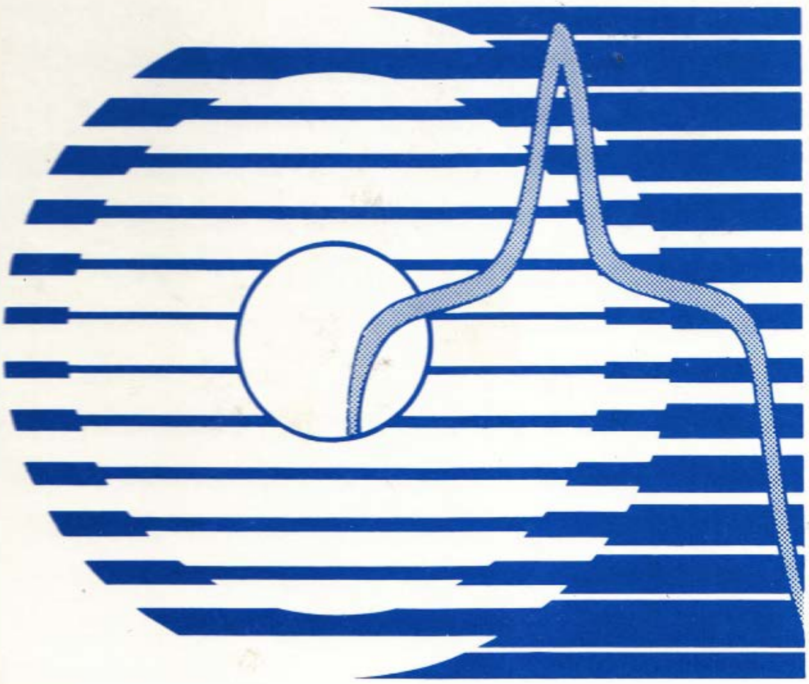
The SCSI Encyclopedia

Volume I
Phases and Protocol
(N–Z)



ENDL
PUBLICATIONS

The SCSI Encyclopedia

Volume I
Phases and Protocols
(N-Z)

Jeffrey D. Stai

ENDL Publications
Saratoga, California

Library of Congress Cataloging-in-Publication Data

Stai, Jeffrey D., 1958-
The SCSI encyclopedia / Jeffrey D. Stai.
p. cm. -- (ENDL SCSI Series)
Contents: v. 1, A-M. Phases and protocols -- v. 1, N-Z. Phases and protocols.
ISBN 1-879936-11-9 (v. 1, A-M). -- ISBN 1-879936-12-7 (v. 1, N-Z). -- ISBN 1-879936-10-0 (set)
1. Computer interfaces--Standards. 2. SCSI (Computer bus) 3. Local area networks (Computer networks) I. Title. II. Series.
TK7887.5.S73 1991
621.39'81--dc20 91-17070
CIP

Though the author and publisher have made every effort to ensure the accuracy and completeness of information contained in this book, we assume no responsibility for errors, inaccuracies, omissions, or any inconsistency therein.

ENDL SCSI Series

The publisher offers generous discounts on this book when ordered in bulk quantities. For more information, write:

Special Sales/Library Marketing
ENDL Publications
14426 Black Walnut Court
Saratoga, CA 95070

The complete set of books in the SCSI Encyclopedia may be ordered under
ISBN 1-879936-10-0

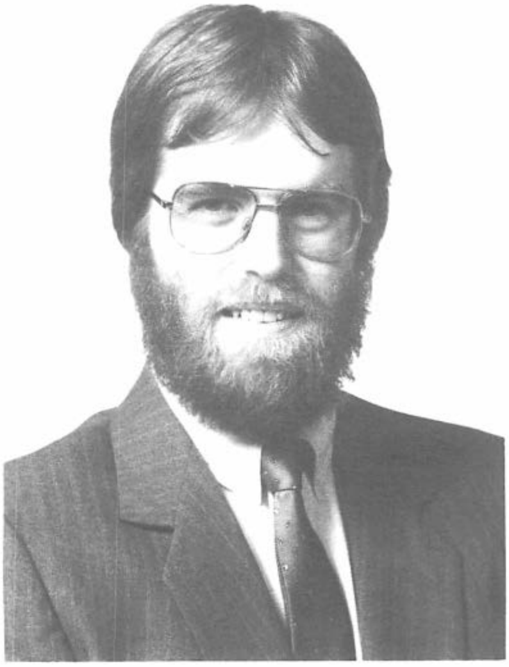
Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

ISBN 1-879936-12-7

ENDL Publications, Saratoga, California

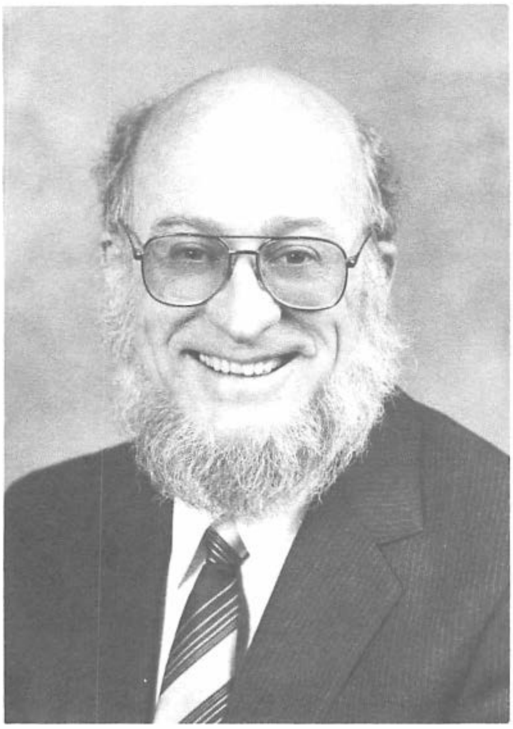
For Mary,
who put up with it.

About the Author



Jeffrey Stai has learned more about SCSI than anyone should be forced to know. He has designed SCSI chips, SCSI controllers, and SCSI software over an eleven year career. He has been the principal representative for Western Digital to X3T9.2 for several years.

About the Publisher



Dal Allan has spent over 30 years working with computers, and specializes in storage systems and related issues. He claims to have had a lot of hair before he became Vice Chairman of X3T9.2 (he lies! - JDS). He has been, and continues to be, active in the development of other industry standard interfaces such as the Intelligent Peripheral Interface, AT Attachment, and Fiber Channel.

Preface

A work like this cannot be called a Labor of Love; it is more like a Labor of Luck. I was lucky enough to be chosen to represent a little company called Adaptive Data Systems (ADSI) to X3T9.2 and be part of the effort to make SCSI a standard. I was fortunate enough to be involved in several projects at ADSI and Western Digital Corporation that exposed me to so many different facets of SCSI. And, I am amazed to have family and friends who put up with the two years of nights and weekends spent grinding this thing out.

There are those (and you know who you are!) who would argue the kind of luck that plunged me into the SCSI effort. Good or Bad?

I have made every effort to be accurate, but I am a mere human. If there is any discrepancy between the SCSI Encyclopedia and the SCSI Standard, follow the Standard. Please be sure to contact either or both ENDL and X3T9.2, so the discrepancy can be addressed. Heck, there is probably room for improvement in here somewhere, so all comments are more than welcome.

I do not apologize for the irreverence of the style. Let's face it; SCSI is dull-dull-dull. Anything that can be done to make the reading easier seems necessary and appropriate. Since Initiators and Targets figure prominently in almost every SCSI transaction, we have occasionally replaced those titles with proper names; Targets become "Tanya" or "Tom", Initiators become "Ian" or "Iris". Proper names allow the use of active verbs, which tend to improve readability. I have even attempted humor on occasion....

My opinions are sprinkled throughout the Encyclopedia; in fact, there is one section dedicated to my opinions, which I call SCSI Etiquette. Since the SCSI Standard often allows more than one way to do something (there's an understatement!), opinions become inevitable.

A work of this magnitude could not be completed without a careful and thoughtful review. I would like to thank Kurt Chan, Erik Jessen, Larry Lamers, John Lohmeyer, Bill Spence, and Gary Stephens for their tremendous help and encouragement.

It has been said that someone who can correctly answer ten SCSI questions out of a hundred is a SCSI Guru, someone who can correctly answer twenty SCSI questions out of a hundred is a SCSI God, while someone who claims they can correctly answer all hundred SCSI questions must be a SCSI Devil, because he must be lying! These guys live on Olympus!

Jeffrey D. Stai
Placentia, California
March 1991

About the SCSI Series

This volume is the second part of the SCSI Series by ENDL Publications. The first part of the series is the SCSI Bench Reference, which re-packages the SCSI Standard with timing charts, examples, easy references, and improved table structures. That book is for intermediate and experienced SCSI users.

Volume I of the SCSI Encyclopedia is intended for all SCSI users, such as the SCSI beginner, who should start with the Study Guide at the front of each volume. The intermediate user can use the SCSI Encyclopedia to bolster understanding of difficult subjects, such as the SCSI Message System. The experienced user will find here discussions on such new subjects as Wide Data Transfer and Command Queuing.

The encyclopedic format was chosen so that you can easily access only the information desired. The need to provide sufficient detail on each topic caused Volume I of the SCSI Encyclopedia to grow to a size that required a split into two separate books, A-M and N-Z.

Volume I deals with the phases and protocols of the low-level interface. In other words, sections 1 through 5 of the SCSI-2 Standard, and part of section 6, are covered in this volume. Volume I covers anything to do with cables and connectors, drivers and receivers, signals and phases, messages and nexuses. (Nexuses? So read the book!)

Volume II will cover the Direct Access Device (disk drive) command set. This includes sections 7 and 8 of the SCSI Standard, and the rest of section 6 that Volume I did not cover. The "generic" commands in section 7 are discussed from a disk point of view. Volume II covers READ and WRITE, MODE SELECT and INQUIRY, FORMAT and REQUEST SENSE, among others.

Volume III will cover Sequential Access Devices (tape drives). This includes sections 7 and 9 of the SCSI Standard, and the rest of section 6 that Volume I did not cover. This time, the "generic" commands in section 7 are discussed from a tape point of view, to give the tape-oriented reader the fullest value. Volume III covers READ and WRITE, MODE SELECT and INQUIRY, RECOVER BUFFERRED DATA and REQUEST SENSE, among others.

Volume IV will cover all Optical Devices, such as write-once read-multiple (WORM) drives, CDROM, and magneto-optic (MO) drives. This includes sections 7, 12, 13, and 15 of the SCSI Standard, and the rest of section 6 that Volume I did not cover. As you might hope, the "generic" commands in section 7 are discussed from the "optical" point of view, again for the fullest value. Volume IV covers READ and WRITE, MODE SELECT and INQUIRY, ERASE and REQUEST SENSE, among others.

The following subjects are covered in Volume I (A-M) of the SCSI Encyclopedia:

A Cable.
ABORT Message.
ABORT TAG message.
ACK Signal.
ACKB Signal.
Active I/O Process.
Active Pointers.
Active Pull-Up.
Arbitration Delay.
ARBITRATION Phase.
Assertion Period.
Assert Signal.
Asynchronous Data Transfer.
Asynchronous Event Notification (AEN).
ATN Signal.
Attention Condition.
B Cable.
Between Phases.
BSY Signal.
Bus Clear Delay.
BUS DEVICE RESET Message.
Bus Free Delay.
BUS FREE Phase.
Bus ID.
Bus Phases.
Bus Phase Signals.
Bus Set Delay.
Bus Settle Delay.
Bus Timing.
C/D Signal.
Cables.
Cable Skew Delay.
CAM (Common Access Method).
Chips.
CLEAR QUEUE Message.
COMMAND COMPLETE Message.
Command Descriptor Block.
COMMAND Phase.
Command Pointer.
COMMAND TERMINATED Status.
Condition.
Connect.
Connected I/O Process.
Connection.
Connection Phases.
Connectors.
Contingent Allegiance.
Control Byte.
Controller.
Control Signals.
Current I/O Process.
Data Bus Signals.
DATA IN Phase.
DATA OUT Phase.
DATA Phase.
Data Pointer.
Data Release Delay.
Deassert.
Deskew Delay.
Device.
Differential Interface.
DIFFSENS.
Disconnect.
Disconnection Delay.
DISCONNECT Message.
ECA.
Error Recovery.
Etiquette.
Extended Contingent Allegiance (ECA) Condition
Extended Messages.
False.
Fast Assertion Period.
Fast Cable Skew Delay.
Fast Data Transfer.
Fast Deskew Delay.
Fast Hold Time.
Fast Negation Period.
Forced Perfect Termination (FPT).
Hard Reset.
HEAD OF QUEUE TAG Message.
Hold Time.
Host Adapter.
I/O Process.
I/O Signal.
IDENTIFY Message.
IGNORE WIDE RESIDUE Message.
Information Transfer Phases.
Initial Connection.
Initialization.
INITIATE RECOVERY Message.
Initiator.
INITIATOR DETECTED ERROR Message.
LINKED COMMAND COMPLETE Messages.
Linked Commands.
Logical Block.
Logical Block Address (LBA).
Logical Unit.
Logical Unit Number (LUN).
Message.
MESSAGE IN Phase.
MESSAGE OUT Phase.
MESSAGE PARITY ERROR Message.
MESSAGE REJECT Message.
Message System.
MODIFY DATA POINTER Message.
MSG Signal.

Study Guide

We think you'll all agree that reading an encyclopedia starting at "A" and going through to "Z" is:

(a) Dullsville.
(b) A real drag, man.
(c) A bummer.
(d) Bogus, dude.

Pick your era.... To save you from that drudgery, we have a Study Guide. Each general topic noted below has a list of subject topics within the Encyclopedia that combine to describe the general topic. So, have fun!

Learning SCSI

Lesson #1: Bus and Devices

- SCSI Device
- SCSI Bus
- Initiator
- Target
- Logical Unit

Lesson #2: Processes and Phases

- Nexus
- I/O Process
- Phase
- Bus Phases
- Condition

Lesson #3: Bus Control

- Path Control
- Pointers
- Message
- Message System
- Error Recovery

Lesson #4: Protocols

- BUS FREE Phase
- ARBITRATION Phase
- SELECTION Phase
- Selection Time-out
- RESELECTION Phase
- Reselection Time-out
- Between Phases
- Asynchronous Data Transfer
- Attention Condition
- Reset Condition
- Unexpected BUS FREE Phase

Lesson #5: Physical Issues

- Single-Ended Interface
- Differential Interface
- Assert, Negate, and Release Signal
- Cables
- Connectors
- Termination
- Terminator Power
- Parity
- Wire-OR Glitch

Lesson #6: High Level Issues

- Command Descriptor Block (CDB)
- Status
- Logical Block
- Contingent Allegiance Condition
- Unit Attention Condition
- Hard Reset
- Linked Commands
- Extended Messages

Lesson #7: Advanced Data Transfer

- Synchronous Data Transfer
- Synchronous Data Transfer Negotiation
- Fast Data Transfer
- Wide Data Transfer
- Wide Data Transfer Negotiation
- P Cable

Lesson #8: Advanced Topics

- Queue
- Soft Reset
- Asynchronous Event Notification (AEN)
- Extended Contingent Allegiance (ECA) Condition
- TERMINATE I/O PROCESS Message and COMMAND TERMINATED Status
- Target Routine

Lesson #9: Implementation

- Chips
- Host Adapter
- Controller

Assorted Subjects

Helpful Hints

- Etiquette

Data Transfer Protocols

- REQ Signal
- ACK Signal
- DATA Phases
- Asynchronous Data Transfer
- Synchronous Data Transfer
- Fast Data Transfer
- Wide Data Transfer
- P Cable

Error Recovery

- Pointers
- Message System
- Error Recovery

Timing

- Bus Timing

Bus Signals

- ACK Signal
- ATN Signal
- BSY Signal
- C/D Signal
- I/O Signal
- MSG Signal
- REQ Signal
- RST Signal
- SEL Signal
- Data Bus Signals
- Parity
- Terminator Power

Reset

- Reset Condition
- RST Signal
- Hard Reset
- Soft Reset

Aborting an Operation in order of severity:

- TERMINATE I/O Message
- ABORT TAG Message
- ABORT Message
- CLEAR QUEUE Message
- BUS DEVICE RESET Message
- Reset Condition

Messages

- ABORT
- ABORT TAG
- BUS DEVICE RESET
- CLEAR QUEUE
- COMMAND COMPLETE
- DISCONNECT
- HEAD OF QUEUE TAG
- IDENTIFY
- IGNORE WIDE RESIDUE
- INITIATE RECOVERY
- INITIATOR DETECTED ERROR
- LINKED COMMAND COMPLETE
- MESSAGE PARITY ERROR
- MESSAGE REJECT
- MODIFY DATA POINTER
- NO OPERATION
- ORDERED QUEUE TAG
- RELEASE RECOVERY
- RESTORE POINTERS
- SAVE DATA POINTER
- SIMPLE QUEUE TAG
- SYNCHRONOUS DATA TRANSFER REQUEST
- TERMINATE I/O PROCESS
- WIDE DATA TRANSFER REQUEST

Using the SCSI Encyclopedia

The **SCSI Encyclopedia** is organized, naturally enough, in alphabetical order. There are two types of topics: major topics and minor topics. Major topics cover subjects like ARBITRATION Phase, which is a SCSI topic that calls for a lot of discussion and examples. Major topics typically begin with a general overview of the subject, and this will typically include an illustration or flow diagram. After the overview, there are several detailed illustrations and examples.

Minor topics cover subjects like Arbitration Delay, which is a subject encompassed by one or more major topics. Minor topics will have a detailed description of the subject, with a reference to a major topic for more information.

A few editorial conventions have been adopted to make information easier to find. The beginning of a topic is identified by a special font; e.g., **ARBITRATION Phase**. Within a topic, a reference to another topic is highlighted as follows: ***see Arbitration Delay***. This convention is very analogous to a HyperText system (except, of course, you can click your mouse on a piece of paper all day with no effect...). Highlighting is done only once on a page or once within a topic to reduce "font fatigue". Also, since they are used so often within each topic, the names "Initiator" and "Target" are not highlighted.

In several places, the terms "Initiator" and "Target" are replaced by proper names, like "Ian" and "Tanya" or "Iris" and "Tom". While this seems odd for a technical work, anthropomorphism allowed us to use simpler language for complicated concepts. And, it does spice things up a little...

We have included the "Examples" section at the end of both volumes so that they are available when reading either volume.

Most Important Note: This is **not** the SCSI Standard! Please refer to the Standard when judging the compliance of any implementation.

Within timing diagrams, certain conventions are used, which are illustrated in Figure 1 below.

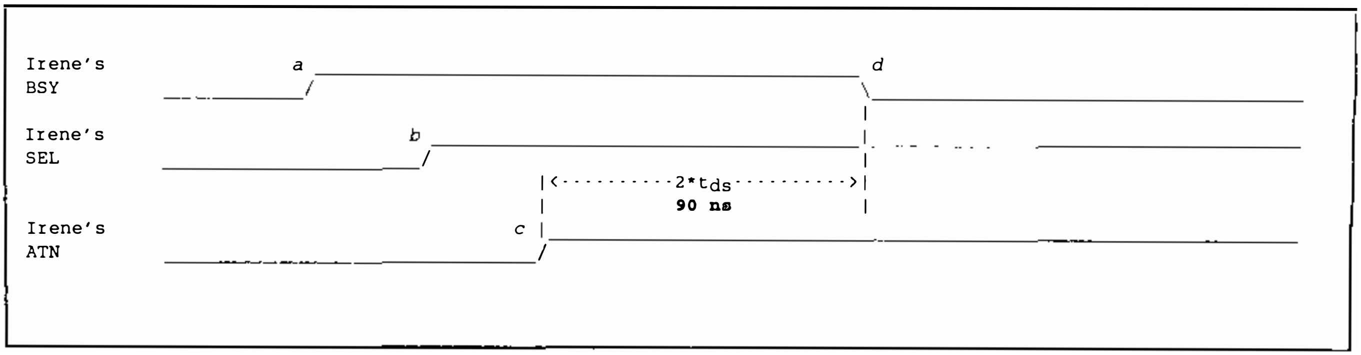
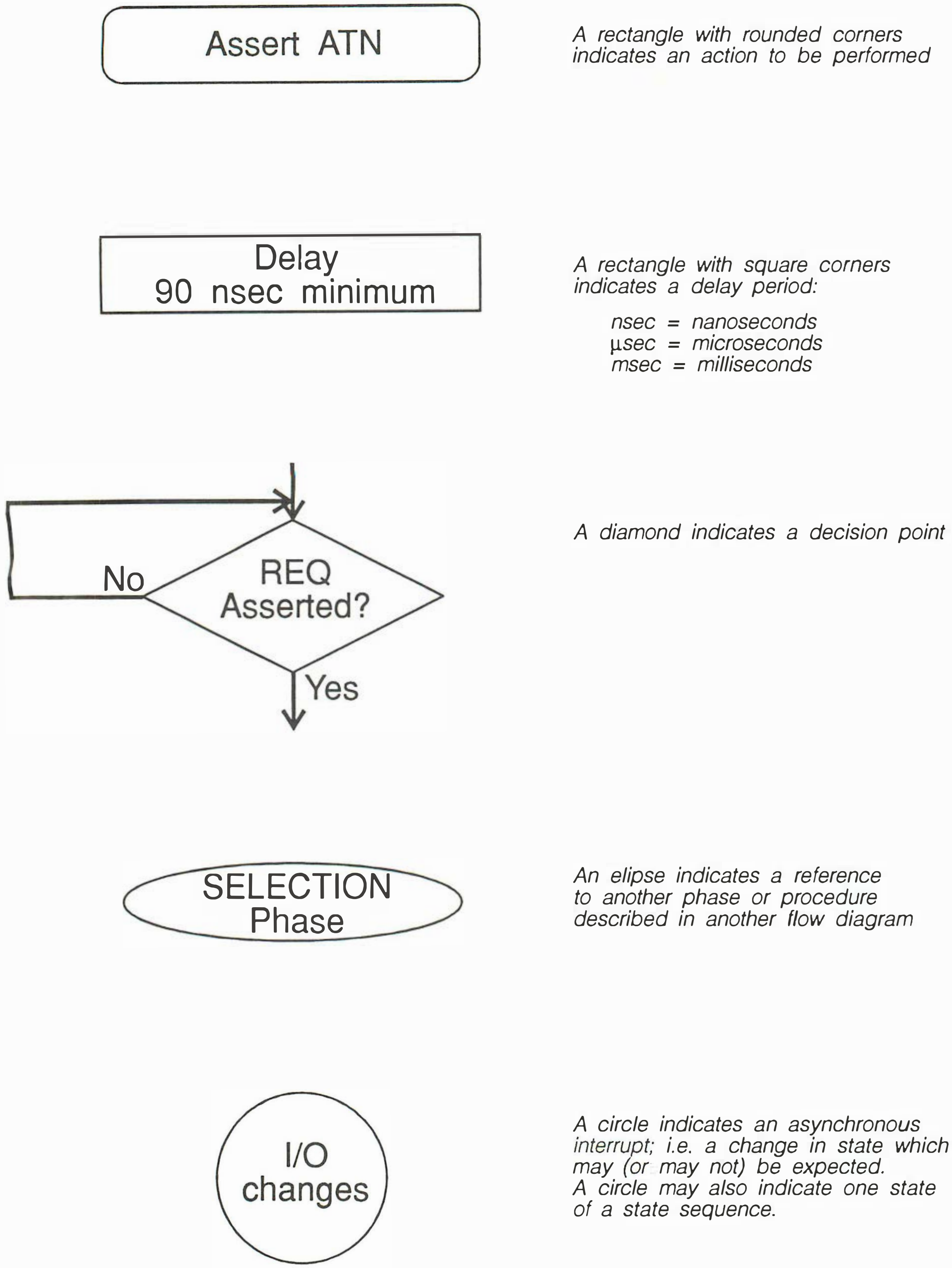


FIGURE 1: EXAMPLE TIMING DIAGRAM

Timing diagram conventions:

- The "owner" of the signal (either as a driver or receiver) is shown on the left hand side.
- All signals are shown as "high-true/low-false", independent of the electrical interface type (***Single Ended Interface*** or ***Differential Interface***). For instance, Single Ended Interface signals are electrically low when true, but the timing diagram shows them high when true. While this may seem confusing, it keeps the timing diagrams generic for all electrical interfaces.
- A signal that is not driven is shown as dashes "-------". A signal that is in a "don't care" state is shown as "XXXXXXX".
- A lower case letter (e.g., *a*) within the timing diagram indicates an event that is described in text following the diagram that begins with the same letter (e.g., (a)).
- Times between two edges are given both as actual time (e.g., **90 ns**), and as defined by SCSI ***Bus Timing*** values (e.g., t_{ds}, which is a ***Deskew Delay***).

The various symbols used in flow diagrams are illustrated in Diagram 1.



A rectangle with rounded corners indicates an action to be performed

A rectangle with square corners indicates a delay period:

nsec = nanoseconds
µsec = microseconds
msec = milliseconds

A diamond indicates a decision point

An elipse indicates a reference to another phase or procedure described in another flow diagram

A circle indicates an asynchronous interrupt; i.e. a change in state which may (or may not) be expected.
A circle may also indicate one state of a state sequence.

DIAGRAM 1: KEY TO SCSI FLOW DIAGRAM SYMBOLS

Negate Signal.
Negation Period.
Nexus.
NO OPERATION Message.

Negate Signal. To drive a signal to the ***False*** state. "Negated" is used in the SCSI standard and in this reference to indicate that a signal or signal pair is driven to the False or "zero" state. Compare to ***Assert*** and ***Release***. See ***Signal Levels***.

Negation Period. *t_{np} - 90 nsec.* The Negation Period is the minimum period of time from the negating edge of the ***REQ Signal*** or ***ACK Signal*** to the asserting edge of the next REQ or ACK, respectively. The Negation Period measures from REQ pulse to REQ pulse, and ACK pulse to ACK pulse, during a synchronous data transfer. It does not restrict REQ to ACK or ACK to REQ timing. See ***Synchronous Data Transfer*** and ***Assertion Period***. More precisely:

- For a device sending data, the Negation Period is the minimum pulse spacing of the signal that it sends to strobe data into the receiving device.
- For a device receiving data, the Negation Period is the minimum pulse spacing of the signal it sends to the other device that allows another transfer to occur.

Nexus. Say What?!?!? You should have heard the SCSI Committee when this name was proposed by the editors:

- "Sounds like a disease!"
- "Sounds like a gland!"
- "Sounds like a shampoo!"

Back to Webster's: "Nexus [Latin]: a link or connection."

- "Hey, that's just what we're talking about!"

In SCSI, Nexus refers to a relationship between two SCSI devices, one an Initiator and the other a Target. The word was chosen so that the different types of relationships could be easily described without using a word that had a prior connotation. A SCSI device must have a particular Nexus with another device before a particular operation may be performed.

It is important to note that a Nexus is a relationship, it does not refer to a specific action. Specific actions can establish or revive a Nexus:

- An Initiator establishes a Nexus during an ***Initial Connection***.
- A Target revives a Nexus during a ***Reconnection***.

- An Initiator may also revive a previously established Nexus via another Connection.

Other actions can further restrict the relationships that are established or revived by the use of the ***IDENTIFY Message*** and the ***Queue Tag Messages***.

The different types of Nexuses (ask Webster about that plural) are:

- **I_T Nexus**: The basic type of Nexus. An I_T Nexus is established after a successful ***SELECTION Phase***. An I_T Nexus is revived after a successful ***RESELECTION Phase***. An I_T Nexus is further restricted to form all other types of Nexuses.

- **I_T_L Nexus**: An I_T_L Nexus is established after an IDENTIFY Message is successfully transferred from the Initiator to the Target following SELECTION Phase. An I_T_L Nexus is revived after an IDENTIFY Message is successfully transferred from the Target to the Initiator following RESELECTION Phase. In both cases the LUNTAR bit of the IDENTIFY message is set to zero, indicating that a ***Logical Unit*** is identified.

- **I_T_R Nexus**: An I_T_R Nexus is established after an IDENTIFY Message is successfully transferred from the Initiator to the Target following SELECTION Phase. An I_T_R Nexus is revived after an IDENTIFY Message is successfully transferred from the Target to the Initiator following RESELECTION Phase. In both cases the LUNTAR bit of the IDENTIFY message is set to one, indicating that a ***Target Routine*** is identified.

- **I_T_x Nexus**: This is a generic term that refers to either an I_T_L or an I_T_R Nexus. It does not refer to an I_T or an I_T_L_Q Nexus.

- **I_T_L_Q Nexus**: An I_T_L_Q Nexus is established after any Queue Tag Message is successfully transferred from the Initiator to the Target following the IDENTIFY Message. An I_T_L_Q Nexus is revived after a ***SIMPLE QUEUE TAG Message*** is successfully transferred from the Target to the Initiator following the IDENTIFY Message. In both cases the LUNTAR bit of the IDENTIFY message is set to zero, indicating that a ***Logical Unit*** is identified.

- **I_T_R_Q Nexus**: An I_T_R_Q Nexus is not allowed! You are not allowed to issue a ***Queued*** command to a ***Target Routine***.

- **I_T_x_y Nexus**: This is a generic term that refers to either an I_T_L, an I_T_R, or an I_T_L_Q Nexus. It does not refer to an I_T Nexus.

Table 1 and Table 2 summarize the different Nexus types:

TABLE 1: NEXUS ESTABLISHED BY AN INITIATOR

Nexus Type	SELECTION Phase	IDENTIFY Message	Queue Tag Message	Preceded by Nexus Type
I_T	Yes	No	No	none
I_T_L	Yes	Yes; LUNTAR=0	No	I_T
I_T_R	Yes	Yes; LUNTAR=1	No	I_T
I_T_L_Q	Yes	Yes; LUNTAR=0	Yes; any type	I_T_L

TABLE 2: NEXUS REVIVED BY A TARGET

Nexus Type	RESELECTION Phase	IDENTIFY Message	Queue Tag Message	Preceded by Nexus Type
I_T	Yes	No	No	none
I_T_L	Yes	Yes; LUNTAR=0	No	I_T
I_T_R	Yes	Yes; LUNTAR=1	No	I_T
I_T_L_Q	Yes	Yes; LUNTAR=0	Yes; SIMPLE QUEUE TAG only	I_T_L

NO OPERATION Message. The NO OPERATION Message does nothing. Period. What more can we say? Here is the message format:

Bit Byte	7	6	5	4	3	2	1	0
0	Message Code = 08 hex							

But, wait, we hear you cry, what's it for? Well, there are a couple of uses....

NO OPERATION Use #1: Let's say you have an Initiator, that decides that its internal buffer is going to overflow during a ***DATA Phase***. It then creates the ***Attention Condition*** with the intent of sending a ***DISCONNECT Message***. But this time, the Initiator catches up and it no longer needs to disconnect. As long as the Attention Condition exists, the Initiator will be expected to send a message. So, the Initiator maintains the Attention Condition until the next ***MESSAGE OUT Phase***. It then sends the NO OPERATION message to the Target, which has no effect, and the transfer continues.

Comment: This is a use that makes sense, and is preferable to creating the Attention Condition and then taking it away (see ***Etiquette***). On the other hand, if the Initiator makes a habit of this the data transfer rate can really go down the drain (i.e., slow down). It might be better to use some of the disconnect control methods defined within the MODE SELECT Command.

NO OPERATION Use #2: What happens when a Target enters the MESSAGE OUT Phase, but the Initiator hasn't created the Attention Condition? Well, the Initiator sends a NO OPERATION message and hopefully everything continues normally.

Comment: A MESSAGE OUT Phase without the Attention Condition? That's less probable than a politician without greed! On the other hand, this could happen if an Attention Condition was created and then ended. Any Initiator presented with this situation either played games with the Attention Condition, or it could assume that the Target is broken and take appropriate action (example: see ***ABORT Message***), or the Target is a ***SCSI-3*** Device and it's doing something we haven't yet imagined!

Summary of Use: The NO OPERATION message is sent only by an Initiator when it has nothing better to send during a MESSAGE OUT Phase.

This page is nearly blank!

ORDERED QUEUE TAG Message.

ORDERED QUEUE TAG Message. The ORDERED QUEUE TAG Message is used by an Initiator to get an ***I/O Process*** executed in the order received (relative to all other I/O Processes previously received by the Target from all Initiators) when a ***Queue*** is used by the Target. ORDERED QUEUE TAG is a two byte message:

Bit Byte	7	6	5	4	3	2	1	0
0	Message Code = 22 hex							
1	Queue Tag							

The second byte of the message specifies the ***Queue Tag*** associated with the ***Nexus*** being established by this message.

This message causes the new I/O Process associated with the Nexus to be put at the rear of the Queue, behind any other I/O Processes that currently may be queued. Any I/O Processes in the Queue prior to the Ordered I/O Process must be executed first. If no I/O Process is currently being executed for that ***Logical Unit***, then the new I/O Process is made active and executed immediately.

This type of Queue Tag also affects the ordering of Tagged I/O Processes received after it. Any incoming I/O Processes using the ***SIMPLE QUEUE TAG Message*** must be queued after an ORDERED QUEUE TAG. The intent of an ORDERED QUEUE TAG is that it is executed in the order received, relative to what came before and what is to follow. See ***Queue*** for an illustration of this behavior.

Note that an I/O Process begun with a ***HEAD OF QUEUE TAG Message*** can still "cut" to the front of the Queue, including any ORDERED QUEUE TAGS in the Queue.

Summary of Use: The ORDERED QUEUE TAG Message is sent only by an Initiator to a Target to cause the I/O Process to be placed at the rear of the Queue.

P Cable.
Parity.
Path Control.
Peripheral Device.
Phase.
Pointers.
Power-On to Selection Time.
Protocol Chips.
Pull-Up.

P Cable. "I didn't see any 'P Cable' in my copy of SCSI-2!" That's right, you didn't. The P Cable is a new feature for ***SCSI-3***, which has so much more utility for new 16-bit designs in the near future than the combination of the ***A Cable*** and the ***B Cable***. The P Cable is the best ***Cable*** to use for 16-bit designs.

Why a P Cable? The P cable was defined in response to concern about the cost and space requirements of the ***B Cable***. With the B Cable, if you only want 16-bit data transfers, you must still use the A Cable and the full B cable and connectors. Two connectors and cables to each device in this era of smaller and smaller devices and systems is painful:

- Where are you going to route those fat cables?
- Where are you going to put those two big fat connectors on the device?
- Who wants to pay for all that extra metal, anyway?

Thus, the P Cable was born.

Since the P Cable is not in your copy of SCSI-2, we will provide the ***Connector*** pinouts for the P Cable here. The P Cable is a 68 conductor cable. Table 5 shows the P Cable connections for the ***Single-Ended Interface***, and Table 6 shows the P Cable connections for the ***Differential Interface***. See ***Wide Data Transfer*** for details on P Cable data transfer.

TABLE 5: SINGLE-ENDED P CABLE CONNECTIONS

Signal Name	Connector Contact Number	Cable Conductor Numbers		Connector Contact Number	Signal Name
GROUND	1	1	2	35	-DB(12)
GROUND	2	3	4	36	-DB(13)
GROUND	3	5	6	37	-DB(14)
GROUND	4	7	8	38	-DB(15)
GROUND	5	9	10	39	-DB(P1)
GROUND	6	11	12	40	-DB(0)
GROUND	7	13	14	41	-DB(1)
GROUND	8	15	16	42	-DB(2)
GROUND	9	17	18	43	-DB(3)
GROUND	10	19	20	44	-DB(4)
GROUND	11	21	22	45	-DB(5)
GROUND	12	23	24	46	-DB(6)
GROUND	13	25	26	47	-DB(7)
GROUND	14	27	28	48	-DB(P)
GROUND	15	29	30	49	GROUND
GROUND	16	31	32	50	GROUND
TERMPWR	17	33	34	51	TERMPWR
TERMPWR	18	35	36	52	TERMPWR
RESERVED	19	37	38	53	RESERVED
GROUND	20	39	40	54	GROUND
GROUND	21	41	42	55	-ATN
GROUND	22	43	44	56	GROUND
GROUND	23	45	46	57	-BSY
GROUND	24	47	48	58	-ACK
GROUND	25	49	50	59	-RST
GROUND	26	51	52	60	-MSG
GROUND	27	53	54	61	-SEL
GROUND	28	55	56	62	-C/D
GROUND	29	57	58	63	-REQ
GROUND	30	59	60	64	-I/O
GROUND	31	61	62	65	-DB(8)
GROUND	32	63	64	66	-DB(9)
GROUND	33	65	66	67	-DB(10)
GROUND	34	67	68	68	-DB(11)

The leading minus sign (e.g., "-BSY") refers to the active low nature of a Single-Ended Interface signal.

TABLE 6: DIFFERENTIAL P CABLE CONNECTIONS

Signal Name	Connector Contact Number	Cable Conductor Numbers		Connector Contact Number	Signal Name
+DB(12)	1	1	2	35	-DB(12)
+DB(13)	2	3	4	36	-DB(13)
+DB(14)	3	5	6	37	-DB(14)
+DB(15)	4	7	8	38	-DB(15)
+DB(P1)	5	9	10	39	-DB(P1)
GROUND	6	11	12	40	GROUND
+DB(0)	7	13	14	41	-DB(0)
+DB(1)	8	15	16	42	-DB(1)
+DB(2)	9	17	18	43	-DB(2)
+DB(3)	10	19	20	44	-DB(3)
+DB(4)	11	21	22	45	-DB(4)
+DB(5)	12	23	24	46	-DB(5)
+DB(6)	13	25	26	47	-DB(6)
+DB(7)	14	27	28	48	-DB(7)
+DB(P)	15	29	30	49	-DB(P)
DIFFSENS	16	31	32	50	GROUND
TERMPWR	17	33	34	51	TERMPWR
TERMPWR	18	35	36	52	TERMPWR
RESERVED	19	37	38	53	RESERVED
+ATN	20	39	40	54	-ATN
GROUND	21	41	42	55	GROUND
+BSY	22	43	44	56	-BSY
+ACK	23	45	46	57	-ACK
+RST	24	47	48	58	-RST
+MSG	25	49	50	59	-MSG
+SEL	26	51	52	60	-SEL
+C/D	27	53	54	61	-C/D
+REQ	28	55	56	62	-REQ
+I/O	29	57	58	63	-I/O
GROUND	30	59	60	64	GROUND
+DB(8)	31	61	62	65	-DB(8)
+DB(9)	32	63	64	66	-DB(9)
+DB(10)	33	65	66	67	-DB(10)
+DB(11)	34	67	68	68	-DB(11)

The leading plus and minus signs (e.g., "-BSY" and "+BSY") refers to the differential pairs of a Differential Interface signal.

The SCSI-3 proposal notes that "Conductor Number" refers to the conductor position when using 0.025 inch centerline flat ribbon cable. Other cable types may be used to implement equivalent contact assignments.

Upgrade Path. But wait, there's more! Because the P Cable carries a 16-bit data path on a single cable, it can also carry an 8-bit data path on the same cable. This is pretty nifty, because this means you can build Initiators and Targets that do 8-bit today, and tomorrow replace them with 16-bit devices without changing your existing cabling! This really simplifies field upgrades.

You do have to follow a rule, though. You must leave the following signals unconnected when using the P Cable on an 8-bit device:

- Single-Ended: -DB(8), -DB(9), -DB(10), -DB(11), -DB(12), -DB(13), -DB(14), -DB(15), and -DB(P1).
- Differential: -DB(8), -DB(9), -DB(10), -DB(11), -DB(12), -DB(13), -DB(14), -DB(15), -DB(P1), +DB(8), +DB(9), +DB(10), +DB(11), +DB(12), +DB(13), +DB(14), +DB(15), and +DB(P1).

Connect the other signals normally, with RESERVED signals left open.

But what about 32-bits? The ***X3T9.2 Committee*** is considering a P Cable-compatible method for expanding to 32-bits. A "Q Cable" would carry the additional 16-bits and the ***REQB Signal*** and ***ACKB Signal*** needed for the 32-bit data path.

Warning! The P Cable is still under development by the X3T9.2 Committee. Contact them for the latest information.

Parity. To quote Seymour Cray: "Parity is for farmers!". Mr. Cray will probably never use SCSI since parity is a fact of life in SCSI bus transactions. Parity is the main method used by SCSI for error detection on the bus (see ***Error Recovery***). Parity covers eight data bits; if ***Wide Data Transfers*** are used, then there is a parity bit for each 8-bit byte (2 parity bits for 16-bit data, 4 parity bits for 32-bit data).

Parity on the SCSI bus is odd Parity (no we don't mean strange!). Odd parity means that the total number of ***True*** (one) bits in the eight data bits plus the parity bit is odd. Parity is calculated on the logical state of the bus; i.e., one or zero. For example, if the data bits are all zero, the parity bit would be one, since then the total number of bits will be one, which is odd. If the data bits are all one, the parity bit would also be one, with a total of nine bits set to one.

Table 7 and Table 8 give a complete list of parity values for all possible eight bit data bytes.

TABLE 7: TABLE OF EIGHT BIT ODD PARITY VALUES (0-127)

Binary	Hex	Parity	Binary	Hex	Parity	Binary	Hex	Parity	Binary	Hex	Parity
00000000	00	1	00100000	20	0	01000000	40	0	01100000	60	1
00000001	01	0	00100001	21	1	01000001	41	1	01100001	61	0
00000010	02	0	00100010	22	1	01000010	42	1	01100010	62	0
00000011	03	1	00100011	23	0	01000011	43	0	01100011	63	1
00000100	04	0	00100100	24	1	01000100	44	1	01100100	64	0
00000101	05	1	00100101	25	0	01000101	45	0	01100101	65	1
00000110	06	1	00100110	26	0	01000110	46	0	01100110	66	1
00000111	07	0	00100111	27	1	01000111	47	1	01100111	67	0
00001000	08	0	00101000	28	1	01001000	48	1	01101000	68	0
00001001	09	1	00101001	29	0	01001001	49	0	01101001	69	1
00001010	0A	1	00101010	2A	0	01001010	4A	0	01101010	6A	1
00001011	0B	0	00101011	2B	1	01001011	4B	1	01101011	6B	0
00001100	0C	1	00101100	2C	0	01001100	4C	0	01101100	6C	1
00001101	0D	0	00101101	2D	1	01001101	4D	1	01101101	6D	0
00001110	0E	0	00101110	2E	1	01001110	4E	1	01101110	6E	0
00001111	0F	1	00101111	2F	0	01001111	4F	0	01101111	6F	1
00010000	10	0	00110000	30	1	01010000	50	1	01110000	70	0
00010001	11	1	00110001	31	0	01010001	51	0	01110001	71	1
00010010	12	1	00110010	32	0	01010010	52	0	01110010	72	1
00010011	13	0	00110011	33	1	01010011	53	1	01110011	73	0
00010100	14	1	00110100	34	0	01010100	54	0	01110100	74	1
00010101	15	0	00110101	35	1	01010101	55	1	01110101	75	0
00010110	16	0	00110110	36	1	01010110	56	1	01110110	76	0
00010111	17	1	00110111	37	0	01010111	57	0	01110111	77	1
00011000	18	1	00111000	38	0	01011000	58	0	01111000	78	1
00011001	19	0	00111001	39	1	01011001	59	1	01111001	79	0
00011010	1A	0	00111010	3A	1	01011010	5A	1	01111010	7A	0
00011011	1B	1	00111011	3B	0	01011011	5B	0	01111011	7B	1
00011100	1C	0	00111100	3C	1	01011100	5C	1	01111100	7C	0
00011101	1D	1	00111101	3D	0	01011101	5D	0	01111101	7D	1
00011110	1E	1	00111110	3E	0	01011110	5E	0	01111110	7E	1
00011111	1F	0	00111111	3F	1	01011111	5F	1	01111111	7F	0

Table 8: Table of Eight Bit Odd Parity Values (128-255)

Binary	Hex	Parity	Binary	Hex	Parity	Binary	Hex	Parity	Binary	Hex	Parity
10000000	80	0	10100000	A0	1	11000000	C0	1	11100000	E0	0
10000001	81	1	10100001	A1	0	11000001	C1	0	11100001	E1	1
10000010	82	1	10100010	A2	0	11000010	C2	0	11100010	E2	1
10000011	83	0	10100011	A3	1	11000011	C3	1	11100011	E3	0
10000100	84	1	10100100	A4	0	11000100	C4	0	11100100	E4	1
10000101	85	0	10100101	A5	1	11000101	C5	1	11100101	E5	0
10000110	86	0	10100110	A6	1	11000110	C6	1	11100110	E6	0
10000111	87	1	10100111	A7	0	11000111	C7	0	11100111	E7	1
10001000	88	1	10101000	A8	0	11001000	C8	0	11101000	E8	1
10001001	89	0	10101001	A9	1	11001001	C9	1	11101001	E9	0
10001010	8A	0	10101010	AA	1	11001010	CA	1	11101010	EA	0
10001011	8B	1	10101011	AB	0	11001011	CB	0	11101011	EB	1
10001100	8C	0	10101100	AC	1	11001100	CC	1	11101100	EC	0
10001101	8D	1	10101101	AD	0	11001101	CD	0	11101101	ED	1
10001110	8E	1	10101110	AE	0	11001110	CE	0	11101110	EE	1
10001111	8F	0	10101111	AF	1	11001111	CF	1	11101111	EF	0
10010000	90	1	10110000	B0	0	11010000	D0	0	11110000	F0	1
10010001	91	0	10110001	B1	1	11010001	D1	1	11110001	F1	0
10010010	92	0	10110010	B2	1	11010010	D2	1	11110010	F2	0
10010011	93	1	10110011	B3	0	11010011	D3	0	11110011	F3	1
10010100	94	0	10110100	B4	1	11010100	D4	1	11110100	F4	0
10010101	95	1	10110101	B5	0	11010101	D5	0	11110101	F5	1
10010110	96	1	10110110	B6	0	11010110	D6	0	11110110	F6	1
10010111	97	0	10110111	B7	1	11010111	D7	1	11110111	F7	0
10011000	98	0	10111000	B8	1	11011000	D8	1	11111000	F8	0
10011001	99	1	10111001	B9	0	11011001	D9	0	11111001	F9	1
10011010	9A	1	10111010	BA	0	11011010	DA	0	11111010	FA	1
10011011	9B	0	10111011	BB	1	11011011	DB	1	11111011	FB	0
10011100	9C	1	10111100	BC	0	11011100	DC	0	11111100	FC	1
10011101	9D	0	10111101	BD	1	11011101	DD	1	11111101	FD	0
10011110	9E	0	10111110	BE	1	11011110	DE	1	11111110	FE	0
10011111	9F	1	10111111	BF	0	11011111	DF	0	11111111	FF	1

Path Control. "Path Control" is an archaic SCSI term that disappeared after the early revs of SCSI-1 (back in the pre-history of the early 1980's...). When an Initiator is ***Connected*** to a Target, it may be said that a "path" exists between the two devices. All commands, status, and data may pass over this path as long as the ***Connection*** exists. The ***Connection Phases*** are used to begin to establish this path.

Now, we have a path between two devices. In order to ensure the integrity of the commands, status, and data on the path, we use ***Parity*** to check each byte on the path. And technology marches on; we have new options for the high-speed data transfer on the path, such as ***Synchronous Data Transfer*** and ***Wide Data Transfer***.

