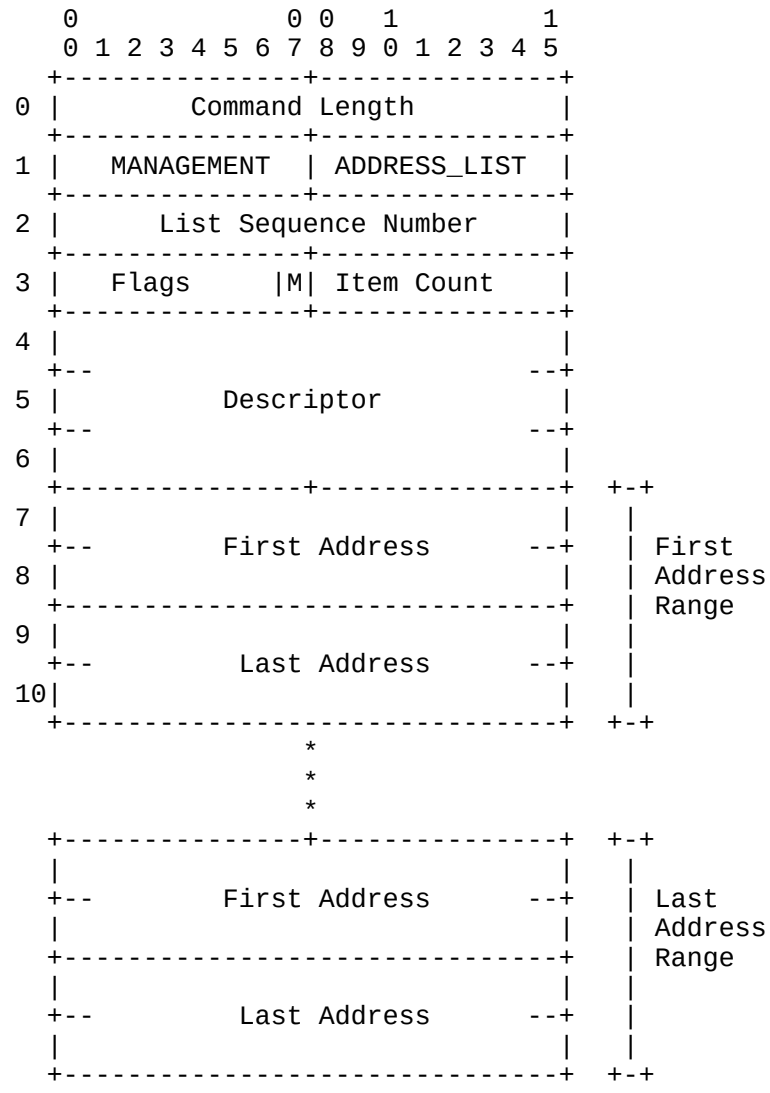
LIST_ADDRESSES FIELDS:

Object Descriptor

Specifies the object whose address ranges are to be listed. Valid modes include PHYS_MACRO, PHYS_MICRO, PROCESS_CODE, and PROCESS_DATA.

8.6 ADDRESS_LIST Reply

The target sends an ADDRESS_LIST reply to the host in response to a successful LIST_ADDRESSES command. The reply contains the sequence number of the LIST_ADDRESSES request, the descriptor of the object being listed, and a list of the valid address ranges within the object.



ADDRESS_LIST Reply Format
Figure 50

ADDRESS_LIST FIELDS:

List Sequence Number

The sequence number of the LIST_ADDRESSES command to which this is the reply.

Flags

If M=1, the address list is continued in one or more subsequent ADDRESS_LIST replies. If M=0, this is the final ADDRESS_LIST.

Item Count

The number of address ranges described in this command.

Descriptor

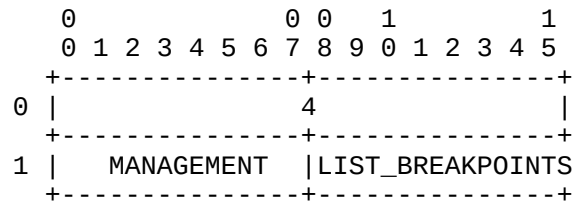
The descriptor of the object being listed.

Address Range

Each address range is composed of a pair of 32-bit addresses which give the first and last addresses of the range. If there are 'holes' in the address space of the object, then multiple address ranges will be used to describe the valid address space.

8.7 LIST_BREAKPOINTS Command

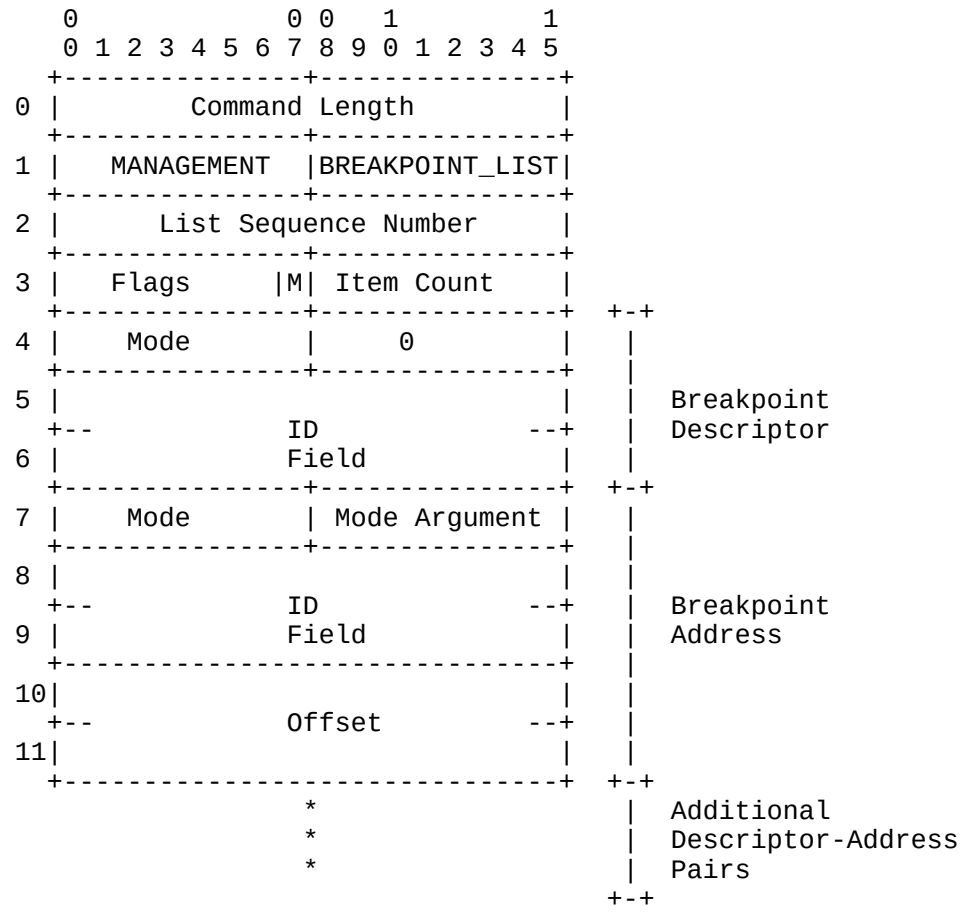
The host sends a LIST_BREAKPOINTS command to request a list of all breakpoints associated with the current connection. The target replies with BREAKPOINT_LIST.