With all of these parity checks and options, there must be some way of handling these details:

- Establishing the path to the appropriate ***Nexus***.
- Agreement between two ***SCSI Devices*** on the data transfer option to use for the path.
- Reporting parity errors and retrying the transfer.
- Breaking the connection, and therefore the path, with the agreement of both devices.
- Ending an ***I/O Process***.

With all of these things to do besides commands, status, and data, it seems like we could use a path control system. That's what the ***Pointers*** and the ***Message System*** do. Using Pointers allows both devices to have agreement on where a ***COMMAND Phase***, ***STATUS Phase***, or ***DATA Phase*** began, so that the transfer in the phase may be retried. The Message System allows for small "messages" to be sent to control the pointers and all of the other details listed above. The messages that control these details are listed in Table 9 below.

We suggest you now read ***Pointers***, and then ***Message System***.

TABLE 9: PATH CONTROL MESSAGE FUNCTIONS

Message Name	Path Control Function
ABORT (Out only)	Initiator Request End Connection and End I/O Process(es) Abnormally
ABORT TAG (Out only)	Initiator Request End Connection and End one I/O Process Abnormally
BUS DEVICE RESET (Out only)	Initiator Request End Connection and Reset Target
CLEAR QUEUE (Out only)	Initiator Request End Connection and all End I/O Processes Abnormally
COMMAND COMPLETE (In only)	Target Report End Connection and End I/O Process Normally
DISCONNECT (Out)	Initiator Request End the Current Connection
DISCONNECT (In)	Target Report End of the Current Connection
HEAD OF QUEUE TAG (Out only)	Initiator Establish Path for Queued I/O Process (I_T_L_Q Nexus)
IDENTIFY (Out)	Initiator Establish Path for Logical Unit or Target Process (I_T_x Nexus)
IDENTIFY (In)	Target Re-establish Path For Logical Unit or Target Process (I_T_x Nexus)
IGNORE WIDE RESIDUE (In only)	Target Report Adjustment to Wide Data Transfer Length
INITIATE RECOVERY (Out)	Begin Extended Contingent Allegiance Error Recovery
INITIATE RECOVERY (In)	Begin Extended Contingent Allegiance Error Recovery
INITIATOR DETECTED ERROR (Out only)	Initiator Report Command, Status, or Data Transfer Error with Target (Request Retry)
LINKED COMMAND COMPLETE (In only)	Target Request Switch to Next Saved Pointer Set for I/O Process (Do Next Command for I/O Process)
LINKED COMMAND COMPLETE (WITH FLAG) (In only)	Target Request Switch to Next Saved Pointer Set for I/O Process (Do Next Command for I/O Process). Echo "Flag" back to Host System.
MESSAGE PARITY ERROR (Out only)	Initiator Report MESSAGE IN Transfer Error (Request Retry)
MESSAGE REJECT (Out or In)	Report Inappropriate Message to Other Device
MODIFY DATA POINTER (In only)	Target Request to Apply Specified Offset to Active Data Pointer
NO OPERATION (Out only)	Do Nothing
ORDERED QUEUE TAG (Out only)	Initiator Establish Path for Queued I/O Process (I_T_L_Q Nexus)
RELEASE RECOVERY (Out only)	End Extended Contingent Allegiance Error Recovery
RESTORE POINTER (In only)	Target Request to Copy all Saved Pointers to Active Pointers
SAVE DATA POINTER (In only)	Target Request to Copy Active Data Pointer to Saved Data Pointer
SIMPLE QUEUE TAG (Out)	Initiator Establish Path for Queued I/O Process (I_T_L_Q Nexus)
SIMPLE QUEUE TAG (In)	Target Re-establish Path for Queued I/O Process (I_T_L_Q Nexus)
SYNCHRONOUS DATA TRANSFER REQUEST (Out or In)	Request/Agree to Synchronous Data Transfer Option
TERMINATE I/O PROCESS (In only)	Initiator Request Immediate Controlled End of Current I/O Process
WIDE DATA TRANSFER REQUEST (Out or In)	Request/Agree to Wide Data Transfer Option

Peripheral Device. A Peripheral Device is defined by SCSI to refer to the physical components of, or attached to, a ***SCSI Device***, which is usually a Target. A peripheral device is often a ***Logical Unit***, but you can also have a Logical Unit made up of several peripheral devices (for example, some disk drive arrays). Or, you can even have more than one Logical Unit access the same peripheral device (for example, logical partitions of a disk drive). The Peripheral Device is often coupled to a ***Controller*** to create a complete SCSI Target.

Other examples of peripheral devices include:

- Tape Drives
- Optical Disk Drives
- Printers
- Scanners
- etc....

Phase. See ***Bus Phases***. See also ***Conditions***.

Pointers. No this is not the "Hints and Tips" section. What we mean is Pointers to memory, as in "indirect" or "indexed" addressing. Before reading further here, be sure to read the section on ***Path Control*** as an introduction to the topic, and specifically the need for Pointers.

Okay, now you are thoroughly confused. It is often difficult to describe one topic without already describing other related topics. This section will begin to explain the concepts hinted at in Table 9 under Path Control, and the section on the ***Message System*** will finish it.

There are three types of ***Information Transfer Phase***, other than the ***Message Phases***: ***COMMAND Phase***, ***STATUS Phase***, and ***DATA Phase*** (only one data transfer direction is possible during a ***SCSI Command***). Therefore, we need pointers for each of these phases:

- Command Pointer
- Status Pointer
- Data Pointer

(By the way, only one data transfer direction is possible because the SCSI data transfer model has only one data pointer...)

We have to remember, though, that we have more to do than have Pointers to Command, Status, and Data, we must also be able to retry any of these transfers. To do that, we need to remember where the transfer started. This means we need a set of two Pointers for each phase; an ***Active Pointer*** and a ***Saved Pointer***. Our set of Pointers is now:

- Active Command Pointer
- Active Status Pointer
- Active Data Pointer

- Saved Command Pointer
- Saved Status Pointer
- Saved Data Pointer

The Message System has its own position management system to handle ***MESSAGE IN Phase*** and ***MESSAGE OUT Phase***.

Diagram 2 shows a graphical representation of these six Pointers and how they relate to host memory. The left side of the diagram shows the Saved Pointers. The top set of three Pointers is for the ***Current I/O Process***. The other three sets represent the Saved Pointers for each pending ***I/O Processes*** in the Initiator. One set of Saved Pointers exists for each ***Nexus*** that is currently established between the Initiator and all Targets. The Saved Pointers must exist for at least the duration of the SCSI Command, and for any additional time that the host requires them. When a ***Disconnect*** occurs, the Saved Pointers are set aside until a later ***Reconnect*** occurs. Put another way, the Saved Pointers become active when a Nexus is established or revived, and become inactive (but are retained) when the Nexus is Disconnected. The Saved Pointers may be discarded when the Command is completed or when the I/O Process is completed (see ***Linked Commands***).

The middle of the diagram shows the Active Pointers. There is only one set of Active Pointers on each Initiator. The Active Pointers are used only for the Current I/O Process. When an I/O Process is Disconnected by a ***DISCONNECT Message***, the contents of the Active Pointers are discarded. When a Nexus is revived by a Reconnect, the Saved Pointers for the Nexus are copied to the Active Pointers.

The right side of the diagram shows the Host Memory in the usual "arbitrary square box format". This box is not to be taken literally! What it represents is a place somewhere in the host where Commands, Status, and Data are kept. These could be main host memory, a special cache, or even inside some kind of super protocol ***Chip***. These blocks (pointed to by each Active Pointer) are contiguous as far as the SCSI bus is concerned; if the host can manage scattered data buffers such that they appear contiguous, that is fine (in fact, some do).

A word now on exactly what these Pointers represent. To the target, a Pointer represents the Initiator's byte position in the transfer. This is also true for the Initiator, though it may be associated with a memory location. For example:

- The Saved Command Pointer points at byte 0 of the command. To the Target, this might represent the first byte in its local command buffer. To the Initiator, the Pointer might represent the cache memory location in the ***Host Adapter*** where the Command is stored.

- The Active Data Pointer may point at byte 1023 of the data transfer. To the Target, this might be the last byte of the 2nd disk sector of the transfer. To the Initiator, the Pointer might be a memory location in a main memory buffer.

There is a key point here. **It does not matter what the physical nature or representation of the Pointer, as long as each device manages the Pointer in a manner consistent with the SCSI Pointer model.**

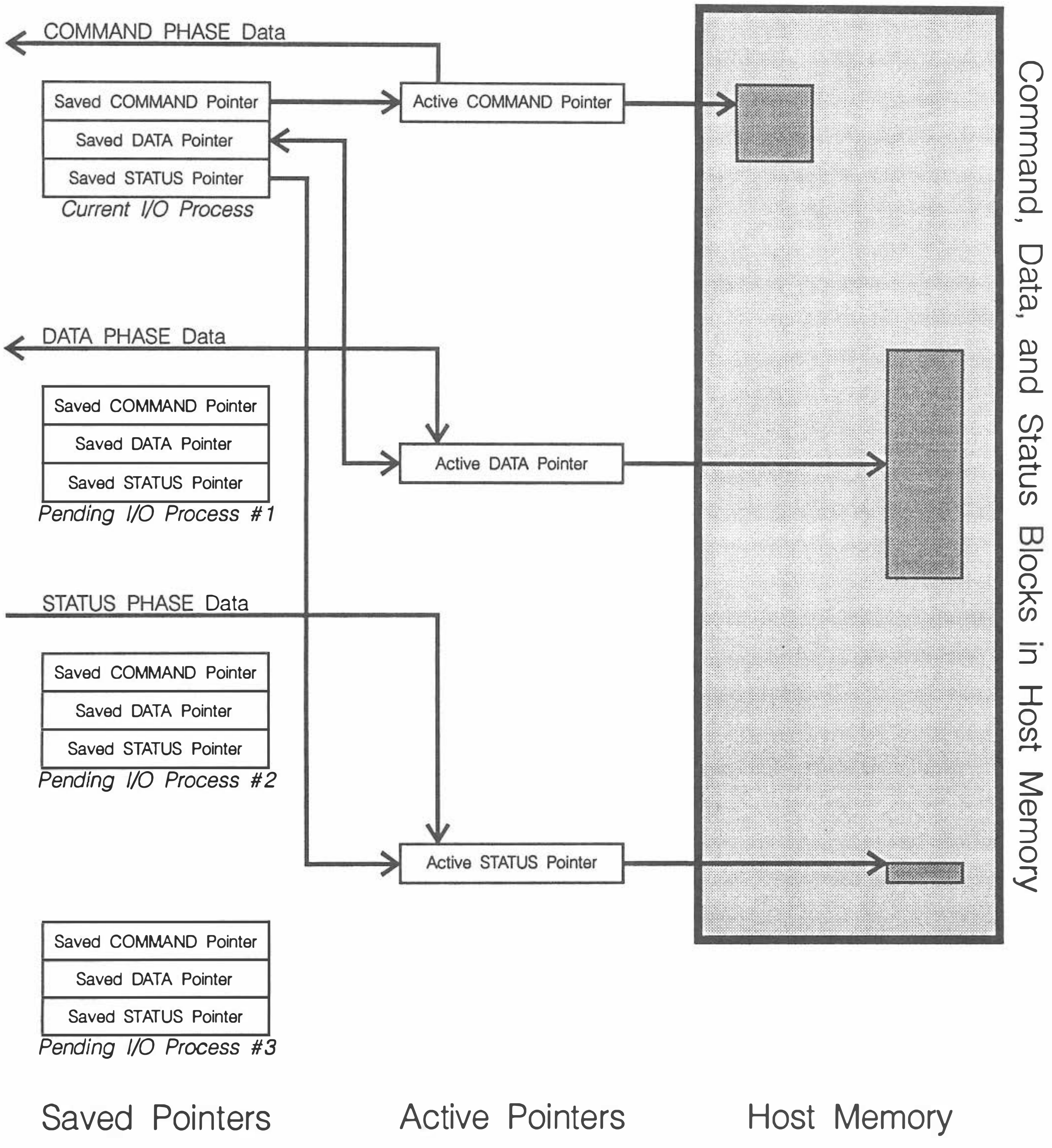


DIAGRAM 2: POINTERS AND HOST MEMORY

Diagram 3 shows the state of the Pointers prior to the start of a new Command. The Active Pointers are empty/null/"Don’t Care" and the Saved Pointers have been loaded and are ready for use. Note that the Active Pointers are not loaded until the Nexus is established.

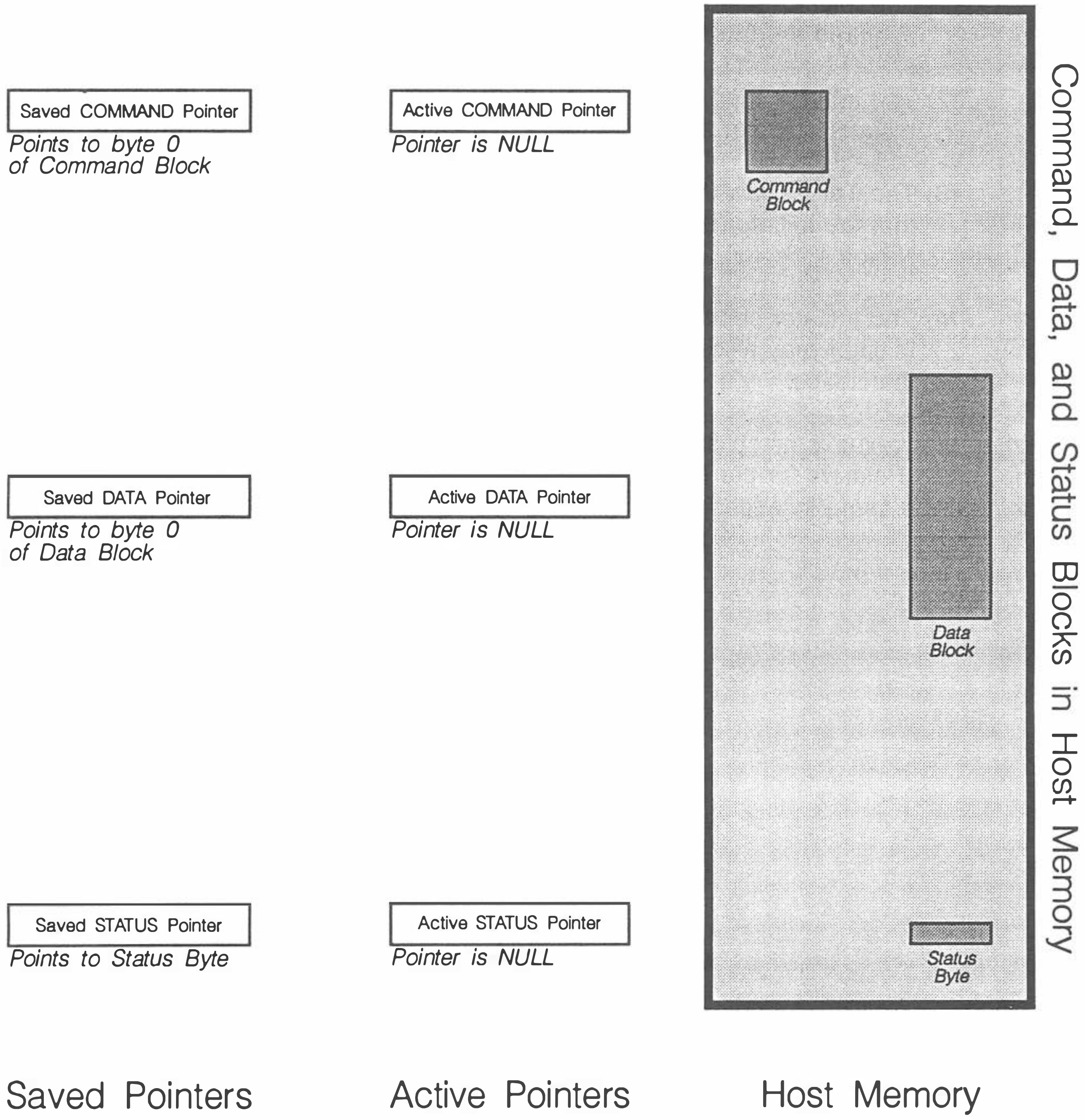


DIAGRAM 3: POINTERS AT THE START OF A COMMAND

Diagram 4 shows the state of the Pointers after the Nexus has been established. Specifically, the following events have happened:

(1) The Initiator has entered the ***ARBITRATION Phase*** and successfully acquired the bus.

(2) The Initiator then entered the ***SELECTION Phase*** to select the Target. The Initiator also set the ***ATN Signal*** to create the ***Attention Condition***.

(3) The Target saw the ATN Signal and entered the ***MESSAGE OUT Phase*** so that the Initiator could send the messages to the Target that establish the Nexus.

(4) The Initiator sends the ***IDENTIFY Message*** to establish the Nexus. The Initiator may also send a ***Queue Tag Message*** to further define and restrict the Nexus.

When the Nexus is established to the Initiator's satisfaction, it copies the contents of the Saved Pointers for the Nexus to the Active Pointers. Both sets of Pointers now point at the same locations in the Command, Status, and Data blocks.

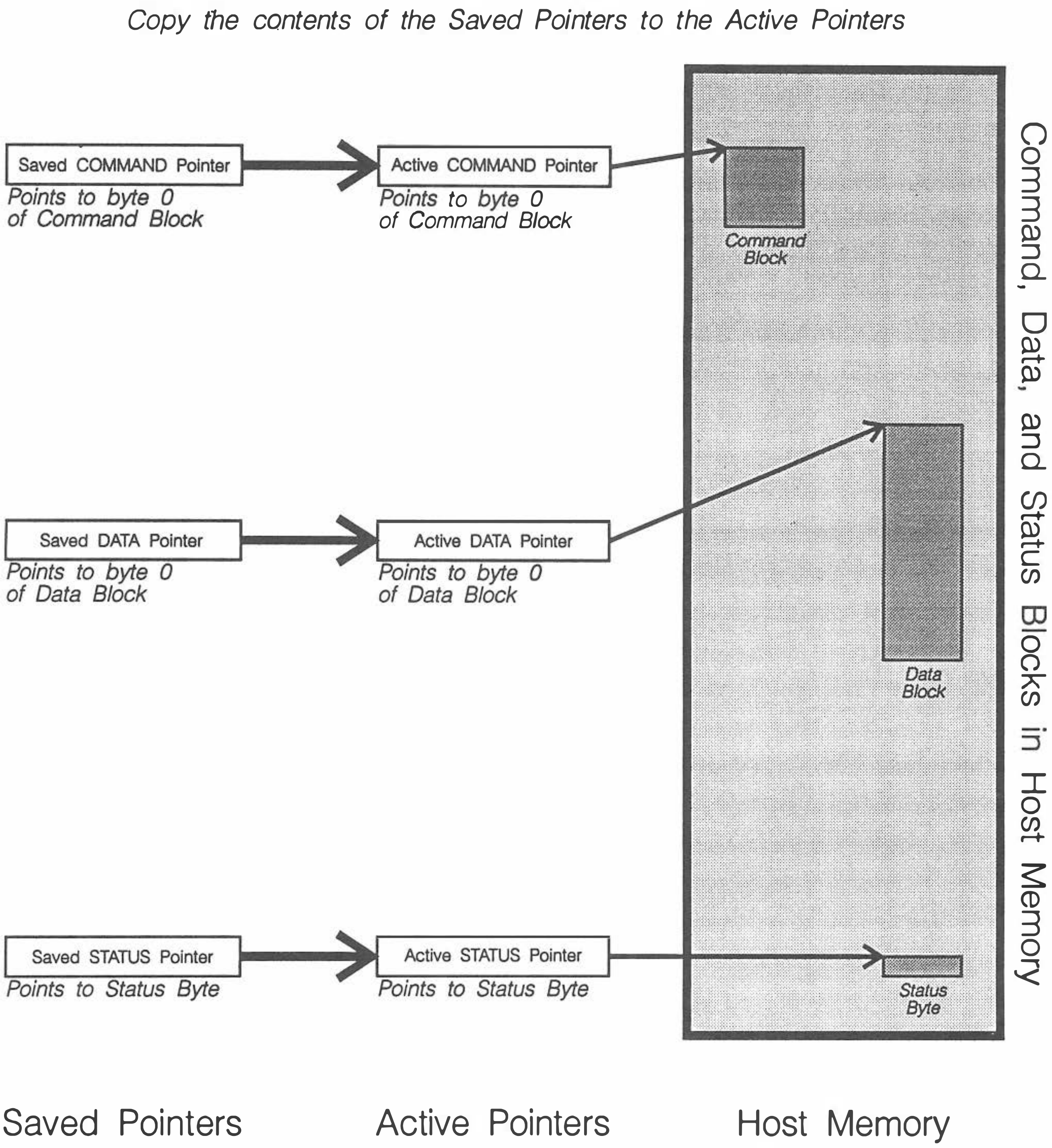


DIAGRAM 4: POINTERS AFTER THE NEXUS IS ESTABLISHED

Diagram 5 shows the state of the Pointers after the Command Block transfer:

(1) After the MESSAGE OUT Phase, the Target enters ***COMMAND Phase***.

(2) The Initiator sends a twelve byte ***Command Descriptor Block (CDB)***.

After the CDB is transferred, the Active Command Pointer has been incremented 12 times. An Active Pointer increments once after every byte that is transferred. An Active Pointer always increments (or changes) by bytes no matter what the actual transfer width is (see ***Wide Data Transfer***). As a result, an Active Pointer always points at the next byte that will be sent or received by the Initiator.

In this case, the Active Command Pointer points at the starting address plus 12. This is not an overrun of the Command block, since the next byte pointed to by the Active Command Pointer hasn't been transferred, and shouldn't be.

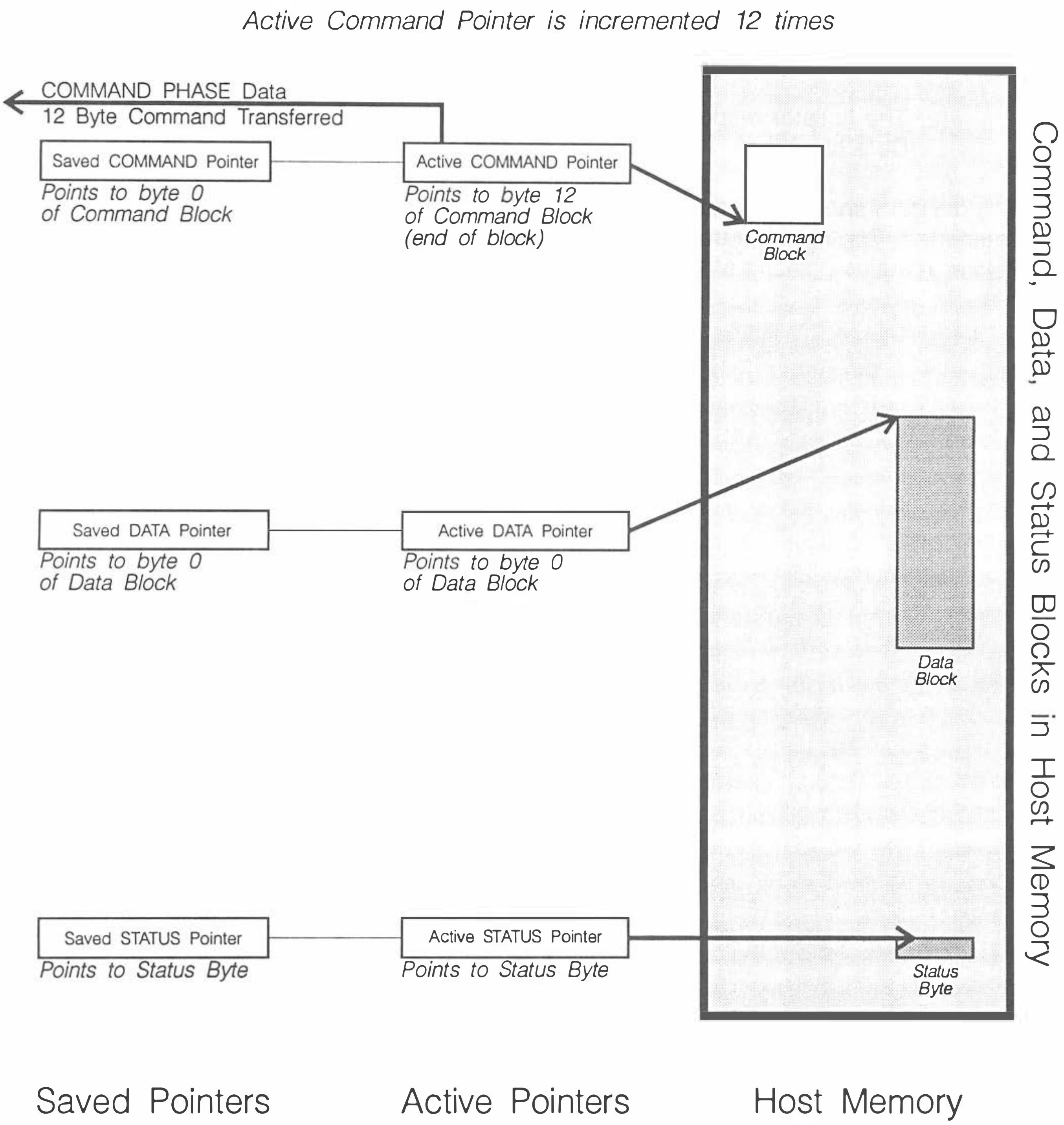


DIAGRAM 5: POINTERS AFTER THE COMMAND TRANSFER

Diagram 6 shows the state of the Pointers part of the way through a data transfer:

(1) After processing the Command, which happens to be a disk READ command for two 512 byte sectors, the Target begins transferring the data.

(2) The Initiator responds to the ***DATA IN Phase*** and transfers the data to the Data block.

This diagram shows the state of the Pointers after one sector (512 bytes) have been transferred. The Active Data Pointer is incremented 512 times, while the Saved Data Pointer remains pointing at the start of the Data block.

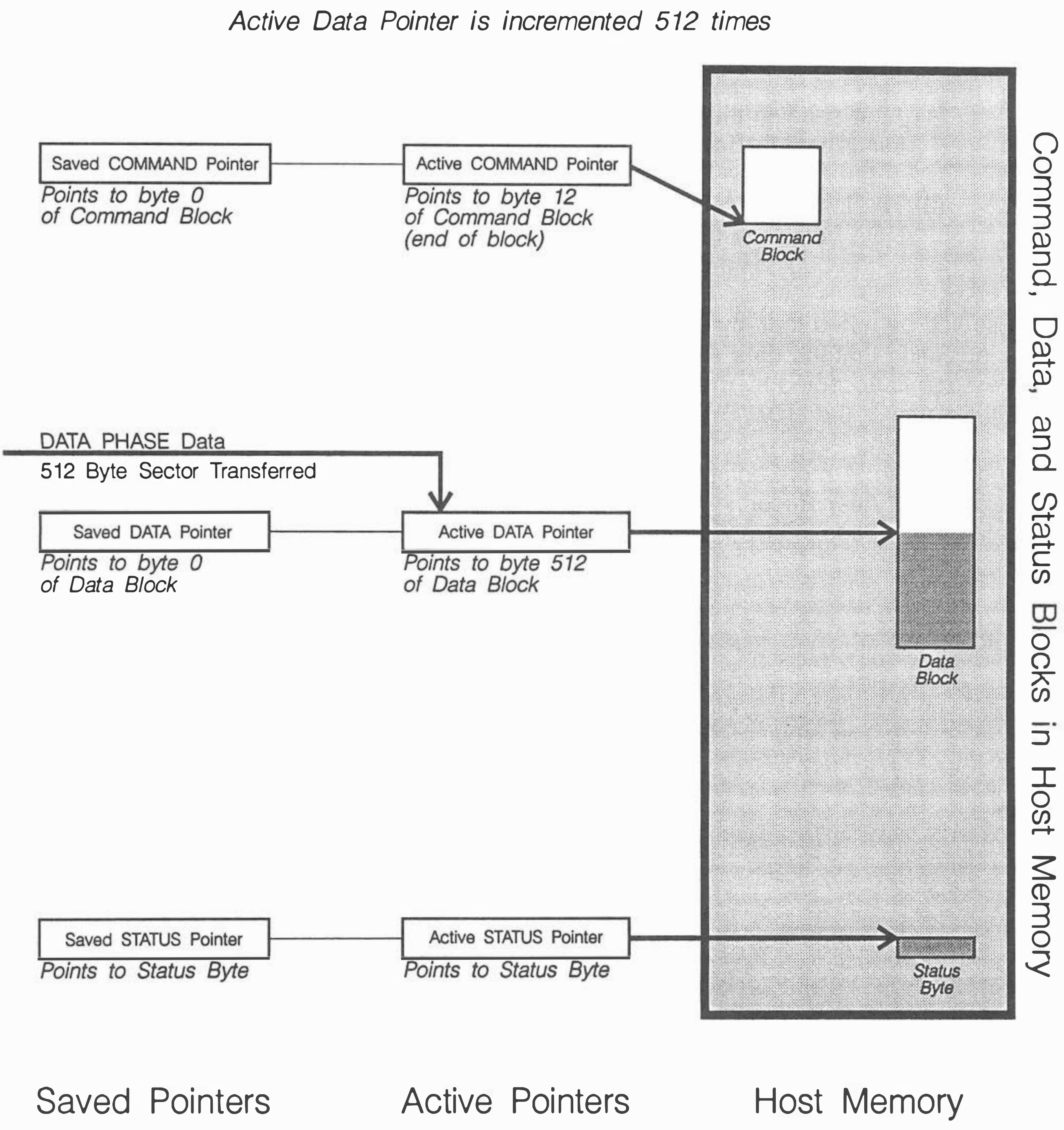

Active Data Pointer is incremented 512 times
Saved COMMAND Pointer
Points to byte 0 of Command Block
Active COMMAND Pointer
Points to byte 12 of Command Block (end of block)
Command Block
DATA PHASE Data
512 Byte Sector Transferred
Saved DATA Pointer
Points to byte 0 of Data Block
Active DATA Pointer
Points to byte 512 of Data Block
Data Block
Saved STATUS Pointer
Points to Status Byte
Active STATUS Pointer
Points to Status Byte
Status Byte
Command, Data, and Status Blocks in Host Memory
Saved Pointers
Active Pointers
Host Memory

DIAGRAM 6: POINTERS IN THE MIDDLE OF A DATA TRANSFER

Diagram 7 shows an example of the use of the ***SAVE DATA POINTER Message***. We are not recommending the use of this Message after every sector (use it to break up longer transfers). The example does show how a transfer can be broken into smaller parts to limit the amount of data that must be re-transferred when an error occurs:

(1) After sending the first sector, the Target changes to ***MESSAGE IN Phase*** to send the SAVE DATA POINTER Message.

(2) The Initiator receives the Message and copies the contents of the Active Data Pointer to the Saved Data Pointer. Both Data Pointers are now equal to the beginning of the data block plus 512 bytes.

It should be noted that, by accepting the SAVE DATA POINTER Message, the Initiator is indicating that the data was transferred successfully with no errors. Once the Message is accepted by the Initiator, there is no possibility of a retry of that Data without use of the ***MODIFY DATA POINTER Message***, or repeating the Command.

Another note: A more common use of the SAVE DATA POINTERS is prior to a Disconnect. After Reconnection, the transfer may continue from the point where the data transfer was interrupted.

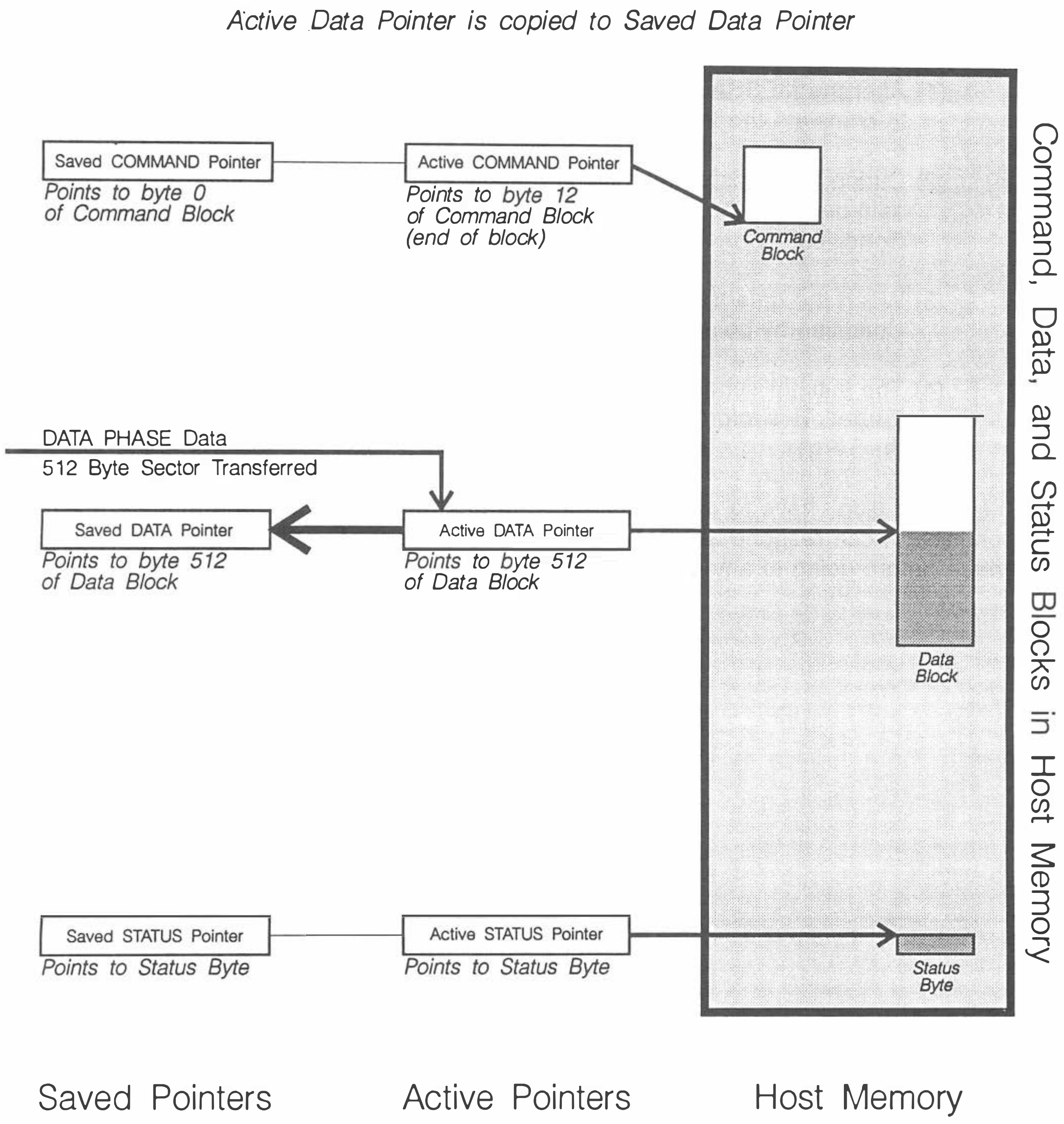

Active Data Pointer is copied to Saved Data Pointer
Saved COMMAND Pointer
Points to byte 0 of Command Block
Active COMMAND Pointer
Points to byte 12 of Command Block (end of block)
Command Block
DATA PHASE Data
512 Byte Sector Transferred
Saved DATA Pointer
Points to byte 512 of Data Block
Active DATA Pointer
Points to byte 512 of Data Block
Data Block
Saved STATUS Pointer
Points to Status Byte
Active STATUS Pointer
Points to Status Byte
Status Byte
Command, Data, and Status Blocks in Host Memory
Saved Pointers
Active Pointers
Host Memory

DIAGRAM 7: POINTERS AFTER SAVE DATA POINTER MESSAGE

Diagram 8 shows the state of the Pointers after the completion of the data transfer. But there was a catch: The Initiator detected a ***Parity*** error during the transfer, and must perform an ***Error Recovery***:

(1) After the MESSAGE IN Phase, the Target returned to the DATA IN Phase to complete the transfer.

(2) During the transfer the Initiator detected a Parity error on the bus. As soon as it detected the error, it asserted the ATN Signal to create the Attention Condition.

(3) When the data transfer is complete, the Target responds to the Attention Condition by going to the MESSAGE OUT Phase.

(4) The Initiator sends an ***INITIATOR DETECTED ERROR Message*** to the Target. This informs the Target of the error, and leaves the next step up to the Target.

The Target didn't have to wait until the end of the transfer to respond to the Attention Condition. It can respond at any time, as long as it responds with a ***MESSAGE OUT Phase*** before going to any other Phase.

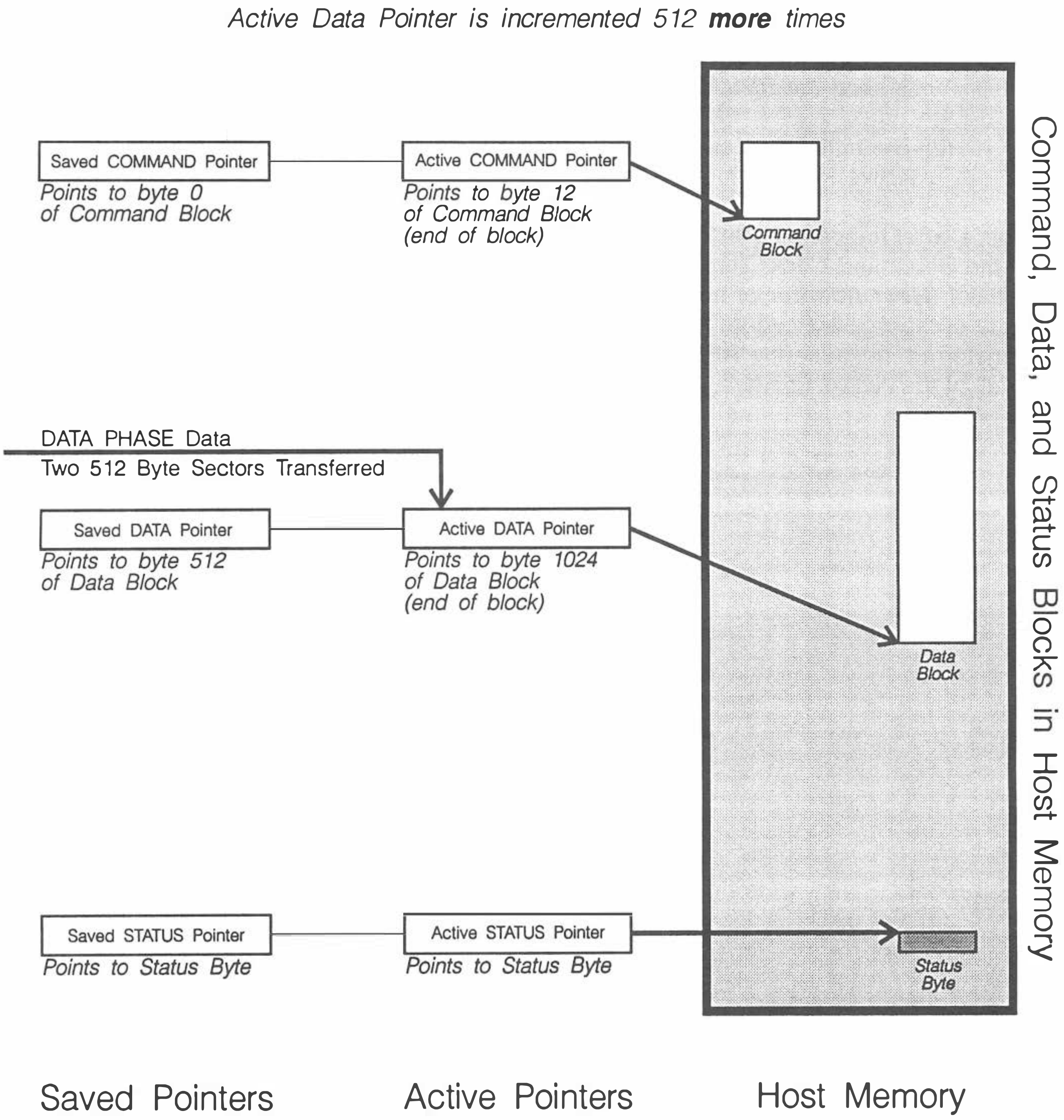

Active Data Pointer is incremented 512 **more** times
Saved COMMAND Pointer
Points to byte 0 of Command Block
Active COMMAND Pointer
Points to byte 12 of Command Block (end of block)
Command Block
DATA PHASE Data
Two 512 Byte Sectors Transferred
Saved DATA Pointer
Points to byte 512 of Data Block
Active DATA Pointer
Points to byte 1024 of Data Block (end of block)
Data Block
Saved STATUS Pointer
Points to Status Byte
Active STATUS Pointer
Points to Status Byte
Status Byte
Command, Data, and Status Blocks in Host Memory
Saved Pointers
Active Pointers
Host Memory

DIAGRAM 8: POINTERS AFTER DATA TRANSFER & PARITY ERROR

Diagram 9 shows the state of the Pointers after the preparation for the data transfer retry has been completed:

(1) After the MESSAGE OUT Phase, the Target goes to MESSAGE IN Phase to send the ***RESTORE POINTERS Message*** to retry the DATA Phase.

(2) The Initiator receives the message and copies all Saved Pointers to the Active Pointers.

Notice what happens here: The Active Command Pointer now points back at the start of the Command block. This is not a problem; the COMMAND Phase is history. The Initiator (***Host Adapter*** or host driver software) cannot (repeat, cannot) rely on the state of the Active Pointers to determine if a Phase occurred or how many bytes were transferred, because the RESTORE POINTERS function can cause this modification to all of the Active Pointers.

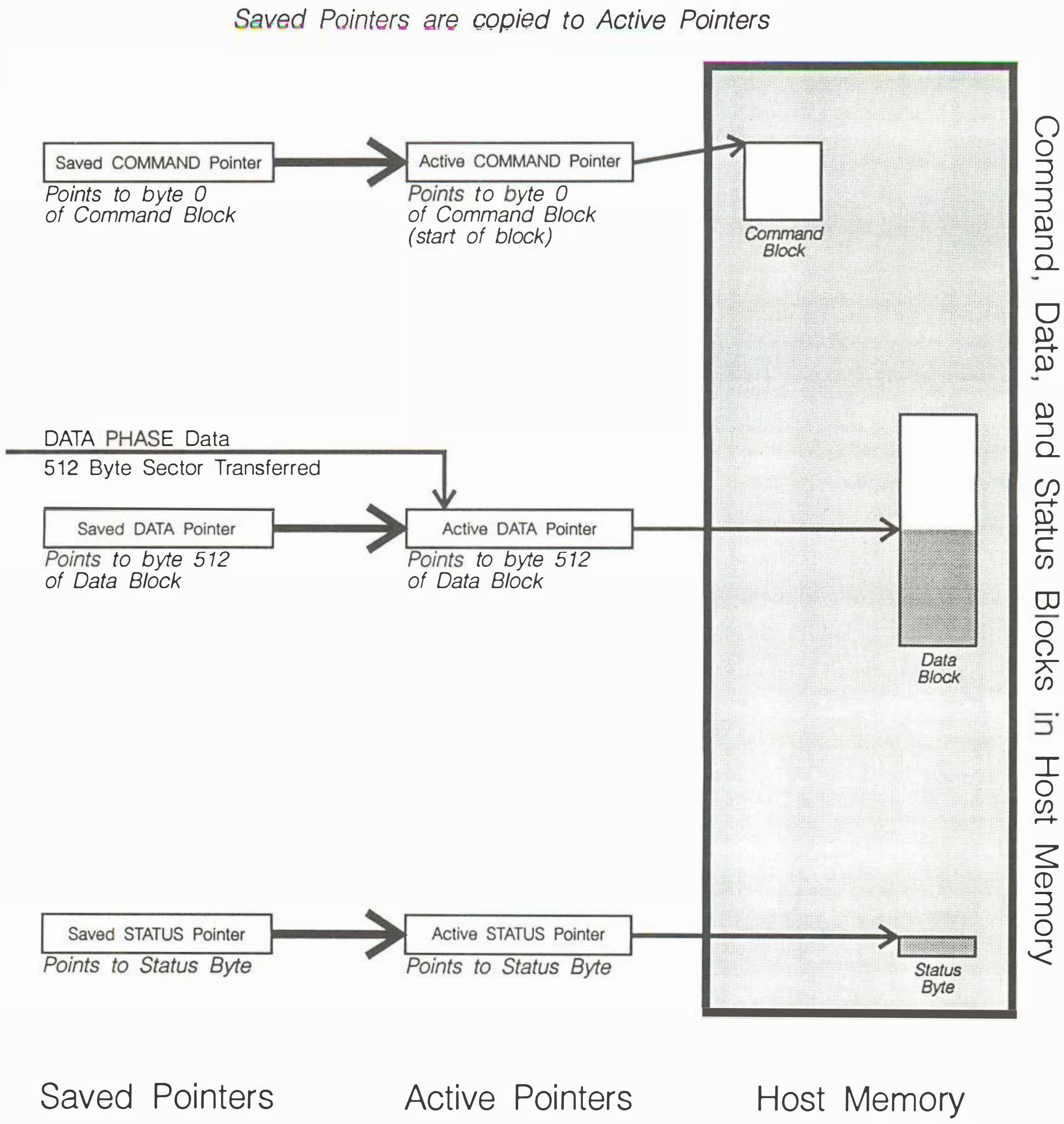

Saved Pointers are copied to Active Pointers
Saved COMMAND Pointer
Points to byte 0 of Command Block
Active COMMAND Pointer
Points to byte 0 of Command Block (start of block)
Command Block
DATA PHASE Data
512 Byte Sector Transferred
Saved DATA Pointer
Points to byte 512 of Data Block
Active DATA Pointer
Points to byte 512 of Data Block
Data Block
Saved STATUS Pointer
Points to Status Byte
Active STATUS Pointer
Points to Status Byte
Status Byte
Command, Data, and Status Blocks in Host Memory
Saved Pointers
Active Pointers
Host Memory

DIAGRAM 9: POINTERS AFTER RESTORE POINTERS MESSAGE

Diagram 10 shows the state of the Pointers after the data transfer retry. This time, the data transfer completed without incident. The Active Data Pointer increments to point at the end of the Data block. The Saved Data Pointer still points at byte 512.