LIST_BREAKPOINTS Command Format
Figure 51

8.8 BREAKPOINT_LIST Reply

The target sends a BREAKPOINT_LIST reply to the host in response to a LIST_BREAKPOINTS command. The reply contains the sequence number of the LIST_BREAKPOINTS request, and a list of all breakpoints associated with the current connection. The descriptor and address of each breakpoint are listed.



BREAKPOINT_LIST Reply Format
Figure 52

BREAKPOINT_LIST FIELDS:

List Sequence Number

The sequence number of the LIST_BREAKPOINTS command to which this is the reply.

Flags

If M=1, the breakpoint list is continued in one or more subsequent BREAKPOINT_LIST replies. If M=0, this is the final BREAKPOINT_LIST.

Item Count

The number of breakpoints described in this list.

Breakpoint Descriptor

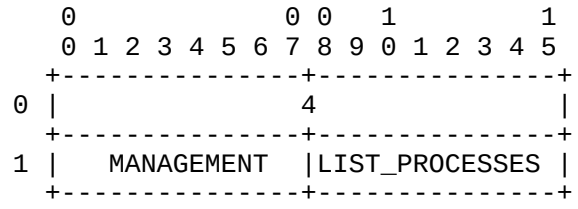
A descriptor assigned by the target to this breakpoint. Used by the host to specify this breakpoint in BREAKPOINT_DATA and DELETE commands.

Breakpoint Address

The address at which this breakpoint is set.

8.9 LIST_PROCESSES Command

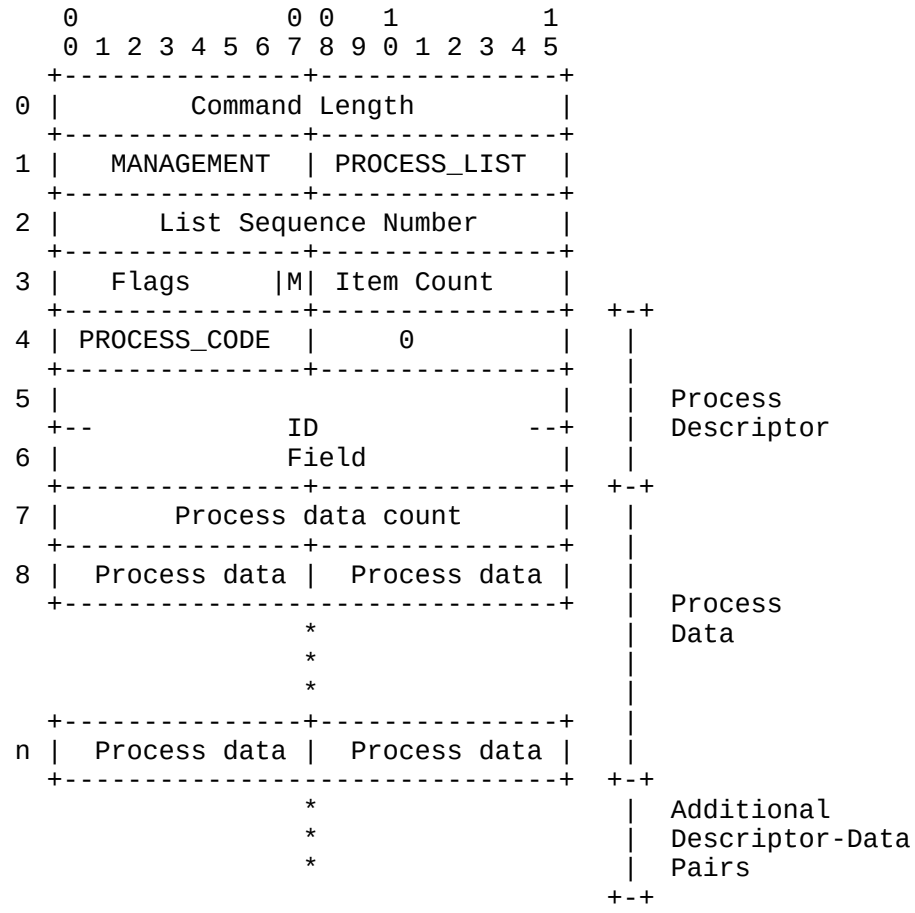
The host sends a LIST_PROCESSES command to request a list of descriptors for all processes on the target. The target replies with PROCESS_LIST.



LIST_PROCESSES Command Format
Figure 53

8.10 PROCESS_LIST Reply

The target sends a PROCESS_LIST reply to the host in response to a LIST_PROCESSES command. The reply contains the sequence number of the LIST_PROCESSES request, and a list of all processes in the target. For each process, a descriptor and a target-dependent amount of process data are given.



PROCESS_LIST Reply Format
Figure 54

PROCESS_LIST FIELDS:

List Sequence Number

The sequence number of the LIST_PROCESSES command to which this is the reply.

Flags

If M=1, the process list is continued in one or more subsequent PROCESS_LIST replies. If M=0, this is the final PROCESS_LIST.

Item Count

The number of processes described in this list. For each process there is a descriptor and a variable number of octets of process data.

Process Descriptor

A descriptor assigned by the target to this process. Used by the host to specify this PROCESS in a DELETE command.

Process Data Count

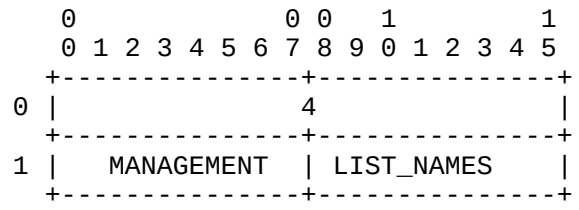
Number of octets of process data for this process. Must be even.

Process Data

Target-dependent information about this process. Number of octets is given by the process data count.