Active Data Pointer is incremented 512 ***more*** *times again*

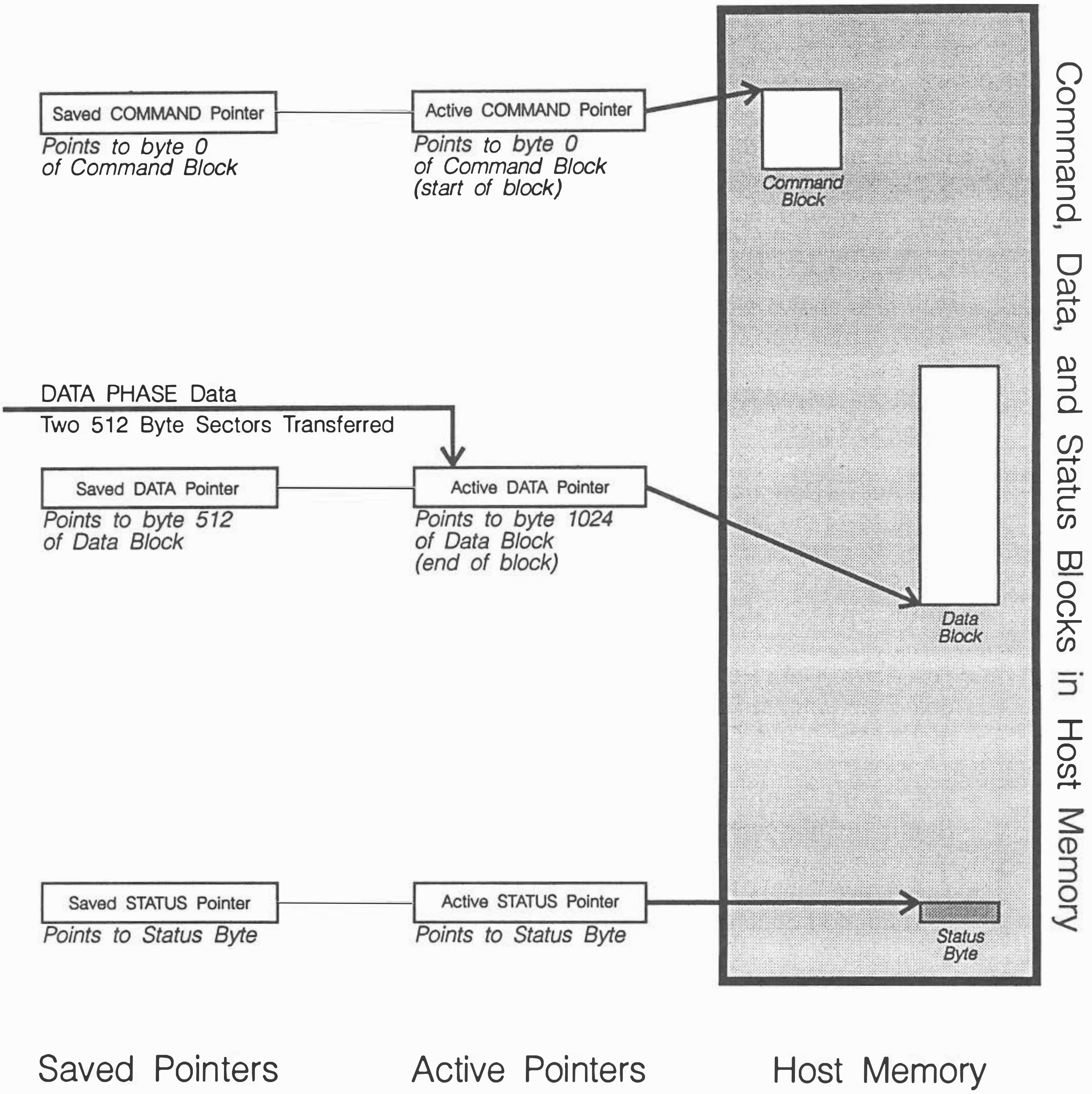


DIAGRAM 10: POINTERS AFTER THE DATA TRANSFER RETRY

Diagram 11 shows the state of the Pointers after ***Status*** byte transfer:

(1) The Target has completed the DATA IN Phase and switches to ***STATUS Phase***.

(2) The Initiator takes the Status byte from the Target and stores it in the Status byte location in host memory.

The Status byte needs a Pointer pair only because of the need to be able to retry the Status transfer. It was simpler to define Status Pointers rather than define a special "Status retry" procedure similar to the Message System. (Also, back in the earlier days of the ***SCSI-1*** development, the draft Standard had two Status bytes!)

This diagram also shows the final state of the Pointers just prior to Disconnect:

- The Saved Command Pointer still points at the start of the Command block, since the Saved Command Pointer cannot be changed during the execution of the Command.

- The Active Command Pointer also points at the start of the Command block because of the RESTORE POINTERS Message during the DATA IN Phase.

- The Saved Data Pointer points at the middle of the Data block because of the SAVE DATA POINTERS Message during the DATA IN Phase.

- The Active Data Pointer points at the end of the Data block as a result of being incremented during the DATA IN Phase. No other events intervened to modify the Active Data Pointer after the DATA IN Phase.

- The Saved Status Pointer still points at the Status byte, since the Saved Status Pointer cannot be changed during the execution of the Command.

- The Active Status Pointer points after the Status byte as a result of being incremented once during the STATUS Phase.

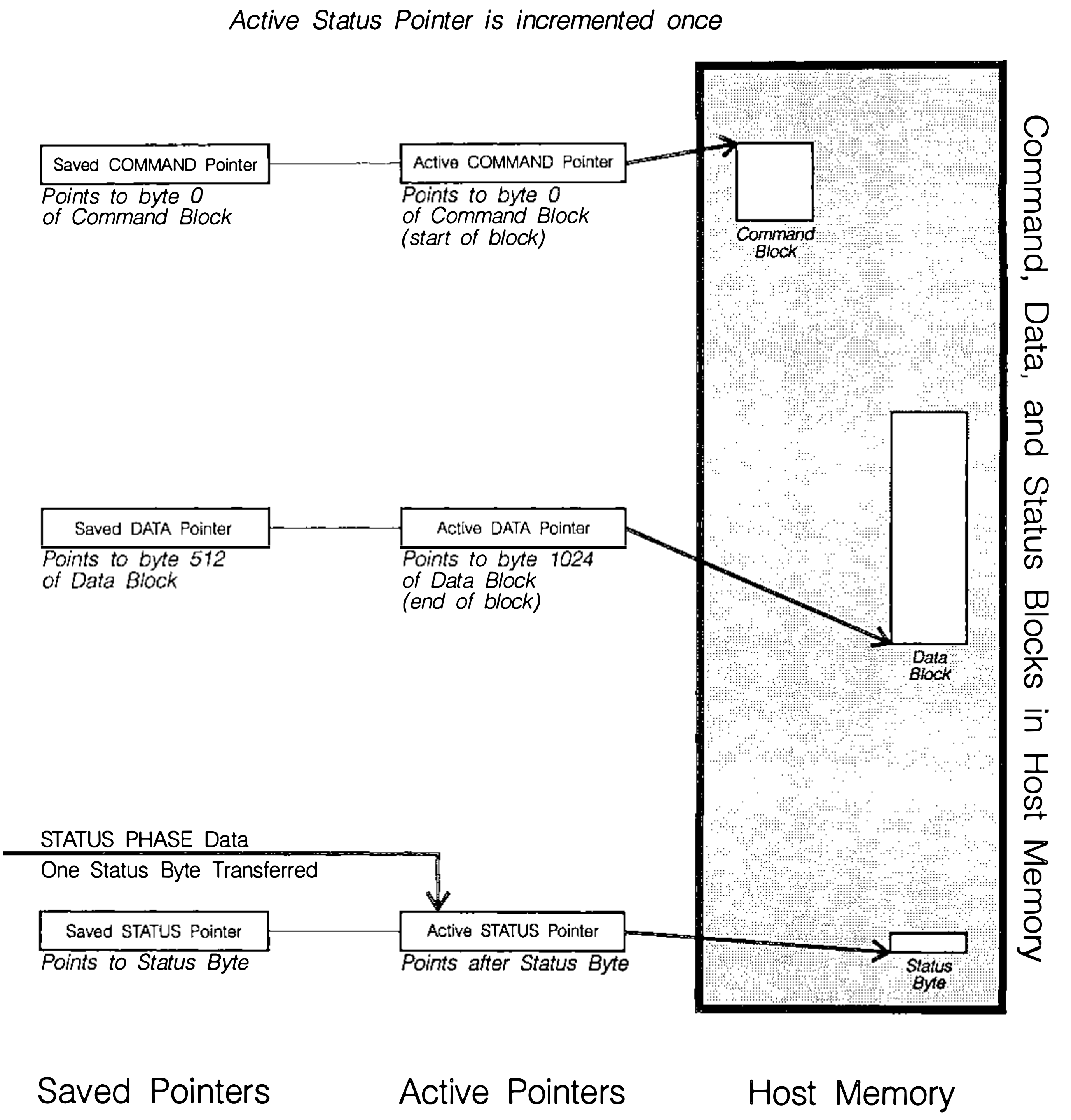

Active Status Pointer is incremented once
Saved COMMAND Pointer
Points to byte 0 of Command Block
Active COMMAND Pointer
Points to byte 0 of Command Block (start of block)
Command Block
Saved DATA Pointer
Points to byte 512 of Data Block
Active DATA Pointer
Points to byte 1024 of Data Block (end of block)
Data Block
STATUS PHASE Data
One Status Byte Transferred
Saved STATUS Pointer
Points to Status Byte
Active STATUS Pointer
Points after Status Byte
Status Byte
Command, Data, and Status Blocks in Host Memory
Saved Pointers
Active Pointers
Host Memory

DIAGRAM 11: POINTERS AFTER THE STATUS TRANSFER

Diagram 12 shows the state of the Pointers after the completion of the Command:

(1) The Target changed from STATUS Phase to MESSAGE IN Phase to send the ***COMMAND COMPLETE Message***.

(2) The Initiator accepts the COMMAND COMPLETE Message and discards the Active Pointers.

(3) The Target goes to ***BUS FREE Phase*** and the Command (and I/O Process) is completed.

The Saved Pointers should be discarded since they cannot be counted upon to contain meaningful data. For example, the Saved Data Pointer now points into the middle of the Data block. In many systems the host would have no knowledge of messages exchanged with the Target, and so would not know why the Saved Data Pointer had changed.

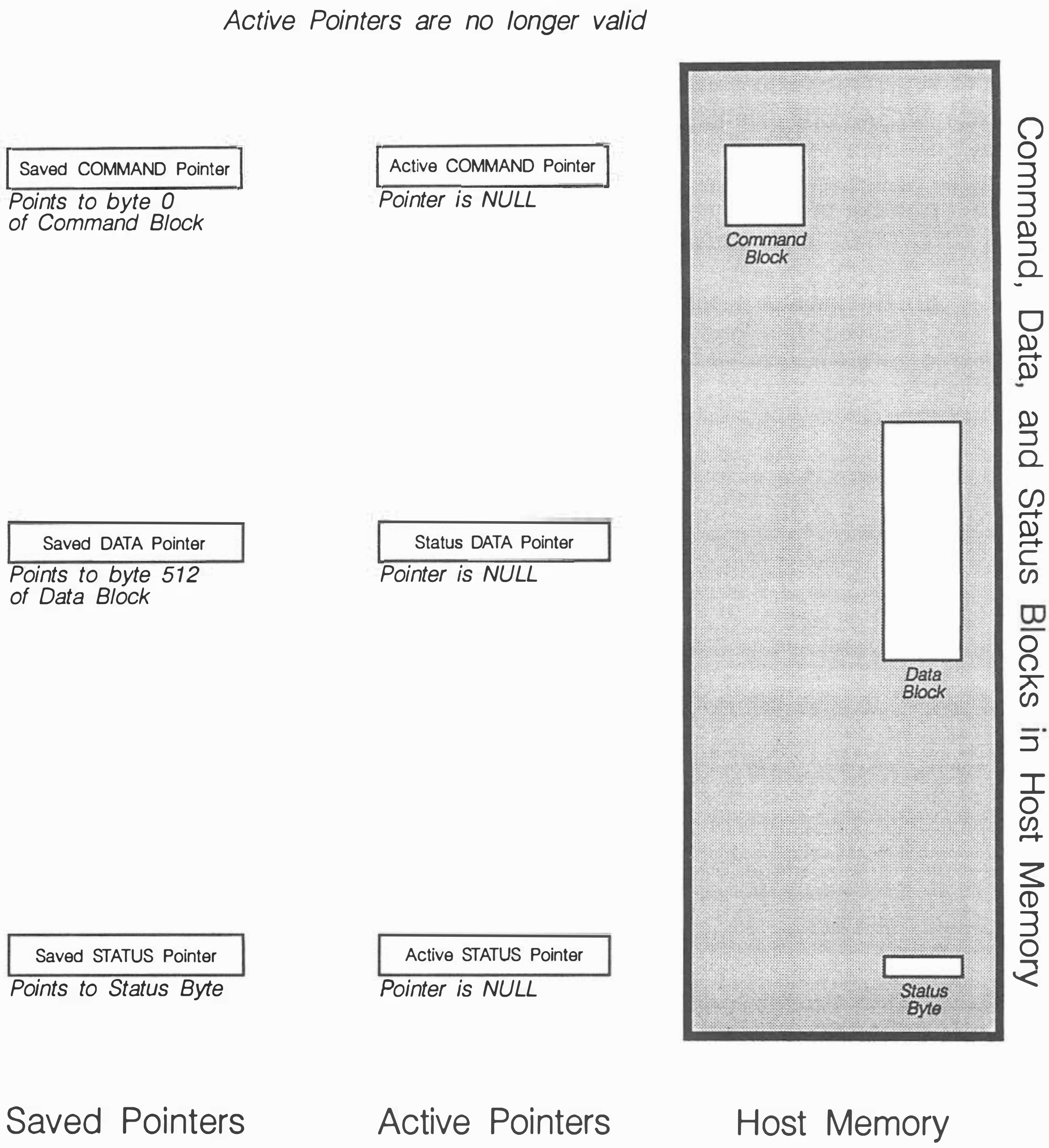


DIAGRAM 12: POINTERS AFTER THE END OF A COMMAND

We will now proceed with some other examples of Path Control using the ***DISCONNECT Message*** and the ***MODIFY DATA POINTER Message***.

Diagram 13 picks up the action in the previous example after the transfer of the SAVE DATA POINTERS Message from the Target to the Initiator (as shown in Diagram 7). Unlike the previous example, the Target will now Disconnect from the bus so that it may perform a seek to the other sector:

(1) After the 512 byte DATA IN Phase, the Target changes to MESSAGE IN Phase and sends the SAVE DATA POINTERS Message.

(2) The Initiator accepts the Message and copies the Active Data Pointer to the Saved Data Pointer.

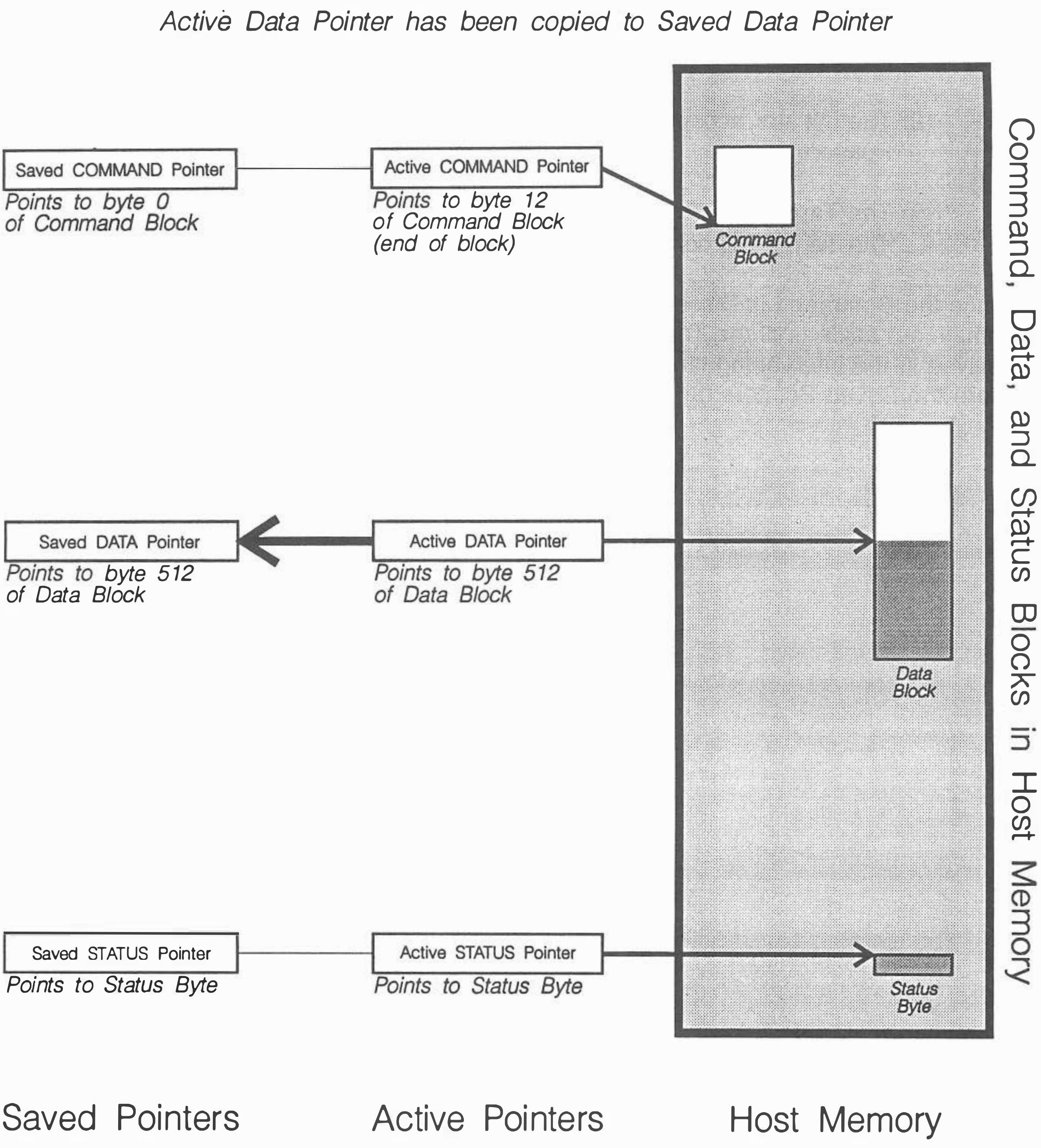

Activė Data Pointer has been copied to Saved Data Pointer
Saved COMMAND Pointer
Points to byte 0 of Command Block
Active COMMAND Pointer
Points to byte 12 of Command Block (end of block)
Command Block
Saved DATA Pointer
Points to byte 512 of Data Block
Active DATA Pointer
Points to byte 512 of Data Block
Data Block
Saved STATUS Pointer
Points to Status Byte
Active STATUS Pointer
Points to Status Byte
Status Byte
Command, Data, and Status Blocks in Host Memory
Saved Pointers
Active Pointers
Host Memory

DIAGRAM 13: POINTERS BEFORE A DISCONNECT

Diagram 14 shows the state of the Pointers after the Disconnect:

(1) After sending the SAVE DATA POINTER Message, the Target stays in MESSAGE IN Phase to send the ***DISCONNECT Message***.

(2) The Initiator accepts the DISCONNECT Message and discards the Active Pointers.

(3) The Target goes to ***BUS FREE Phase*** and the Command is Disconnected. The Target will continue to process the Command off-line.

Once the Command is Disconnected, the Initiator sets the Saved Pointers aside. They remain set aside until the Target ***Reconnects*** to that ***Nexus***. Other Nexuses may be revived in the intervening time, so the Initiator must be sure to have enough space to store the Saved Pointers for all pending Commands.

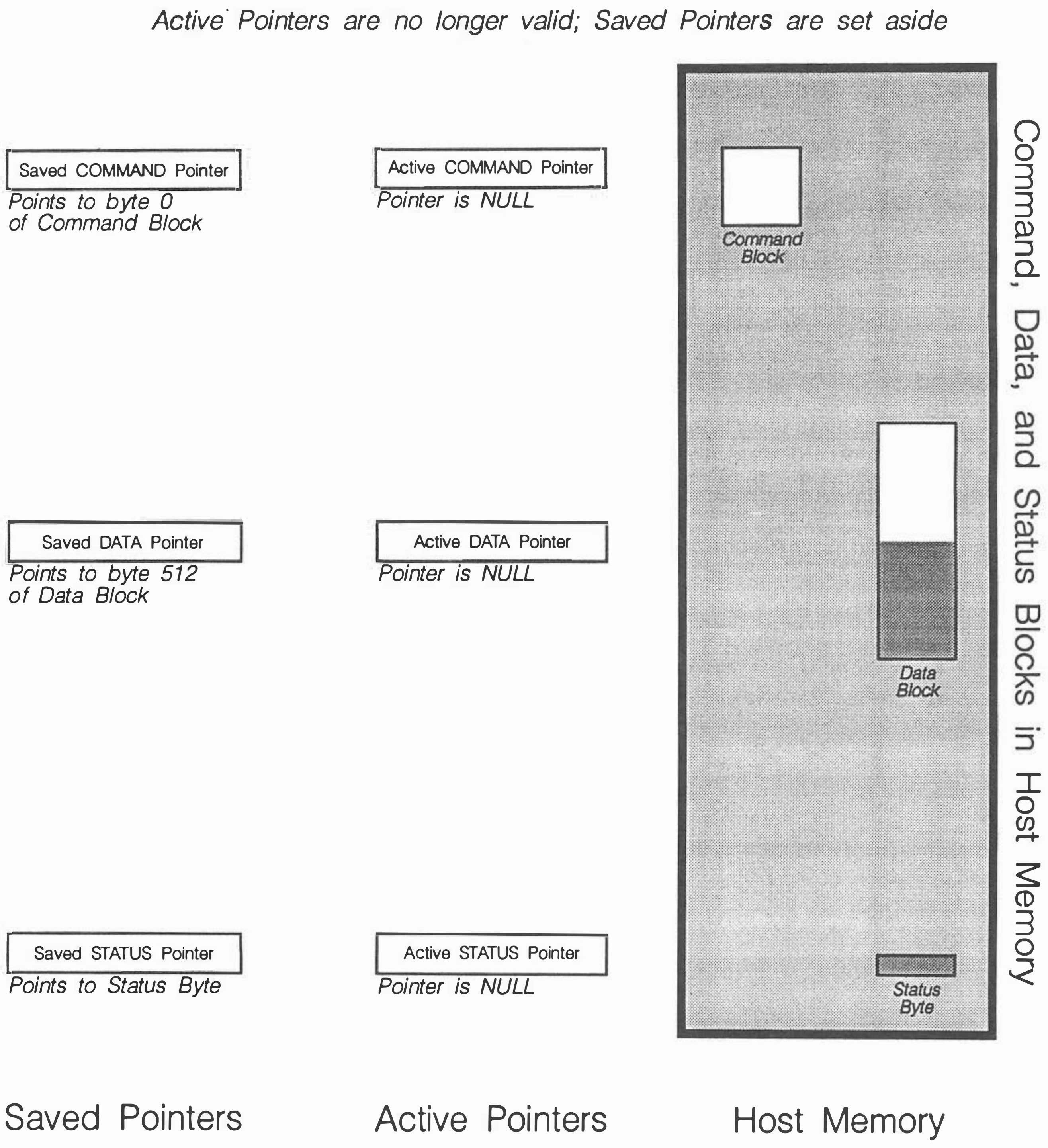


DIAGRAM 14: POINTERS AFTER A DISCONNECT

Diagram 15 shows the state of the Pointers after a Reconnect:

(1) The Target enters the ***ARBITRATION Phase*** and acquires the bus.

(2) The Target then enters the ***RESELECTION Phase*** to reconnect to the Initiator.

(3) The Target then enters the ***MESSAGE IN Phase*** to send the Messages to the Initiator that revive the Nexus.

(4) The Initiator receives the ***IDENTIFY Message*** to revive the Nexus. The Target may also send a ***Queue Tag Message*** to further define and restrict the Nexus, but only if the Nexus was originally established by the Initiator with a Queue Tag Message.

When the Nexus is identified to the Initiator's satisfaction, the contents of the corresponding Saved Pointers are copied to the Active Pointers. Both sets of Pointers now point at the same locations in the Command, Status, and Data blocks.

The command now proceeds as in Diagram 6.

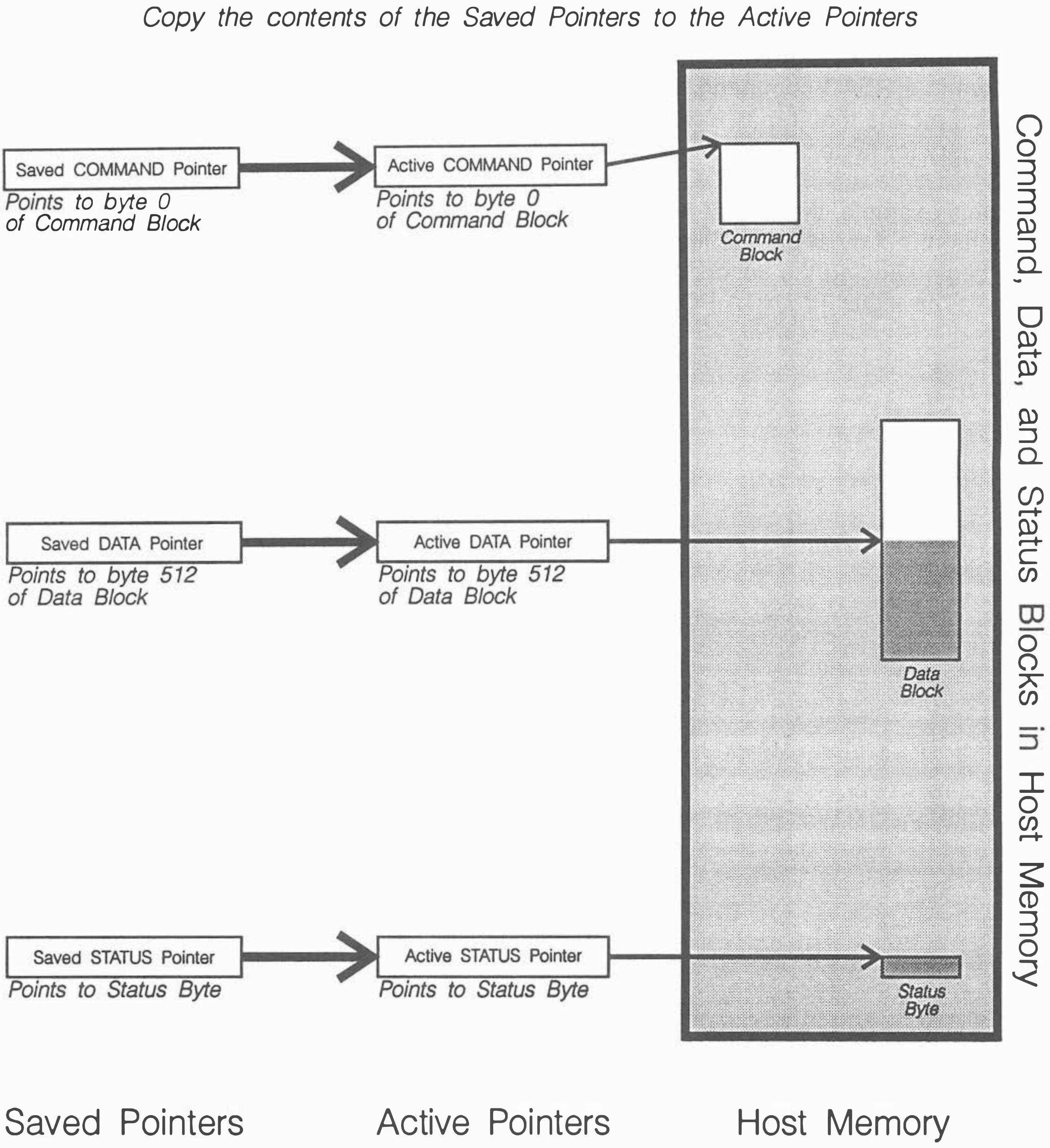

Copy the contents of the Saved Pointers to the Active Pointers
Saved COMMAND Pointer
Points to byte 0 of Command Block
Active COMMAND Pointer
Points to byte 0 of Command Block
Command Block
Saved DATA Pointer
Points to byte 512 of Data Block
Active DATA Pointer
Points to byte 512 of Data Block
Data Block
Saved STATUS Pointer
Points to Status Byte
Active STATUS Pointer
Points to Status Byte
Status Byte
Command, Data, and Status Blocks in Host Memory
Saved Pointers
Active Pointers
Host Memory

DIAGRAM 15: POINTERS AFTER A RECONNECT

The next example demonstrates the ***MODIFY DATA POINTER Message***.

Diagram 16 shows the state of the Pointers before a MODIFY DATA POINTER Message. The Target has just completed the ***COMMAND Phase*** (see Diagram 5) and taken in a READ command. In this example, however, the Target is a caching device, which means that it may have some of the requested sectors resident in a cache memory for faster access.

In this example, the Target has the second sector requested already available in cache, but it must get the first sector from disk. The optimum solution for the Target is to begin sending the second sector to the Initiator while fetching the first sector from disk. The MODIFY DATA POINTER Message is used to achieve this solution.

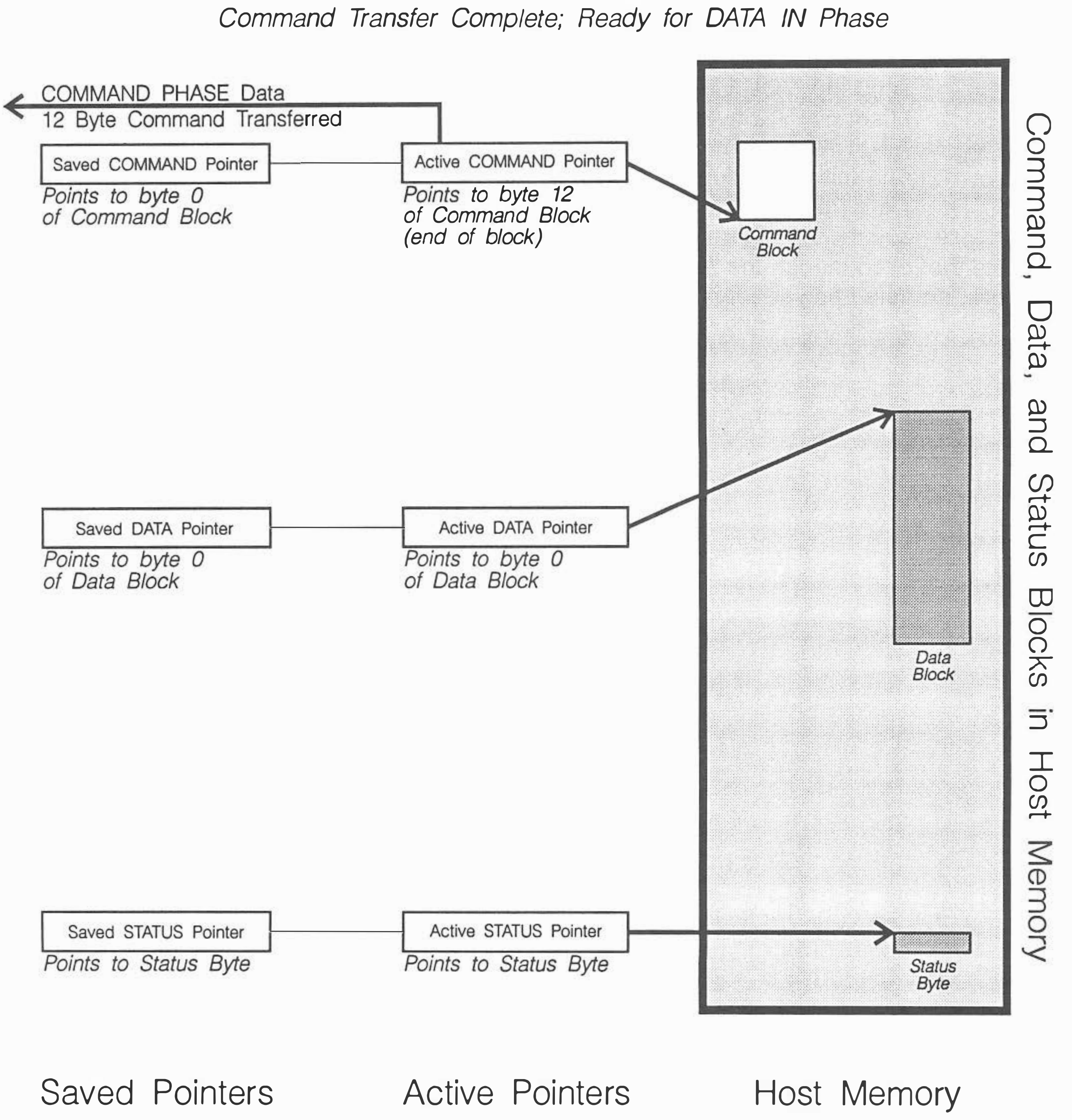

Command Transfer Complete; Ready for DATA IN Phase
COMMAND PHASE Data
12 Byte Command Transferred
Saved COMMAND Pointer
Points to byte 0 of Command Block
Active COMMAND Pointer
Points to byte 12 of Command Block (end of block)
Command Block
Saved DATA Pointer
Points to byte 0 of Data Block
Active DATA Pointer
Points to byte 0 of Data Block
Data Block
Saved STATUS Pointer
Points to Status Byte
Active STATUS Pointer
Points to Status Byte
Status Byte
Command, Data, and Status Blocks in Host Memory
Saved Pointers
Active Pointers
Host Memory

DIAGRAM 16: POINTERS BEFORE MODIFY DATA POINTER

Diagram 17 shows the state of the Pointers after the MODIFY DATA POINTER Message:

(1) The Target changes from COMMAND Phase to the ***MESSAGE IN Phase*** to send the MODIFY DATA POINTER Message. The Message contains an offset of +512 bytes.

(2) The Initiator accepts the Message and adds 512 bytes of offset to the Active Data Pointer. The Saved Data Pointer is not affected.

Note that even though the Active Data Pointer now points at byte 512 of the Data block, no data has yet been transferred.

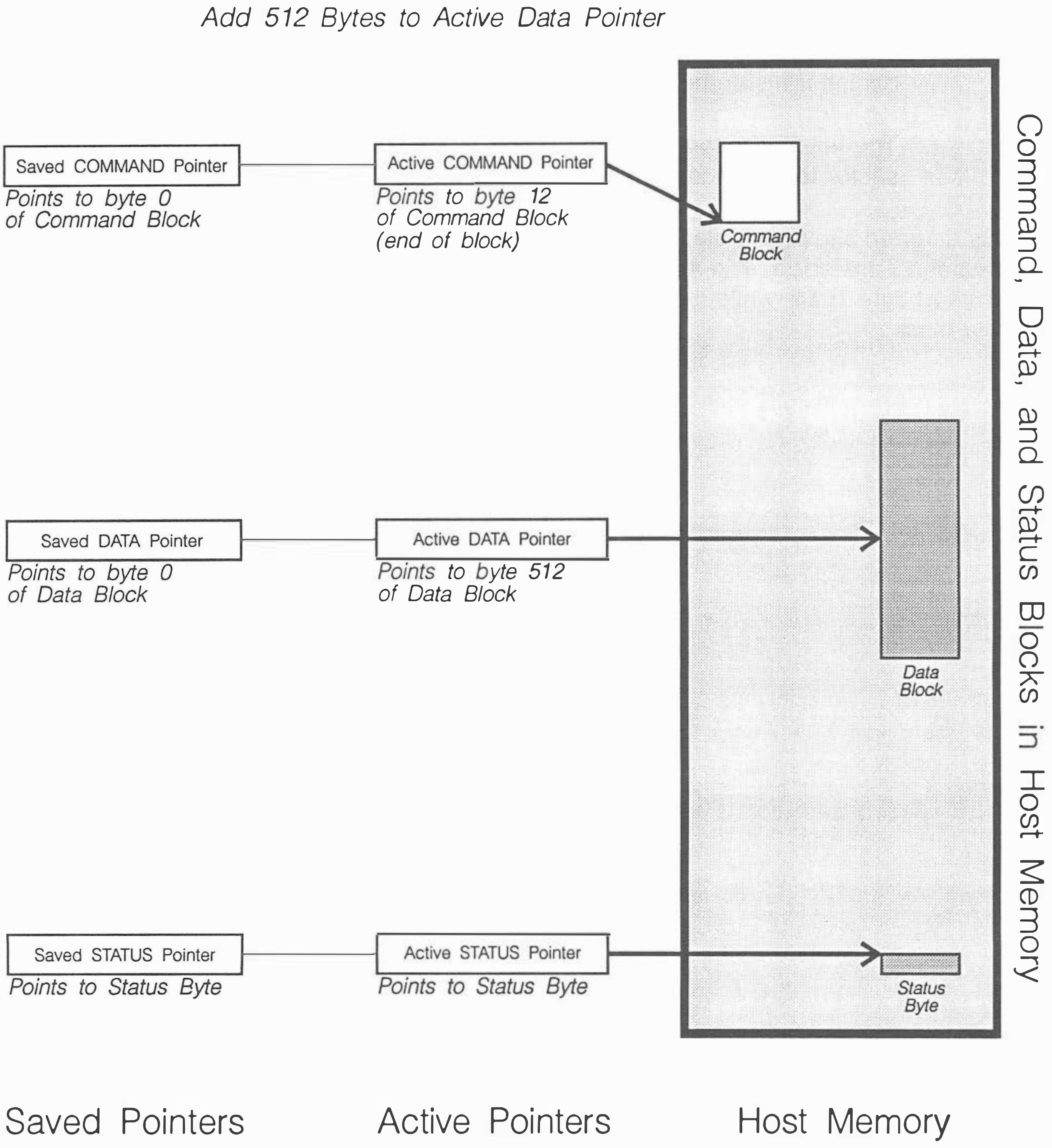


DIAGRAM 17: POINTERS AFTER A MODIFY DATA POINTER

Diagram 18 shows the state of the Pointers after the first sector data transfer after the MODIFY DATA POINTER Message:

(1) The Target switches from MESSAGE IN Phase to ***DATA IN Phase*** and begins transferring the data for the second sector.

(2) The Initiator responds to the ***DATA IN Phase*** and transfers the second sector to the second half of the Data block.

This diagram shows the state of the Pointers after only the second sector (512 bytes) has been transferred. The Active Data Pointer is incremented 512 times and now points at byte 1024, while the Saved Data Pointer remains pointing at the start of the Data block.

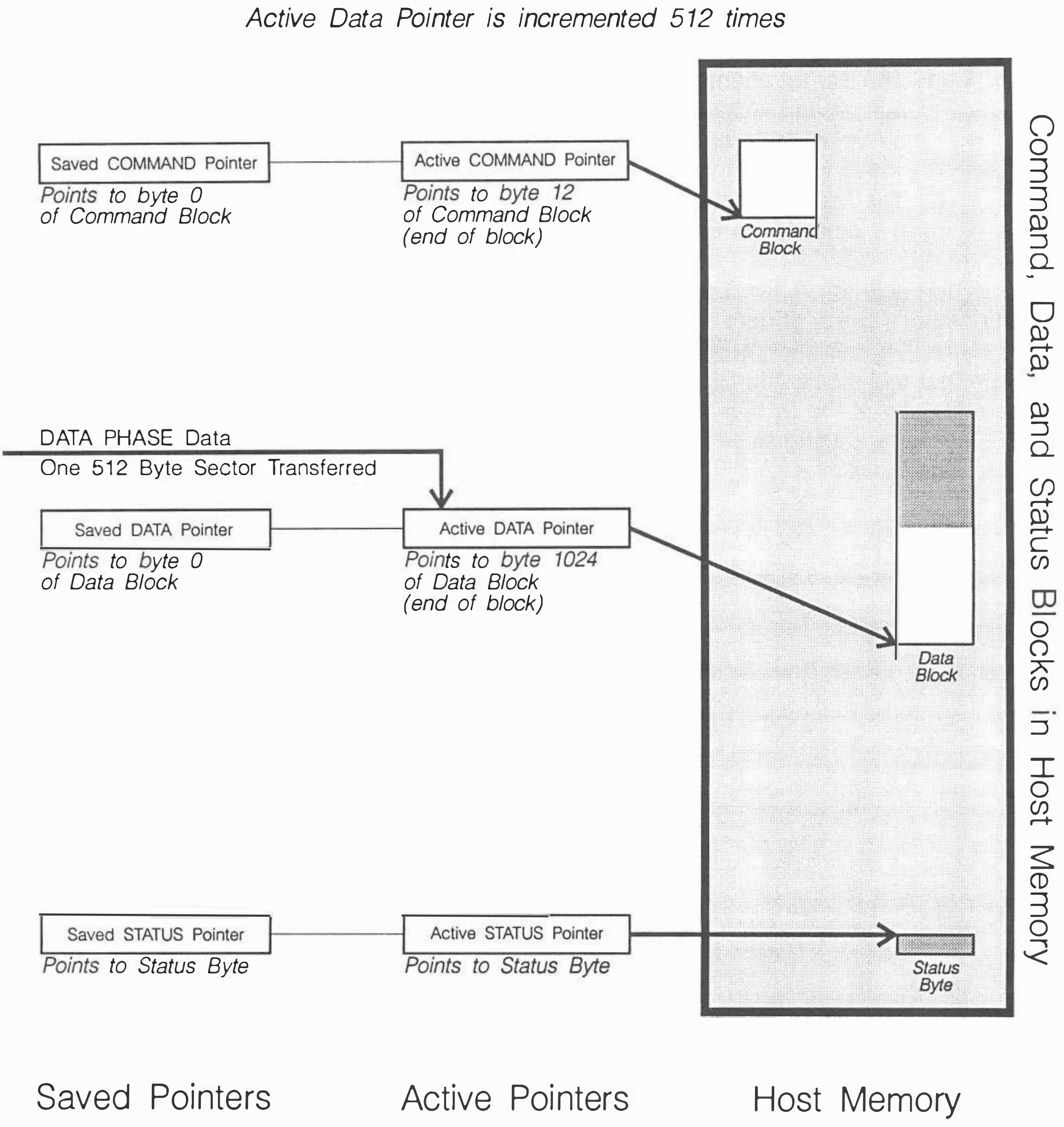

Active Data Pointer is incremented 512 times
Saved COMMAND Pointer
Points to byte 0 of Command Block
Active COMMAND Pointer
Points to byte 12 of Command Block (end of block)
Command Block
DATA PHASE Data
One 512 Byte Sector Transferred
Saved DATA Pointer
Points to byte 0 of Data Block
Active DATA Pointer
Points to byte 1024 of Data Block (end of block)
Data Block
Saved STATUS Pointer
Points to Status Byte
Active STATUS Pointer
Points to Status Byte
Status Byte
Command, Data, and Status Blocks in Host Memory
Saved Pointers
Active Pointers
Host Memory

DIAGRAM 18: 1ST SECTOR AFTER MODIFY DATA POINTER

In order to send the first requested sector, the Target must modify the Active Data Pointer again. Diagram 19 shows the state of the Pointers after the second MODIFY DATA POINTER Message:

(1) The Target changes from DATA IN Phase to ***MESSAGE IN Phase*** to send the second MODIFY DATA POINTER Message. The Message contains an offset of -1024 bytes.

(2) The Initiator accepts the Message and subtracts 1024 bytes of offset from the Active Data Pointer. Again, the Saved Data Pointer is not affected.

Note that now the Active Data Pointer points at byte 0 of the Data block, even though one sector's worth of data has been transferred.

Note that technically the RESTORE POINTERS Message could also have been used here instead of the MODIFY DATA POINTER Message. The MODIFY DATA POINTER Message is used since it is a more general solution for random access within the host data block.

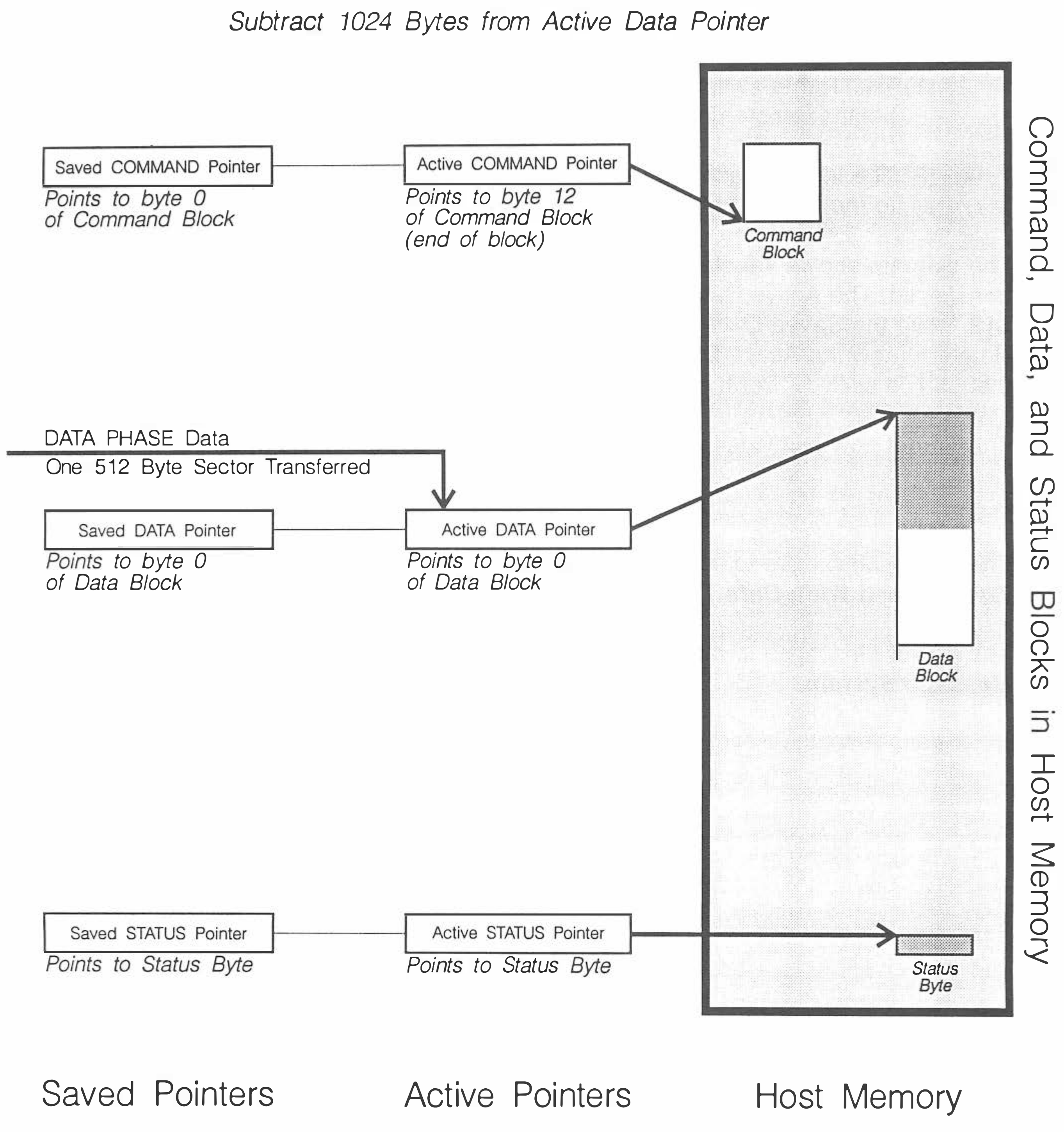

Subtract 1024 Bytes from Active Data Pointer
Saved COMMAND Pointer
Points to byte 0 of Command Block
Active COMMAND Pointer
Points to byte 12 of Command Block (end of block)
Command Block
DATA PHASE Data
One 512 Byte Sector Transferred
Saved DATA Pointer
Points to byte 0 of Data Block
Active DATA Pointer
Points to byte 0 of Data Block
Data Block
Saved STATUS Pointer
Points to Status Byte
Active STATUS Pointer
Points to Status Byte
Status Byte
Command, Data, and Status Blocks in Host Memory
Saved Pointers
Active Pointers
Host Memory

DIAGRAM 19: POINTERS AFTER 2ND MODIFY DATA POINTER

The first sector requested is now available for transfer to the Initiator. Diagram 20 shows the state of the Pointers at the end of the data transfer after the second MODIFY DATA POINTER Message:

(1) The Target switches from MESSAGE IN Phase to ***DATA IN Phase*** and begins transferring the data for the first sector.

(2) The Initiator responds to the ***DATA IN Phase*** and transfers the first sector to the first half of the Data block.

The diagram shows the state of the Pointers after two sectors (1024 bytes) have been transferred. The Active Data Pointer is incremented 512 times and now points at byte 512, while the Saved Data Pointer remains pointing at the start of the Data block. Note that now the Active Data Pointer points at byte 512 of the Data block, even though two sectors' worth of data have been transferred.

The Target now continues with ***STATUS Phase*** as shown in Diagram 11.

The use of Messages to control path options are covered under ***Synchronous Data Transfer*** and ***Wide Data Transfer***. See also ***Messages***.

The use of Messages to manage the exchange of other Messages is covered in ***Message System***.

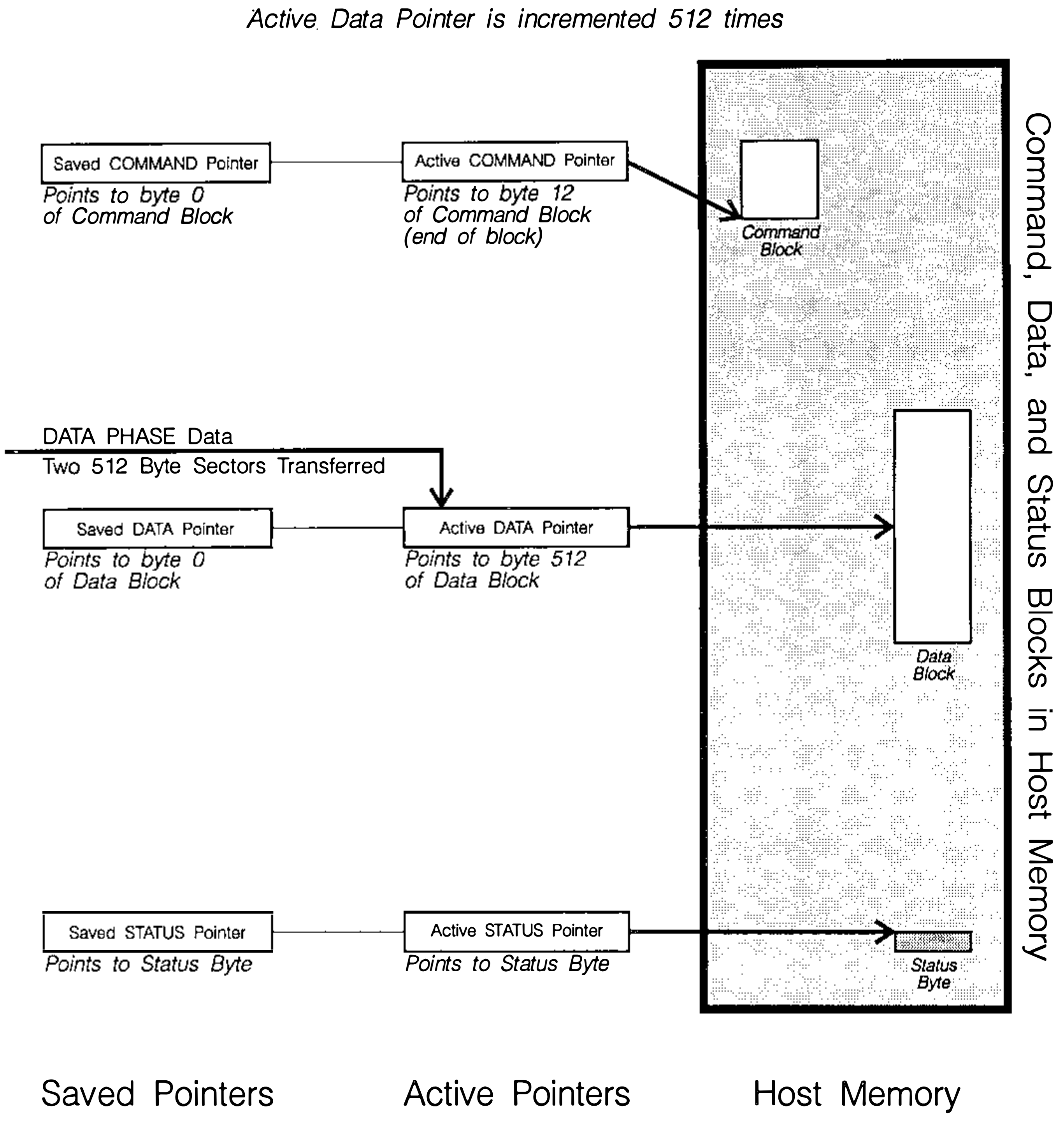

Active Data Pointer is incremented 512 times
Saved COMMAND Pointer
Points to byte 0 of Command Block
Active COMMAND Pointer
Points to byte 12 of Command Block (end of block)
Command Block
DATA PHASE Data
Two 512 Byte Sectors Transferred
Saved DATA Pointer
Points to byte 0 of Data Block
Active DATA Pointer
Points to byte 512 of Data Block
Data Block
Saved STATUS Pointer
Points to Status Byte
Active STATUS Pointer
Points to Status Byte
Status Byte
Command, Data, and Status Blocks in Host Memory
Saved Pointers
Active Pointers
Host Memory

DIAGRAM 20: 2ND SECTOR AFTER MODIFY DATA POINTER

Power-On to Selection Time. *tpost = 10 sec recommended*. This is defined as the time from "power application" to when a Target is able to accept certain Commands from an Initiator. This is one of those "funny" things in SCSI where you aren't even sure how anyone will know the difference. In those cases where the Host actually knows when a Target's power was applied (e.g., they share the same power supply or the host controls the Target's power sequence), this helps design timeouts in the Host System.

The Commands that the Target must be able to execute (see later Volumes of the SCSI Encyclopedia for details):

- TEST UNIT READY
- INQUIRY
- REQUEST SENSE

Protocol Chips. See ***Chips***.

Pull-Up. See ***Active Pull-Up***.

Queue.
Queue Tag.
Queue Tag Messages.

Queue. A Queue is used by a Target to manage more than one ***I/O Process*** at a time. My dictionary says that a Queue is a pigtail (whoops); the second definition is "A line, as of persons waiting to be served". A SCSI Queue is a line of I/O Processes waiting to be executed.

The need for a Queue is illustrated in Diagram 21. The need arises because of the ability of a Target to ***Disconnect*** from the bus in the middle of an I/O Process.

- Initiator #1 (we could call him Isaac) ***Connects*** to the Target (who we could call Tanya) to start an I/O Process and send her a ***SCSI Command***. Tanya decides to complete the operation without further interaction with Isaac, so she Disconnects from Isaac (recall that she says good-bye by sending the ***DISCONNECT Message***!).

- Initiator #2 (who we could call Izzy) calls up, er, Connects to Tanya to send her a SCSI Command.

Tanya now has a problem. She has a request (I/O Process) outstanding from Isaac, and here comes Izzy with another request:

- She could just say to Izzy "Sorry I'm Busy" by returning BUSY ***Status***. This is legal under SCSI, but not very nice; this is something an old ***SASI*** Target would do. Also, it kind of defeats the benefits of Disconnect.

- A better thing to do is to Queue the new I/O Process from Izzy. All Tanya has to do is Disconnect from Izzy (who was likely expecting it anyway; Tanya is popular). The other thing she has to do is save Izzy's Command in a safe place (i.e., a "Queue") until she is done with Isaac.

After she finishes Isaac's request, she retrieves Izzy's request from the Queue (activates his I/O Process), and starts executing it. In this case, Izzy's request takes longer to complete because of the delay caused by finishing Isaac's request.

An Important Note. During this discussion, we are referring to a Queue for a single ***Logical Unit***. Each Logical Unit has its own Queue. If a Target has more than one Logical Unit, then it will have a Queue for each. Each Queue operates independently for the most part. The only exception is that an I/O Process ready to be executed at the front of a Queue may have to wait if an ***Active I/O Process*** is using Target resources required by the Queued I/O Process.

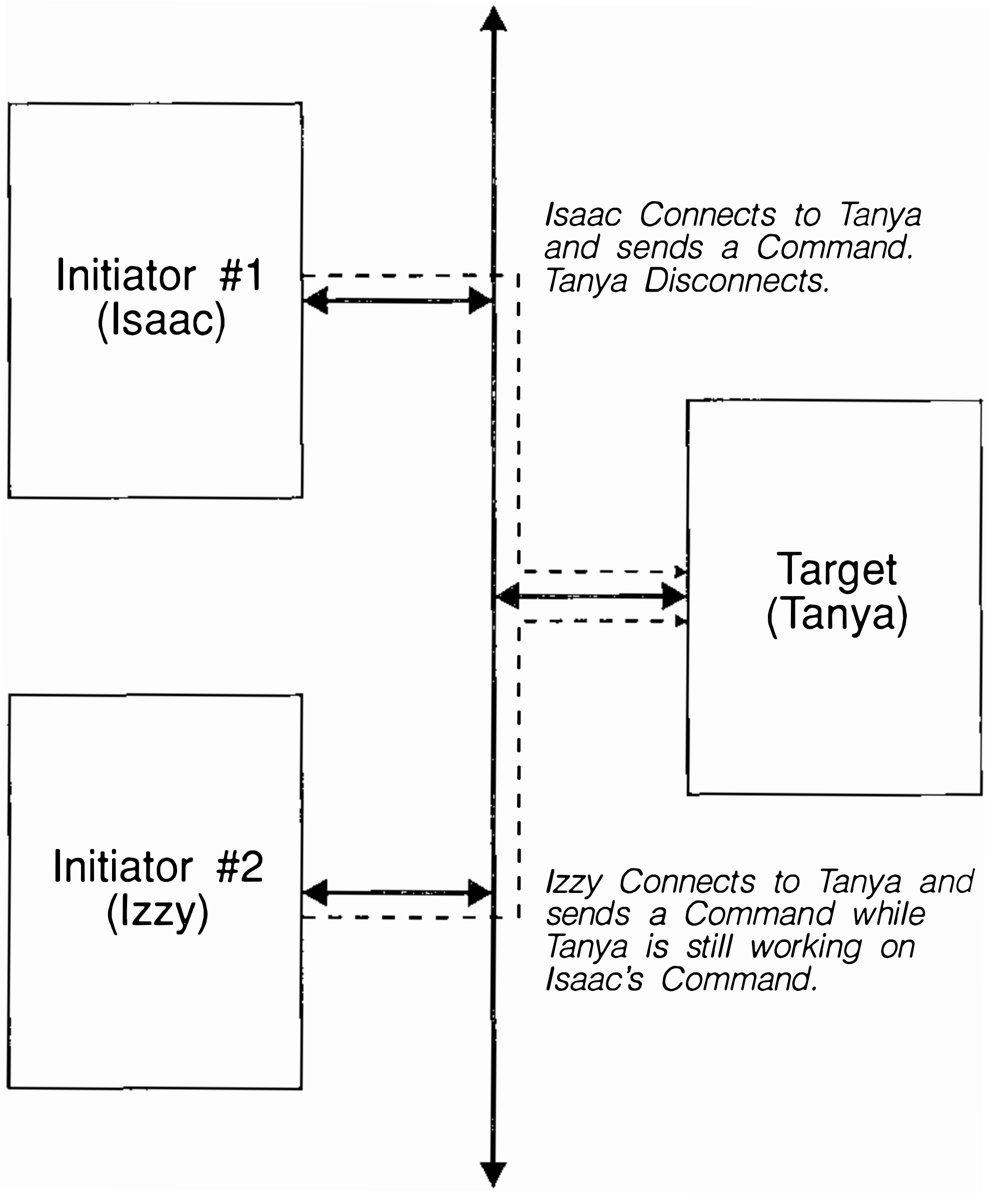

Initiator #1
(Isaac)
Isaac Connects to Tanya and sends a Command. Tanya Disconnects.
Target
(Tanya)
Initiator #2
(Izzy)
Izzy Connects to Tanya and sends a Command while Tanya is still working on Isaac's Command.

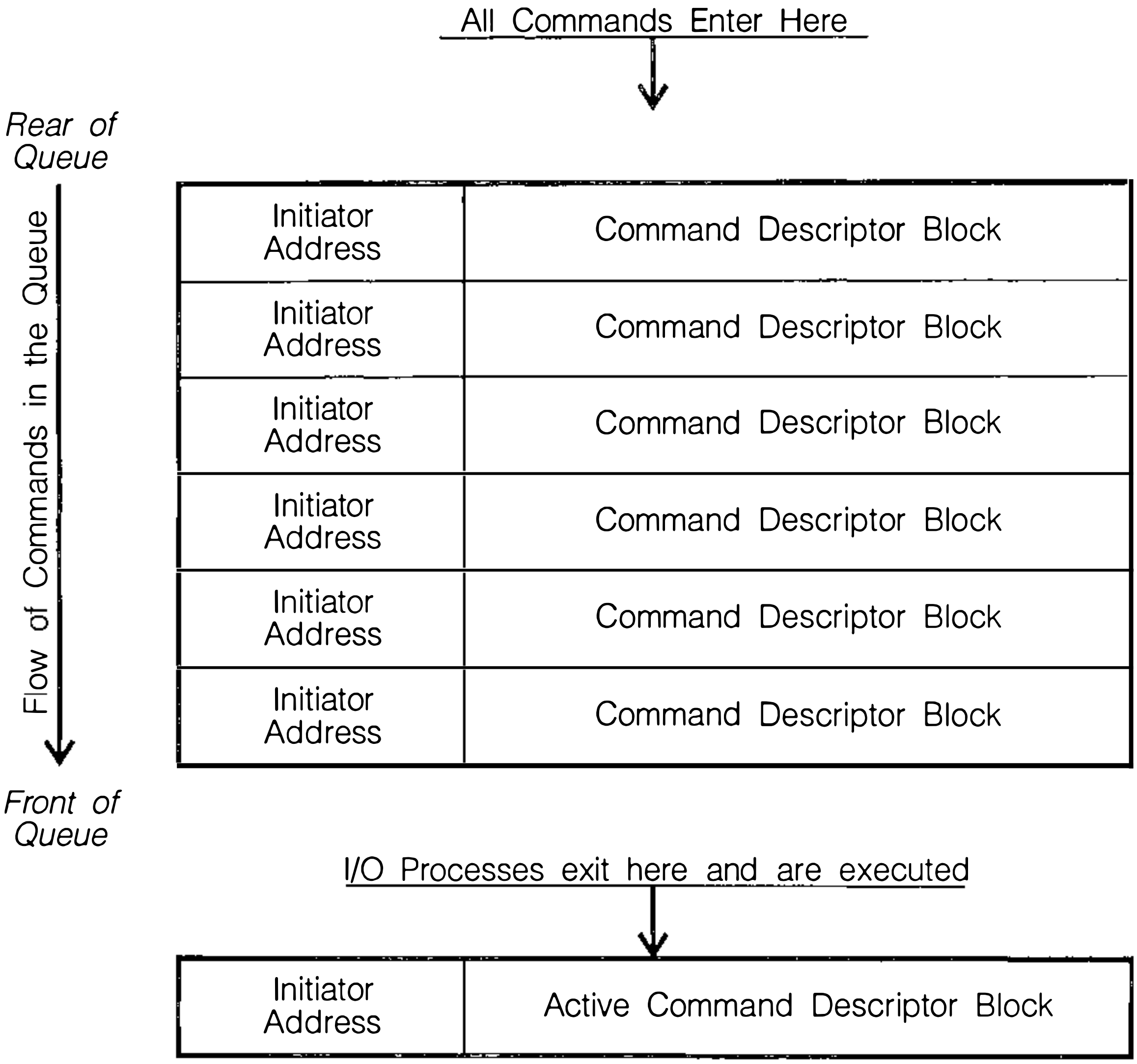
DIAGRAM 21: THE CASE FOR QUEUING

There are two types of Queuing defined by SCSI:

- Untagged Queuing. This is the simplest type of Queue, and very easy for a Target to manage. At most, one command is accepted from each Initiator. More than one command from each Initiator cannot be accepted because, on ***Reconnect***, the Initiator would not know which I/O Process was Reconnecting. (Why? Because the ***IDENTIFY Message*** only indicates which Logical Unit, not which I/O Process. See ***Nexus***.) The maximum Queue length is a number of elements per Logical Unit equal to the possible number of ***SCSI-1*** or ***SCSI-2*** Initiators in the system, minus one for the actively executing command. For example, if up to 7 Initiators are supported in the system, then the maximum Queue length per Logical Unit is 6. A shorter Queue length may also be implemented.

- Tagged Queuing. This type of Queuing solves the limitation of Untagged Queuing, and is really not that much harder for a Target to manage. The difference is that the Initiator follows the IDENTIFY Message with a ***Queue Tag Message***. This allows the Target to accept up to 256 commands from each Initiator. The maximum Queue length is a number of elements per Logical Unit equal to 256 Tags times the possible number of SCSI-2 Initiators in the system, minus one for the actively executing command. For example, if up to 7 Initiators are supported in the system, then the maximum Queue length per Logical Unit is (256 * 7) - 1, or 1791 elements. A shorter Queue length may also be implemented.

Untagged Queuing. Diagram 22 shows how Untagged Queuing works. The Queue itself records the Initiator and (optionally) the ***Command Descriptor Block (CDB)*** itself. Note that the Standard does not require First-In First-Out (FIFO) queuing of the commands, but we feel this is implementation is the fairest (of them all...).



Each Initiator Address in the Queue or Actively Executing must be unique

DIAGRAM 22: UNTAGGED QUEUING

The Queue can be described as a "C" language data structure:

```
// define Untagged Queue data structures and constants

#define MAX_CDB_SIZE     10   // 6 and 10 byte CDBs accepted

#define MAX_QUEUE_SIZE   6    // number of Initiators supported


struct  queue_element {
     unsigned       initiator : 3;      // Initiator SCSI Address
     unsigned char  cdb[MAX_CDB_SIZE];  // Command Block
} ;


struct queue_element queue[MAX_QUEUE_SIZE]; // this is the queue
                                            // itself


int last_element = -1;    // index to last element in queue
                          // -1 means the Queue is empty


struct queue_element new_process;  // new incoming process
```

NOTE: The "//" indicates a code comment to the end of the line. This convention is relatively new for the ANSI "C" definition.

Lotsa notes:

- `MAX_CDB_SIZE` can be set to 12, if 12 byte CDBs are supported.
- `MAX_QUEUE_SIZE` can be reduced to save storage allocated to the Queue. It does not have to be 6; a Target will seldom, if ever, be presented with a full complement of commands.
- The `struct` statement defines the fields of each element in the Queue. The "3" makes the `initiator` fields 3 bits wide.
- The second `struct` declares the queue itself as an array of queue elements, and the third one declares a single element used as a holder for the new incoming I/O Process.
- The variable `last_element` is an index for the `queue` array, which always points at the last valid entry in the queue. When `last_element` is set to "-1", the `queue` is empty.

If you don't know "C", try and treat this (and later examples) as "pseudo-code". We will endeavor, for clarity, not to use "clever C tricks". We know you "C" hacks out there could polish up this code nicely. Also, we know you assembly language hacks out there can convert this to assembly in your sleep...

Queue management is a snap (well, compared to the ***Message System***....). When a Target receives a new I/O Process from an Initiator, the first step is to see if there is a place for it:

```
// test whether to accept the new I/O Process

// test that the I/O Process is valid and is not redundant

for (i = 0; i <= last_element; i++)
     if (queue[i].initiator == new_process.initiator)
          abort_command(queue[i]); // abort the queued command
                                   // and dump the new one

// if no other I/O Process is active,
// and the Queue is empty, then just start the I/O Process

if ((lun_active == INACTIVE) && (last_element == -1))
     activate_process(new_process);


// if the IDENTIFY Message does not allow the Target to
// Disconnect

if (disconnect_okay == NOT_OKAY)
     return_status(BUSY_STATUS);

// if the last element in the queue is occupied

if (last_element == MAX_QUEUE_SIZE-1)
     return_status(BUSY_STATUS);
```

Notes:

- The first test determines whether the new I/O Process is correct. If the new I/O Process is for a Nexus that already exists (i.e., there is already a command queued for the Initiator), then both the new and the queued I/O Process are aborted.

- The second test determines whether queuing is even necessary. If the Logical Unit is currently inactive, meaning no other I/O Process is active, then there may be no need to queue the new I/O Process. We also check for an empty Queue just in case the Target has not had a chance to start the next I/O Process at the front of the Queue.

- The third test checks to see that the Initiator allowed the Target to Disconnect when it sent the IDENTIFY Message (a bit in the Message selects this). If not, then the Target cannot Disconnect to finish the currently Active I/O Process. A Target doing Untagged Queuing should then change to ***STATUS Phase*** and return BUSY ***Status***.

- The fourth test checks to see if the Queue is full. If the Queue is full, the test fails and the Target returns BUSY Status. If the `MAX_QUEUE_SIZE` is 6 (the maximum possible number of Initiators), then the test will always pass.

At this point the new I/O Process can be accepted into the Queue, and the following code fragment illustrates how:

```
// put the new I/O Process into the Queue


// Put it at the front

last_element++;                   // increment to next element
queue[last_element] = new_process; // copy new to queue
```

Notes:

- First, we increment `last_element` so that it points at the first empty element at the end of the queue. The `++` is an increment operator in "C".
- Then we copy the new_process information to the end of the queue.

Told you it was easy. Taking the next element from the front of the queue is almost as easy ("Next Please!"):

```
// get the next I/O Process off the Queue


activate_process(queue[0]);   // start the next process

for (i = 1; i <= last_element; i++)
     queue[i-1] = queue[i];   // shift everybody up
last_element--;               // decrement element index
```

Notes:

- First, we activate the process at the front of the Queue, which by definition is `queue[0]`. Depending on what the `activate_process` function does, it may be better to do this last. If the code that follows may not get executed, then `queue[0]` will have to be copied to a temporary variable before activating it. (This is illustrated in Tagged Queuing below.)

- The next step is a `for` loop that copies the Nth element to the N-1th element. This effectively "moves up" the line.

- `last_element` is decremented to account for the element removed from the list.

Easy!

What about the Initiator? The only responsibilities of the Initiator are:

- Issue the I/O Process using the IDENTIFY Message with the "Disconnect OK" bit set to one.

- Only have at most one I/O Process issued to each Logical Unit. Don't issue two I/O Processes to the same Logical Unit at the same time. At best, both processes are aborted. At worst, you'll screw up the Target.

Easy!

Tagged Queuing. We know this sounds scary, but it really isn't too bad. The good news is that the ***SCSI-2*** Standard gives a very good description of how to reorder Tagged I/O Processes and handle new Tagged I/O Processes coming in (really rather remarkable...). See the section in the standard on "Queued I/O Processes", and the Appendix on "Data Integrity and I/O Process Queuing". We'll be elaborating and filling in the cracks here.

Diagram 23 shows the flow of commands in a Tagged Queue. Commands Tagged with Ordered or Simple Queue Tags, and untagged commands, are always placed at the rear of the Queue initially. Tagged commands with Ordered Queue Tags and untagged commands keep their place in line. Tagged commands with Simple Queue Tags may be rearranged amongst themselves.

Commands tagged with Head of Queue Tags always enter at the front of the Queue. If another command with a Head of Queue Tag comes into a Queue that already has a Head of Queue Tag, then the new command bumps the existing command and takes the front of the Queue for itself.

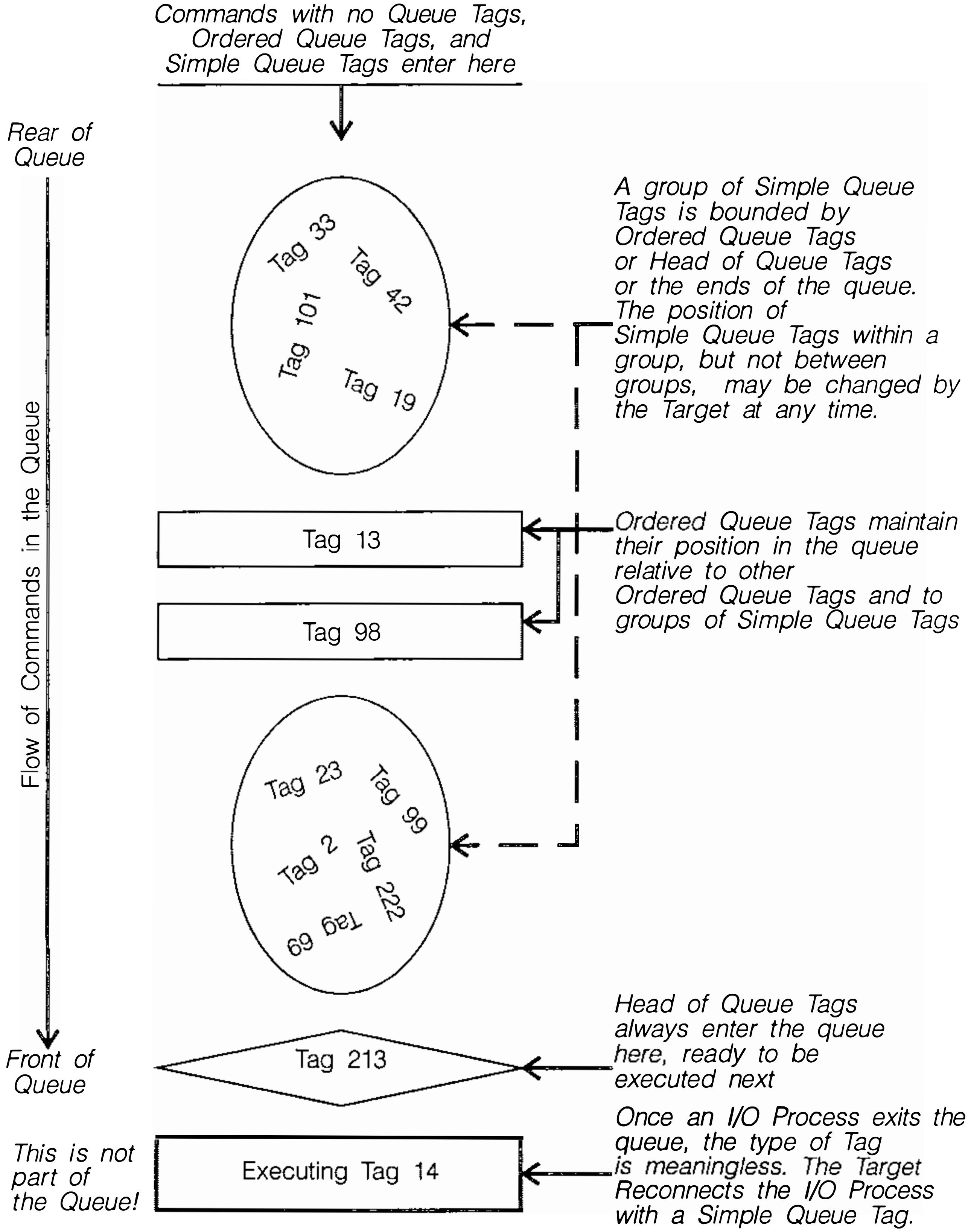

Commands with no Queue Tags, Ordered Queue Tags, and Simple Queue Tags enter here
Rear of Queue
Flow of Commands in the Queue
Front of Queue
Tag 33
Tag 42
Tag 101
Tag 19
A group of Simple Queue Tags is bounded by Ordered Queue Tags or Head of Queue Tags or the ends of the queue. The position of Simple Queue Tags within a group, but not between groups, may be changed by the Target at any time.
Tag 13
Tag 98
Ordered Queue Tags maintain their position in the queue relative to other Ordered Queue Tags and to groups of Simple Queue Tags
Tag 23
Tag 99
Tag 2
Tag 222
Tag 69
Head of Queue Tags always enter the queue here, ready to be executed next
Tag 213
This is not part of the Queue!
Executing Tag 14
Once an I/O Process exits the queue, the type of Tag is meaningless. The Target Reconnects the I/O Process with a Simple Queue Tag.

DIAGRAM 23: TAGGED QUEUING

Like we did for Untagged Queuing, the Tagged Queue can be described as a "C" language data structure:

```
// define Tagged Queue data structures and constants


#define MAX_CDB_SIZE     10   // 6 and 10 byte CDBs accepted

#define MAX_QUEUE_SIZE   100  // max space available in Target


struct  queue_element {
     unsigned        initiator : 3;      // Initiator SCSI Address
     unsigned        tag_type : 2;       // Queue Tag Message type
     unsigned char   tag;                // Queue Tag
     unsigned char   cdb[MAX_CDB_SIZE];  // Command Block
} ;


// possible values for tag_type

#define   TAG_NONE       0         // no tag (I_T_x Nexus)
#define   TAG_SIMPLE     1         // simple queue tag
#define   TAG_HEAD       2         // head of queue tag
#define   TAG_ORDERED    3         // ordered queue tag


struct queue_element queue[MAX_QUEUE_SIZE]; // this is the queue
                                            // itself


int last_element = -1;    // index to last element in queue
                          // -1 means the Queue is empty


struct queue_element new_process;  // new incoming process

struct queue_element temp_element; // temporary element storage
```

Notes:

- `MAX_CDB_SIZE` can be set to 12, if 12 byte CDBs are supported.

- `MAX_QUEUE_SIZE` for a Logical Unit can be chosen to save storage allocated to the Queue. It does not have to be 256 * the number of Initiators supported. A Target is unlikely to be presented with a full complement of I/O Processes.

- The `struct` statement defines the fields of each element in the Queue. The "3" makes the `initiator` fields 3 bits wide. The "2" makes the `tag_type` field 2 bits wide. No sense wasting memory.

- The `tag_type` field indicates which ***Queue Tag Message*** was received. This field is used to decide where to put newly received I/O Processes, and also how to handle re-ordering them. The `tag` field is the ***Queue Tag*** from the Queue Tag Message.

- The second `struct` declares the queue itself as an array of queue elements, and the third one declares a single element used as a holder for the new incoming I/O Process.

- The variable `last_element` is an index for the `queue` array, which always points at the last valid entry in the queue. When `last_element` is set to "-1", `queue` is empty.

Like Untagged Queuing, when a Target receives a new I/O Process from an Initiator, the first step is to see if there is a place for it. There is the same number of steps as Untagged Queuing, only a couple of details change:

```
// test whether to accept the new I/O Process

// test that the I/O Process is valid and is not redundant
// Abort all commands for the Initiator on any conflict

// see if there is an I_T_x Nexus already in the Queue
// for the Initiator

for (i = 0; i <= last_element; i++)
     if ((queue[i].initiator == new_process.initiator)
     &&  (queue[i].tag_type == TAG_NONE))    // "&&" is "AND"
          abort_all(new_process.initiator); // abort all
                                   // commands

// see if there is an I_T_L_Q Nexus already in the Queue
// for the same Initiator and Queue Tag

for (i = 0; i <= last_element; i++)
     if ((queue[i].initiator == new_process.initiator)
     &&  (queue[i].tag == new_process.tag))
          abort_all(new_process.initiator); // abort all
                                   // commands

// if no other I/O Process is active,
// and the Queue is empty, then just start the I/O Process

if ((lun_active == INACTIVE) && (last_element == -1))
     activate_process(new_process);


// if the IDENTIFY Message does not allow the Target to
// Disconnect

if (disconnect_okay == NOT_OKAY)
     return_status(BUSY_STATUS);


// if the last element in the queue is occupied

if (last_element == MAX_QUEUE_SIZE-1)
     return_status(QUEUE_FULL_STATUS);
```

Notes:

- The first two tests determine whether the new I/O Process is correct. If the new I/O Process is for an I_T_x Nexus that already exists (i.e., there is already an untagged command queued for the Initiator), then both the new and the queued I/O Process are aborted. If the new I/O Process is for an I_T_L_Q Nexus that already exists (i.e., there is already a tagged command queued for the Initiator for the same Tag value), then the new I/O Process and all other queued I/O Processes for that Initiator are aborted.

- The third test determines whether queuing is even necessary. If the Logical Unit is currently inactive, meaning no other I/O Process is active, then there may be no need to queue the new I/O Process. We also check for an empty Queue just in case the Target has not had a chance to start the next I/O Process at the front of the Queue.

- The fourth test checks to see that the Initiator allowed the Target to Disconnect when it sent the IDENTIFY Message (a bit in the Message selects this). If not, then the Target cannot Disconnect to finish the currently Active I/O Process. A Target doing Tagged Queuing should then change to ***STATUS Phase*** and return BUSY ***Status***.

- The fifth test checks to see if the Queue is full. If the Queue is full, the test fails and the Target returns QUEUE FULL Status (note that BUSY Status is returned for Untagged Queuing). If the `MAX_QUEUE_SIZE` is the maximum possible number of Initiators times the maximum number of Queued I/O Processes, then the test will always pass.

At this point the new I/O Process can be accepted into the Queue. For Tagged Queuing, this is a little harder, but not much:

```
// put the new I/O Process into the Queue at the
// appropriate place


// if the Queue is empty, then put it at the front

if (last_element == -1)
     {
     last_element++;               // increment to 0th element
     queue[0] = new_process;       // copy new to queue
     }


else if ((new_process.tag_type == TAG_ORDERED)
     ||  (new_process.tag_type == TAG_NONE)) // "||" is "OR"
     {
     last_element++;               // increment to next element
     queue[last_element] = new_process; // copy new to last
     }


else if (new_process.tag_type == TAG_SIMPLE)
     {
     last_element++;               // increment to next element
     queue[last_element] = new_process; // copy new to last
     re_order(queue);         // redefine the order
     }


else if (new_process.tag_type == TAG_HEAD)
     {
     for (i = 0; i <= last_element; i++)
          queue[i+1] = queue[i];   // shift everybody back
     last_element++;               // increment element index
     queue[0] = new_process;       // copy new to front
     }
```

Notes:

- If the Queue is empty (`last_element = -1`), then we put the new I/O Process at the start of the Queue. This first test is done to skip the overhead that may be associated with later tests (e.g., shifting or re-ordering elements). An index of "0" is specified because it may be a little faster to execute....

- If the Tag is an "Ordered" Queue Tag, or there is no Tag because this is an I_T_x Nexus, then the I/O Process is always added to the end of the Queue. `last_element` is incremented to point at the new last element in `queue`.

- If the Tag is a "Simple" Queue Tag, then the I/O Process is added to the end of the Queue. `last_element` is incremented to point at the new last element in `queue`. Then, a call to a "re-order" function is made to see if the new element can be rearranged relative to the other "Simple" elements in the Queue. More on that later.

- If the Tag is a "Head of Queue" Queue Tag, then the I/O Process is always added to the front of the Queue. First, all of the other elements in the Queue are moved back one place; this is the purpose of the `for` loop. `last_element` is incremented to point at the new last element in `queue`. Finally, the new I/O Process is copied to the empty element at the front of the Queue. Note that any prior "Head of Queue" Tags in the Queue will be pushed back as well.

That wasn't too bad, we hope. Taking the next element from the front of the queue is same as Untagged Queuing. This time, we'll do the "activate" last:

```
// get the next I/O Process off the Queue


temp_element = queue[0];        // copy front element to temporary
                                // variable

for (i = 1; i <= last_element; i++)
     queue[i-1] = queue[i];     // shift everybody back

last_element--;                 // decrement element index

activate_process(temp_element);     // start the next process
```

Notes:

- First, we copy the element at the front of the Queue (`queue[0]`) to a temporary variable for later use. We want to finish the management of the Queue before starting the next Process.

- The next step is a `for` loop that copies the Nth element to the N-1th element. This effectively "moves up" the line.

- `last_element` is decremented to account for the element removed from the list.

- The I/O Process is activated by calling `activate_process` and specifying the temporary variable we set earlier.

Easy!

What about the Initiator? The main responsibilities of the Initiator are:

- Issue the I/O Process using the IDENTIFY Message with the "Disconnect OK" bit set to one.

- Each Queue Tag issued by the Initiator must be unique for each Logical Unit. The Initiator may not start an I/O Process with a Queue Tag that is the same as the Queue Tag for an I/O Process that the Target already has queued or active.

Easy!

Getting the Most out of your Tagged Queue. The only real reason to use Tagged Queuing is to allow the Target to re-order the I/O Processes received. It might be nice to be able to toss every command from the Host System out to the Target, but an Initiator could also do that kind of queuing with little problem. Tagged Queuing really comes into its own when the Target is allowed to decide the execution order of Tagged I/O Processes.

This re-ordering is done for some very typical reasons. For example, when several requests are received by a disk, the amount of time spent seeking for different sectors can be reduced by sorting the requests into the appropriate order. Table 10 shows how this re-ordering into ascending order can improve performance.

TABLE 10: TAGGED QUEUING RE-ORDERING EXAMPLE

Without Tagged Queuing		With Tagged Queuing	
Block Accessed	Seek Time (ms)	Block Accessed	Seek Time (ms)
start at block 0	---	start at block 0	---
1000	10	0	0
2000	10	1	0
0	15	1000	10
3000	20	1003	0
2005	10	2000	10
1003	10	2005	0
1	10	3000	10
Total Seek Time	**85 ms**	Total Seek Time	**30 ms**

On the left side of the table, a sequence of ***Logical Block*** requests is shown. Each block request is serviced in the order received. As a result, a lengthy seek is performed to service each request. On the right side of the table, Tagged Queuing is used, so the requests can be sorted into ascending order. This cuts the amount of time spent seeking to almost a third in this example.

Note that the example shows all requests sorted. In practice, the first request would be acted on before the second request is received. The actual result would depend on when each I/O Process was received by the Target.

Re-ordering may also be called for when the Target has a Cache. The Logical Blocks in a Cache have a faster effective seek time which enters into the re-ordering equation.

In general, Targets should re-order I/O Processes in the Queue if they bother to do Tagged Queuing at all. Targets should also follow the "Data Integrity" guidelines in the Appendix of the SCSI-2 Standard.

Again, What About Initiators? Initiators must use the three Queue Tag Messages properly:

SIMPLE QUEUE TAG Message is the "all-purpose" Queue Tag to use for all I/O Processes. This message should be used unless there is an overriding reason to use one of the other two Messages. This Message gives the Target the most freedom to re-order for best performance.

ORDERED QUEUE TAG Message is used when the Initiator wants to control the ordering of I/O Process execution. We can think of two uses for this:

- Use it to finish off a set of requests. The Initiator issues several READ Commands to get a file from a disk. If it issues the last READ as an Ordered Tag, then it knows that when that last Tagged I/O Process is completed, it has the whole file. Think of it as a "Synchronize Queue" function.

- The Initiator may not trust the Target to handle the re-ordering of WRITE Commands relative to READ Commands. As described in the Appendix of the SCSI-2 standard, if a WRITE to a block is moved in order relative to a READ of the block, the Initiator may get the old version of the block or the updated version. The Initiator can either handle it by never mixing those kinds of requests, or it can just issue Ordered Tags for a WRITE Commands. This can impact performance, so try and treat this as a last solution.

HEAD OF QUEUE TAG Message is used for special cases only. If an Initiator needs to get a Command to the Logical Unit right now, it can use this type of Queue Tag. This only puts it at the front of the Queue; it does not interrupt the currently executing I/O Process(es) (even to break a series of ***Linked Commands***). This makes the HEAD OF QUEUE TAG Message a bit limited as an "emergency" operation, since the currently executing I/O Process could take a long time for all the Initiator knows.

Another use might be for high demand/high bandwidth data, such as data acquisition or video data. These types of data need to be dumped now. In order to get timely service to write this data to a disk, the Initiator can use Head of Queue Tags to get serviced sooner, at the expense of the performance of other processes.

The three Messages previously described allow an Initiator to add I/O Processes to the Queue. The following Messages allow an Initiator to remove and abort I/O Processes from the Queue:

ABORT TAG Message is used to remove a single I/O Process from the Queue. The Initiator usually issues this Message by Connecting for just that purpose. After ***BUS FREE Phase***, the Initiator Connects to the Target via ***ARBITRATION Phase*** and ***SELECTION Phase***. Then during ***MESSAGE OUT Phase***, the following Messages are sent by the Initiator in the following order:

- IDENTIFY Message is sent to establish which Logical Unit is to be accessed; and by extension, which Queue.

- SIMPLE QUEUE TAG Message is sent to establish which I/O Process is to be removed and aborted. The Tag field of the Message is set to the Tag value of the I/O Process to be removed and aborted.

- ABORT TAG Message is sent to remove and abort the I/O Process specified by the SIMPLE QUEUE TAG Message. Note that the I/O Process is aborted whether it is in the Queue or it is currently executing. After this Message, the Target goes to BUS FREE Phase.

ABORT Message is used to remove from the Queue and abort all I/O Processes that belong to the Initiator. Any I/O Processes that belong to other Initiators are not affected. As with the ABORT TAG Message, the Initiator Connects to the Target and then sends the following Messages:

- IDENTIFY Message is sent to establish which Logical Unit is to be accessed; and by extension, which Queue.

- ABORT Message is sent to remove and abort the I/O Processes belonging to the currently Connected Initiator. Note that the I/O Processes are aborted whether they are in the Queue or are currently executing. After this Message, the Target goes to BUS FREE Phase.

CLEAR QUEUE Message is used to remove from the Queue and abort all I/O Processes (queued or actively executing) that belong to any Initiators. This should only be used in extreme circumstances! As with the ABORT Message, the Initiator Connects to the Target and then sends the following Messages:

- IDENTIFY Message is sent to establish which Logical Unit is to be accessed; and by extension, which Queue.

- CLEAR QUEUE Message is sent to remove and abort all I/O Processes for the Logical unit. Note that the I/O Processes are aborted whether they are in the Queue or are currently executing. After this Message, the Target goes to BUS FREE Phase, and the Queue for the Logical Unit is empty.

Other Considerations. Tagged Queuing requires some other special considerations for Initiators and Targets:

- ***Contingent Allegiance Condition***: When the Target returns CHECK CONDITION Status at the end of an I/O Process, both the Initiator and Target drop out of Tagged Queuing temporarily for that Logical Unit. Until the Initiator clears the ***Condition***, the Queue is halted; the Target returns BUSY Status for any subsequent I/O Processes received, and may not execute any I/O Process in the Queue. The Initiator sends the REQUEST SENSE Command that clears the Condition without a Queue Tag Message. After the Condition is cleared, Tagged Queuing is resumed. (There is an option under the MODE SELECT Command to clear the Queue when the Condition is cleared.)

- ***Extended Contingent Allegiance (ECA) Condition***: The ECA Condition follows exactly the same rules as the Contingent Allegiance Condition. All I/O Processes sent by the Initiator while the Condition exists are sent without a Queue Tag Message.

- **Mixing Tagged and Untagged Queuing**: We can easily envision a situation where an old Initiator (that can't do Tagged Queuing) is attached to a SCSI Bus where the other Initiators and Targets support Tagged Queuing. The Target must accept Untagged Commands from the old Initiator in a manner consistent with Untagged Queuing, at least from the old Initiator's point of view.

  The answer is easy: The Target takes the Untagged I/O Process and queues it as if it were an Ordered Tag (there is no real Tag number). This ensures that the I/O Process is executed in the order issued, exactly like the old Initiator expects.

  An Initiator should never mix Untagged I/O Processes with Tagged I/O Processes for a Logical Unit, unless a Contingent Allegiance or ECA Condition currently exists for that Initiator. If it does mix them, the Target will dump all of the Initiator's queued and actively executing I/O Processes and return an error.

Queue Tag. A Queue Tag is an arbitrary 8-bit number that is used to "identify" a particular ***Queued*** I/O Process. The Queue Tag is the final element of a ***Nexus*** that includes the ***SCSI Address*** of the Initiator and Target, and the ***Logical Unit Number (LUN).*** Once a particular Queue Tag value is sent by the Initiator to a ***Logical Unit*** during the ***Initial Connection***, that value is associated with that ***I/O Process*** until the process is terminated. The Queue Tag value may then be used for a new I/O Process on that Logical Unit.

Queue Tag Messages. The "Queue Tag Messages" refers to the three ***Messages*** that are used to pass a ***Queue Tag*** between two ***SCSI Devices***. These Messages are:

- ***SIMPLE QUEUE TAG Message***: The Target ***Queues*** the ***I/O Process*** anywhere between the end of the Queue and the last ORDERED QUEUE TAG received (if any).

- ***ORDERED QUEUE TAG Message***: The Target puts the I/O Process at the end of the Queue, and puts no subsequent SIMPLE QUEUE TAGS or ORDERED QUEUE TAGS in front of it.

- ***HEAD OF QUEUE TAG Message***: The Target puts the I/O Process at the front of the Queue.

Reconnect.
Reconnection.
RELEASE RECOVERY Message.
Release Signal.
REQ/ACK Offset.
REQ Signal.
REQB Signal.
RESELECTION Phase.
Reselection Timeout.
Reserved.
Reset Condition.
Reset Hold Time.
Reset to Selection Time.
RESTORE POINTERS Message.
RST Signal.

Reconnect. "Reconnect" is a verb used in the SCSI standard to describe the action that the Target performs to re-establish a "working relationship" with an Initiator, also known in SCSI as a ***Nexus***. Contrast this with a "***Connect***", which the Initiator performs. A "reconnect" includes:

(1) ***RESELECTION*** of an Initiator by a Target.

(2) The transfer of an ***IDENTIFY*** message from the Target to the Initiator to establish the Logical Unit.

(3) If ***Queuing*** is used, the transfer of (specifically) a ***SIMPLE QUEUE TAG Message*** to establish a tag to identify the ***I/O Process***.

The flow to reconnect to various types of Nexuses is shown in Diagram 24 on the following page.

When the Target has "reconnected" to the Initiator, a Nexus (relationship) that was previously established between the two devices is revived. The SCSI standard calls this a ***Reconnection***. A reconnection is performed to continue an ***I/O Process***.

Usually after the reconnect is completed, the Target requests a ***DATA Phase*** with the Initiator. Or, the Target may go to the ***STATUS Phase*** and complete the command.

Sometimes, as with "connect", it helps to use a word in a sentence to better define it:

> "The Target **reconnects** to an Initiator to continue the execution of a command."
>
> "The Target will now **reconnect** to the Initiator to complete the I/O Process issued the last time the Initiator **connected** to the Target."
>
> "An Initiator connects to a Target, but a Target **reconnects** to an Initiator."