8.11 LIST_NAMES Command

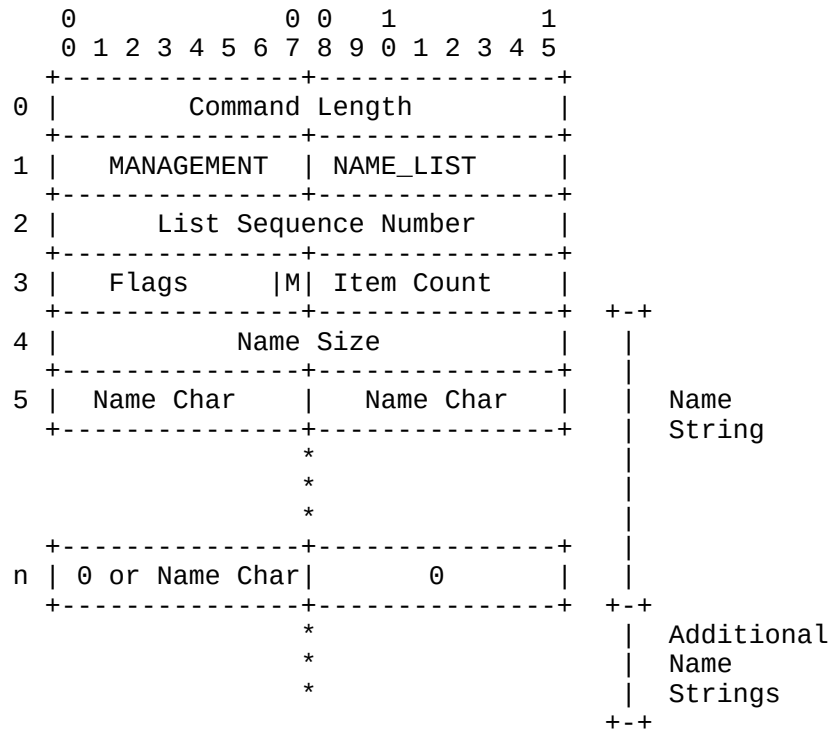
The host sends a LIST_NAMES command to request a list of available names as strings. The target replies with NAME_LIST.



LIST_NAMES Command Format
Figure 55

8.12 NAME_LIST Reply

The target sends a NAME_LIST reply to the host in response to a LIST_NAMES command. The reply contains the sequence number of the LIST_NAMES request, and a list of all target names, as strings.



NAME_LIST Reply Format
Figure 56

NAME_LIST FIELDS:

List Sequence Number

The sequence number of the LIST_NAMES command to which this is the reply.

Flags

If M=1, the name list is continued in one or more subsequent NAME_LIST replies. If M=0, this is the final NAME_LIST.

Item Count

The number of name strings in this list. Each name string consists of a character count and a null-terminated string of characters.

Name Size

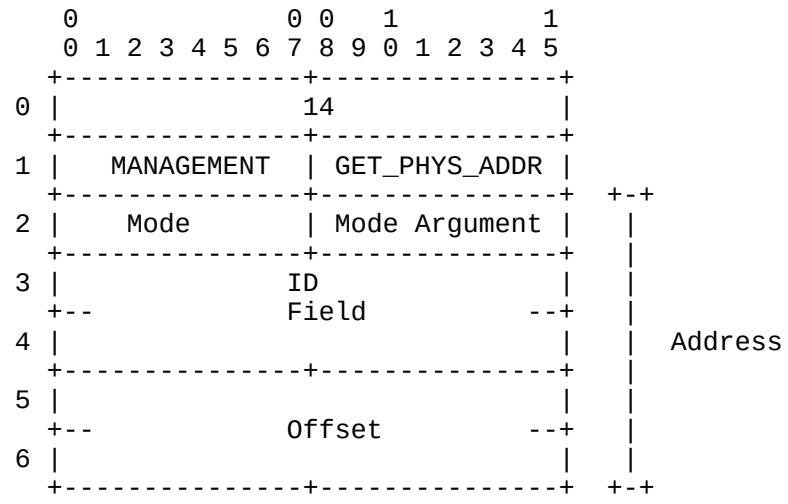
The number of octets in this name string. Must be even.

Name Characters

A string of octets composing the name. Ends with a null octet. The number of characters must be even, so if the terminating null comes on an odd octet, another null is appended.

8.13 GET_PHYS_ADDR Command

The host sends a GET_PHYS_ADDR command to convert an address into physical form. The target returns the physical address in a GOT_PHYS_ADDR reply. For example, the host could send a GET_PHYS_ADDR command containing a register-offset address, and the target would return the physical address derived from this in a GOT_PHYS_ADDR reply.



GET_PHYS_ADDR Command Format
Figure 57

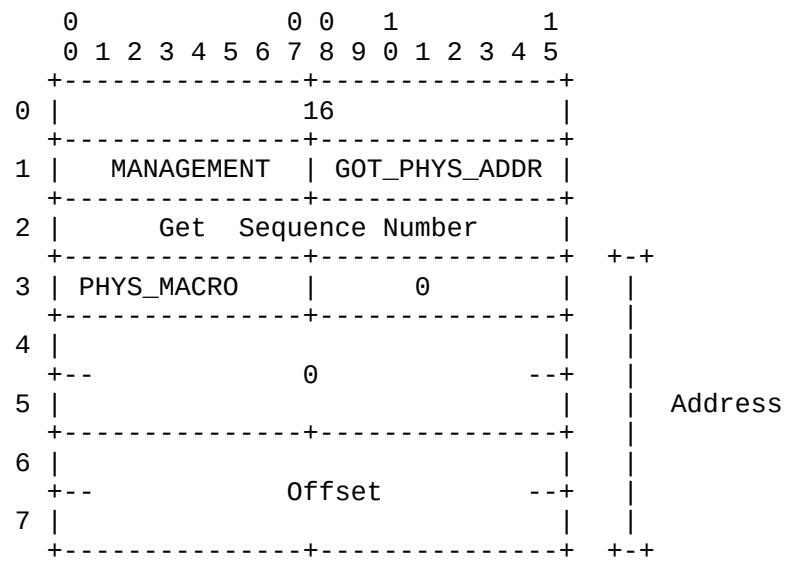
GET_PHYS_ADDR FIELDS:

Address

The address to be converted to a physical address. The mode may be one of PHYS_REG_OFFSET, PHYS_REG_INDIRECT, PHYS_MACRO_PTR, any OBJECT_* mode, and any PROCESS_* mode except for PROCESS_REG.

8.14 GOT_PHYS_ADDR Reply

The target sends a GOT_PHYS_ADDR reply to the host in response to a successful GET_PHYS_ADDR command. The reply contains the sequence number of the GET_PHYS_ADDR request, and the specified address converted into a physical address.