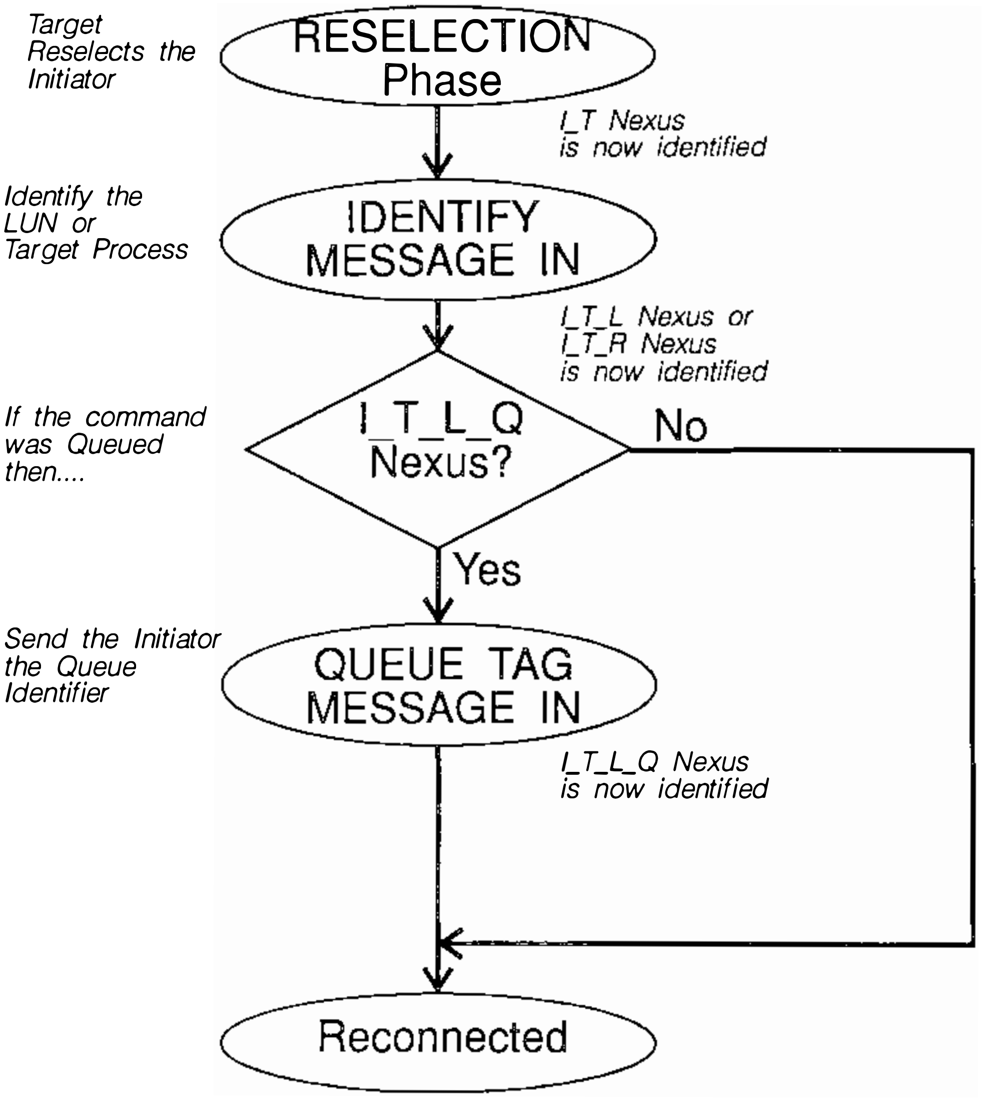


DIAGRAM 24: RECONNECT FLOW DIAGRAM

Reconnection. A "reconnection" is what exists after:

(1) A Target has ***Reconnected*** to an Initiator.

A reconnection ends when the next BUS FREE phase occurs. Note that reconnection is a subset of ***Connection***.

RELEASE RECOVERY Message. The RELEASE RECOVERY message is sent from an Initiator to a Target to clear an ***Extended Contingent Allegiance (ECA) Condition***. RELEASE RECOVERY is a single-byte message:

Bit Byte	7	6	5	4	3	2	1	0
0	Message Code = 10 hex							

This message is only sent in the OUT direction; i.e., a ***MESSAGE OUT Phase*** from the Initiator to the Target. If the Initiator sends this message, it must follow the ***IDENTIFY Message*** of an ***Initial Connection*** (no ***Queue Tag Messages*** allowed!). The Initiator should only send this message after completing any extended recovery that may have been required after the Target sent the ***INITIATE RECOVERY Message***.

When the Target receives this message, it clears the ECA Condition. If there was no ECA Condition in effect at the time the Target received the message, then the message has no effect. In any case, the Target goes to ***BUS FREE Phase*** after receiving this message.

Summary of Use: The RELEASE RECOVERY message is sent only by an Initiator to end an ECA condition.

Release Signal. To allow a signal to return to the "false" state. "Released" is used in the SCSI standard to indicate that a signal or signal pair is no longer driven to the "true" or "one" state, and is also not driven to the "false" or "zero" state; the signal is allowed to return to the "false" by the effect of the bus ***Termination***. Compare to ***Negate*** and ***Assert***. See ***Signal Levels***.

REQ/ACK Offset. See ***Synchronous Offset***.

REQ Signal. The REQ signal is asserted by the Target to begin either an ***Asynchronous Data Transfer*** or a ***Synchronous Data Transfer***. REQ is never asserted by an Initiator. The Initiator responds to REQ with the ***ACK Signal***. For transfers from the Target to the Initiator, the REQ signal indicates that the data is available on the bus. For transfers from the Initiator to the Target, the REQ signal indicates that the Target is ready to accept data from the Initiator.

REQB Signal. The REQB Signal is used in systems that use the ***B Cable*** to do ***Wide Data Transfers***. It has the same control over data transfer on the B Cable as the REQ Signal on the ***A Cable***. The REQB Signal is independent of the REQ signal; it only reacts to the ***ACKB Signal***, and it is only related to the ***Data Bus Signals*** on the B Cable.

RESELECTION Phase. The RESELECTION Phase is used by a Target to ***Reconnect*** to an Initiator and continue an ***I/O Process***. The RESELECTION Phase, like the ***SELECTION Phase***, is always preceded by the ***ARBITRATION Phase***, and is always followed by a ***MESSAGE IN Phase to send the IDENTIFY Message***, and possibly a ***SIMPLE QUEUE TAG Message***.

Diagram 25 and Diagram 26 show how Initiators and Targets handle RESELECTION Phase. Except for some details, RESELECTION Phase is just like SELECTION Phase with the Initiator and Target roles reversed. During RESELECTION Phase, the Target sets the bus signals that define the phase, and then waits for a response from the Initiator. After the Initiator response, the Target releases those signals and begins sequencing through the ***Information Transfer Phases***, starting with the MESSAGE IN Phase.

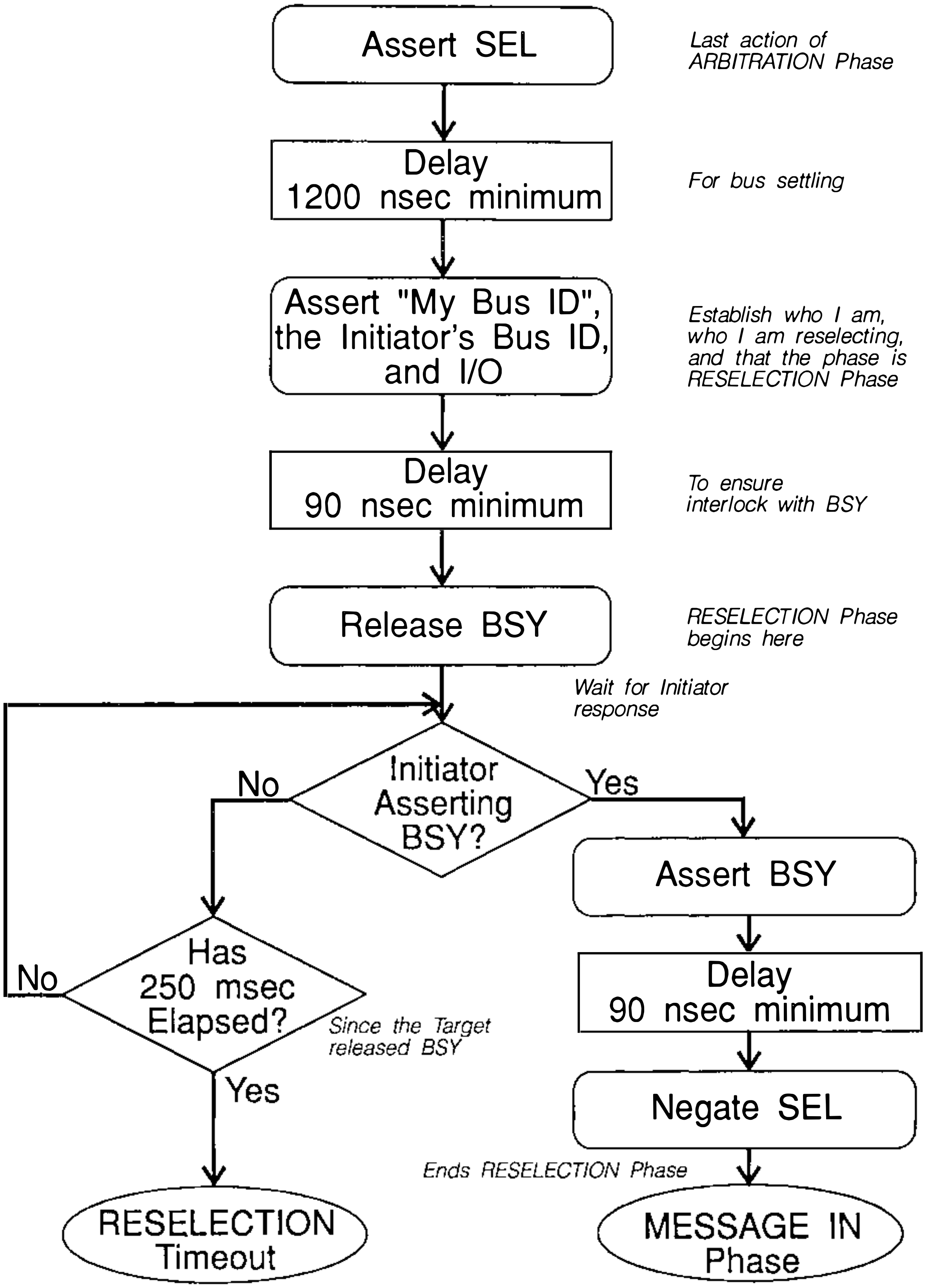
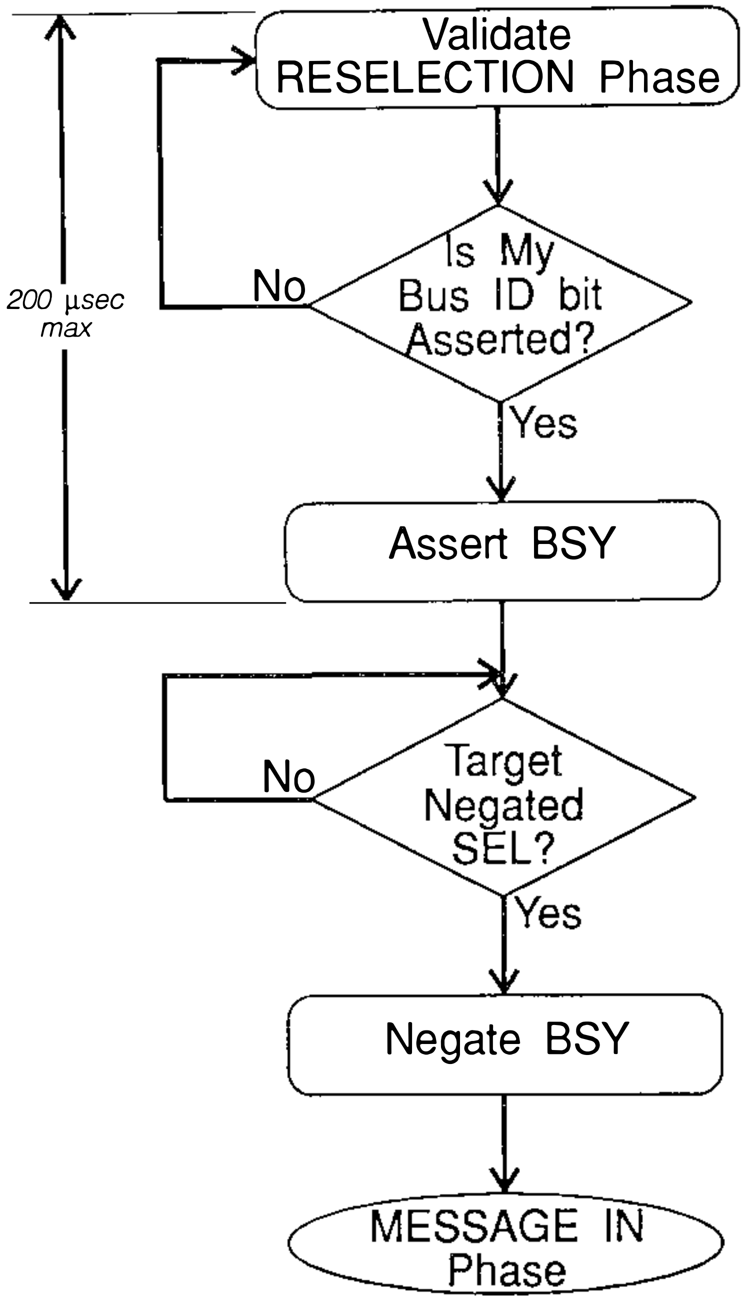

Assert SEL
Last action of ARBITRATION Phase
Delay 1200 nsec minimum
For bus settling
Assert "My Bus ID", the Initiator's Bus ID, and I/O
Establish who I am, who I am reselecting, and that the phase is RESELECTION Phase
Delay 90 nsec minimum
To ensure interlock with BSY
Release BSY
RESELECTION Phase begins here
Wait for Initiator response
Initiator Asserting BSY?
No
Yes
Has 250 msec Elapsed?
Since the Target released BSY
No
Yes
RESELECTION Timeout
Assert BSY
Delay 90 nsec minimum
Negate SEL
Ends RESELECTION Phase
MESSAGE IN Phase

DIAGRAM 25: RESELECTION FLOW DIAGRAM FOR TARGETS



SEL and I/O asserted, BSY negated

Am I reselected?

Yes, respond to the Target

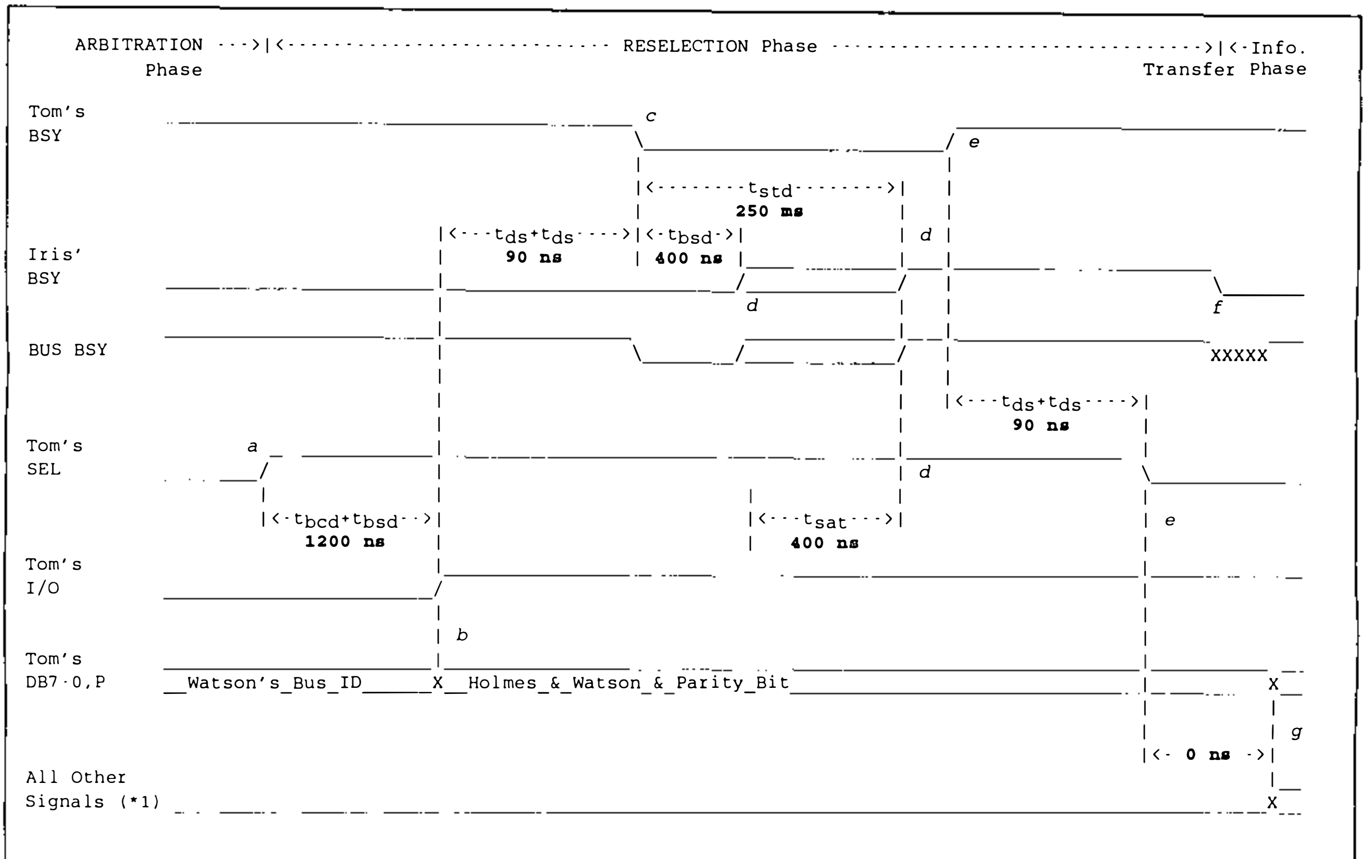
Wait for the Target to see my response (no timeout!)

The Target has BSY asserted now

Target will send IDENTIFY message

DIAGRAM 26: RESELECTION FLOW DIAGRAM FOR INITIATORS

Figure 2 shows the exact timing of RESELECTION Phase:



(*1): Other signals = MSG, C/D, REQ, and ACK. They may not be asserted until RESELECTION Phase is completed. ATN may be asserted but it is not part of the normal protocol and the Target does not respond to it until after the Nexus is established.

FIGURE 2: RESELECTION PHASE TIMING

Here's how events proceed in RESELECTION Phase. Compare them to SELECTION Phase:

(a) Tom has successfully acquired the bus during the preceding ***ARBITRATION Phase*** by asserting the ***SEL Signal*** to take the bus (Iris must not have been trying to call...).

(b) After a ***Bus Clear Delay*** of 800 nsec to allow the losers to release their bus signals, and after an additional ***Bus Settle Delay*** of 400 nsec to allow the bus to settle (imagine that?) from the signal releases (total delay 1200 nsec), Tom begins setting up for RESELECTION Phase. He asserts his ***SCSI Bus ID*** (which is 0) and Iris' SCSI Bus ID (which is 7) on the ***DATA BUS Signals***; Iris is the Initiator that Tom will be reselecting. The data bus value is then **10000001** with the ***Parity*** bit also asserted. Tom also asserts the ***I/O Signal*** to differentiate RESELECTION Phase from SELECTION Phase.

(c) At least two ***Deskew Delays*** (total 90 nsec) after asserting the data bus and the I/O signal, Tom releases the ***BSY Signal*** to begin the selection.

(d) Other devices may now examine the bus to determine who the lucky Initiator to be reselected is. The other devices must delay a minimum of 400 nsec (Bus Settle Delay again) before examining the data bus. Iris, the "One Who is Reselected", examines the bus and sees that she is reselected. Tom has certain expectations about how Iris will respond:

- Iris should respond within 250 msec (***Selection Timeout Delay***) from the start of the RESELECTION Phase (Tom released BSY in step (c)). In other words, a SCSI device should look for a RESELECTION Phase at least every 250 msec if interrupts are not used or are disabled.

- Iris must respond within 200 µsec (***Selection Abort Time***) of noticing that she is reselected. This is done to facilitate a smooth ***Reselection Timeout***.

Iris responds to the Reselection by Tom by asserting the BSY signal.

(e) Tom sees that Iris has asserted BSY in response to the selection. Tom then also asserts the BSY signal. Then, after delaying 90 nsec (two Deskew Delays) to ensure the signal interlock, Tom may release the SEL signal.

(f) As soon as Iris sees the SEL signal go false (no delay), she releases the BSY signal. The release of the BSY signal can cause a ***Wire-OR Glitch*** of up to 400 nsec in duration. This is not a problem if everybody meets the ***BUS FREE Delay*** requirement (see also ***BUS FREE Phase***).

(g) As soon as Tom releases the SEL signal, he may begin setting up for the MESSAGE IN Phase by changing the ***Bus Phase Signals***.

Other observations on RESELECTION Phase:

- Note that ***Parity*** is valid during the RESELECTION Phase. Since two data bits are always asserted true during RESELECTION, the Parity bit (DB(P)) is always asserted as well, since SCSI Parity is odd (that is, as opposed to even....).

- The Selection Timeout Delay exists to provide an upper limit to the time an Initiator takes to respond. If the Initiator hasn't responded within that time, the Target may respond in a couple of ways:

  - The Target may reasonably assume that the Initiator no longer exists at that ***SCSI Address***. When this happens, the Target may discard the pending I/O Process (as with an ***ABORT Message*** or ***ABORT TAG***

Message) and also may hold the Aborted status of the operation in anticipation of a later selection by the Initiator. It's kind of the same as an ***Unexpected BUS FREE Phase***, except the Initiator doesn't see the event. The Target is assuming that the timeout only occurs when the Initiator suffers a catastrophic failure, such as some bozo tripping over the power cord (and who was the bozo who left the cord out there anyway?).

- A smarter Target might assume the Initiator is just too busy to respond to the RESELECTION Phase. In this case the Target waits some long period of time and tries again. The Target may keep trying forever, or it may give up after some number of times and respond as above. Our humble opinion is that modern Initiators are able to at least assert BSY in response to RESELECTION Phase. Once the Initiator responds to BSY, there is no timeout. A well-behaved Initiator will be able to send the ***DISCONNECT Message*** to the Target if it can't handle the Reconnection.

Table 12 shows the timing values used during RESELECTION Phase.

TABLE 12: TIMING VALUES USED DURING RESELECTION PHASE

Symbol	Timing Name	MIN or MAX?	Time
t_{bsd}	Bus Settle Delay	MINIMUM	400 nsec
t_{bcd}	Bus Clear Delay	MINIMUM	800 nsec
t_{std}	Selection Timeout Delay	MINIMUM	250 msec
t_{ds}	Deskew Delay	MINIMUM	45 nsec
t_{sat}	Selection Abort Time	MAXIMUM	200 µsec

This page is nearly blank!
We use the space to improve Readability.

Reselection Timeout. The Reselection Time-out is a formal procedure to prevent bus hangs when a Target decides to give up on a pending ***RESELECTION Phase***. Please see ***Selection Time-out*** for an explanation of how the bus can hang.

The Reselection Time-out procedure is almost the same as the Selection Time-out procedure, except that the Target must manage the ***I/O Signal***:

(1) After the Target has maintained RESELECTION Phase for a Selection Time-out Delay, it decides to abort the reselection. The first thing the Target does is release all ***Data Bus Signals***. This is the same as in Selection Time-out.

(2) The Target waits for at least a ***Selection Abort Delay*** plus two ***Deskew Delays*** (200.09 µsec total), and if the Initiator has not yet responded, releases the ***SEL Signal*** and the ***I/O Signal***, returning to the ***BUS FREE Phase***.

Diagram 27 shows how Targets handle Reselection Timeout.

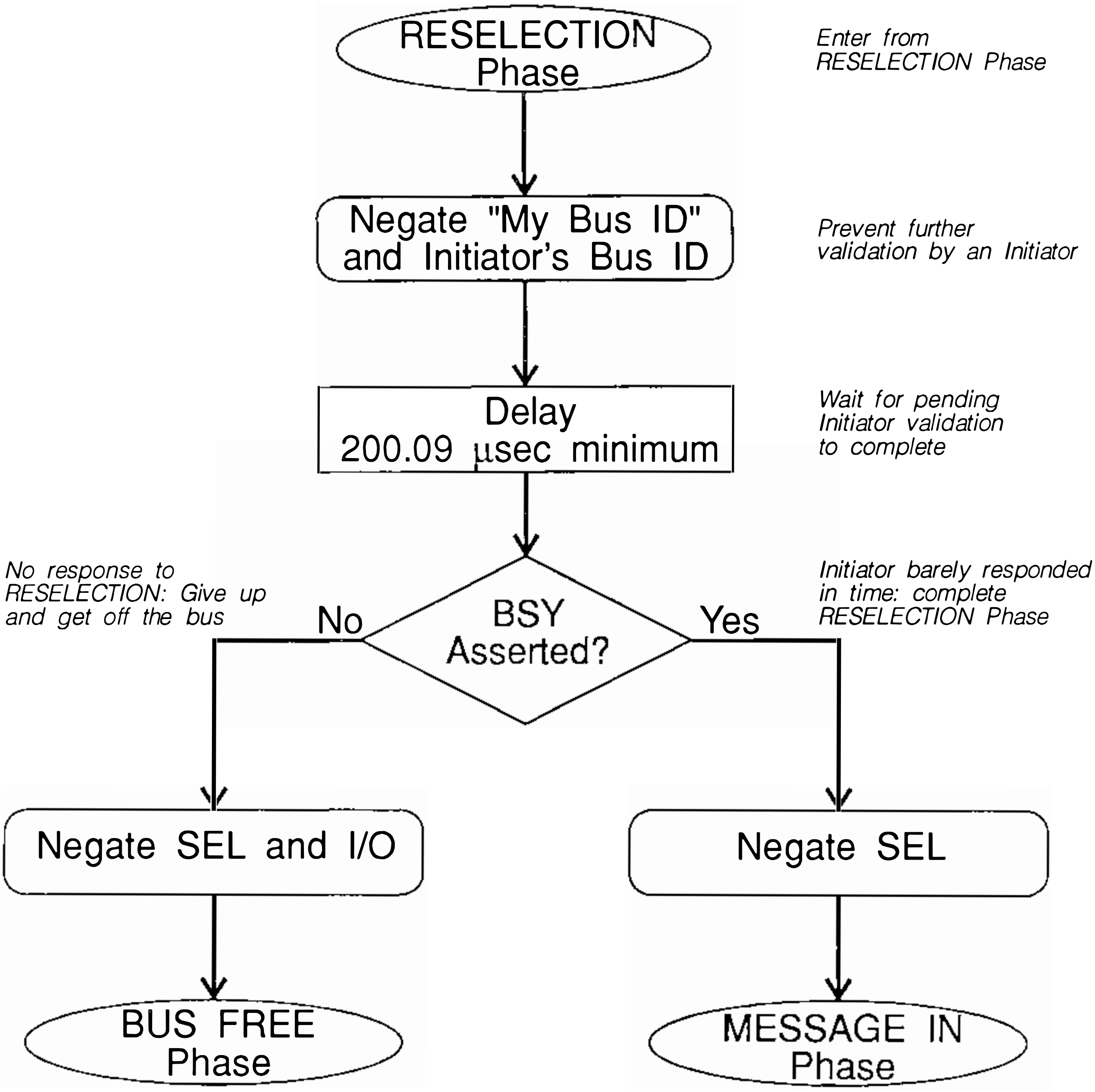

RESELECTION Phase
Enter from RESELECTION Phase
Negate "My Bus ID" and Initiator's Bus ID
Prevent further validation by an Initiator
Delay 200.09 µsec minimum
Wait for pending Initiator validation to complete
No response to RESELECTION: Give up and get off the bus
No
BSY Asserted?
Yes
Initiator barely responded in time: complete RESELECTION Phase
Negate SEL and I/O
Negate SEL
BUS FREE Phase
MESSAGE IN Phase

DIAGRAM 27: RESELECTION TIMEOUT FLOW DIAGRAM

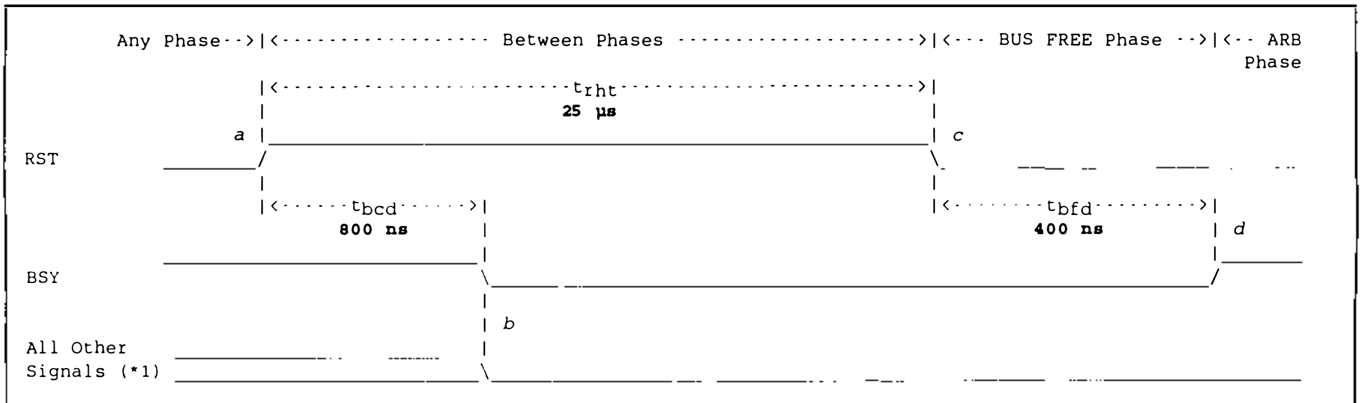
Reserved. In SCSI, "Reserved" refers to any bit, field, byte, or code in a ***Command Descriptor Block (CDB)***, ***Message***, or Parameter Data that serves no current purpose. The ***X3T9.2 Committee*** "reserves" these bits, fields, and bytes for future use. These uses usually include new functions or options for existing functions.

If the SCSI Standard says that a field (or bit or byte or code) is Reserved, believe it!

- For an Initiator sending a CDB or Parameter Data, or any ***SCSI Device*** sending a Message, a Reserved field must be set to zero. If you don't set them to zero, you could be invoking an option you don't know about, and then you wouldn't get the effect you wanted.

- For a Target returning Parameter, INQUIRY, or SENSE Data, setting a Reserved bit to one could cause the Initiator to misinterpret the data.

Reset Condition. The Reset Condition exists as long as the ***RST Signal*** is asserted by any ***SCSI Device***. The Reset Condition also exists for a minimum of a ***Reset Hold Time***. Figure 3 shows the Reset Condition timing.



(*1): Other signals = All Data Bus Signals, SEL, ATN, MSG, C/D, I/O, REQ, and ACK.

FIGURE 3: RESET CONDITION TIMING

Here's how events proceed during the Reset Condition:

(a) A SCSI Device asserts the RST signal for some reason (see below for some possibilities). This begins the Reset Condition on the bus. A ***Hard Reset*** or ***Soft Reset*** is also begun at this time.

(b) All of the other devices detect the RST signal and begin to clear off of the bus if they are currently ***Connected***. All devices must clear their signals off the bus within a ***Bus Clear Delay*** (800 nsec) of receiving the RST signal. At this time, the ***Bus Phase*** is always ***BUS FREE Phase***.

(c) After a Reset Hold Time from asserting the RST signal, the SCSI Device that started the Reset Condition finishes it by releasing the RST signal. This ends the Reset Condition.

(d) The soonest a device may assert the ***BSY Signal*** to begin an ***ARBITRATION Phase*** is 400 nsec (a ***Bus Free Delay***) after the release of RST.

Other observations on the Reset Condition:

- There is no validation time for ensuring that we really have a Reset Condition. According to the standard, a device is not supposed to react to glitches on the RST signal, but there is no other guidance given, other than that glitch rejection must take less than a Bus Clear Delay (800 nsec). Since you are likely at the mercy of the SCSI protocol chip manufacturer for glitch rejection, this is another good reason to use good ***Cables***.

- If for some reason two devices both decide to assert the RST signal, the Reset Condition ends when both devices have released RST. This is because the RST signal is an "OR-tied" signal. This may occur when two devices both recognize the need to reset at the same time. One example is when a Target fails and it is connected to the bus. Two Initiators may decide to "nail" the bus due to a system time-out or operator intervention.

It should be noted that creating the Reset Condition is a pretty drastic step. A SCSI Device should attempt all other options, including:

- The ***TERMINATE I/O PROCESS Message***, ***ABORT Message***, or the ***ABORT TAG Message***. These messages will just clear the command or commands involved (each in their own way, of course). This option is available only to a connected Initiator. Of course, if the Target in question is misbehaving and won't accept the message, you may not have any choice other than the Reset Condition.

- The ***BUS DEVICE RESET Message***. The advantage of this message is that it causes the Target to execute a "Power-On" type reset, and it doesn't directly affect any other device like the Reset Condition can. Of course, this option has the same disadvantage as the first one.

- Beat on the offending device with a hammer. Actually, if the device has a user accessible reset button this may be less traumatic to a system than the Reset Condition....

The Reset Condition also causes all SCSI Devices on the bus (including Devices that are usually Initiators!) to execute a reset procedure; either a Soft Reset or a Hard Reset.

Table 13 shows the timing values used during the Reset Condition.

TABLE 13: TIMING VALUES USED DURING THE RESET CONDITION

Symbol	Timing Name	MIN or MAX?	Time
t_{bfd}	Bus Free Delay	MINIMUM	400 nsec
t_{bcd}	Bus Clear Delay	MAXIMUM	800 nsec
t_{rht}	Reset Hold Time	MINIMUM	25 µsec

Reset Hold Time. *t_{rht} = 25 µsec.* If a device asserts the ***RST Signal***, it must assert RST for a minimum of a Reset Hold Time. See ***Reset Condition***. There is really no good reason why this time is so long. It just is. Some folks on the original ***X3T9.2 Committee*** wanted to be sure everything got reset, even if the device had a really slow clock!

Reset to Selection Time. *t_{rst} =250 msec recommended.* After the ***Reset Condition*** is over (the ***RST Signal*** is negated after it is asserted), all SCSI Targets must be able to respond to a ***SELECTION Phase*** no later than this period of time. Further, it must also be able to execute the following commands:

- TEST UNIT READY
- REQUEST SENSE
- INQUIRY

Now, as a practical matter, this number is fairly meaningless except as a way to encourage Targets to be ready to respond to an Initiator in a reasonably short period of time. Why?

- This is a "recommended" time. As such, a procurement spec may specify a different number, and your Target may be required to meet a looser or tighter number.

- Once the Target responds to SELECTION Phase, it must then be able to execute the commands listed above. But, what does execute mean?

- TEST UNIT READY: Pop back to the Initiator with a ***Status*** code. That's easy. But, the Target may not yet have loaded its microcode (say, from reserved sectors on a disk), and may not be able to respond with the Sense Data that best describes the Status.

- REQUEST SENSE: Return something. Again, the data may not meaningful because of delays loading microcode.

- INQUIRY: Same as REQUEST SENSE. The data may be a minimal set saying: "I am this Device Type and that's all I know right now".

As it turns out, most of the time all the Initiator really wants to know is whether a Target exists at that ***SCSI Address***. It is then usually okay if the Target takes a little while longer to gather more complete information that can be provided at a later time.

RESTORE POINTERS Message. A RESTORE POINTERS Message is sent from a Target to an Initiator to retry an ***Information Transfer Phase***. RESTORE POINTERS is a single byte message:

Bit Byte	7	6	5	4	3	2	1	0
0	Message Code = 03 hex							

Specifically, this message is used to retry a ***COMMAND Phase***, ***STATUS Phase***, or a ***DATA Phase***. The Target sends this message when it gets an indication that one of these phases must be retried. This indication can include one of the following:

- Bus ***Parity*** error detected by the Target.

- Internal error within the Target; e.g., a buffer memory failure.

- An ***INITIATOR DETECTED ERROR Message*** received from the Initiator.

When the Initiator receives this message, it copies all ***Saved Pointers*** to the ***Active Pointers***. In most cases, this effectively goes back to the beginning of the transfer; an exception is when the ***SAVE DATA POINTER Message*** or ***MODIFY DATA POINTER Message*** is used. See ***Pointers*** and ***Path Control*** for all the details.

Summary of Use: The RESTORE POINTERS is sent only by a Target to request that the Initiator copy its Saved Pointers to its Active Pointers for purposes of retrying an Information Transfer Phase.

RST Signal. The RST Signal is asserted by any ***SCSI Device*** to cause a ***Reset Condition*** on the bus. The RST signal must never be ***Negated***; it should only be ***Released*** since this is an "OR-tied" signal. If the RST Signal is to be asserted by a device, it must be asserted for a minimum of a ***Reset Hold Time***, which is 25 µsec. Like the ***BSY Signal***, the RST Signal is "OR-tied"; no device may "negate" the RST signal. It may only be "asserted" or "released". This allows more than one SCSI Device to assert the signal. See ***Signal Levels***.

SASI.
SAVE DATA POINTER Message.
Saved Pointers.
SCSI, SCSI-1, and SCSI-2.
SCSI-3?!
SCSI Address.
SCSI Bus.
SCSI Bus ID.
SCSI Commands.
SCSI Device.
SCSI ID.
Selection Abort Time.
SELECTION Phase.
Selection Time-out.
Selection Time-out Delay.
SEL Signal.
Signal Levels.
SIMPLE QUEUE TAG Message.
Single-Ended Interface.
Soft Reset.
Status.
STATUS Phase.
Status Pointer.
Synchronous Data Transfer.
Synchronous Data Transfer Negotiation.
SYNCHRONOUS DATA TRANSFER REQUEST Message.
Synchronous Offset.
Synchronous Transfer Period.

SASI. The "Shugart Associates System Interface" (or SASI), is the direct predecessor to ***SCSI***. The name was changed to SCSI because ANSI rules prohibited the use of a company name in the name of an ANSI standard. SASI defined the basic ***Bus Phases***, ***Control Signals***, and the 8-bit ***Data Bus Signals***, along with a primitive ***Message System*** and a minimal set of disk-oriented commands.

SAVE DATA POINTER Message. The SAVE DATA POINTER is used to establish a new starting point for any later ***DATA Phase*** retries. This is a single byte message:

Bit Byte	7	6	5	4	3	2	1	0
0	Message Code = 02 hex							

SAVE DATA POINTER is sent by a Target to request that the Initiator copy its ***Current Data Pointer*** to the ***Saved Data Pointer***. This message is usually seen in one of the following situations:

- Prior to a ***Disconnect***. In this case, the Target is preparing to disconnect from the bus for a time and wishes to continue the data transfer from where it left off.

- In the middle of a long data transfer. In this case, the Target has transferred a **lot** of data and wants to establish a new place to begin retries. The simplest motivation for doing this is to avoid a very lengthy retry in case of a transfer error (such as a ***Parity*** error). Another reason, however, may be that the Target is incapable of re-sending the data transferred prior to this point. An example of this might be a SCSI communication bridge to a network.

- At the end of a data transfer, if the Target wants to Disconnect before sending ***Status***. In this case, the Target is ensuring that the Saved Data Pointer matches the Current Data Pointer. Some Initiators have been designed to require this, but this author does not recommend this for future Initiator designs (see ***Etiquette***).

See ***Pointers*** and ***Path Control*** for more details on this message.

Summary of Use: The SAVE DATA POINTER message is sent only by a Target to request that the Initiator copy its Current Data Pointer to its Saved Data Pointer.

Saved Pointers. The Saved Pointers (as opposed to the ***Active Pointers***) are three pointers that refer to the starting command, status, and data locations within the Initiator for an ***I/O Process***. A Saved Pointer indicates the starting byte of each of those locations. There are three Saved Pointers:

(1) Saved Command Pointer
(2) Saved Status Pointer
(3) Saved Data Pointer

Technically, the Saved Data Pointer may not point at the start of the Initiator data location. The Saved Data Pointer may be modified by the Target by the use of the ***SAVE DATA POINTER Message***. Note that the ***MODIFY DATA POINTER Message*** does not modify the Saved Data Pointer; it only modifies the Active Data Pointer.

See ***Pointers*** for the whole story.

SCSI, SCSI-1, and SCSI-2. Yes, you've heard these terms, and you may have noticed that no distinction is made between them in this volume. There is a good reason; this volume is targeted (SCSI pun not intended) at designers and users setting out to learn and initiate (SCSI pun intended) SCSI designs today. SCSI today is SCSI-2. However, we do recognize that the reader may have to know what the difference between SCSI-1 and SCSI-2 is, thus we include this topic.

SCSI: Anything that conforms or refers to the current SCSI standard; as of this writing, SCSI-2.

SCSI-1: Anything that conforms or refers to ANSI Standard X3.131-1986.

SCSI-2: Anything that conforms or refers to ANSI Standard X3.131-199x.

Oh, and one other item:

CCS: Anything that conforms to the working document X3T9.2/85-52 revision 4B, the "Common Command Set". This internally inconsistent document formed a basis for SCSI-2. Modern SCSI products don't design to CCS, they design to SCSI-2. Just say "No" to CCS!

New features, within the scope of this volume, that were added to SCSI-2 include:

- High density connectors
- Fast data transfers
- Active termination
- ARBITRATION and IDENTIFY Message are required
- Wide data transfers
- Command Queuing and Queue Tags
- Extended Contingent Allegiance
- TERMINATE I/O PROCESS Message
- DISCONNECT Message sent by the Initiator

SCSI-3?! No, we will NOT use the old "...just when you thought it was safe to..." line! However, it <u>does</u> seem very appropriate here. It seems that the X3T9.2 committee, whose job it is to maintain the SCSI Standard, just can't keep their hands off....

Enough editorializing. The features currently planned for SCSI-3 as of this writing include:

- Single cable 16-bit wide transfers (already a de facto standard).
- The Standard will be broken into several pieces to simplify the documentation of new features.
- Packetized transfers over fiber optic and copper interfaces.
- Dual or multiple port operation.
- Additional Caching control features.

SCSI Address. See also ***SCSI Bus ID***. The SCSI address is the numerical representation of the bus address assigned to a ***SCSI Device***. Since up to eight devices can be connected to the bus, the possible addresses are 0 thru 7.

Each device has a unique address. If two devices on a single bus cable have the same address, it is likely they will contend for the bus and, at best, hang the bus. At worst, data could be lost. Be careful with your address settings!

Since the address must be unique, SCSI devices are designed so that the address may be changed at installation time. This is usually a DIP switch, jumper, or other mechanical connection. There is no standard method for changing the address over the bus, but you will see some devices that have contrived a non-standard method.

The address also determines the priority of the device during ***ARBITRATION Phase***. The higher the value of the address, the more likely the device is to win ARBITRATION. Therefore, in certain situations the addresses of your SCSI devices must be chosen with some care. For example, if the devices use the bus for long periods of time, then there is less free time on the bus for others to use it. This will force more competition during ARBITRATION Phase.

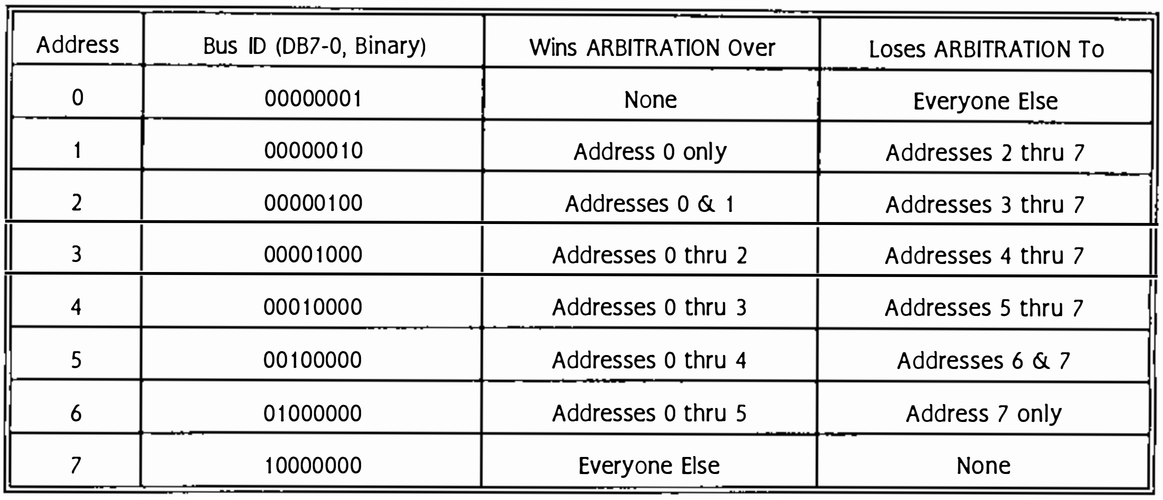
SCSI Bus. The SCSI Bus is a generic term that refers to the complete set of signals that define the activity of the interface. These signals include the ***Control Signals*** and the ***Data Bus Signals***. That's about all that can be said here without writing the rest of this book....

SCSI Bus ID. The Bus ID is the ***SCSI Address*** expressed as a single bit on the ***Data Bus***. The Bus ID is used by a ***SCSI Device*** to:

- Broadcast its address during ***ARBITRATION Phase***. The device that is broadcasting the largest address (e.g., 7 is greater than 6) wins.
- Transmit the address of the device to be selected (or reselected) during ***SELECTION Phase*** (or ***RESELECTION Phase***). The device being selected (or reselected) sees its Bus ID and responds.
- Transmit its address during SELECTION Phase so that the selected device knows who is selecting it. The selected device (the Target) uses this information to ***Reconnect*** later, and to maintain many different Initiator-related types of information and operating conditions.
- Transmit its address during RESELECTION Phase so that the reselected device knows who is reselecting it. The reselected device (the Initiator) uses this information to re-establish the ***Nexus***.

Table 16 shows the SCSI Address, the corresponding Bus ID, and how that address fares during ARBITRATION Phase:

TABLE 16: SCSI BUS ADDRESSES AND IDS



Address	Bus ID (DB7-0, Binary)	Wins ARBITRATION Over	Loses ARBITRATION To
0	00000001	None	Everyone Else
1	00000010	Address 0 only	Addresses 2 thru 7
2	00000100	Addresses 0 & 1	Addresses 3 thru 7
3	00001000	Addresses 0 thru 2	Addresses 4 thru 7
4	00010000	Addresses 0 thru 3	Addresses 5 thru 7
5	00100000	Addresses 0 thru 4	Addresses 6 & 7
6	01000000	Addresses 0 thru 5	Address 7 only
7	10000000	Everyone Else	None

SCSI Commands. A SCSI Command is an operation performed by a Target for an Initiator. A SCSI command is fully specified by the following items:

- The ***Command Descriptor Block*** (CDB) which specifies the function to be performed.
- The ***Logical Unit Number (LUN)*** which specifies the ***Logical Unit*** for the commanded function. This is usually specified in the ***IDENTIFY Message***. In older SCSI-1 devices, the LUN is specified in the second byte of the CDB.
- The optional ***Queue Tag*** which further defines the ***Nexus***.
- If the command function requires that a ***DATA Phase*** must occur, that data may contain any or all of the following:
    - Additional Command Parameters to or from the Initiator.
    - Data to or from the Logical Unit.
- A SCSI command may also be influenced by parameters set up by previous commands such as ***MODE SELECT***.