GOT_PHYS_ADDR Reply Format
Figure 58

GOT_PHYS_ADDR FIELDS:

Get Sequence Number

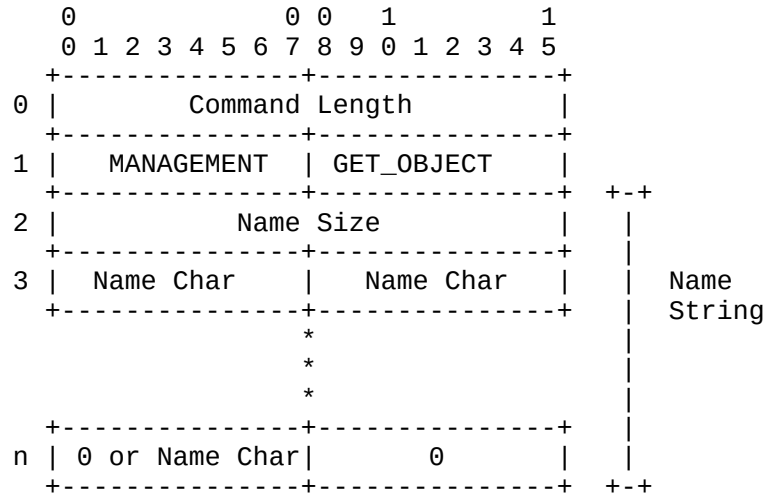
The sequence number of the GET_PHYS_ADDR command to which this is the reply.

Address

The address resulting from translating the address given in the GET_PHYS_ADDR command into a physical address. Mode is always PHYS_MACRO and ID and mode argument are always zero. Offset gives the 32-bit physical address.

8.15 GET_OBJECT Command

The host sends a GET_OBJECT command to convert a name string into a descriptor. The target returns the descriptor in a GOT_OBJECT reply. Intended for use in finding control parameter objects.



GET_OBJECT Command Format
Figure 59

GET_OBJECT FIELDS:

Name String

The name of an object.

Name Size

The number of octets in this name string. Must be even.

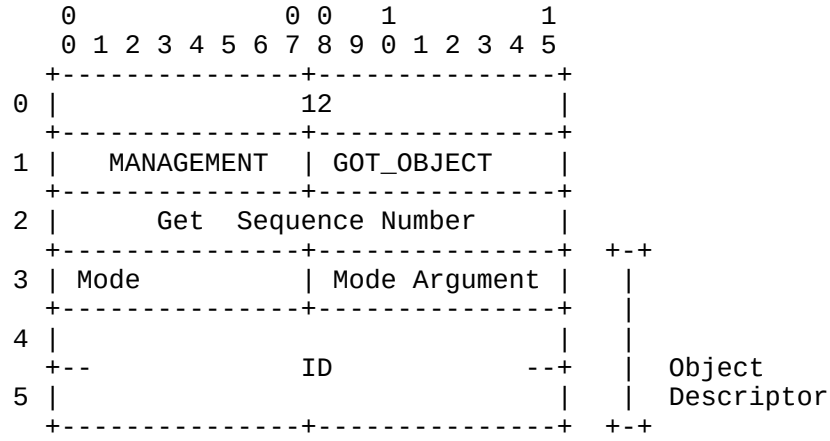
Name Characters

A string of octets composing the name. Ends with a null octet. The number of characters must be even, so if the

terminating null comes on an odd octet, another null is appended.

8.16 GOT_OBJECT Reply

The target sends a GOT_OBJECT reply to the host in response to a successful GET_OBJECT command. The reply contains the sequence number of the GET_OBJECT request, and the specified object name converted into a descriptor.



GOT_OBJECT Reply Format
Figure 60

GOT_OBJECT FIELDS:

Get Sequence Number

The sequence number of the GET_OBJECT command to which this is the reply.

Descriptor

The descriptor of the object named in the GET_OBJECT command.

CHAPTER 9

Breakpoints and Watchpoints

Breakpoints and watchpoints are used in debugging applications. Each breakpoint or watchpoint is associated with one debugger connection and one address. When a breakpoint or watchpoint is triggered, the target executes one or more commands associated with it. A breakpoint is triggered when its address is executed. A watchpoint is triggered when its address is modified. The same mechanism is used for structuring breakpoint and watchpoint commands. For brevity's sake, 'breakpoint' will be used in the remainder of this document to refer to either a breakpoint or a watchpoint.

The commands used by the host to manipulate breakpoints are given in Figure 61, in the order in which they are normally used. All commands are sent from the host to the target, and each specifies the descriptor of a breakpoint.

Command	Description
CREATE	Create a breakpoint
BREAKPOINT_DATA	Send commands to be executed in an FSM breakpoint
START	Activate a breakpoint, set state and initialize breakpoint variables
STOP	Deactivate a breakpoint
CONTINUE	Activate a breakpoint
LIST_BREAKPOINTS	List all breakpoints
REPORT	Report the status of a breakpoint
DELETE	Delete a breakpoint

Commands to Manipulate Breakpoints
Figure 61

There are two kinds of breakpoints: default breakpoints and finite state machine (FSM) breakpoints. They differ in their use of commands.

Default breakpoints do not contain any commands. When triggered, a default breakpoint stops the target object (i.e., target process or application) it is located in. A STATUS report on the stopped object is sent to the host. At this point, the host may send further commands to debug the target.

An FSM breakpoint has one or more conditional command lists, organized into a finite state machine. When an FSM breakpoint is created, the total number of states is specified. The host then sends commands (using BREAKPOINT_DATA) to be associated with each state. The target maintains a state variable for the breakpoint, which determines which command list will be executed if the breakpoint is triggered. When the breakpoint is created its state variable is initialized to zero (zero is the first state). A breakpoint command, SET_STATE, may be used within a breakpoint to change the value of the state variable. A REPORT command applied to a breakpoint descriptor returns its address, whether it is armed or disarmed, and the value of its state variable.

Commands valid in breakpoints include all implemented data transfer and control commands, a set of conditional commands, and a set of breakpoint commands. The conditional commands and the breakpoint commands act on a set of local breakpoint variables. The breakpoint variables consist of the state variable, a counter, and two pointer variables. The conditional commands control the execution of breakpoint command lists based on the contents of one of the breakpoint variables. The breakpoint commands are used to set the value of the breakpoint variables: SET_STATE sets the state variable, SET_PTR sets one of the pointer variables, and INC_COUNT increments the breakpoint counter. There may be implementation restrictions on the number of breakpoints, the number of states, the number of conditions, and the size of the command lists. Management commands and protocol commands are forbidden in breakpoints.

In FSM breakpoints, the execution of commands is controlled as follows. When a breakpoint is triggered, the breakpoint's state variable selects a particular state. One or more conditional command lists is associated with this state. A conditional command list consists of a list of conditions followed by a list of commands which are executed if the condition list is satisfied. The debugger starts a breakpoint by executing the first of these lists. If the condition list is

satisfied, the debugger executes the associated command list and leaves the breakpoint. If the condition list fails, the debugger skips to the next conditional command list. This process continues until the debugger either encounters a successful condition list, or exhausts all the conditional command lists for the state. The relationship of commands, lists and states is shown in Figure 62 (IFs, THENs and ELSEs are used below to clarify the logical structure within a state; they are not part of the protocol).

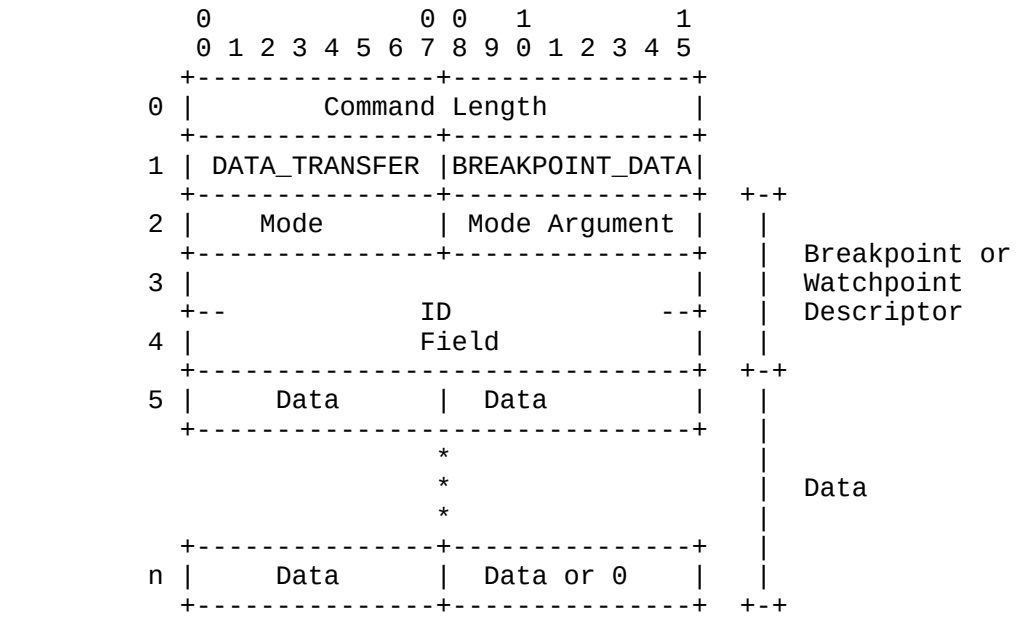
```
State 0
        IF <condition list 0>
           THEN <command list 0>

        ELSE IF <condition list 1>
           THEN <command list 1>

        *
        *
        *

        ELSE IF <condition list n>
            THEN <command list n>

        ELSE <exit>
  *
  *
  *
State n
```

Breakpoint Conditional Command Lists
Figure 62

9.1 BREAKPOINT_DATA Command

BREAKPOINT_DATA is a data transfer command used by the host to send commands to be executed in breakpoints and watchpoints. The command specifies the descriptor of the breakpoint or watchpoint, and a stream of commands to be appended to the end of the breakpoint's command list. BREAKPOINT_DATA is applied sequentially to successive breakpoint states, and successive

command lists within each state. Multiple BREAKPOINT_DATAs may be sent for a given breakpoint. Breaks between BREAKPOINT_DATA commands may occur anywhere within the data stream, even within individual commands in the data. Sufficient space to store the data must have been allocated by the maximum size field in the CREATE BREAKPOINT/WATCHPOINT command.



BREAKPOINT_DATA Command Format
Figure 63

BREAKPOINT_DATA FIELDS:

Command Length

Total length of this command in octets, including data, excluding the final padding octet, if any.

Data

A stream of data to be appended to the data for this breakpoint or watchpoint. This stream has the form of one or more states, each containing one or more conditional

command lists. The first BREAKPOINT_DATA command sent for a breakpoint contains data starting with state zero. The data for each state starts with the state size. A conditional command list is composed of two parts: a condition list, and a command list. Each list begins with a word that gives its size in octets.

```
<state 0 size>
         <condition list 0 size> <condition list 0>
         <command list 0 size>   <command list 0>
                    *
                    *
                    *
         <condition list n size> <condition list n>
         <command list n size>   <command list n>
<state 1 size>
                  <etc>
     *
     *
     *
<state n size>
```

Breakpoint Data Stream Format
Figure 64

Sizes

All sizes are stored in 16-bit words, and include their own length. The state size gives the total number of octets of breakpoint data for the state. The condition list size gives the total octets of breakpoint data for the following condition list. A condition list size of 2 indicates an empty condition list: in this case the following command list is executed unconditionally. The command list size gives the total octets of breakpoint data for the following command list.

Lists

Condition and command lists come in pairs. When the breakpoint occurs, the condition list controls whether the following command list should be executed. A condition list consists of one or more commands from the CONDITION command class. A command list consists one or more LDP commands. Valid commands are any commands from the BREAKPOINT, DATA_TRANSFER or CONTROL command classes.

CHAPTER 10

Conditional Commands

Conditional commands are used in breakpoints to control the execution of breakpoint commands. One or more conditions in sequence form a condition list. If a condition list is satisfied (evaluates to TRUE), the breakpoint command list immediately following it is executed. (See Breakpoints and Watchpoints, above, for a discussion of the logic flow in conditional/command lists.) Conditional commands perform tests on local breakpoint variables, and other locations. Each condition evaluates to either TRUE or FALSE. Figure 65 contains a summary of conditional commands:

Command	Description
CHANGED <loc>	Determine if a location has changed
COMPARE <loc1> <mask> <loc2>	Compare two locations, using a mask
COUNT_[EQ | GT | LT] <value>	Compare the counter to a value
TEST <loc> <mask> <value>	Compare a location to a value

Conditional Command Summary
Figure 65

The rules for forming and evaluating condition lists are:

- consecutive conditions have an implicit logical AND between them. A sequence of such conditions is called an 'and_list'. and_lists are delimited by an OR command and by the end of the condition list.
- the breakpoint OR command may be inserted between any pair of conditions
- AND takes precedence over OR
- nested condition lists are not supported. A condition list is simply one or more and_lists, separated by ORs.

o the condition list is evaluated in sequence until either a TRUE and_list is found (condition list <- TRUE), or the end of the condition list is reached (condition list <- FALSE). An and_list is TRUE if all its conditions are TRUE.