SCSI Device. A SCSI Device is anything that can plug into the SCSI Bus and actively participate in bus activity. The term "SCSI Device" is usually used as an inclusive term for "Initiator or Target".

SCSI ID. See ***SCSI Bus ID***.

Selection Abort Time. *tsat = 200 µsec.* The Selection Abort Time has been defined to prevent bus hangs during ***SELECTION Phase*** or ***RESELECTION Phase***. This time is defined as the maximum amount of time that a device may take between detecting that it is being selected, and when it responds by asserting the ***BSY Signal***. See ***Selection Timeout*** and ***Reselection Timeout*** for how this timing value is used.

SELECTION Phase. The SELECTION Phase is used by an Initiator to choose the Target and begin an ***I/O Process***. The SELECTION Phase is always preceded by the ***ARBITRATION Phase***, and is always followed by:

- A ***MESSAGE OUT Phase*** to send the ***IDENTIFY Message***, and possibly a ***Queue Tag Message***. This is preferred.
- A ***BUS FREE Phase***, because of a ***Selection Timeout***.
- A ***COMMAND Phase*** if the Target is an older SCSI device. Modern SCSI Targets do not transition directly from SELECTION Phase to COMMAND Phase, and modern SCSI Initiators do not make Targets change directly by neglecting the use of the ***Attention Condition***.

Diagram 28 and Diagram 29 show how Initiators and Targets handle SELECTION Phase. During SELECTION Phase, the Initiator sets the bus signals that define the phase, and then waits for a response from the Target. After the Target response, the Initiator releases those signals and waits for the Target to begin sequencing through the ***Information Transfer Phases***, starting with the MESSAGE OUT Phase.

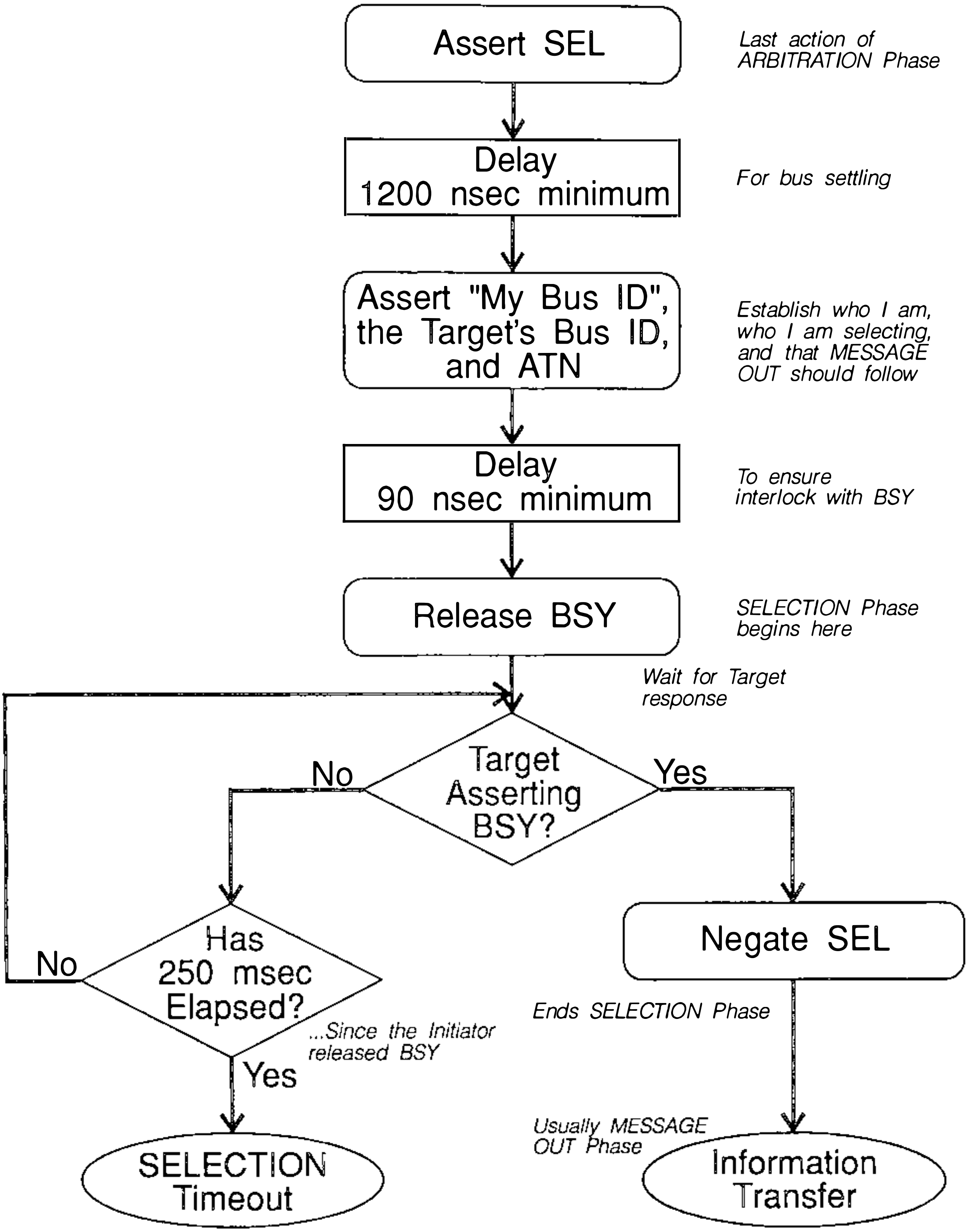

Assert SEL
Last action of ARBITRATION Phase
Delay 1200 nsec minimum
For bus settling
Assert "My Bus ID", the Target's Bus ID, and ATN
Establish who I am, who I am selecting, and that MESSAGE OUT should follow
Delay 90 nsec minimum
To ensure interlock with BSY
Release BSY
SELECTION Phase begins here
Wait for Target response
Target Asserting BSY?
No
Yes
Has 250 msec Elapsed?
...Since the Initiator released BSY
No
Yes
Negate SEL
Ends SELECTION Phase
Usually MESSAGE OUT Phase
SELECTION Timeout
Information Transfer

DIAGRAM 28: SELECTION FLOW DIAGRAM FOR INITIATORS

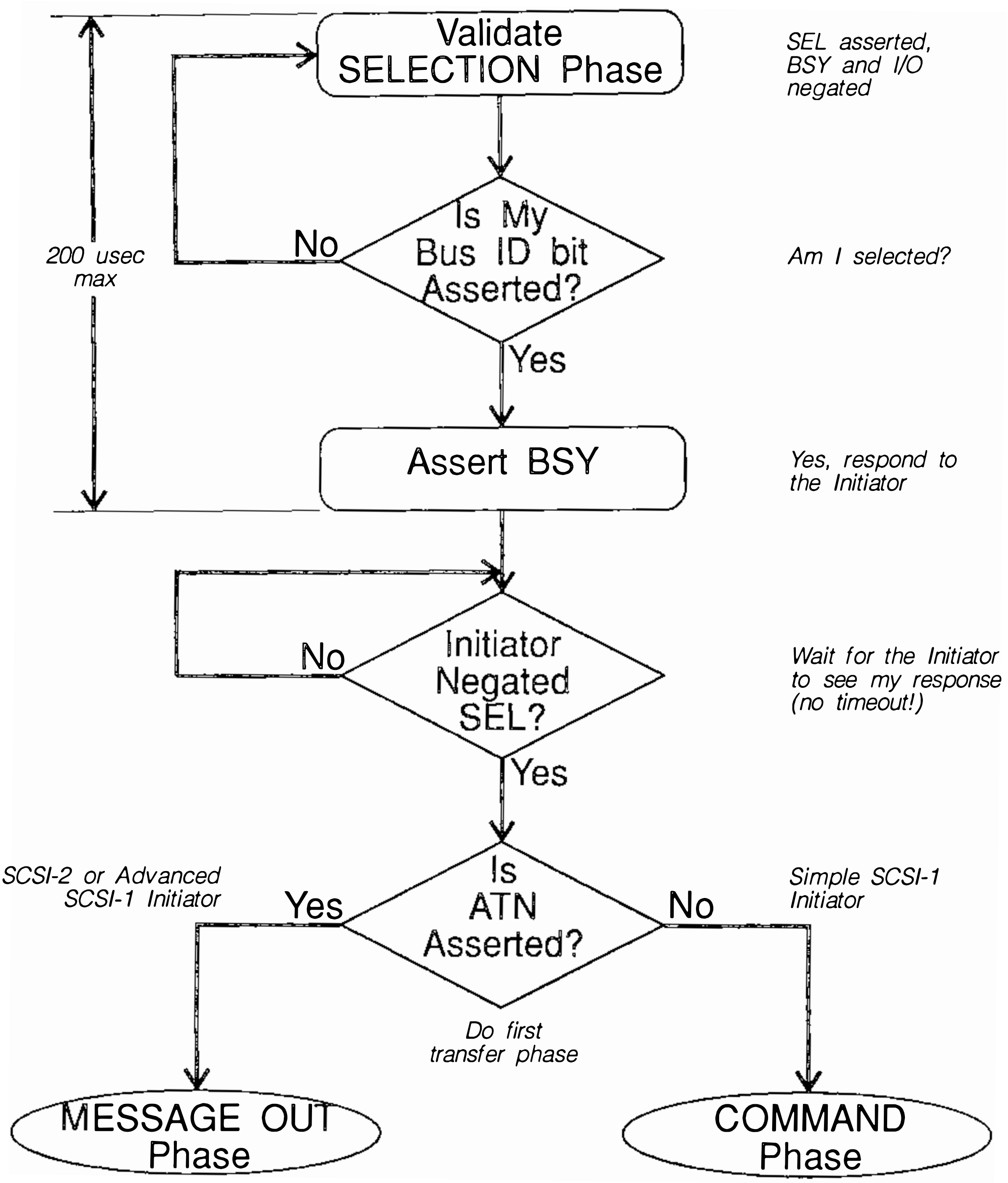
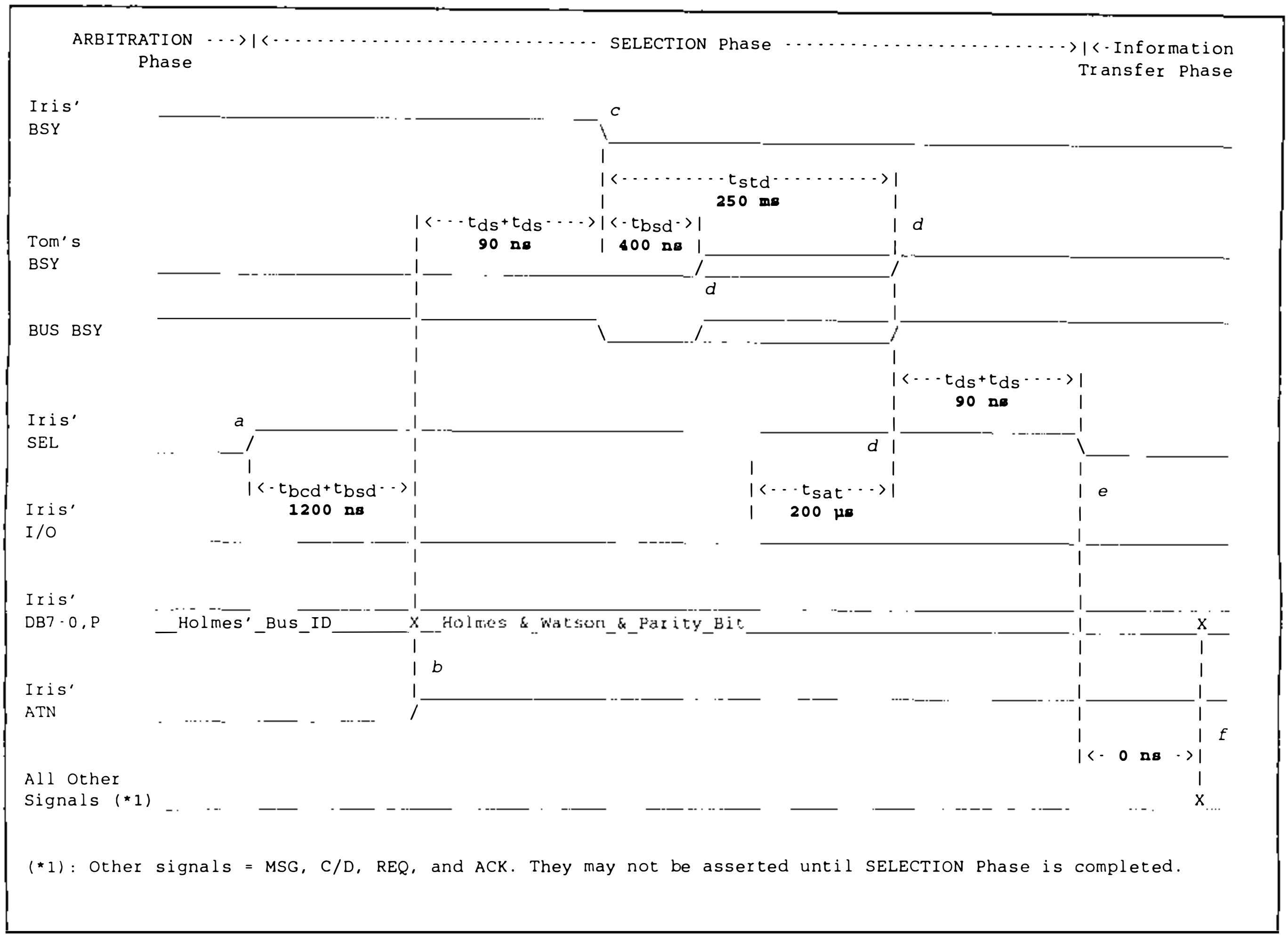


DIAGRAM 29: SELECTION FLOW DIAGRAM FOR TARGETS

Figure 4 shows the exact timing of SELECTION Phase.



(*1): Other signals = MSG, C/D, REQ, and ACK. They may not be asserted until SELECTION Phase is completed.

FIGURE 4: SELECTION PHASE TIMING

Here's how events proceed in SELECTION Phase:

(a) Iris has successfully acquired the bus during the preceding ***ARBITRATION Phase*** by asserting the ***SEL Signal*** to take the bus (Iris is often successful at what she does).

(b) After a ***Bus Clear Delay*** of 800 nsec to allow the losers to release their bus signals, and after an additional ***Bus Settle Delay*** of 400 nsec to allow the bus to settle (imagine that?) from the signal releases (total delay 1200 nsec), Iris begins setting up for SELECTION Phase. She asserts her ***SCSI Bus ID*** (which is 7) and Tom's SCSI Bus ID (which is 0) on the ***DATA BUS Signals***; Tom is the Target that Iris will be selecting. The data bus value is then **10000001** with the ***Parity*** bit also asserted. Iris also asserts the ***ATN Signal*** to indicate that a ***MESSAGE OUT Phase*** should follow the SELECTION Phase. The ***I/O Signal*** must be false during SELECTION Phase; Iris must leave it released.

(c) At least two ***Deskew Delays*** (total 90 nsec) after asserting the data bus and the ATN signal, Iris releases the ***BSY Signal*** to begin the selection.

(d) Other devices may now examine the bus to determine who the lucky Target to be selected is. The other devices must delay a minimum of 400 nsec (Bus Settle Delay again) before examining the data bus. Tom, the "One Who is Selected", examines the bus and sees that he is selected. Iris has certain expectations about how Tom will respond:

- Tom should respond within 250 msec (***Selection Timeout Delay***) from the start of the SELECTION Phase (Iris released BSY in step (c)). In other words, a SCSI device should look for a SELECTION Phase at least every 250 msec if interrupts are not used or are disabled.

- Tom must respond within 200 µsec (***Selection Abort Time***) of noticing that he is selected. This is done to facilitate a smooth ***Selection Timeout***.

Tom responds to the selection by Iris by asserting the BSY signal.

(e) Iris sees that Tom has asserted BSY in response to the selection. After delaying 90 nsec (two Deskew Delays) to ensure signal interlock, Iris may release the SEL signal.

(f) As soon as Tom sees the SEL signal go false (no delay), he may begin setting up for the MESSAGE OUT Phase by changing the ***Bus Phase Signals***.

Other observations on SELECTION Phase:

- Note that ***Parity*** is valid during the SELECTION Phase. Since two data bits are always asserted true during SELECTION, the Parity bit (DB(P)) is always asserted as well, since SCSI Parity is odd (that is, as opposed to even....).

- The Selection Timeout Delay exists to provide an upper limit to the time a Target takes to respond. If the Target hasn't responded within that time, the Initiator may reasonably assume that no Target exists at that ***SCSI Address***. In practice, the timeout only becomes a factor during ***Initialization***, which is when the Initiator is trying to determine the devices that are available. Most Targets today will respond within a very short period of time. A Target that fails to respond is usually non-existent, powered-down, or broken.

Table 17 shows the timing values used during SELECTION Phase.

TABLE 17: TIMING VALUES USED DURING SELECTION PHASE

Symbol	Timing Name	MIN or MAX?	Time
t_{bsd}	Bus Settle Delay	MINIMUM	400 nsec
t_{bcd}	Bus Clear Delay	MINIMUM	800 nsec
t_{std}	Selection Timeout Delay	MINIMUM	250 msec
t_{ds}	Deskew Delay	MINIMUM	45 nsec
t_{sat}	Selection Abort Time	MAXIMUM	200 µsec

This page is nearly blank!
We use the space to improve Readability.

Selection Time-out. The Selection Time-out is a formal procedure to prevent bus hangs when an Initiator decides to give up on a pending ***SELECTION Phase***. How does this prevent bus hangs? Let's see what happens without a formal time-out procedure:

(1) A device (call him "Tom"; deja vu....) enters SELECTION phase (intending to become an Initiator), attempting to select another device (call her "Iris") as a Target.

(2) Iris sees Tom trying to select her, but she's too busy at the moment to respond (something about keeping a tape streaming), so she ignores Tom for the moment.

(3) Tom gets tired of waiting (probably after a ***Selection Time-out Delay***, but before all night....) and gives up, going directly to ***BUS FREE Phase***; a bad thing to do, as we will see.

(4) Just as Tom gives up, Iris decides she has time for him now, so without checking the bus, she asserts BSY to respond to his selection.

(5) Poor Iris. She is connected to the bus, but she has no one to talk to. And Poor Tom. He gave up and isn't there to acknowledge her requests.... As a result the bus is hung with no recourse but a ***Reset Condition***.

As you can see, there has to be some way of giving up gracefully. In the above example, even if Iris tried to respond right away, but Tom gave up the selection just after Iris examined the bus, a hang would still occur. To solve this problem, the following procedure has been defined:

(1) After Iris (the Initiator) has maintained SELECTION Phase for a Selection Time-out Delay, she decides to abort the selection. The first thing Iris does is release all ***Data Bus Signals***.

(2) Iris waits for at least a ***Selection Abort Delay*** plus two ***Deskew Delays*** (200.09 μsec total), and if Tom (the Target) has not yet responded, releases the ***SEL Signal*** and the ***ATN Signal***, returning to the BUS FREE Phase.

Let's see how this prevents a bus hang. If Tom does not check the bus for a Selection until after step (1) above, then he will not see his ***SCSI Bus ID*** asserted on the Data Bus. If Tom detects selection just before Iris decides to give up, Iris will still catch Tom's response just in time.

Diagram 30 shows the Selection Time-out Procedure.

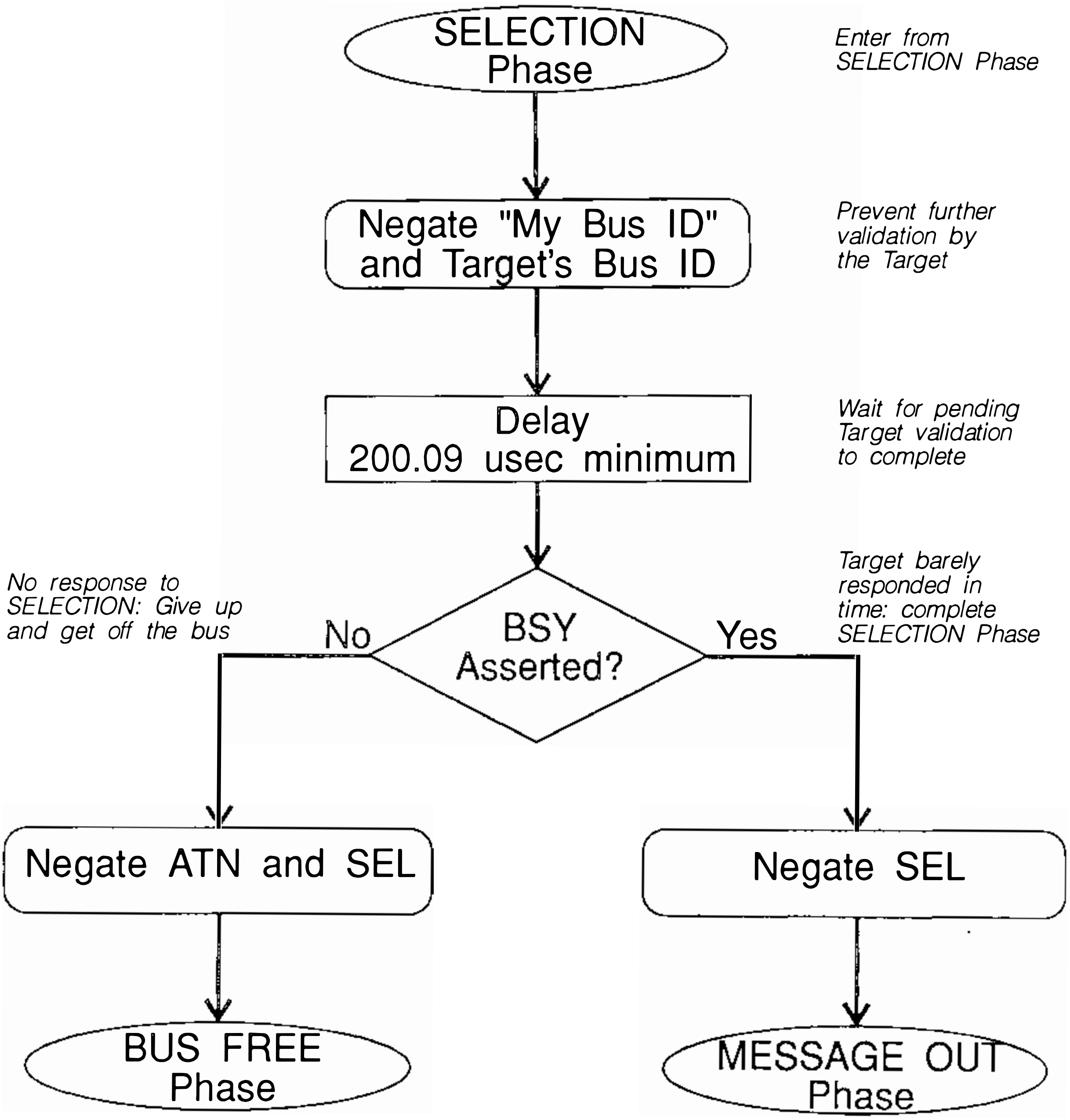

SELECTION Phase
Enter from SELECTION Phase
Negate "My Bus ID" and Target's Bus ID
Prevent further validation by the Target
Delay 200.09 usec minimum
Wait for pending Target validation to complete
No response to SELECTION: Give up and get off the bus
No
BSY Asserted?
Yes
Target barely responded in time: complete SELECTION Phase
Negate ATN and SEL
Negate SEL
BUS FREE Phase
MESSAGE OUT Phase

DIAGRAM 30: SELECTION TIMEOUT PROCEDURE

Selection Time-out Delay. *tstd - 250 msec recommended.* The Selection Time-out Delay establishes an upper limit on how long an Initiator should wait before deciding to give up on selecting a Target. If the Target hasn't responded after this delay, the Initiator begins a ***Selection Time-out*** procedure. This delay also applies to Targets trying to reselect an Initiator; in the case of a time-out, the Target would begin a ***Reselection Time-out*** procedure.

Note that this time is "recommended": The SCSI committee shied away from defining a firm time-out delay. It may be good that they didn't, because new SCSI chips are capable of automatically responding to ***SELECTION Phase*** immediately. It is very uncommon today to see devices that take longer than a millisecond to respond. If your application requires a faster ***Initialization*** time, then you may be able to use a shorter time-out delay.

SEL Signal. The SEL signal is used to indicate a ***SELECTION Phase*** or a ***RESELECTION Phase***, or an impending transition to one of those phases:

- A ***SCSI Device*** asserts SEL at the end of ***ARBITRATION Phase*** when it has won arbitration and wishes to take the bus.

- The SEL signal asserted with the ***I/O Signal*** negated or released (and the ***BSY Signal*** released) indicates a SELECTION Phase.

- The SEL signal asserted with the ***I/O Signal*** asserted (and the ***BSY Signal*** released) indicates a RESELECTION Phase.

One other note; the SEL signal is never driven to the false state. It is always released. See ***Signal Levels***.

Signal Levels. The signal levels on the bus determine the state of the signal; either true or false. A signal that is true is ***Asserted***. A signal that is false is either actively driven to the ***Negated*** level, or passively allowed to return to the false state by ***Releasing*** the signal. The following paragraphs describe these levels; they are also illustrated under ***Single-Ended Interface*** and ***Differential Interface***.

The signal levels for the ***Single-Ended Interface*** are:

- A signal is asserted when it is driven low on the cable. "Low" is defined as driving to no more than 0.5 Volts at 48 mA sinking. A receiving device sees an asserted signal when the input is lower than 0.8 Volts.

- A signal is negated when it is driven high on the cable. "High" is defined as driving to at least 2.5 Volts. A receiving device sees a negated signal when the input is higher than 2.0 Volts.

- A signal is released when it is allowed to float high on the cable. "Float High" is defined as the voltage to which the terminator holds the bus when no active device is driving. A receiving device sees a negated signal.

The signal levels for the ***Differential Interface*** are:

- A signal is asserted when its "+" signal is driven higher than the "−" signal is driven low. The driver drives the "+" signal at least 1.0 Volts higher than the "−" signal. A receiving device sees an asserted signal when the "+" input voltage is higher than the "−" input voltage.

- A signal is negated when its "+" signal is driven lower than the "−" signal is driven low. The driver drives the "−" signal at least 1.0 Volts higher than the "+" signal. A receiving device sees a negated signal when the "−" input voltage is higher than the "+" input voltage.

- A signal is released when its "−" signal is allowed to float higher than the "+" signal is allowed to float. "Float High" is defined as the voltage to which the terminator holds the bus when no active device is driving. A receiving device sees a negated signal.

SIMPLE QUEUE TAG Message. The SIMPLE QUEUE TAG Message is used by an Initiator to get a command executed in an optimum order decided by the Target when a ***Queue*** is used by the Target. The Target may also send this Message during a ***Reconnection***. SIMPLE QUEUE TAG is a two byte message:

Bit Byte	7	6	5	4	3	2	1	0
0	Message Code = 20 hex							
1	Queue Tag							

The second byte of the message specifies the ***Queue Tag*** associated with the ***Nexus*** being established by this message.

This message causes the new ***I/O Process*** associated with the Nexus to be put in the Queue, and allows the Target to reorder it relative to other commands that currently may be queued. If no I/O Process is currently being executed for that ***Logical Unit***, then the new I/O Process is executed immediately.

An I/O Process with a SIMPLE QUEUE TAG may be placed anywhere in the Queue, subject to the following rules:

- If there are no I/O Processes with ***ORDERED QUEUE TAGS*** or ***HEAD OF QUEUE TAGS*** in the Queue, the Target may place the new I/O Process anywhere in the Queue.

- If there are I/O Processes with ORDERED QUEUE TAGS or HEAD OF QUEUE TAGS in the Queue, the Target may place the new I/O Process in the Queue between the rear of the Queue and the ORDERED QUEUE TAG or HEAD OF QUEUE TAG I/O Process closest to the rear of the queue received by the Target.

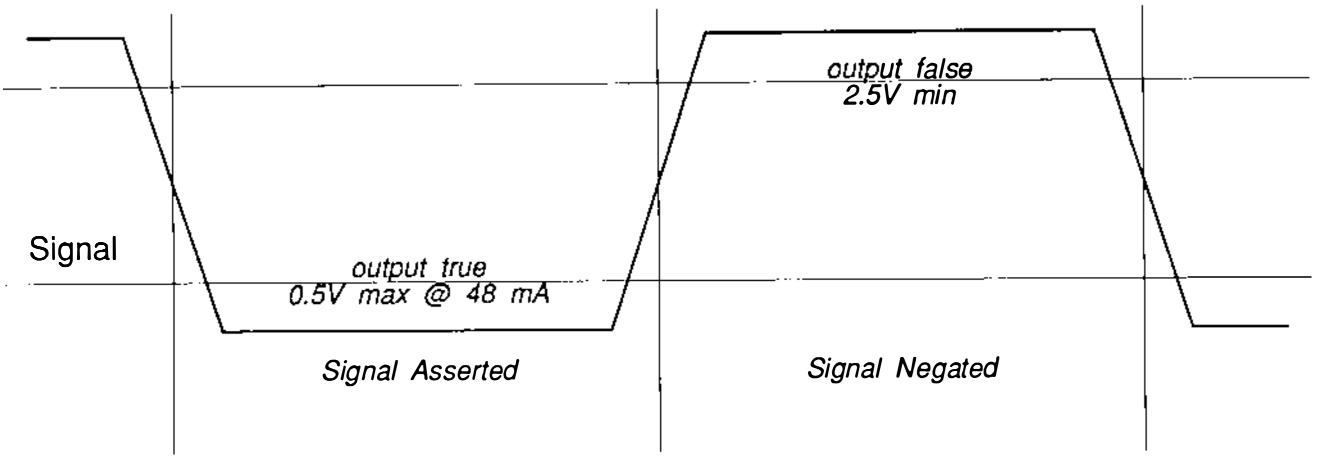
There are other rules for reordering I/O Processes. See ***Queue*** for an illustration of this behavior.

Summary of Use: The SIMPLE QUEUE TAG Message is sent by an Initiator to a Target to cause the I/O Process to be placed in the queue. The SIMPLE QUEUE TAG Message is sent by a Target to an Initiator to re-establish a Nexus during a Reconnect.

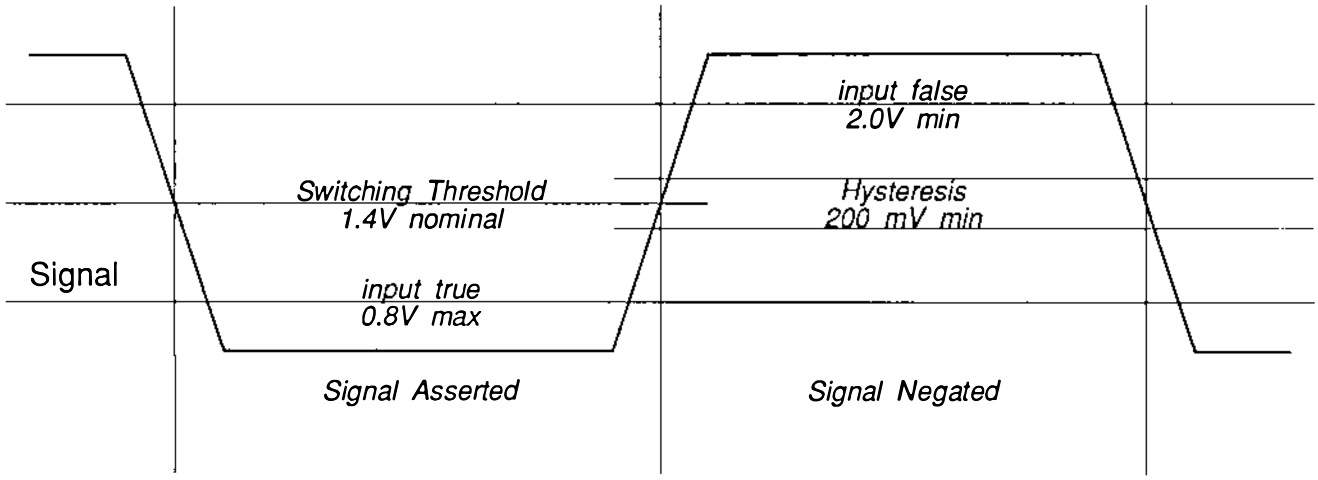
Single-Ended Interface. The Single-Ended Interface is one of two electrical interfaces (see ***Differential Interface***) defined for SCSI. The electrical interfaces are the means by which the ***Data Bus Signals*** and ***Control Signals*** are transmitted and received by ***SCSI Devices***.

The Single-Ended Interface transmits the signals using a single wire with a ground return. The driver pulls the wire to ground to ***Assert*** the signal, and either goes to a high impedance state or pulls the wire high to ***Negate*** the signal. In any case, the driver must be able to go to a high impedance state when the device is not enabled on the bus. The receiver detects the voltage and decides the state of the signal. The top half of Diagram 31 shows the operating characteristics of the driver, and the bottom half shows the operating characteristics of the receiver.

Note that the "Signal Negated" portion of the Driver Requirements is only meaningful for SCSI Devices with ***Active Pull-ups***. Devices with "open-collector" drivers can only ***Release*** a signal.



Driver Requirements



Receiver Requirements

DIAGRAM 31: SINGLE-ENDED DRIVER AND RECEIVER VOLTAGES

Other characteristics of the Single-Ended Interface are:

- The current at any signal input must not exceed:
    - -0.4 mA when the input voltage is 0.5 V.
    - 0.1 mA when the input voltage is 2.7 V.

  These must also hold when the device is powered down.

- The capacitance on any signal input must not exceed 25 pF. Note that this is measured at the connector and includes:
    - Connector capacitance;
    - Printed circuit trace capacitance;
    - SCSI protocol ***Chip*** capacitance.

- The ***Termination*** biases each signal when ***Released*** between 2.85 V and 2.7 V when using the "Preferred" Circuit. When using the "Old Style" Circuit, the bias is between 3.5 V and 2.1 V. The variation in bias voltage is due to resistor tolerances at the Terminator, and, in the case of the "Old-Style" Circuit, is also due to variation in the ***Terminator Voltage***.

The advantages of the Single-Ended Interface are:

- The Single-Ended Interface uses less power than the Differential Interface. Single-Ended drivers have less current to sink than Differential drivers, and there are half as many lines to drive.

- The Single-Ended Interface dissipates less power than the Differential Interface. An 8-bit Differential driver set will dissipate up to about 4 watts, while an 8-bit Single-Ended driver set dissipates less than 1 watt.

- As of this writing, the Single-Ended Interface is several dollars cheaper to implement than a Differential Interface.

The disadvantages of the Single-Ended Interface are:

- Cable length (6 meters) is actually pretty short. By the time the cable is routed through a decent sized computer cabinet, there is little extra length available to go outside the box. You should use the Differential Interface when the cable must leave a cabinet for a total cable length longer than 6 meters.

- Signal quality is "poorer" than with the Differential Interface. This must be qualified: If you use cheap flat ribbon cable, route it poorly within the cabinet, exceed stub lengths, and use passive termination with low terminator power voltage, you can expect to have signal quality problems. If you use good quality cable, route with care, use active termination, and generally observe the requirements and recommendations of the standard, you should have no trouble. In other words, the Single-Ended Interface is more susceptible to carelessness in following the standard.

Soft Reset. "Gently return to your initial state." Not quite. Soft Reset is one of two ways that a ***SCSI Device*** can respond to the ***Reset Condition*** (the other way is, of course, ***Hard Reset***). The problem with Hard Reset is that everything on the Device gets cleared. This includes Devices that were minding their own business off the bus.

That seems kind of unfair. Why should a behaving Device get nailed? The Reset Condition is usually used to get a failed Device off the bus, usually when the Device is refusing to respond to the ***Attention Condition***. This could be bad, particularly if the Device gets nailed in the middle of a sector write.

Soft Reset attempts to address that problem. A Device that has implemented Soft Reset does not get nailed by the Reset Condition. In fact, the first rule is easy:

- If the Device that has implemented Soft Reset is not currently ***Connected*** to the bus, it ignores the Reset Condition.

If the Device is Connected, it gets a little stickier, but not too bad. What? You've read the standard? It's ugly? Well, I'm here to tell you to forget what you read, because there is a little loophole. It turns out that if the Target can't determine where exactly it was interrupted during the Connection, it can abort the ***I/O Process*** and return SENSE Data (if asked) that says the I/O Process was Aborted. The Target can choose when to continue or abort the I/O Process based on the capabilities of your SCSI Protocol ***Chip***.

Nonetheless, we will comment on each of the nine requirements/conditions for implementing Soft Reset (the numbers refer to the numbers in the ***SCSI-2*** Standard under Soft Reset):

(1) An Initiator cannot be sure the Target really has safely stored the information required for the ***Nexus*** to be established unless the Target has been able to change to another ***Bus Phase***. Until then, the Target could still be examining the contents of the ***Messages***.

(2) If the Target knows what it's got, then it knows the Nexus, and can consider the I/O Process fully identified.

(3) In this case, the Initiator believed that the Target had discarded the I/O Process in question. Or, the Initiator implemented Hard Reset and forgot all about the previous I/O Process. The result is overlapped I/O Processes.

(4) Same reasons as (3) with reversed roles.

(5) The falling edge of the ***ACK Signal*** defines the end of this Bus Phase. If the Initiator knows it did it, then it has accepted the Message and can consider the I/O Process completed.

(6) Same reasons as (5). The Target knows that the Initiator understands and accepts a Message when it negates ACK (with the ***ATN Signal*** false) for the last (or only) byte of the Message.

(7) In general, the Initiator should perform an action called for by a Message before negating the ACK Signal for the Message.

(8) Same reasons as (6) and (7).

(9) This is one of those situations where a Device can't be sure that something has happened. This allows the I/O Process to be aborted.

There! Now that isn't so hard....

Status. Status is a brief indication of what the Target did with the ***SCSI Command*** or ***I/O Process***. The Status Code is sent from the Target to the Initiator during the ***STATUS Phase***. Before we get into specifics, we can summarize the kinds of conditions reported in the Status Code:

- Successful completion of the command.
- The command failed to be completed; the I/O Process terminates.
- The Target could not do the I/O Process now, but will be able to later.

Table 19 shows the currently defined SCSI Status Codes:

TABLE 19: STATUS BYTE CODE VALUES

Hex Code	Status Indicated
00	GOOD STATUS
02	CHECK CONDITION
04	CONDITION MET/GOOD STATUS
08	BUSY
10	INTERMEDIATE GOOD STATUS
14	INTERMEDIATE/CONDITION MET/GOOD STATUS
18	RESERVATION CONFLICT
22	COMMAND TERMINATED
28	QUEUE FULL
All others	Reserved

We will now briefly define each code within the categories described above. More detailed definitions are found in the Device-Specific volumes of the SCSI Encyclopedia. The first group of codes report successful completion:

GOOD STATUS: This is the most typical status returned since it indicates that the command was performed as requested with no problems or other conditions deemed necessary (by the Target) to report.

CONDITION MET: This status is reported for a very short list of commands. This status indicates that there were no problems with the command, and a further indication that a condition specified by the command was met. This status causes the ***Contingent Allegiance Condition*** to be created. Specifically, the commands that can return this status are:

- SEARCH DATA commands for Direct Access Devices; the data to be searched for was found

- PRE-FETCH command for Direct Access Devices; there was sufficient cache space for all blocks requested to be pre-fetched.

- MEDIUM SCAN command for Optical Memory Devices; the area of the media that met the scan condition (e.g., blank media) was found.

INTERMEDIATE GOOD STATUS: This is the same as GOOD STATUS, except that the command was part of a sequence of ***Linked Commands***. If the Target is to return GOOD STATUS, and the LINK bit in the ***Command Descriptor Block*** (CDB) is set to one, then this Status Code is returned instead. If the command is the last of a sequence of Linked Commands, then GOOD STATUS is returned since the LINK bit is zero.

INTERMEDIATE/CONDITION MET/GOOD STATUS: This is the same as CONDITION MET, except that the command was part of a sequence of ***Linked Commands***. If the Target is to return CONDITION MET, and the LINK bit in the ***Command Descriptor Block*** (CDB) is set to one, then this Status Code is returned instead. If the command is the last of a sequence of Linked Commands, then CONDITION MET is returned since the LINK bit is zero.

The second group reports the failure of the command to complete. Note that the state of the LINK bit in the CDB does not affect this kind of Status Code; in fact, the sequence of Linked Commands is broken when one of these codes is returned (the I/O Process is terminated).

CHECK CONDITION: This code is returned for any failure or fault in the execution of the command. Technically, the "CHECK" means to check SENSE DATA, by issuing a REQUEST SENSE command. This status causes the Contingent Allegiance Condition to be created; the Target returns this code if it has some condition to report by Sense Data.

COMMAND TERMINATED: This code is returned only if the Initiator sent a ***TERMINATE I/O PROCESS Message***. When the Initiator sends that message, and the Target did not fully complete the command, and no other failure occurred that would require CHECK CONDITION Status, this Status Code is sent. This status causes the Contingent Allegiance Condition to be created.

The third group of codes reports conditions that prevented the acceptance of the I/O Process at this time. In all cases, the Initiator can try again later in hopes that the condition of the Target has changed.

BUSY: This Status Code is returned when the Target is incapable of accepting the command. This can occur:

- When the Target does not implement a ***Queue*** and is busy with another I/O Process from another Initiator.
- When a Contingent Allegiance Condition exists and the Target cannot allocate more space for SENSE DATA.
- When an ***Extended Contingent Allegiance Condition*** exists for another Initiator for the Logical Unit.

RESERVATION CONFLICT: This code is returned when a reserved unit or portion of the unit must be accessed in order to execute the command.

QUEUE FULL: This code is returned when the Queue has no space to receive another I/O Process. This status is only used for Tagged Queuing.

STATUS Phase. The ***Status*** byte is transferred from the Initiator to the Target during the STATUS Phase. The STATUS Phase can follow one of these four phases:

- Following a ***COMMAND Phase***. In this case, the ***SCSI Command*** had no ***DATA Phase*** associated with it, or the command could not be accepted at this time, or the ***I/O Process*** was terminated before the DATA Phase could occur.

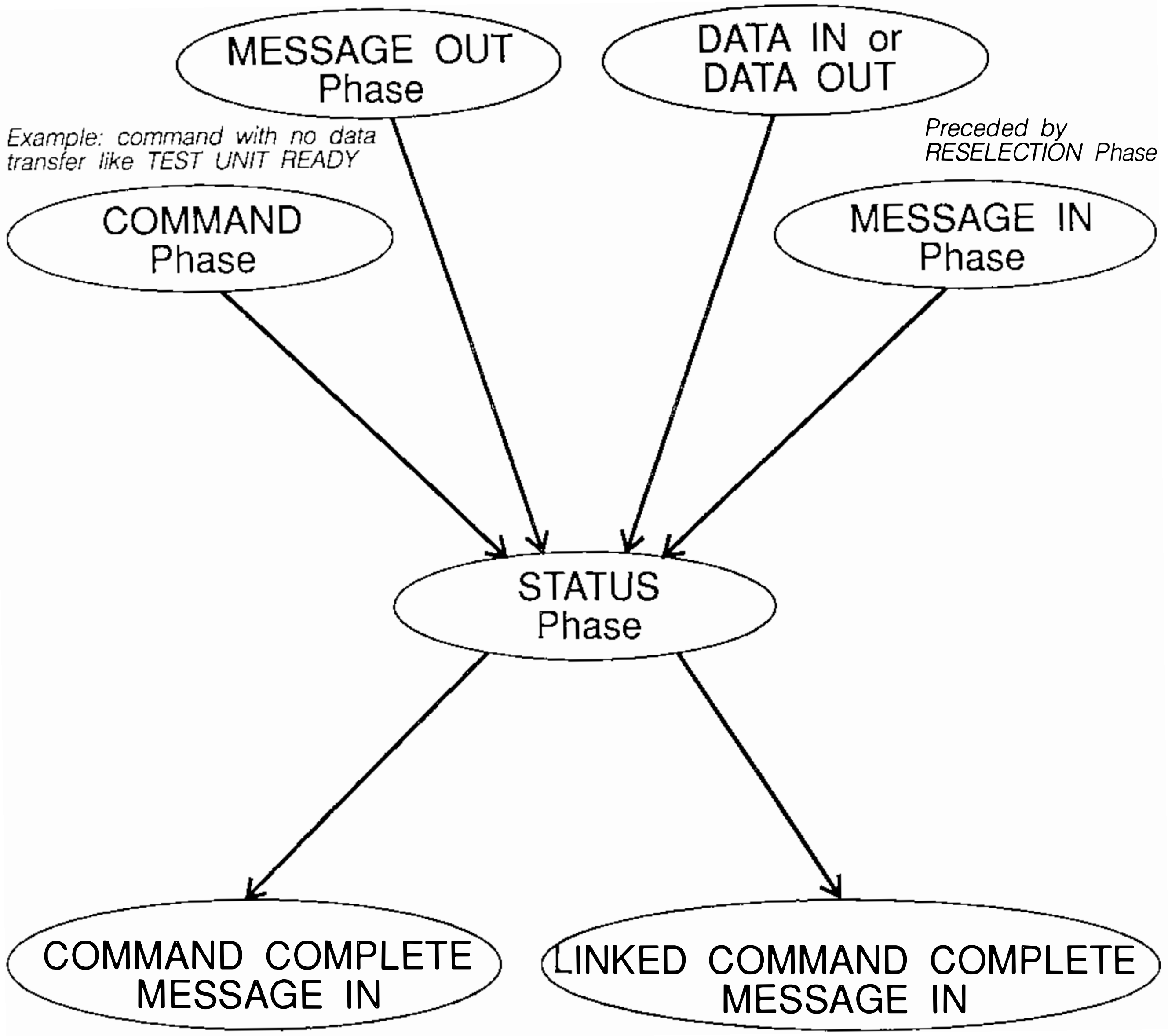
- Following an ***IDENTIFY Message*** or ***Queue Tag Message*** from the Initiator to the Target. In this case, the command could not be accepted at this time, or the IDENTIFY Message was for an unsupported LUN or function.

- Following an IDENTIFY Message or Queue Tag Message from the Target to the Initiator. This is usually for the completion of a command that was executing after a ***Disconnect***. An example of this is a disk write which may be completed while disconnected after the data is transferred to the Target's data buffer.

- Following a DATA Phase (IN or OUT). This signals the completion status of the command following the data transfer associated with the command.

The STATUS Phase may only be followed by a completion ***MESSAGE IN Phase***; either a ***COMMAND COMPLETE Message***, one of the two ***LINKED COMMAND COMPLETE Messages***, or the ***INITIATE RECOVERY Message*** (see ***Extended Contingent Allegiance (ECA)***.

Diagram 32 on the following page illustrates these possible phase transitions into and out of the STATUS Phase.