The distillation of these rules into BNF is:

```
<condition_list> :== <and_list>  [OR <and_list>]*
<and_list>       :== <condition> [AND <condition>]*
<condition>      :== CHANGED | COMPARE | COUNT | TEST

where:  OR  is a breakpoint command
        AND is implicit for any pair of consecutive conditions
```

For example, the following condition list, with one command per line,

```
COUNT_EQ 1
OR
COUNT_GT 10
COUNT_LT 20
```

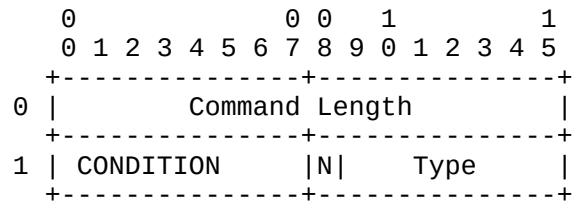
evaluates to:

```
(COUNT = 1) OR (COUNT > 10  AND COUNT < 20)
```

and will cause the command list that follows it to be executed if the counter is equal to one, or is between 10 and 20.

10.1 Condition Command Format

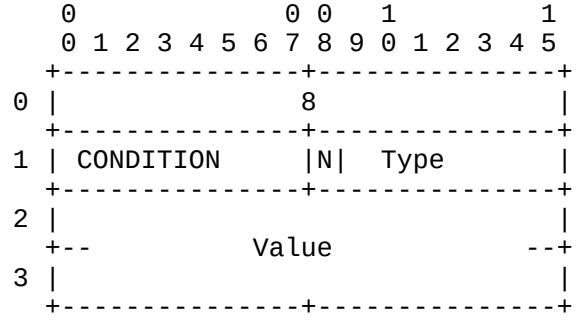
Condition commands start with the standard four-octet command header. The high-order bit of the command type byte is used as a negate flag: if this bit is set, the boolean value of the condition is negated. This flag applies to one condition only, and not to other conditions in the condition list.



Condition Command Header
Figure 66

10.2 COUNT Conditions

The COUNT conditions (COUNT_EQ, COUNT_GT and COUNT_LT) are used to compare the breakpoint counter to a specified value. The counter is set to zero when the breakpoint is STARTed, and is incremented by the INC_COUNT breakpoint command. The format is the same for the COUNT_EQ, COUNT_GT and COUNT_LT conditions.



COUNT Condition Format
Figure 67

COUNT_* Condition FIELDS:

Type

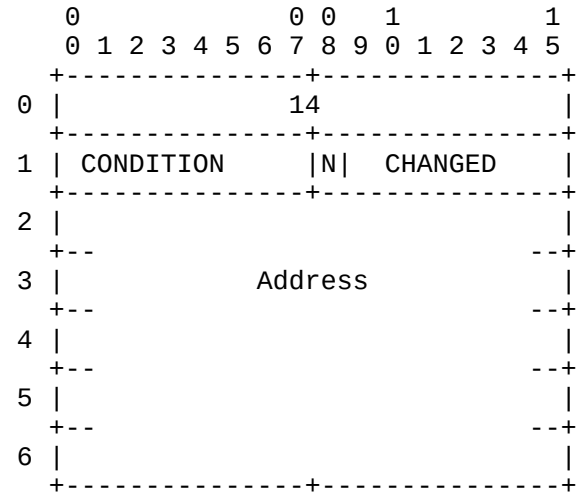
One of COUNT_EQ, COUNT_LT and COUNT_GT. The condition is TRUE if the breakpoint counter is [EQ | LT | GT] the specified value.

Value

A 32-bit value to be compared to the counter.

10.3 CHANGED Condition

The CHANGED condition is TRUE if the contents of the specified location have changed since the last time this breakpoint occurred. Only one location may be specified as the object of CHANGED conditions per breakpoint. The CHANGED condition is always FALSE the first time the breakpoint occurs.



CHANGED Condition
Figure 68

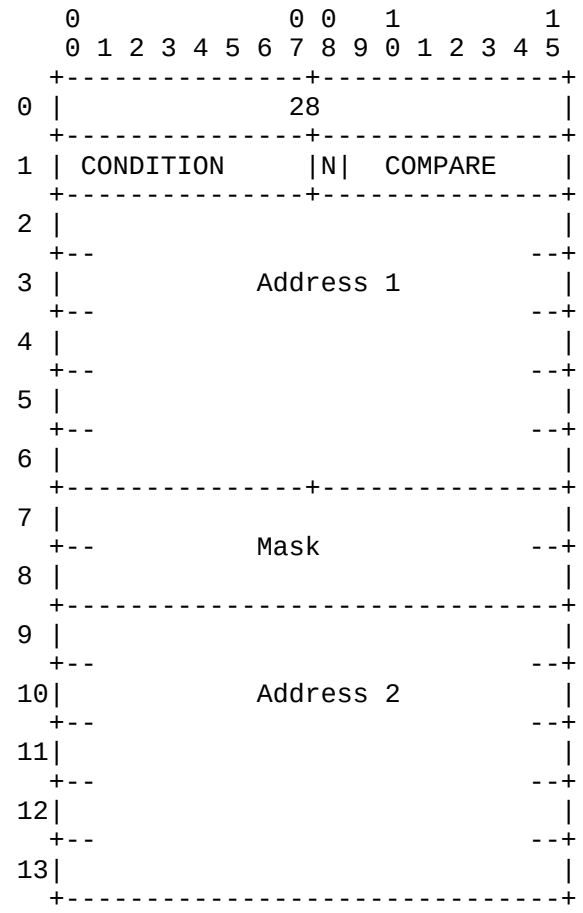
CHANGED FIELDS:

Address

The full 5-word address of the location to be tested by the CHANGED command.

10.4 COMPARE Condition

The COMPARE condition compares two locations using a mask. The condition is TRUE if (<loc1> & <mask>) = (<loc2> & <mask>).



COMPARE Condition
Figure 69

COMPARE FIELDS:

Address 1
Address 2

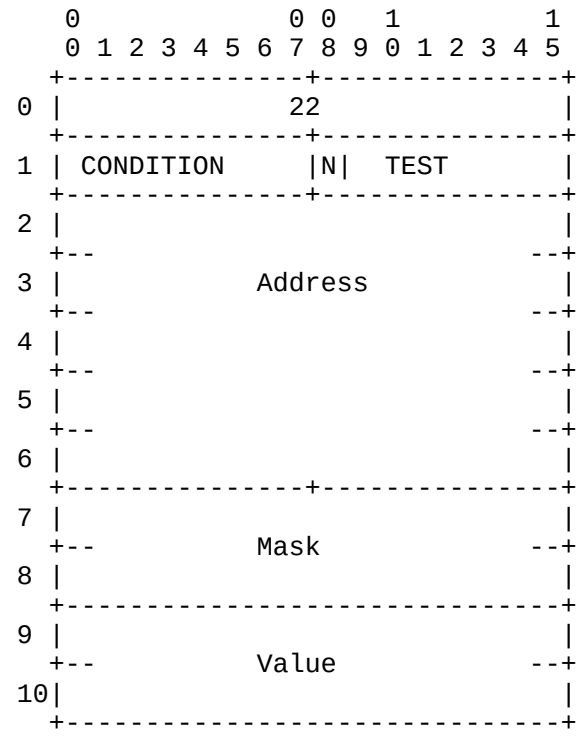
The 5-word addresses of the locations to be compared.

Mask

A 32-bit mask specifying which bits in the locations should be compared.

10.5 TEST Condition

The TEST condition is used to compare a location to a value, using a mask. The condition is TRUE if (<loc> & <mask>) = <value>.



TEST Condition
Figure 70

TEST FIELDS:

Address

The 5-word address of the location to be compared to the value.

Mask

A 32-bit mask specifying which bits in the location should be compared.

Value

A 32-bit value to compare to the masked location.

CHAPTER 11

Breakpoint Commands

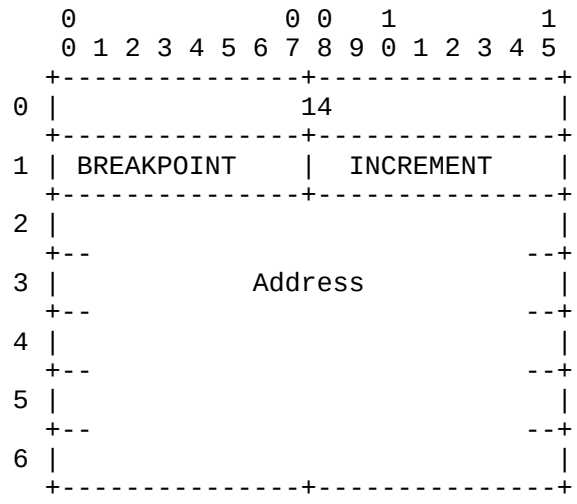
Breakpoint commands are used to set the value of breakpoint variables. These commands are only valid within breakpoints and watchpoints. They are sent from the host to the target as data in BREAKPOINT_DATA commands. Figure 71 contains a summary of breakpoint commands:

Command	Description
INCREMENT <location>	Increment the specified location
INC_COUNT	Increment the breakpoint counter
OR	OR two breakpoint condition lists
SET_PTR <n> <location>	Set pointer <n> to the contents of <location>
SET_STATE <n>	Set the breakpoint state variable to <n>

Breakpoint Command Summary
Figure 71

11.1 INCREMENT Command

The INCREMENT command increments the contents of a specified location. The location may be in any address space writable from LDP.



INCREMENT Command Format
Figure 72

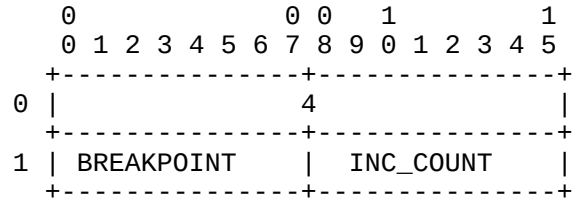
INCREMENT FIELDS:

Address

The full address of the location whose contents are to be incremented.

11.2 INC_COUNT Command

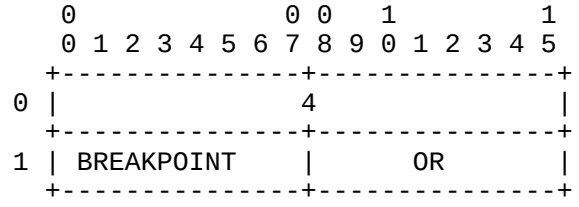
The INC_COUNT command increments the breakpoint counter. There is one counter variable for each breakpoint. It is initialized to zero when the breakpoint is created, when it is armed with the START command, and whenever the breakpoint state changes. The counter is tested by the COUNT_* conditions.



INC_COUNT Command Format
Figure 73

11.3 OR Command

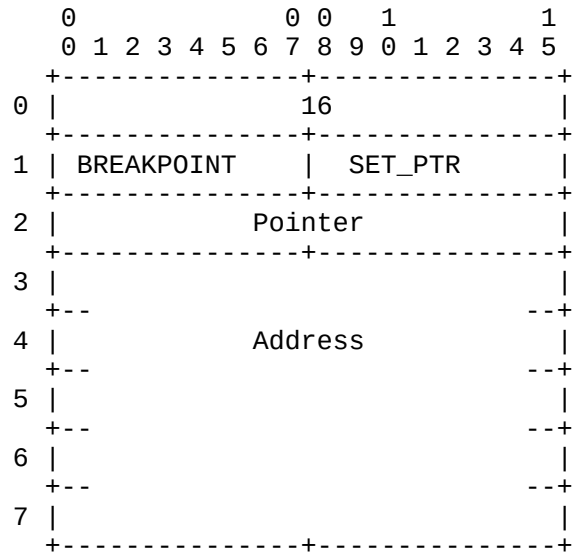
The OR command delineates two and_lists in a breakpoint condition list. A condition list is TRUE if any of the OR separated and_lists in it are TRUE. A breakpoint condition list may contain zero, one or, many OR commands. See 'Condition Commands' for an explanation of condition lists.



OR Command Format
Figure 74

11.4 SET_PTR Command

The SET_PTR command loads the specified breakpoint pointer with the contents of a location. The pointer variables and the SET_PTR command are intended to provide a primitive but unlimited indirect addressing capability. Two addressing modes, BPT_PTR_OFFSET and BPT_PTR_INDIRECT, are used for referencing the breakpoint pointers. For example, to follow a linked list, use SET_PTR to load a pointer with the start of the list, then use successive SET_PTR commands with addressing mode BPT_PTR_OFFSET to get successive elements.



SET_PTR Command Format
Figure 75

SET_PTR FIELDS:

Pointer

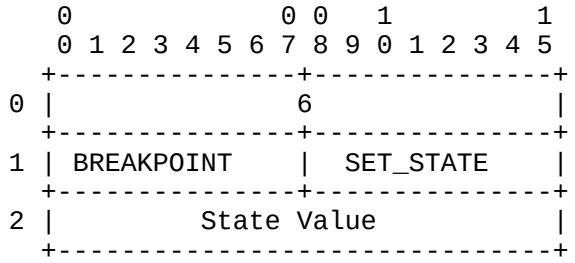
The pointer to be changed. Allowable values are 0 and 1.