Preceded by SELECTION with ATN asserted

Example: command with no data transfer like TEST UNIT READY

Preceded by RESELECTION Phase

May be Preceded by INITIATE RECOVERY Message In

DIAGRAM 32: STATUS PHASE TRANSITIONS

Status Pointer. The Status Pointer refers to the destination of ***Status*** within the Initiator. This pointer, like all pointers, has a ***Saved Pointer*** and a ***Active Pointer***. Since there can only be a single Status byte, it might seem silly to have a whole pointer for that one byte. It turns out that it is easier to handle Status the same way as Commands and Data for phase retry purposes. (Also, in the early stages of the SCSI standard development, STATUS Phase could have multiple bytes). This pointer can therefore have only two states; it is either equal to the Saved Status Pointer, or equal to the Saved Status Pointer plus one. See ***Pointers***.

Synchronous Data Transfer. Synchronous Data Transfer is an alternate, optional method for moving data between two ***SCSI Devices***. Depending on implementation details, a significant (2 times or more) performance gain can be realized. Synchronous Data Transfer may only be used during a ***DATA IN Phase*** or a ***DATA OUT Phase***.

We recommend that you understand ***Asynchronous Data Transfer*** before diving into this subject!

Let's review the Asynchronous Data Transfer, which proceeds with the steps shown in Table 20.

TABLE 20: REVIEW OF ASYNCHRONOUS DATA TRANSFER

Steps During Transfer OUT	Steps During Transfer IN
Target Asserts REQ	Target Asserts Data
Initiator Asserts Data	Target Asserts REQ
Initiator Asserts ACK	Initiator Asserts ACK
Target Negates REQ	Target Negates REQ
Initiator Negates ACK	Initiator Negates ACK

This works very well, but there is a lot of wasted time. The actual work of sending the data occurs in the first two or three steps. Also, since each step must wait for the previous step to complete, there are several delays due to the speed of signals and data propagating down the ***Cable***. Recall the performance equation from the Asynchronous Data Transfer:

$$
\begin{aligned}
t_{xfer} &= t_{cd} + t_{pd} + t_{ds} + t_{cs} + t_{cd} + t_{pd} + t_{cd} + t_{pd} + t_{cd} + t_{pd} \\
&= 4t_{cd} + 4t_{pd} + t_{ds} + t_{cs}
\end{aligned}
$$

Where: t_{cd} is the cable propagation delay.
t_{pd} is the internal device delay from receiving a signal to responding with another signal.
t_{ds} is the ***Deskew Delay***.
t_{cs} is the ***Cable Skew Delay***.

It would be nice to eliminate, or at least minimize, the wasted time. Funny thing, that's what Synchronous Data Transfer attempts to do; the transfer time is not a function of these delays. Instead, the transfer time is equal to the clock period, known as the ***Synchronous Transfer Period***.

Synchronous Data Transfer uses the ***REQ Signal*** and the ***ACK Signal*** as clocks to directly latch data into the destination, and to clock circuits that keep track of the state of the transfer. In this transfer mode, the REQ and ACK signals do not interlock; they are issued independently by each device. In other words, the REQ and ACK signals are sent out as a string of clock pulses. In a nutshell:

- During a transfer IN, the Target clocks data into the Initiator on each REQ pulse. The Initiator returns an ACK pulse for each REQ pulse.

- During a transfer OUT, the Target sends a REQ pulse for each data transfer requested from the Initiator. The Initiator returns an ACK that clocks the data into the Target, one for each REQ pulse.

Note that the number of REQ pulses sent by the Target must be equal to the number of ACK pulses sent by the Initiator at the end of the data transfer. If the Target has not received ACK pulses equal to the REQ pulses, it may not change to another phase until all ACKs are received. Note also that a "received ACK pulse" is the whole pulse; the trailing edge must also be received. These facts allow the Initiator to hold the end of the Phase until it can check for ***Parity*** or other errors.

The diagrams that follow on the next pages illustrate a Synchronous Data Transfer system. It should be noted that this is a simplified model, and that other implementations (e.g., a single FIFO) are possible.

Each Target contains the following elements (Diagram 33):

- A Data Send FIFO: The data to be sent to the Initiator is passed through this FIFO. Data is clocked into the FIFO in a ***Controller*** specific manner; e.g., a DMA channel. Data is clocked out of the FIFO on the trailing edge of the REQ signal (with a catch; more on this later). If the Send FIFO is empty, the transfer is held up (see below).

- A Data Receive FIFO. The data received from the Initiator is passed through this FIFO. Data is clocked into the FIFO on the leading edge of the ACK signal. Data is clocked out of the FIFO in a Controller specific manner. If the Receive FIFO is full, the transfer is held up (see below).

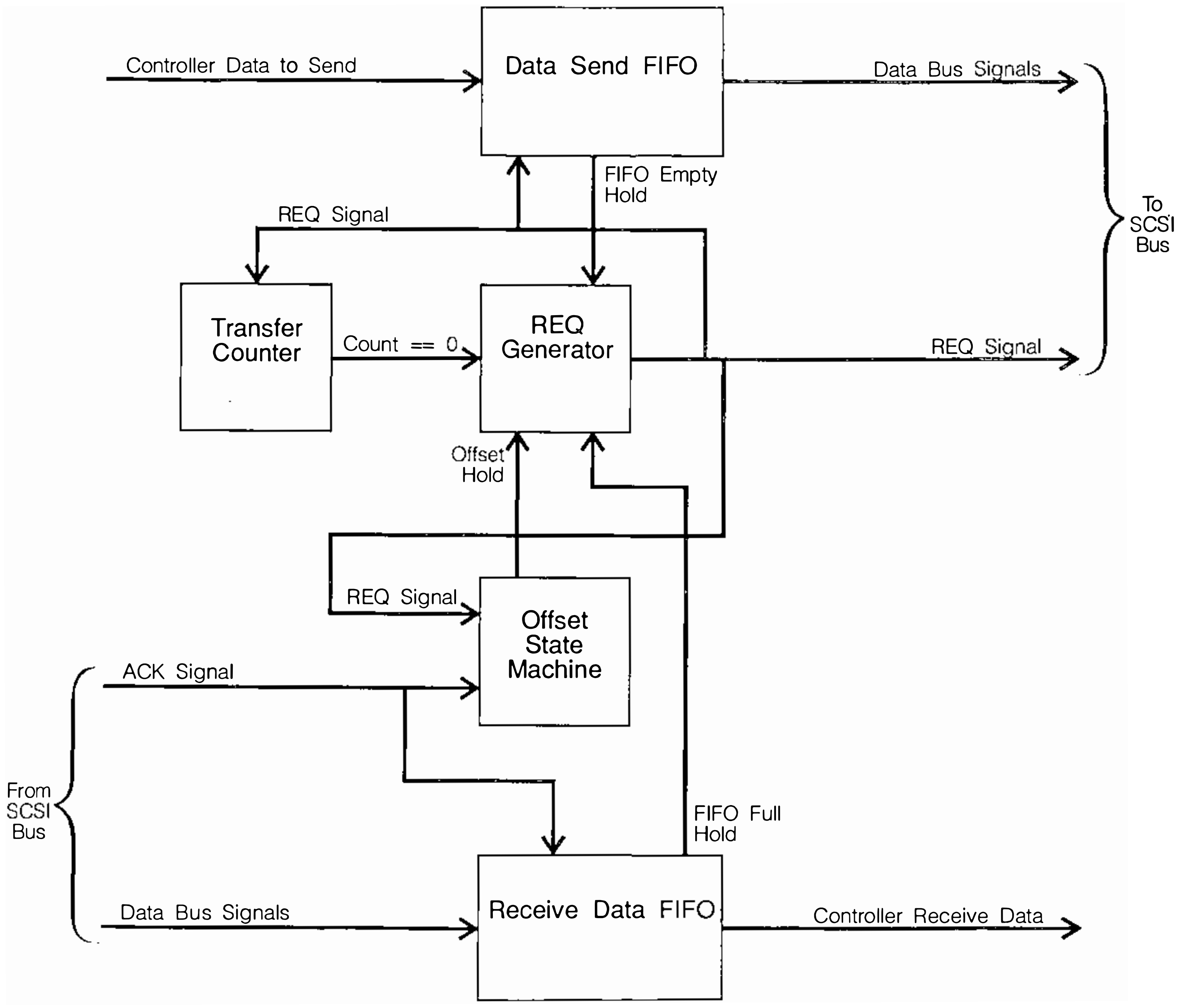
- A Transfer Counter. This counter keeps track of the number of bytes transferred with the Initiator. The counter is decremented once on every leading edge of the REQ signal. When the count is zero, the transfer is completed.

- An Offset State Machine. This is actually a state machine and counter. This block keeps track of the ***Synchronous Offset***. When the offset is exceeded, the transfer is held up. More on this in a couple of pages.

- A REQ Generator. The REQ generator creates the REQ timing for the transfer. The REQ generator emits REQ pulses as long as one of the 'hold' conditions listed above doesn't occur. The spacing between the pulses is the previously agreed to ***Synchronous Transfer Period***.

To begin a transfer, the Target sets the Transfer Counter. For a transfer IN, the Target loads data into the Send FIFO. For a transfer OUT, the Target lets the REQ signals go out, and when the Initiator responds with ACK signals, takes the data out of the Receive FIFO. Details of this flow follow the block diagrams.

DIAGRAM 33: SYNC TRANSFER TARGET BLOCK DIAGRAM



Each Initiator contains the following elements (Diagram 34):

- A Data Send FIFO: The data to be sent to the Target is passed through this FIFO. Data is clocked into the FIFO in a ***Host*** specific manner; e.g., a DMA channel. Data is clocked out of the FIFO on the trailing edge of the ACK signal (with a catch; more on this later). If the Send FIFO is empty, the transfer is held up (see below).

- A Data Receive FIFO. The data received from the Target is passed through this FIFO. Data is clocked into the FIFO on the leading edge of the REQ signal. Data is clocked out of the FIFO in a Host specific manner. If the Receive FIFO is full, the transfer is held up (see below).

- A Transfer Counter. This counter keeps track of the number of bytes transferred with the Target. The counter is decremented once on every leading edge of the ACK signal. When the count is zero, the transfer is completed.

- An Offset State Machine. This is actually a state machine and counter. This block keeps track of the ***Synchronous Offset***. When the offset is exceeded, the transfer is held up. More on this in a couple of pages.

- An ACK Generator. The ACK generator creates the ACK timing for the transfer. The ACK generator emits ACK pulses as long as one of the 'hold' conditions listed above doesn't occur. The spacing between the pulses is the previously agreed to ***Synchronous Transfer Period***.

As will be seen, the Initiator has a tougher job than the Target (it's about time!!). Since the Initiator cannot be sure exactly when the Target will change to a ***DATA Phase***, the Initiator circuit must always be ready to receive REQ pulses and possibly data (on a ***DATA IN Phase***). The Initiator detects that the Target has begun a DATA Phase when the Offset State Machine indicates a non-zero current offset. Then the Initiator can set the Transfer Counter. For a transfer OUT, the Initiator loads data into the Send FIFO. For a transfer IN, the Initiator detects the REQ signals coming in, and responds with ACK signals while taking the data out of the Receive FIFO. Details of this flow follow the block diagrams.

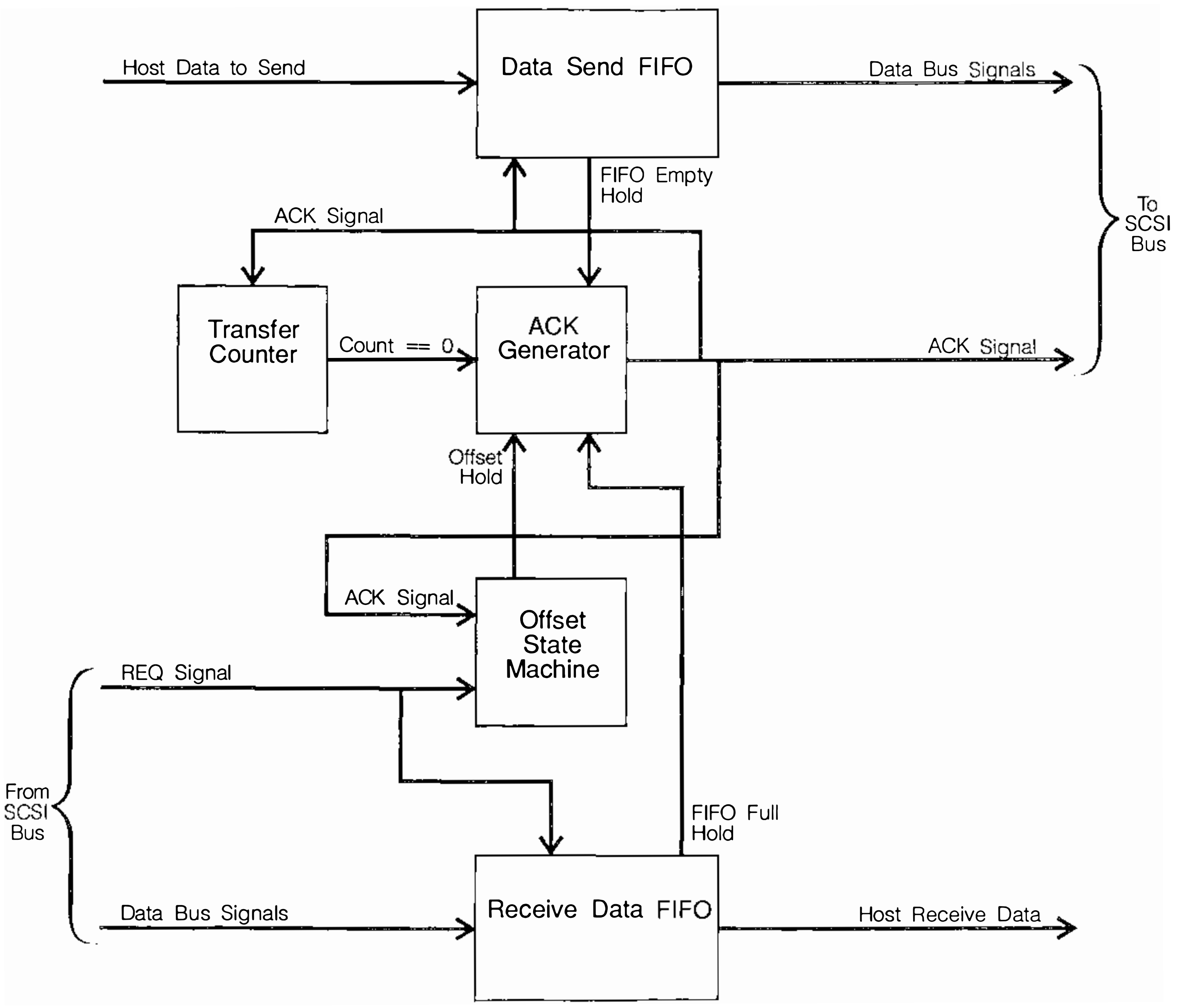


DIAGRAM 34: SYNC TRANSFER INITIATOR BLOCK DIAGRAM

The alert student will notice how similar the two block diagrams are. The exact differences are:

- The roles of the REQ Signal and the ACK Signal are reversed.
- The Target has a REQ Generator, and the Initiator has an ACK Generator.
- There are some minor differences in the Offset State Machine, which are not evident in the block diagrams. We will discuss these differences now.

The Offset State Machine state transition diagram for the Target is shown in Diagram 35 on the next page. The Offset State Machine actually consists of a finite state machine and a Current Offset Counter. The counter is able to compare the setting of the ***Synchronous Offset*** with the current state of the counter. The Offset State Machine provides a HOLD output to the REQ Generator. There are four states:

(1) Okay to Transfer. The Target begins all ***DATA Phases*** in this state. As long as the Offset State Machine is in this state, the "HOLD" output is false. The Offset State Machine stays in this state until the leading edge of a REQ pulse or an ACK pulse.

(2) Increment Offset. When the Offset State Machine detects the leading edge of a REQ pulse, it momentarily enters this state to increment the Current Offset Counter. If the current state of the counter is less than the Synchronous Offset, the next state is Okay to Transfer. If the current state of the counter is equal to the Synchronous Offset, the next state is Transfer Hold. The Offset State Machine should stay in this state for less than an ***Assertion Period*** to be sure that it can detect an outstanding ACK pulse and the next REQ pulse.

(3) Decrement Offset. When the Offset State Machine detects the leading edge of an ACK pulse, it momentarily enters this state to decrement the Current Offset Counter. The next state is always Okay to Transfer. The Offset State Machine should stay in this state for less than an Assertion Period to be sure that it can detect an outstanding REQ pulse and the next ACK pulse.

(4) Transfer Hold. When the Offset State Machine is in this state, all transfers are inhibited; the HOLD output is true. The Offset State Machine stays in this state until the next ACK pulse is received. This causes a transition to the Decrement Offset state.

Note that the Offset State Machine must be in the Okay to Transfer state, with the Current Offset Counter at zero, before the Target can proceed to another phase.

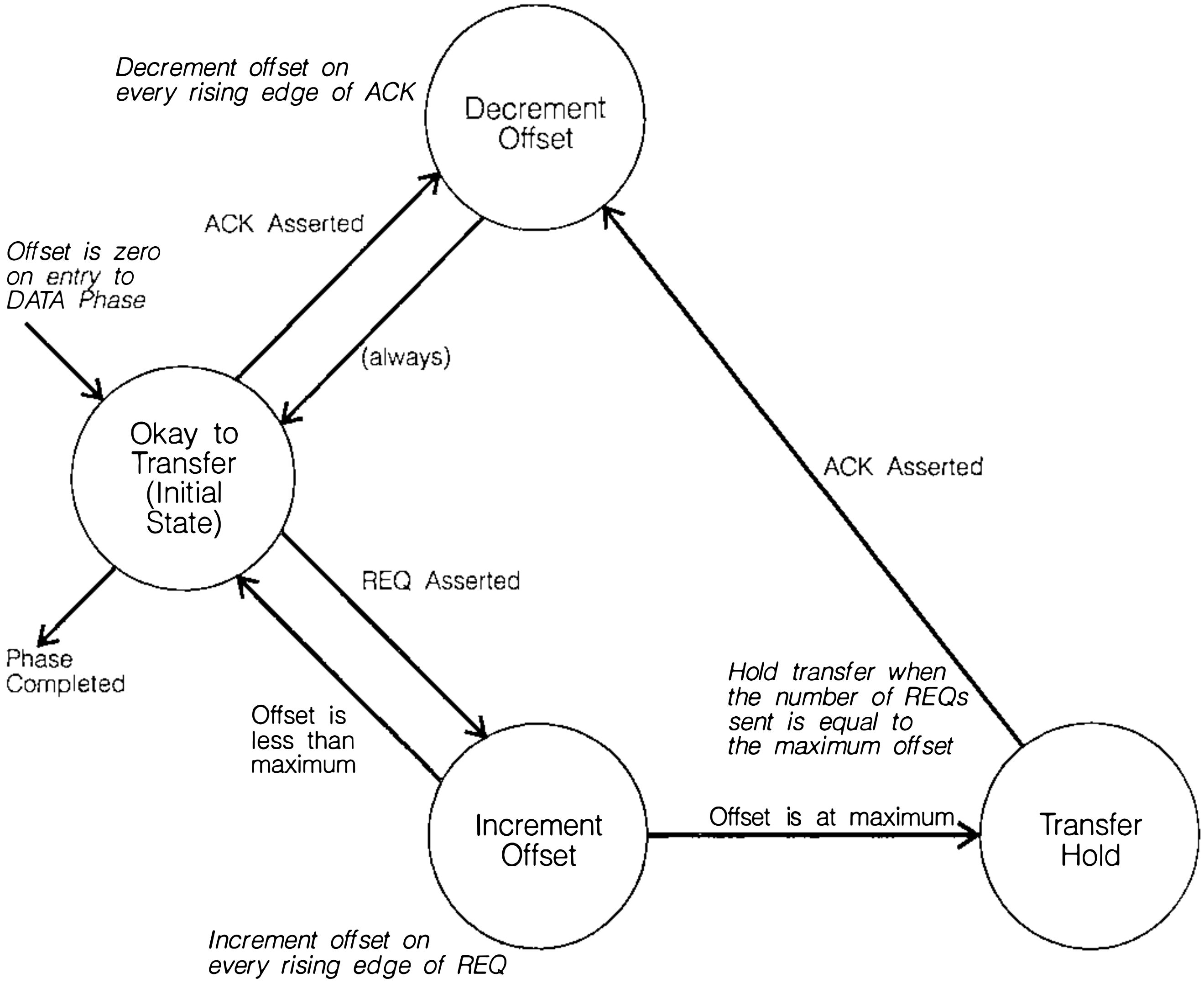

Decrement offset on every rising edge of ACK
Decrement Offset
ACK Asserted
Offset is zero on entry to DATA Phase
(always)
Okay to Transfer (Initial State)
ACK Asserted
REQ Asserted
Phase Completed
Hold transfer when the number of REQs sent is equal to the maximum offset
Offset is less than maximum
Increment Offset
Offset is at maximum
Transfer Hold
Increment offset on every rising edge of REQ

DIAGRAM 35: SYNC OFFSET STATE MACHINE FOR TARGETS

The Offset State Machine state transition diagram for the Initiator is shown in Diagram 36 on the next page. As with the Target, the Offset State Machine consists of a finite state machine and a Current Offset Counter. The use of the counter is different for the Initiator: the counter is compared with zero instead of the ***Synchronous Offset***. The Offset State Machine provides a HOLD output to the ACK Generator. There are four states:

(1) Transfer Hold. The Initiator begins all ***DATA Phases*** in this state. In this state, the number of REQ pulses received equals the number of ACK pulses sent. When the Offset State Machine is in this state, all transfers are inhibited; the HOLD output is true. The Offset State Machine stays in this state until the next REQ pulse is received. This causes a transition to the Increment Offset state.

(2) Increment Offset. When the Offset State Machine detects the leading edge of a REQ pulse, it momentarily enters this state to increment the Current Offset Counter. The next state is always Okay to Transfer. The Offset State Machine should stay in this state for less than an ***Assertion Period*** to be sure that it can detect an outstanding ACK pulse and the next REQ pulse.

(3) Decrement Offset. When the Offset State Machine detects the leading edge of an ACK pulse, it momentarily enters this state to decrement the Current Offset Counter. If the current state of the counter is greater than zero, the next state is Okay to Transfer. If the current state of the counter is zero, the next state is Transfer Hold. The Offset State Machine should stay in this state for less than an Assertion Period to be sure that it can detect an outstanding REQ pulse and the next ACK pulse.

(4) Okay to Transfer. As long as the Offset State Machine is in this state, the "HOLD" output is false. The Offset State Machine stays in this state until the leading edge of a REQ pulse or an ACK pulse. In this state, the number of REQ pulses received is greater than the number of ACK pulses sent.

Note that the Offset State Machine must be in the Transfer Hold state, with the Current Offset Counter at zero, before the Target can proceed to another phase.

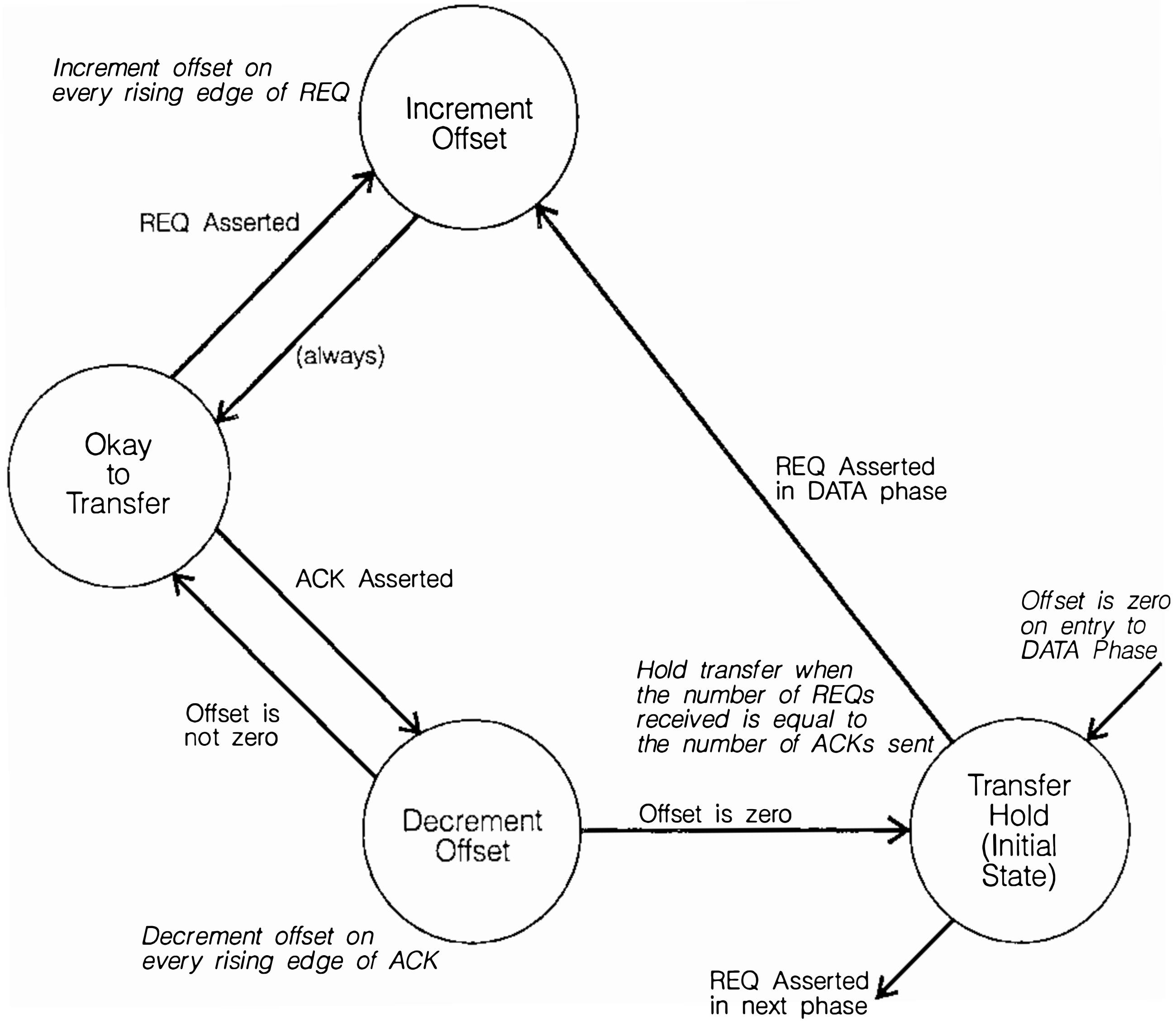

Increment offset on
every rising edge of REQ
Increment Offset
REQ Asserted
(always)
Okay to Transfer
REQ Asserted in DATA phase
ACK Asserted
Offset is zero on entry to DATA Phase
Hold transfer when the number of REQs received is equal to the number of ACKs sent
Offset is not zero
Transfer Hold (Initial State)
Decrement Offset
Offset is zero
Decrement offset on every rising edge of ACK
REQ Asserted in next phase

DIAGRAM 36: SYNC OFFSET STATE MACHINE FOR INITIATORS

Table 21 summarizes the differences between the Target and the Initiator machines.

TABLE 21: OFFSET STATE MACHINE: INITIATOR VS. TARGET

Target Offset State Machine	Initiator Offset State Machine
Initial State is Okay to Transfer	Initial State is Transfer Hold
Hold when current offset is equal to Synchronous Offset	Hold when current offset is equal to zero
Exit Transfer Hold state on leading edge of ACK	Exit Transfer Hold state on leading edge of REQ
Enter Transfer Hold state after an offset increment	Enter Transfer Hold state after an offset decrement
Final State is Okay to Transfer; current offset is zero	Final State is Transfer Hold; current offset is zero

There are two fundamental reasons for these differences:

- The Target issues REQ pulses, and these cause the current offset to increase. The Target can only cause the current offset to increase; it cannot decrease it. Therefore, the Target must inhibit itself so that it does not exceed the limit imposed by the Synchronous Offset.

- The Initiator issues ACK pulses, and these cause the current offset to decrease. The Initiator can only cause the current offset to decrease; it cannot increase it. Therefore, the Initiator must inhibit itself so that it does not issue more ACK pulses than the number of REQ pulses received.

The next four diagrams show the details of Synchronous Data Transfer flow for Targets and Initiators and for ***DATA IN Phase*** and ***DATA OUT Phase***. Refer to the previous block diagrams while studying these diagrams.

Diagram 37 shows the flow for DATA IN Phase for the Target. The Target sets up the Synchronous Data Transfer circuit by loading the Transfer Counter. It then begins presenting data to the Initiator by loading the Send FIFO. This allows the REQ Generator to begin sending REQ pulses to the Initiator.

If the Initiator doesn't start responding with ACK pulses immediately, the Offset State Machine will cause a Transfer Hold. A Transfer Hold could also occur if the Send FIFO is empty.

The data transfer continues until the Transfer Count decrements to zero. At this point, all data has been transferred from the Target to the Initiator. The Target now waits for the Initiator to finish sending the rest of the ACK pulses. When the current offset returns to zero, the Phase is completed.

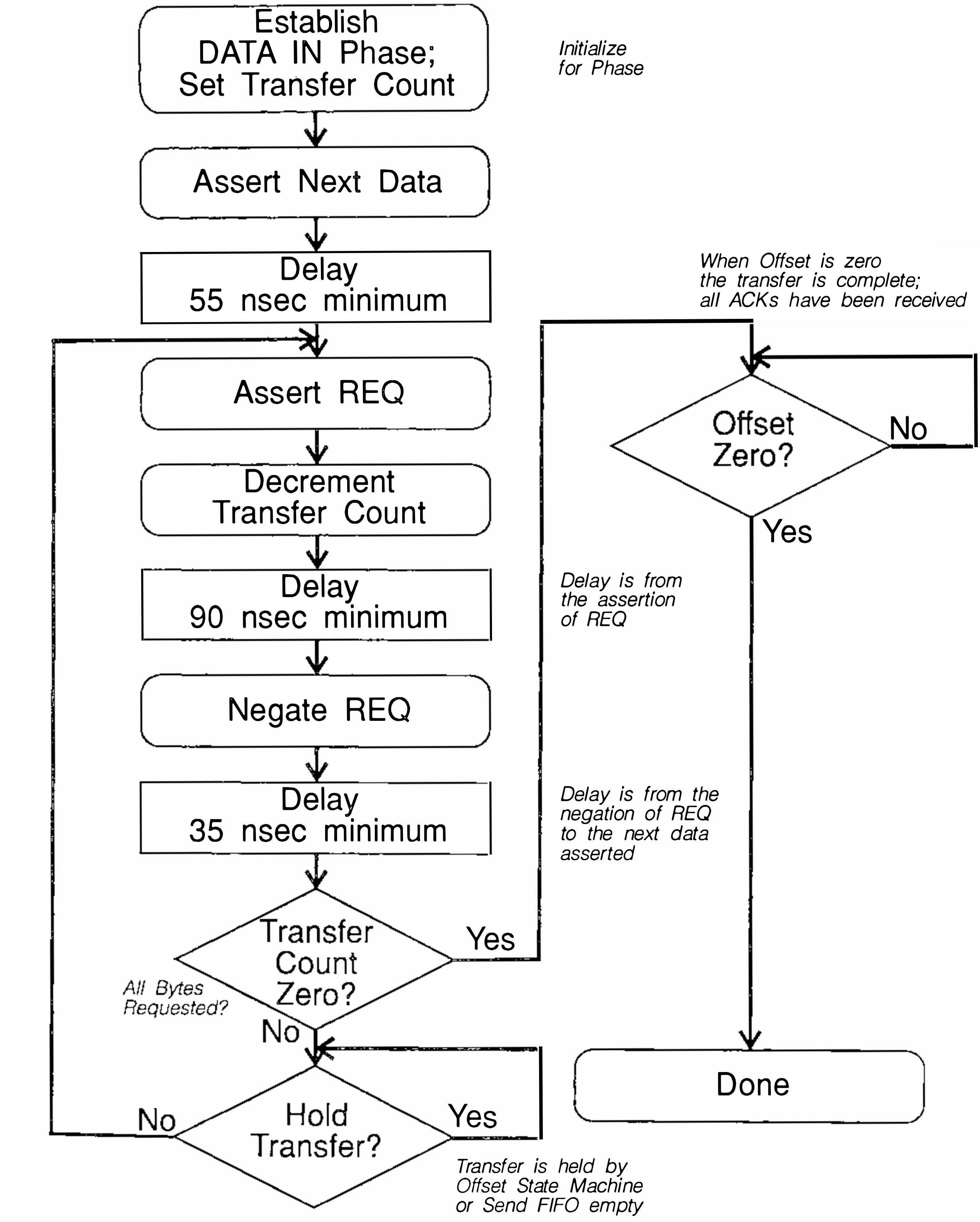

Establish DATA IN Phase; Set Transfer Count
Initialize for Phase
Assert Next Data
Delay 55 nsec minimum
When Offset is zero the transfer is complete; all ACKs have been received
Assert REQ
Offset Zero?
No
Yes
Decrement Transfer Count
Delay 90 nsec minimum
Delay is from the assertion of REQ
Negate REQ
Delay 35 nsec minimum
Delay is from the negation of REQ to the next data asserted
Transfer Count Zero?
Yes
All Bytes Requested?
No
Hold Transfer?
No
Yes
Done
Transfer is held by Offset State Machine or Send FIFO empty

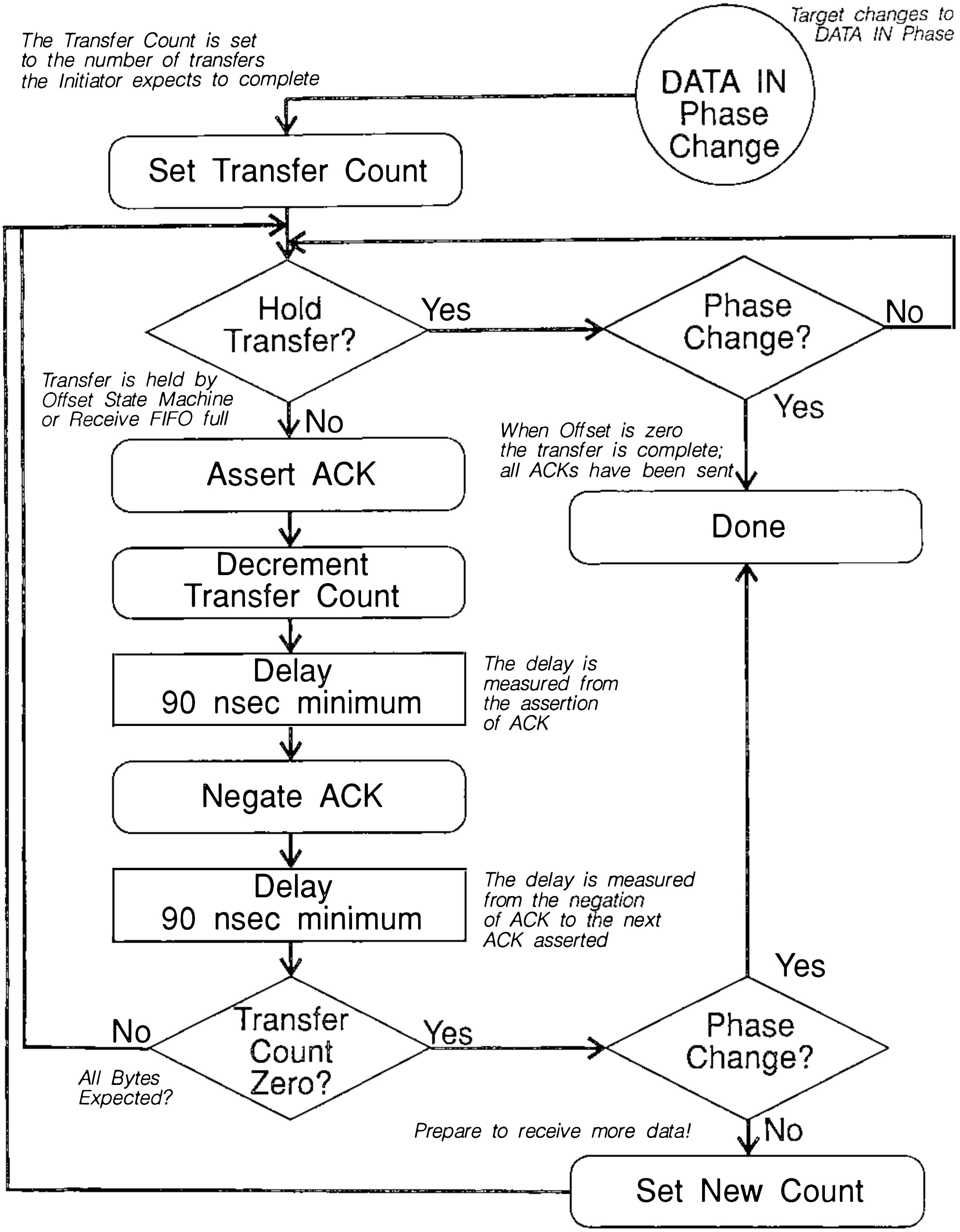
DIAGRAM 37: SYNC TRANSFER: DATA IN PHASE FOR TARGETS

Diagram 38 shows the flow for DATA IN Phase for the Initiator. When the Target changes to DATA IN Phase, it begins sending data immediately with the first REQ pulse, which clocks the data into the Receive FIFO. As a consequence, unless the Initiator is very fast, the Initiator's Receive DATA FIFO is already full of data from the Target. The Initiator sets up the Synchronous Data Transfer circuit by loading the Transfer Counter. It then begins taking data from the Target by unloading the Receive FIFO. This allows the ACK Generator to begin sending ACK pulses to the Target.

If the Target doesn't start responding with REQ pulses immediately, the Offset State Machine will cause a Transfer Hold. A Transfer Hold could also occur if the Receive FIFO is full.

The data transfer continues until the Transfer Count decrements to zero. At this point, all data expected by the Initiator has been transferred from the Target to the Initiator. When the current offset returns to zero, the Phase is (hopefully) completed.

If the current offset is not zero, then more REQ pulses are still outstanding. The Target is expecting to send more data. The Initiator must take this data from the Target even if it must discard it. An appropriate Initiator response is to create the ***Attention Condition*** and continue to take and discard data until the ***ATN Signal*** is recognized by the Target, and then issue the ***ABORT Message***. In general, the Initiator should stay prepared for more DATA IN Phase data until the REQ Signal is asserted in another ***Information Transfer Phase***. (Note: Some SCSI protocol controllers have a mode called "Transfer Pad", which is used for this situation).



The Transfer Count is set to the number of transfers the Initiator expects to complete
Target changes to DATA IN Phase
DATA IN Phase Change
Set Transfer Count
Hold Transfer?
Yes
Phase Change?
No
Transfer is held by Offset State Machine or Receive FIFO full
No
Yes
When Offset is zero the transfer is complete; all ACKs have been sent
Assert ACK
Done
Decrement Transfer Count
Delay 90 nsec minimum
The delay is measured from the assertion of ACK
Negate ACK
Delay 90 nsec minimum
The delay is measured from the negation of ACK to the next ACK asserted
Yes
No
Transfer Count Zero?
Yes
Phase Change?
All Bytes Expected?
Prepare to receive more data!
No
Set New Count

DIAGRAM 38: SYNC TRANSFER: DATA IN PHASE FOR INITIATORS

Diagram 39 shows the flow for DATA OUT Phase for the Target. The Target sets up the Synchronous Data Transfer circuit by loading the Transfer Counter. It then begins requesting data from the Initiator because the REQ Generator is allowed to begin sending REQ pulses to the Initiator.

If the Initiator doesn't start responding with ACK pulses immediately, the Offset State Machine will cause a Transfer Hold. A Transfer Hold could also occur when the Receive FIFO is full. The Initiator ACK pulses clock data into the Receive FIFO, and the Target takes the data by removing it from the Controller end.

The data transfer continues until the Transfer Count decrements to zero. At this point, all data has been requested from the Initiator. The Target now waits for the Initiator to finish sending the rest of the ACK pulses and data. When the current offset returns to zero, the Phase is completed.

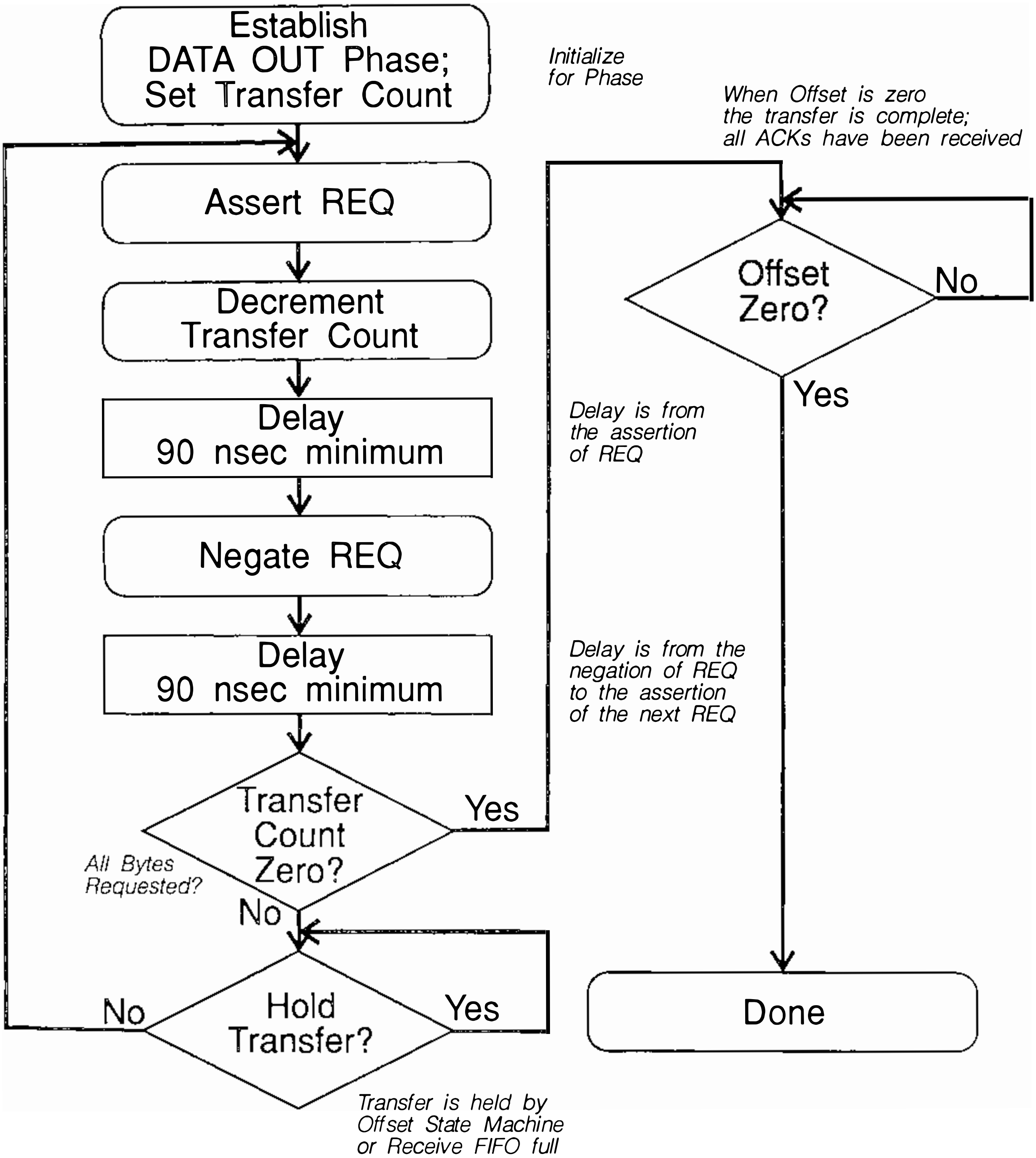


DIAGRAM 39: SYNC TRANSFER: DATA OUT FOR TARGETS

Diagram 40 shows the flow for DATA OUT Phase for the Initiator. The Initiator sets up the Synchronous Data Transfer circuit by loading the Transfer Counter with the expected transfer count. It then begins sending data to the Target by loading the Send FIFO. This allows the ACK Generator to begin sending ACK pulses to the Target and clocking data into the Target's Receive FIFO.

If the Target doesn't start responding with REQ pulses immediately, the Offset State Machine will cause a Transfer Hold. A Transfer Hold could also occur if the Send FIFO is empty.

The data transfer continues until the Transfer Count decrements to zero. At this point, all data expected by the Initiator has been transferred to the Target from the Initiator. When the current offset returns to zero, the Phase is (hopefully) completed.

If the current offset is not zero, then more REQ pulses are still outstanding. The Target is expecting to receive more data. Just like with DATA IN Phase, the Initiator must send data to the Target even if it is garbage data. An appropriate Initiator response is to create the ***Attention Condition*** and continue to send "data" until the ***ATN Signal*** is recognized by the Target, and then issue the ***ABORT Message***. In general, the Initiator should stay prepared for more DATA OUT Phase data until the REQ Signal is asserted in another ***Information Transfer Phase***. (Note: The "Transfer Pad" mode can also be used for this situation).