Address

The full address of the location whose contents are to be loaded into the given pointer variable.

11.5 SET_STATE Command

The SET_STATE command sets the breakpoint state variable to the specified value. This is the only method of changing a breakpoint's state from within a breakpoint. The breakpoint's state may be also be changed by a START command from the host. The state variable is initialized to zero when the breakpoint is created.



SET_STATE Command Format
Figure 76

SET_STATE FIELDS:

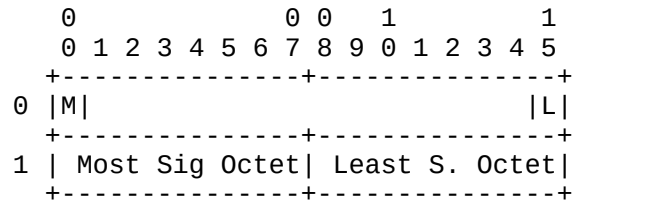
State Value

The new value for the breakpoint state variable. Must not be greater than the maximum state value specified in the CREATE BREAKPOINT command that created this breakpoint.

APPENDIX A

Diagram Conventions

Command and message diagrams are used in this document to illustrate the format of these entities. Words are listed in order of transmission down the page. The first word is word zero. Bits within a word run left to right, most significant to least. However, following a convention observed in other protocol documents, bits are numbered in order of transmission; the most significant bit in a word is transmitted first. The bit labelled '0' is the most significant bit.



M = most significant bit in word zero, transmitted first
L = least significant bit in word zero, transmitted last

Sample Diagram
Figure 77

APPENDIX B

Command Summary

The following table lists all non-breakpoint LDP commands in alphabetical order, with a brief description of each.

Command	Sender: Host	Sender: Target	Function
ABORT	X		Abort outstanding commands
ABORT_DONE		X	Acknowledge ABORT
ADDRESS_LIST		X	Return valid address ranges
BREAKPOINT_DATA	X		Send breakpoint commands
BREAKPOINT_LIST		X	Return list of breakpoints
CONTINUE	X		Resume execution
CREATE	X		Create target object
CREATE_DONE		X	Acknowledge CREATE
DELETE	X		Delete target object
DELETE_DONE		X	Acknowledge DELETE
EXCEPTION		X	Report target exception
ERROR		X	Report error with a host command
ERRACK	X		Acknowledge ERROR
GET_OBJECT	X		Get object descriptor from name
GET_PHYS_ADDRESS	X		Get address in physical form
GOT_OBJECT		X	Return object descriptor
GOT_PHYS_ADDRESS		X	Return physical address
HELLO	X		Initiate LDP session
HELLO_REPLY		X	Return LDP parameters
LIST_ADDRESSES	X		Request valid address ranges
LIST_BREAKPOINTS	X		Request breakpoint list
LIST_NAMES	X		Request name list
LIST_PROCESSES	X		Request process list
MOVE	X		Read data from target
MOVE_DONE		X	Acknowledge MOVE completion
MOVE_DATA		X	Send data request by MOVE
NAME_LIST		X	Return name list
PROCESS_LIST		X	Return process list
READ	X		Read data from target
READ_DATA		X	Return data requested by READ
READ_DONE		X	Acknowledge READ completion
REPEAT_DATA	X		Write copies of data
REPORT	X		Request status of object
START	X		Start target object
STATUS		X	Return status of object
STEP	X		Step execution of target object
STOP	X		Stop target object
SYNCH	X		Check sequence number
SYNCH_REPLY		X	Confirm sequence number
WRITE	X		Write data
WRITE_MASK	X		Write data with mask

Command Summary
Figure 78

APPENDIX C

Commands, Responses and Replies

The following table shows the relationship between commands, responses and replies. Commands are sent from the host to the target. Some commands elicit responses and/or replies from the target. Responses and replies are sent from the target to the host. The distinction between them is that the target sends only one reply to a command, but may send multiple responses. Responses always contain data, whereas replies may or may not.

Command	Response	Reply
ABORT		ABORT_DONE
BREAKPOINT_DATA		
CONTINUE		
CREATE		CREATE_DONE
DELETE		DELETE_DONE
GET_OBJECT		GOT_OBJECT
GET_PHYS_ADDRESS		GOT_PHYS_ADDRESS
HELLO		HELLO_REPLY
LIST_ADDRESSES		ADDRESS_LIST
LIST_BREAKPOINTS		BREAKPOINT_LIST
LIST_NAMES		NAME_LIST
LIST_PROCESSES		PROCESS_LIST
MOVE	MOVE_DATA	MOVE_DONE
READ	READ_DATA	READ_DONE
REPEAT_DATA		
REPORT		STATUS
START		
STEP		
STOP		
SYNCH		SYNCH_REPLY
WRITE		
WRITE_MASK		

Commands, Responses and Replies
Figure 79

APPENDIX D

Glossary

FSM

Finite state machine. Commands of each breakpoint or watchpoint are implemented as part of a finite state machine. A list of breakpoint commands is associated with each state. There are several breakpoint commands to change from one state to another.

host

The ’host’ in an LDP session is the timesharing system on which the user process runs.

long

A long is a 32-bit quantity.

octet

An octet is an eight-bit quantity.

RDP

The Reliable Data Protocol (RDP) is a transport layer protocol designed as a low-overhead alternative to TCP. RDP is a connection oriented protocol that provides reliable, sequenced message delivery.

server process

The LDP server process is the passive participant in an LDP session. The server process usually resides on a target machine such as a PAD, PSN or gateway. The server process waits for a user process to initiate a session, and responds to commands from the user process. In response to user commands, the server may perform services on the target like reading and writing memory locations or setting breakpoints. ’Server’ is sometimes employed as a shorthand for ’server process’.

target

The 'target' in an LDP session is the PSN, PAD or gateway that is being loaded, dumped or debugged by the host. Normally, LDP will be implemented in the target as a server process. However, in some targets with strange requirements, notably the Butterfly, the target LDP may be a user process.

user process

The LDP user process is the active participant in an LDP session. The user process initiates and terminates the session and sends commands to the server process which control the session. The user process usually resides on a timesharing host and is driven by a higher-level entity (e.g., an application program like an interactive debugger). 'User' is sometimes employed as a shorthand for 'user process'.

word

A word is a sixteen-bit quantity.

INDEX