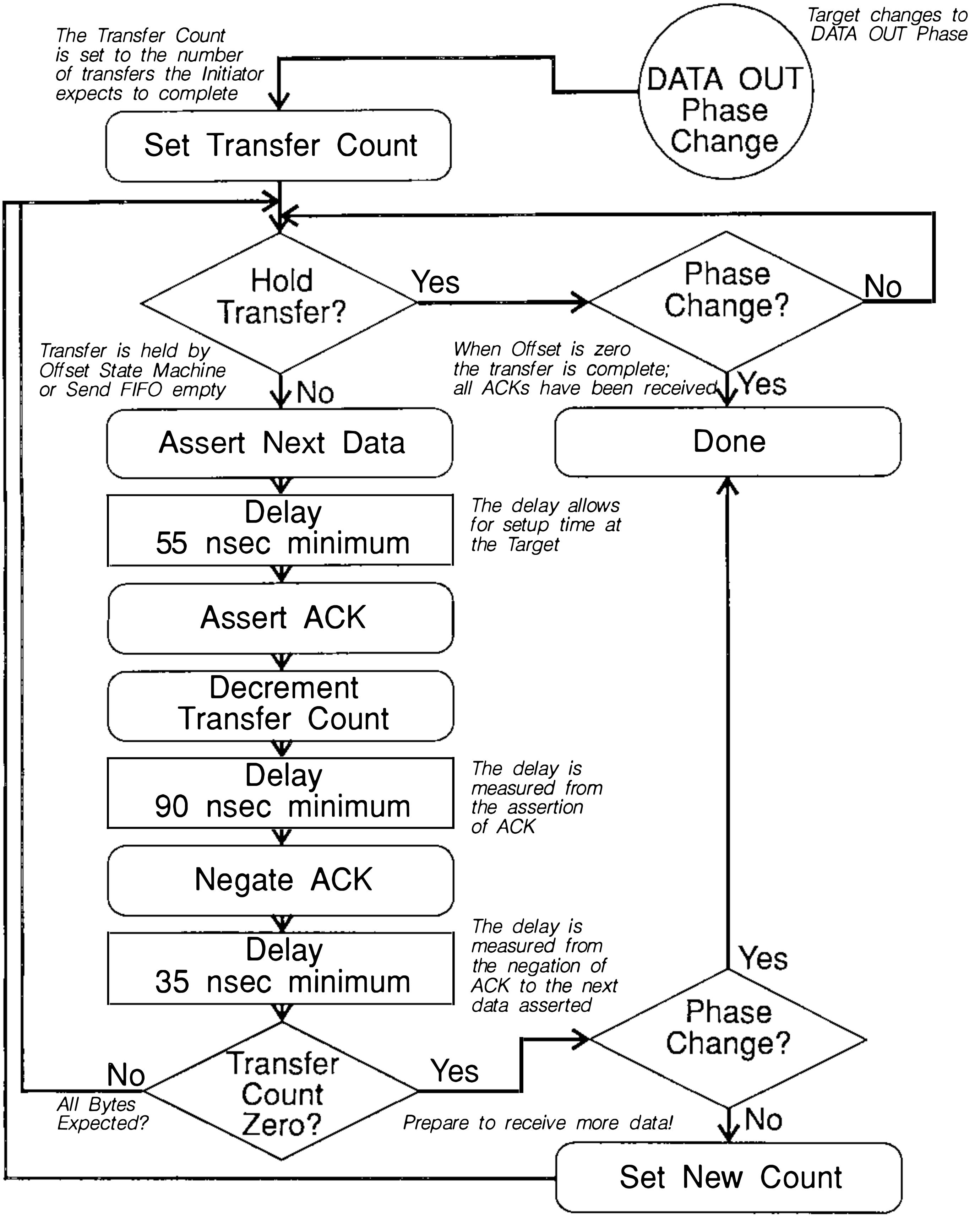

Target changes to DATA OUT Phase
DATA OUT Phase Change
The Transfer Count is set to the number of transfers the Initiator expects to complete
Set Transfer Count
Hold Transfer?
Yes
Phase Change?
No
Transfer is held by Offset State Machine or Send FIFO empty
No
When Offset is zero the transfer is complete; all ACKs have been received
Yes
Assert Next Data
Done
Delay 55 nsec minimum
The delay allows for setup time at the Target
Assert ACK
Decrement Transfer Count
Delay 90 nsec minimum
The delay is measured from the assertion of ACK
Negate ACK
Delay 35 nsec minimum
The delay is measured from the negation of ACK to the next data asserted
Yes
Phase Change?
No
All Bytes Expected?
Transfer Count Zero?
Yes
Prepare to receive more data!
No
Set New Count

DIAGRAM 40: SYNC TRANSFER: DATA OUT FOR INITIATORS

What? There's more? Well, we thought it would be nice to show the exact timing of a Synchronous Data Transfer...

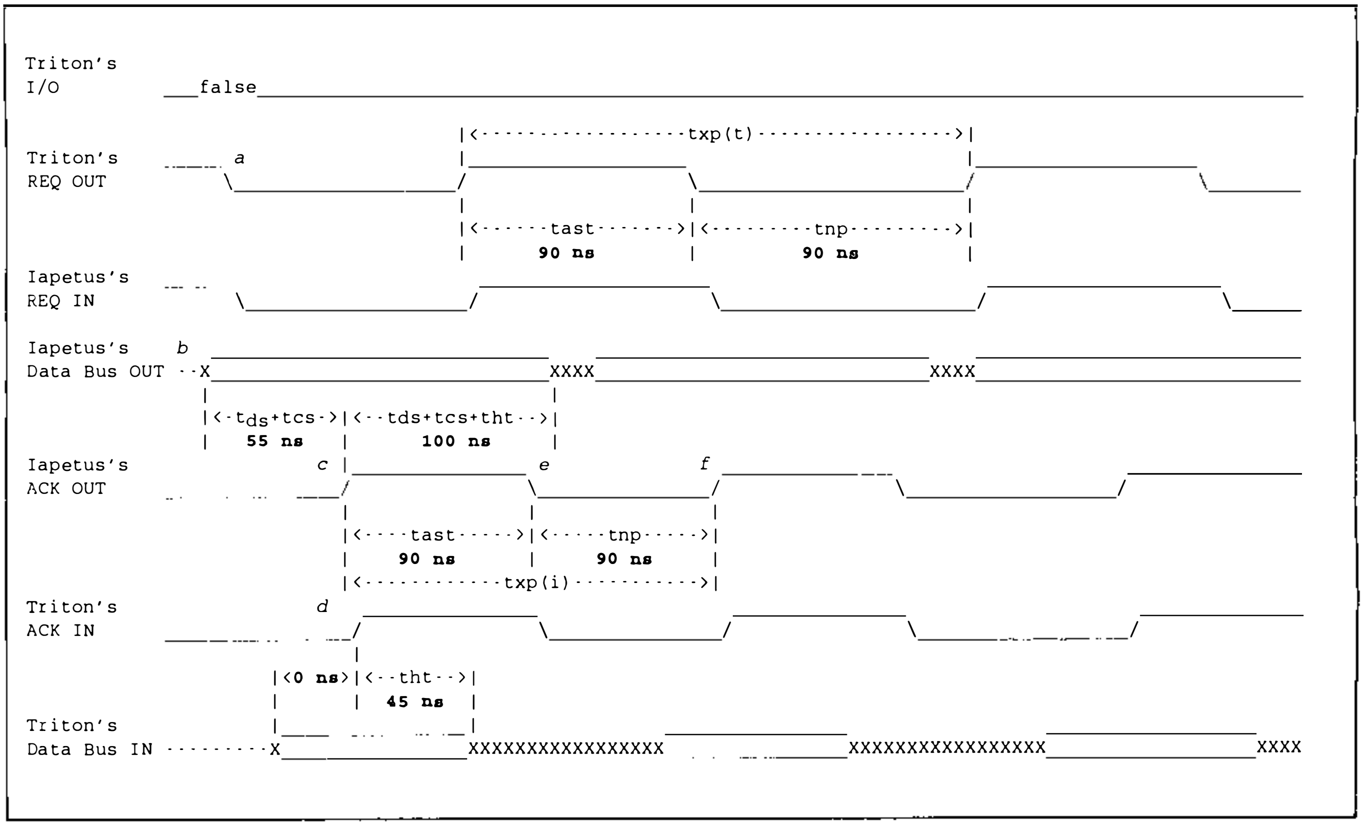


FIGURE 5: SYNCHRONOUS DATA TRANSFER OUT - INITIATOR TO TARGET

The Details are shown in Figure 5:

(a) Triton has begun sending a sequence of REQ pulses to Iapetus. The leading edge of the first REQ pulse defined this as a ***DATA OUT Phase***. Triton sends out a number of REQ pulses appropriate to the amount of data that it can accept from the Initiator. While the number of REQ pulses outstanding is usually equal to the Synchronous Offset, Triton may decide not to send REQ pulses until he is ready to receive more data. This usually happens when Triton's data path stalls; he can't take data until an internal resource becomes available.

After the signal propagates down the cable for a time, Iapetus detects the REQ signal going true at her input, which increments her Current Offset Counter. Depending on the chip and other implementation details, Iapetus may be able to respond to the new phase immediately (particularly if the phase was expected next), or she may need to do some cleanup of the previous phase and bookkeeping before she can get started. In any case, the Current Offset Counter must accept the incoming REQ pulses to the limit of the ***Synchronous Offset***.

(b) Now Iapetus is ready to start sending data to the Target. The first thing she does is to set the data bus to the value of the first data byte (or word, see ***Wide Data Transfer***) to send.

(c) After a 55 nsec delay, Iapetus asserts the ACK signal true for no less than 90 nsec, after which she may negate ACK. The 55 nsec delay provides deskew time for the Target receiving latch or FIFO. The Initiator may extend the assertion of the ACK signal to prevent the Target from changing Phase for purposes of checking for errors and possibly creating the ***Attention Condition***.

(d) After the data and ACK signal propagate down the cable for a time, Triton detects the ACK signal going true at his input. When he sees the ACK, he knows the data is valid on the bus, so he clocks it from the bus directly with the ACK signal and eventually stores the data to its final destination. The ACK also decrements his Current Offset Counter.

Note what happened to the set up time and hold time for the data. When it left Iapetus, there was 55 nsec of set up time between the data arriving and the leading edge of the ACK signal, and 100 nsec of hold time after the ACK. But when it reached Triton, there was no more lead time, and the hold time was 45 nsec. A total of 55 nsec disappeared from both times.

The 55 nsec time is intended to account for skew in the circuit, and is made up of two parts. The first part is called the ***Deskew Delay*** (45 nsec), which is intended to compensate for internal skews in both devices (Initiator and Target). The second part is called the ***Cable Skew Delay*** (10 nsec), which is intended to compensate for propagation speed differences between signals on the cable. The SCSI standard allows 10 nsec for this signal skew, and therefore the user ought to use a ***Cable*** that at least meets this requirement.

In practice the set up time and/or hold time will be larger, since the worst case conditions cannot occur both ways. These numbers are intended as a guideline for designers of the receive data circuit.

(e) After a delay of 100 nsec from the leading edge of the ACK pulse, Iapetus is allowed to change the data bus. This delay provides hold time for the Target receiving latch or FIFO.

(f) Triton continues sending REQ pulses until he has requested all of the data. The leading edges of the REQ pulses must be separated by no less than time $t_{xp(t)}$, which is the ***Synchronous Transfer Period*** that Iapetus said she could handle during ***Synchronous Data Transfer Negotiation***. Iapetus continues sending data and ACK pulses until the Offset State Machine and transfer count reach zero, as described above. As with the REQ pulses, the leading edges of the ACK pulses must be separated by no less than time $t_{xp(i)}$, which is the Synchronous Transfer Period that Triton said he could handle during Synchronous Data Transfer Negotiation.

Some additional observations on Synchronous Data Transfer OUT:

- OUT and IN: These terms are used throughout the SCSI standard to indicate the current data direction. They are relative to the Initiator; data goes OUT from the Initiator to the Target and IN from the Target to the Initiator.

- Since the transfer is synchronous, cable propagation time does not affect the transfer rate. One device throws data and pulses out on the cable. The other device has plenty of time (relatively) to respond with its pulses. Cable skew (that is, the difference in delay between any two signals), on the other hand, is significant with regard to set up and hold time, as noted above.

- The ***MSG, C/D, and I/O Signals*** must be stable for the 400 nsec (or more) delay after changing to the ***DATA OUT Phase*** before the Target may begin sending REQ pulses to begin the transfer. The phase does not begin until the leading edge of the first REQ pulse. Some Initiator chip implementations, in a sincere effort to gain some performance, detected phase changes when the MSG, C/D, and/or I/O signal(s) changed state. If the three lines changed at different times, the device would indicate the one or more wrong phases before indicating the correct phase. A person using one of these devices should ensure that the phase detected in such a manner is the same phase when the REQ pulses begin. See ***Bus Phases*** and ***Between Phases*** for more on this.

Now we'll look at how to get data **in from** a Target:

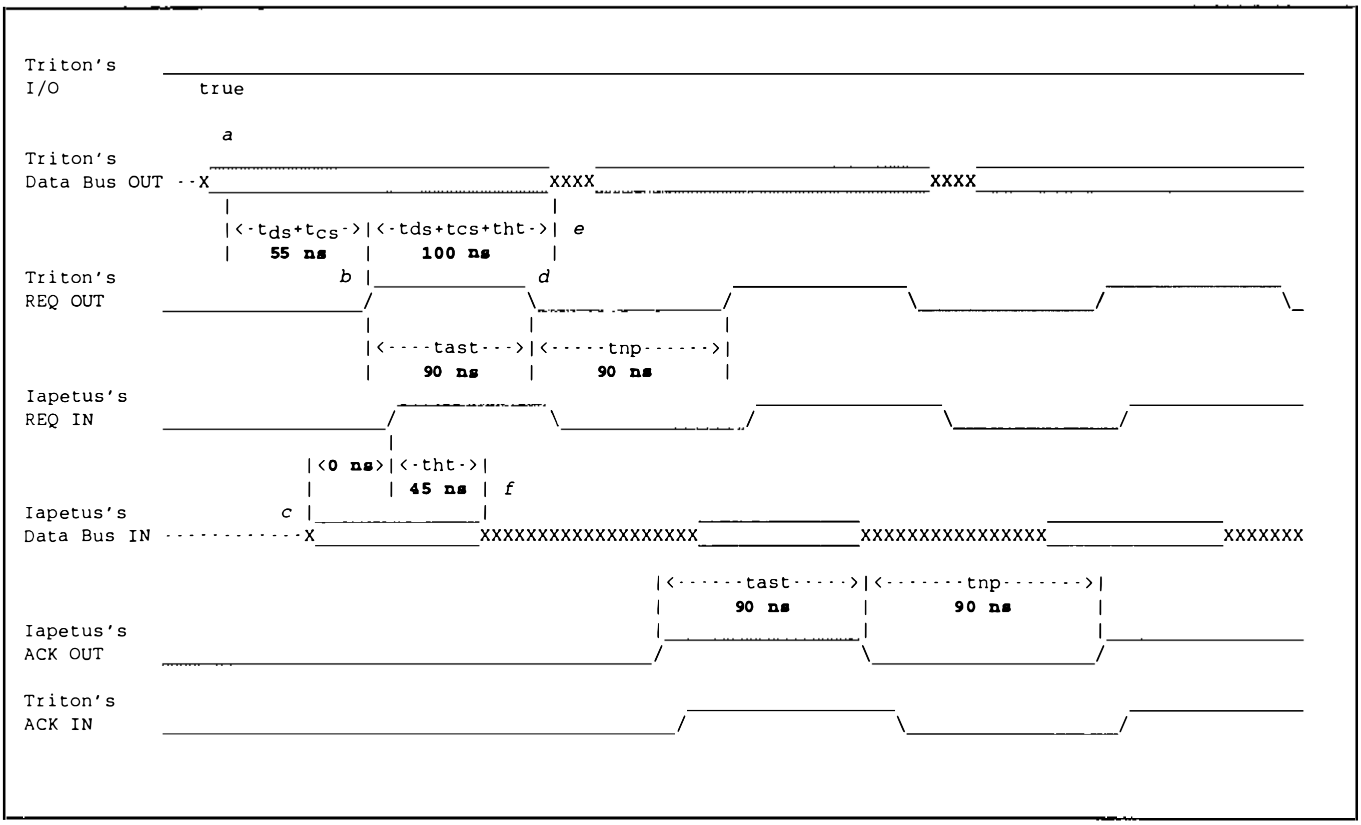


FIGURE 6: SYNCHRONOUS DATA TRANSFER IN - TARGET TO INITIATOR

The Details are shown in Figure 6:

(a) Triton begins by setting the data bus to the value of the first data byte (or word, see ***Wide Data Transfer***) to send.

(b) After a 55 nsec delay, Triton asserts the REQ signal true for no less than 90 nsec, after which he may negate REQ. The 55 nsec delay provides deskew time for the Initiator receiving latch or FIFO. The leading edge of the first REQ pulse defined this as a ***DATA IN Phase***. Triton continues sending out a number of REQ pulses (with data) appropriate to the amount of data that the Initiator can accept, which is the ***Synchronous Offset*** agreed to during ***Synchronous Data Transfer Negotiation***.

Note that Iapetus must be able to take what Triton sends, no matter what. This means that, no matter what circuitry Iapetus uses to receive data, he must always be ready to accept the data after the phase change to DATA IN Phase. The Target gives no warning to the Initiator until the first REQ pulse with data comes down the cable.

(c) After the signal propagates down the cable for a time, Iapetus detects the REQ signal going true at her input, which increments her Current Offset Counter and clocks the data into her Receive FIFO. Depending on the chip and other implementation details, Iapetus may be able to respond to the new phase immediately (particularly if the phase was expected next), or she may need to do some cleanup of the previous phase and bookkeeping before she can get started. In any case, the Current Offset Counter and Receive FIFO must accept the incoming REQ pulses and data to the limit of the Synchronous Offset.

Again, the set up time and hold time for the data is reduced. When it left Triton, there was 55 nsec of set up time between the data arriving and the leading edge of the ACK signal, and 100 nsec of hold time after the ACK. But when it reached Iapetus, there was no more lead time, and the hold time was 45 nsec. Like the transfer OUT, the 55 nsec that disappeared from both times is intended to account for skew in the circuit, and is made up of the two parts described there; the ***Deskew Delay*** (45 nsec) and the ***Cable Skew Delay*** (10 nsec).

(d) After a delay of 100 nsec from the leading edge of the ACK pulse, Triton is allowed to change the data bus. This delay provides hold time for the Initiator receiving latch or FIFO.

(e) When Iapetus is able to absorb the data from Triton, she begins returning ACK pulses to Triton. Each ACK pulse returned indicates to Triton that another REQ pulse and data may be sent. The ACK pulse decrements Triton's Current Offset Counter.

(f) Triton continues sending REQ pulses until he has sent all of the data. The leading edges of the REQ pulses must be separated by no less than time $t_{xp(t)}$, which is the ***Synchronous Transfer Period*** that Iapetus said she could handle during ***Synchronous Data Transfer Negotiation***. Iapetus continues sending ACK pulses until the Offset State Machine and transfer count reach zero, as described above. The leading edges of the ACK pulses must be separated by no less than time $t_{xp(i)}$, which is the Synchronous Transfer Period that Triton said he could handle during Synchronous Data Transfer Negotiation.

At the end of the Phase, the Initiator may extend the assertion of the ACK signal to prevent the Target from changing Phase for purposes of checking for errors and possibly creating the ***Attention Condition***.

The additional observations on Synchronous Data Transfer IN are the same as for OUT (see above). Table 22 shows the timing values used during Synchronous Data Transfer.

TABLE 22: TIMING VALUES USED DURING SYNCHRONOUS DATA TRANSFER

Symbol	Timing Name	MIN or MAX?	Time
t_{ast}	Assertion Period	MINIMUM	90 nsec
t_{ds}	Deskew Delay	MINIMUM	45 nsec
t_{cs}	Cable Skew Delay	MAX/MIN	10 nsec
t_{ht}	Hold Time	MINIMUM	45 nsec
t_{np}	Negation Period	MINIMUM	90 nsec

Synchronous Data Transfer Negotiation. Synchronous Data Transfer Negotiation (SDTN) is used to establish the data transfer method to be used between two ***SCSI Devices***. If two devices never engage in a SDTN, the default transfer method is ***Asynchronous Data Transfer***. After the SDTN process has completed, the two devices will (hopefully) have agreed to a set of parameters that result in a ***Synchronous Data Transfer***.

The specific purpose of SDTN is to provide a procedure where two devices can agree on a set of parameters for Synchronous Data Transfer. The two parameters are ***Synchronous Transfer Period*** and ***Synchronous Offset*** (these two topics are defined in their own sections below). The intent is to arrive at an agreement such that the maximum performance is achieved.

Note that the agreement is between two SCSI devices. This agreement is independent of Initiator and Target role, and is also independent of ***Logical Unit*** on each device. In other words, the same agreement is used in all of the following situations:

- Device A takes the Initiator role and Selects Device B (which then takes the Target role) and transfers data with Logical Unit #X.

- Device B takes the Target role and Reselects Device A (which then takes the Initiator role) and transfers data with Logical Unit #Y.

- Device B takes the Initiator role and Selects Device A (which then takes the Target role) and transfers data with Logical Unit #Y.

- Device A takes the Target role and Reselects Device B (which then takes the Initiator role) and transfers data with Logical Unit #X.

If there is a Device C on the bus, both Device A and Device B must each reach their own agreement with Device C.

Any device may begin the SDTN process at any time (see ***Etiquette***). Typically, the process is begun on the first ***Connection*** following a ***Hard Reset*** on either device. For example:

- An Initiator ***Connects*** to a Target. After the ***MESSAGE OUT Phase*** in which the ***IDENTIFY Message*** is sent by the Initiator, the Target changes to ***MESSAGE IN Phase*** and sends the ***Synchronous Data Transfer Request (SDTR)*** to begin the SDTN process.

- A Target ***Reconnects*** to an Initiator. During the MESSAGE IN Phase in which the IDENTIFY Message is sent by the Target, the Initiator creates the ***Attention Condition***. In response, the Target changes to MESSAGE OUT Phase, and the Initiator sends the SDTR message to begin the SDTN process.

You might also see the SDTN process begin just prior to the ***DATA Phase***, after the ***COMMAND Phase***, or after a ***Queue Tag Message***.

Diagram 41 and Diagram 42 illustrate how SCSI devices negotiate synchronous data transfer parameters. Note that the diagrams do not refer to Initiators and Targets, rather, they refer to the Originator and Responder.

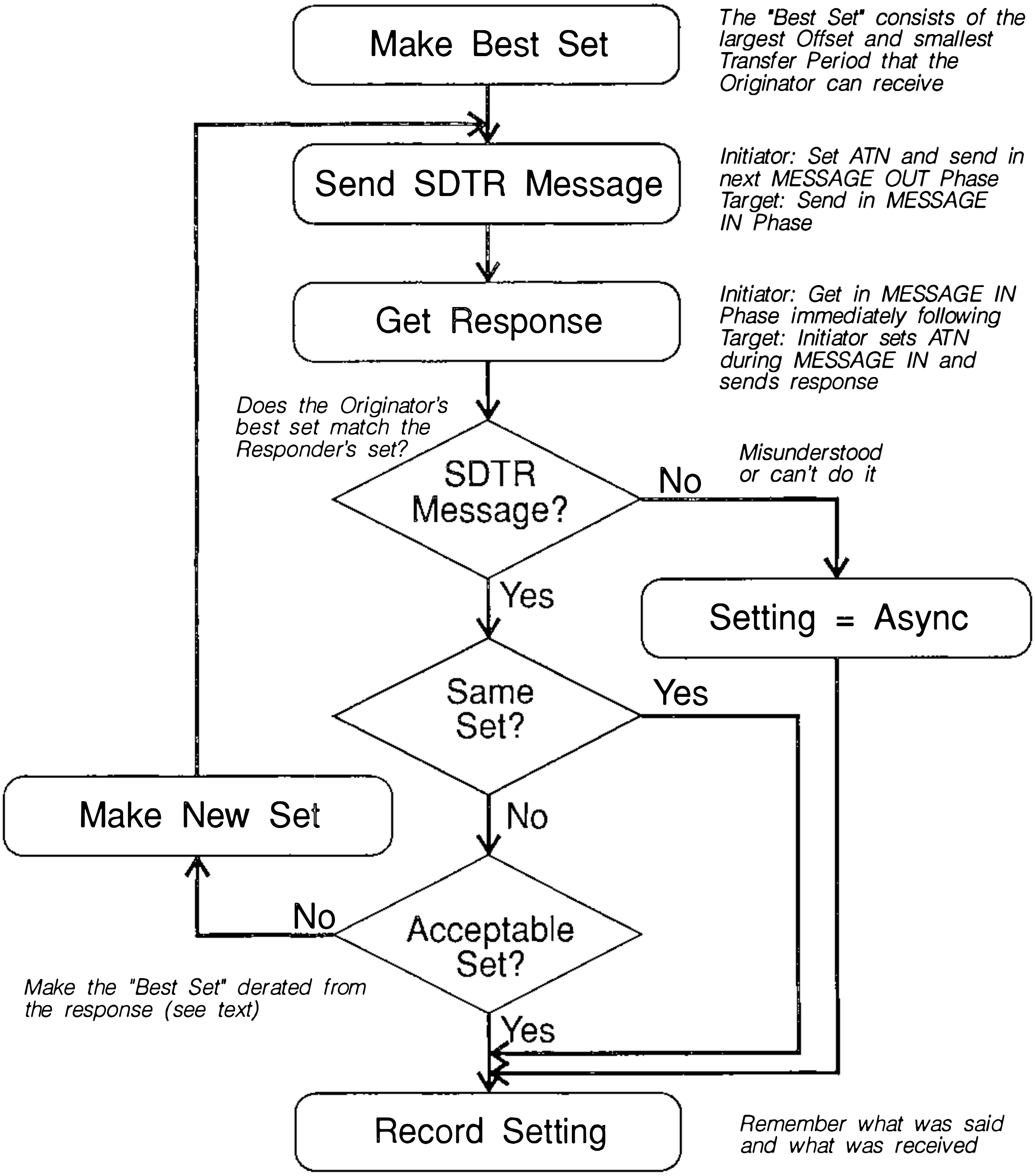

Make Best Set
The "Best Set" consists of the largest Offset and smallest Transfer Period that the Originator can receive
Send SDTR Message
Initiator: Set ATN and send in next MESSAGE OUT Phase
Target: Send in MESSAGE IN Phase
Get Response
Initiator: Get in MESSAGE IN Phase immediately following
Target: Initiator sets ATN during MESSAGE IN and sends response
Does the Originator's best set match the Responder's set?
SDTR Message?
No
Misunderstood or can't do it
Yes
Setting = Async
Same Set?
Yes
No
Make New Set
No
Acceptable Set?
Make the "Best Set" derated from the response (see text)
Yes
Record Setting
Remember what was said and what was received

DIAGRAM 41: SDTR FLOW DIAGRAM FOR ORIGINATORS

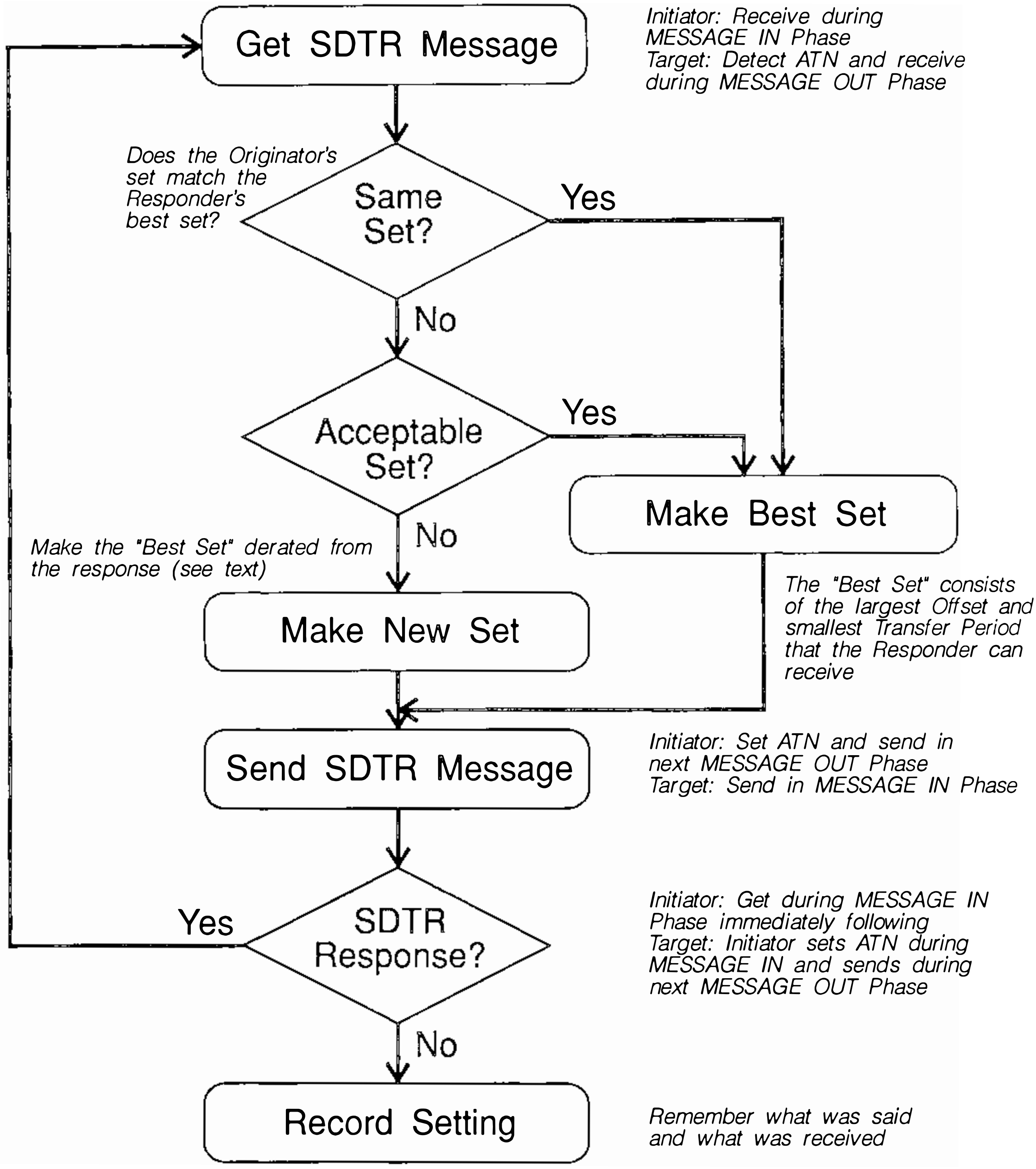

Get SDTR Message
Initiator: Receive during MESSAGE IN Phase
Target: Detect ATN and receive during MESSAGE OUT Phase
Does the Originator's set match the Responder's best set?
Same Set?
Yes
No
Acceptable Set?
Yes
No
Make Best Set
Make the "Best Set" derated from the response (see text)
Make New Set
The "Best Set" consists of the largest Offset and smallest Transfer Period that the Responder can receive
Send SDTR Message
Initiator: Set ATN and send in next MESSAGE OUT Phase
Target: Send in MESSAGE IN Phase
Yes
SDTR Response?
Initiator: Get during MESSAGE IN Phase immediately following
Target: Initiator sets ATN during MESSAGE IN and sends during next MESSAGE OUT Phase
No
Record Setting
Remember what was said and what was received

DIAGRAM 42: SDTR FLOW DIAGRAM FOR RESPONDERS

When a device decides it needs to begin the SDTN process (more on why it might decide that later), it first sets up its "Best Set" of synchronous transfer parameters. This Best Set is usually the maximum Offset and minimum Transfer Period that the device can accept. The Best Set is sent as part of the SDTR message. The device has now taken the role of the Originator in the SDTN process.

The other device, upon receiving the SDTR message from the Originator, becomes the Responder. In response, it can do one the following:

- If the parameters are acceptable, and also happen to be equivalent to the Responder's Best Set, it repeats the same SDTR message back to the Originator. Parameters are acceptable in this case when the Responder is able to perform synchronous data transfer with the Originator, and:

    - Accept a ***REQ Signal*** or ***ACK Signal*** that meets the maximum Offset and minimum Transfer Period specified by the Originator; and

    - Not exceed the Originator's maximum Offset and minimum Transfer Period when sending the REQ Signal or ACK Signal.

- If the parameters are acceptable, but they are not equivalent to the Responder's Best Set, the Responder creates an SDTR message with the Responder's Best Set that is also less aggressive than the Originator's parameters, and sends it to the Originator. "Less aggressive" means that one or both of the following are true:

    - The Offset is less than the Originator's Offset;

    - The Transfer Period is greater than the Originator's Transfer Period.

  The parameters are acceptable in this case when the Responder is able to perform synchronous data transfer with the Originator, and not exceed the Originator's maximum Offset and minimum Transfer Period when sending the REQ Signal or ACK Signal.

- If the parameters are not acceptable, the Responder creates an SDTR message with the Offset set to zero, and sends it to the Originator. With this response, the Originator and Responder are agreeing that Asynchronous Data Transfer is the data transfer method that will be used.

  The parameters are not acceptable in this situation because the Responder is able to perform synchronous data transfer with the Originator, but is unable to proceed without exceeding the Originator's maximum Offset and/or minimum Transfer Period when sending the REQ Signal or ACK Signal. In practice, modern SCSI devices are able handle a wide range of parameters. This case can only happen if the Responder is not flexible enough to come down to the level of the Originator.

- If the Responder hasn't a clue about Synchronous Data Transfer, or just doesn't support it, it will probably respond to the SDTR Message with a ***MESSAGE REJECT Message***. In this case the Originator should get the hint and stick to Asynchronous Data Transfer with the Responder from now on.

When is Negotiation Required? A device should originate an SDTN whenever it has no record of a previous negotiation. This, of course, implies that a device that completes an SDTN should keep a record of it. This record is maintained for each SCSI Device on the bus, and could be represented as a simple data structure consisting of the following elements:

- SDTN flag, where:
  - 0 = No SDTN completed with that device.
  - 1 = SDTN has been completed with that device.
- Transfer Period, if the SDTN flag is 1.
- REQ/ACK Offset, if the SDTN flag is 1.

The ways that a device could lose the value stored in this data structure are somewhat unique to each device, but we think we can all agree that at least the following events would clear the data structure:

- A power-on reset.

- A ***Hard Reset***.

- A ***BUS DEVICE RESET Message***.

SYNCHRONOUS DATA TRANSFER REQUEST Message.

The SYNCHRONOUS DATA TRANSFER REQUEST (SDTR) Message is used to establish the parameters that will be used during a ***Synchronous Data Transfer***. This is an ***Extended Message***:

Bit Byte	7	6	5	4	3	2	1	0
0	Message Code = 01 hex							
1	Additional Message Length = 03 hex							
2	Extended Message Code = 01 hex							
3	Transfer Period							
4	REQ/ACK Offset							

Besides the message code, this message also carries the ***Synchronous Transfer Period*** (byte 3) and the ***Synchronous Offset*** or "REQ/ACK Offset" (byte 4).

The Transfer Period is specified as a multiple of four. For example, a value of 50 (32 hex) indicates a 200 nsec Transfer Period. If ***Fast Data Transfer*** timing is used, the minimum value for byte 3 is 25 (19 hex); otherwise, the minimum value is 50 (32 hex).

The REQ/ACK Offset is specified as the number of pulses of the ***REQ Signal*** that the Target may issue beyond the number of pulses of the ***ACK Signal*** is has received. If a device sets the REQ/ACK Offset to zero, it is requesting an ***Asynchronous Data Transfer***. If a device sets the REQ/ACK Offset to FF hex, it is capable of handling an unlimited Synchronous Offset.

See ***Synchronous Data Transfer Negotiation*** for a complete description of the use of this message.

Summary of Use: The SDTR Message is sent by any SCSI Device to establish the parameters to be used during Synchronous Data Transfer.

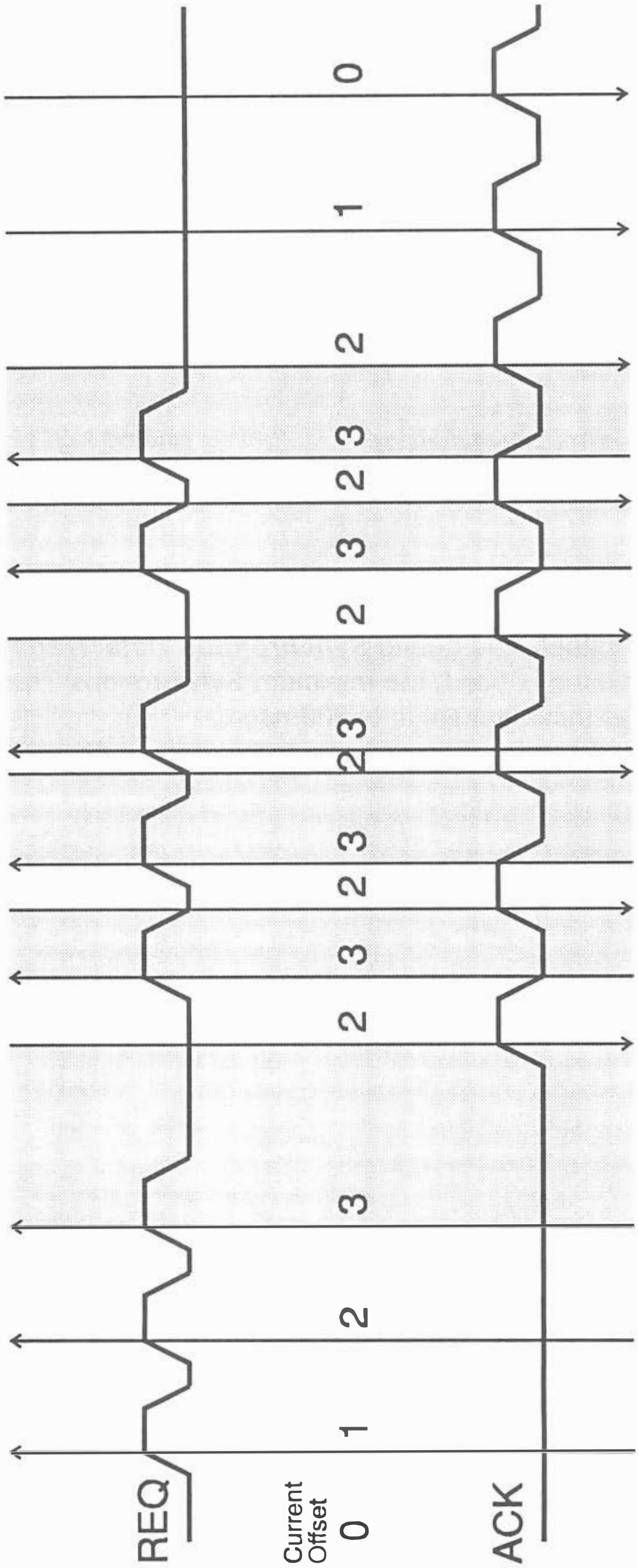
Synchronous Offset. The Offset specifies the number of pulses on the ***REQ Signal*** that the Target may have outstanding before receiving a pulse on the ***ACK Signal***.

We'll say it another way: The REQ Pulses of the Target may lead the ACK pulses of the Initiator by a number less than or equal to the Synchronous Offset.

Or how about: The "Current Offset" is the number of REQ pulses issued by the Target minus the number of ACK pulses issued by the Initiator. The Current Offset must not exceed the Synchronous Offset at any instant during a ***Synchronous Data Transfer***.

OK, how about a picture? The example in Diagram 43 shows a Synchronous Offset of 3. The Current Offset is shown in the space between the REQ Signal and ACK Signal. The sequence begins when the Target issues three REQ pulses. At that point, the Current Offset reaches 3 and the Target is held from issuing any more REQ pulses (as indicated by the shaded areas of the example).

Some time later, the Initiator starts issuing ACK pulses. With the rising edge of ACK, the Current Offset decrements. The Target is then allowed to issue another REQ pulse, which increments the Current Offset. This incrementing by REQ and decrementing by ACK continues until the Target has sent all of its REQ pulses. The Initiator sends all of its ACK pulses and the Current Offset returns to zero.



The shaded areas represent times where the Target must pause and wait for an Initiator response

DIAGRAM 43: SYNCHRONOUS OFFSET

Synchronous Transfer Period. The Synchronous Transfer Period specifies how close together pulses on the ***REQ Signal*** or the ***ACK Signal*** can be during ***Synchronous Data Transfer***. The Transfer Period is the minimum time between leading edges of pulses on the REQ Signal, or the ACK signal. Note that there is <u>no</u> time specification between any edges of the REQ and ACK pulses. Figure 7 illustrates the Transfer Period.

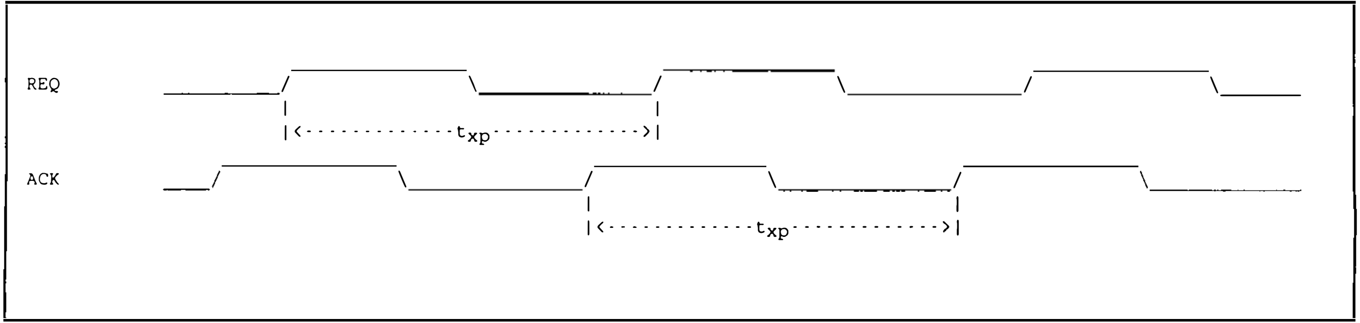


FIGURE 7: SYNCHRONOUS TRANSFER PERIOD

The Transfer Period is established during ***Synchronous Data Transfer Negotiation***. If ***Fast Data Transfer*** timing is used, the minimum Synchronous Transfer Period is 100 nsec; otherwise, the minimum value is 200 nsec.

Tagged Queuing.
Target.
Target Routine.
Target Routine Number (TRN).
TERMINATE I/O PROCESS Message.
Termination.
Terminator Power (TERMPWR).
Transfer Period.
True.

Tagged Queuing. Tagged Queuing is a ***Queue*** for ***I/O Processes*** for a ***Logical Unit*** which uses ***Queue Tags***. Contrast to ***Untagged Queuing***. Tagged Queuing allows for several I/O Processes to be active from a single Initiator for that Logical Unit. See ***Queue*** for the big description....

Target. From the dictionary: "...any object that is shot at...". Well, that works! On the ***SCSI Bus***, the Target receives a bus transaction (which is called an ***I/O Process***) "shot" at it by an ***Initiator***.

In general, the Target is a role assumed by any ***SCSI Device*** able to receive a ***SCSI Command*** from another SCSI Device that can assume the Initiator role. In traditional terms, the Target is usually part of, or attached to, a peripheral of some sort; usually referred to as a ***Controller*** or a ***Peripheral Device***. A ***Host Adapter*** may also assume the Target role to receive an ***Asynchronous Event Notification (AEN)*** or data from another Host Adapter acting as an Initiator.

Diagram 44 shows the Target role in a system, and also shows which signals on the SCSI Bus the Target drives and receives. The following bullets give a flavor of the Target's contribution to the execution of an I/O Process:

- A Target responds to ***Selection*** by an Initiator to begin an I/O Process.

- After ***SELECTION Phase***, the Target is in charge of changing to different ***Bus Phases***. The Initiator must respond to these Phases and transfer information as requested by the Target.

- The Target must respond to the ***Attention Condition*** when it is created by an Initiator.

- The Target is responsible for ***Error Recovery***. The Initiator can request a recovery through the Attention Condition, but the Target controls the actual recovery process. This includes error recovery within the ***Message System***.

- The Target manages the Initiator's ***Pointers*** during the ***Current I/O Process***. The Initiator uses Pointers to maintain the current state of Command, ***Status***, and Data transfer.

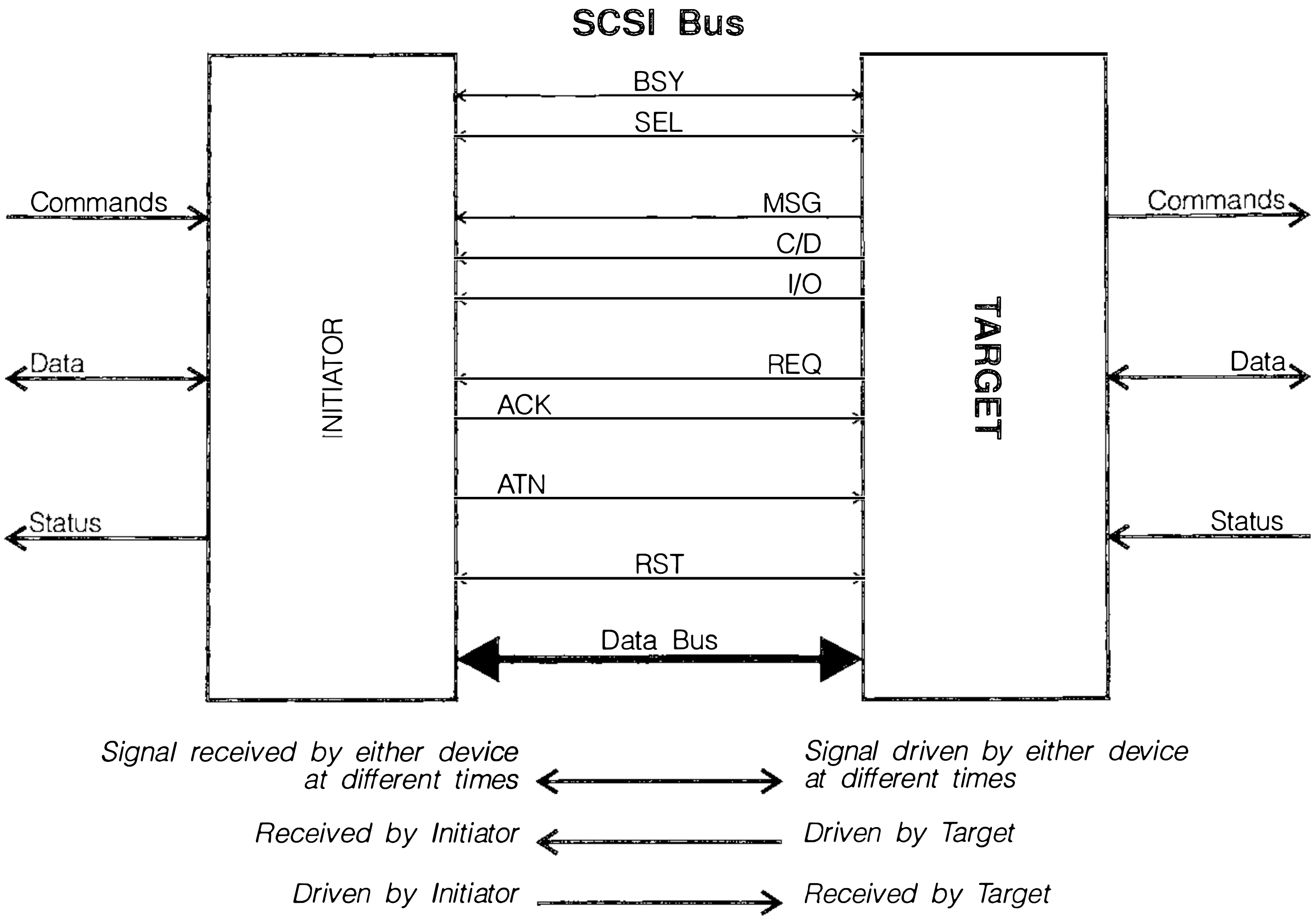


DIAGRAM 44: TARGET ROLE FOR SCSI DEVICES

Target Routine. A Target Routine is an arbitrary "unit" which represents the Target itself. The Initiator can issue commands directly to a Target Routine instead of a ***Logical Unit***. As of ***SCSI-2***, the only commands that can be issued to a Target Routine are:

- INQUIRY: This is used to return data about the Target that is not appropriate to any Logical Unit.
- REQUEST SENSE: This is used mostly to return error information regarding the INQUIRY command....

The use of a Target Routine is optional. Most Target implementations return information about the Target as part of the information for each Logical Unit.

Target Routine Number (TRN). The Target Routine Number (TRN) is the address of the ***Target Routine*** within a Target. If there is a Target Routine there must be a Target Routine #0.

The TRN is used by the Initiator and Target to specify that a Target Routine is being referred to. The ***IDENTIFY Message*** is used to exchange the TRN between the Initiator and Target. The exchange of TRN is part of establishing an I_T_R ***Nexus*** between two devices.

TERMINATE I/O PROCESS Message. The TERMINATE I/O PROCESS Message is used by the Initiator to "interrupt" and terminate an ***I/O Process***. This is a single byte Message:

Bit Byte	7	6	5	4	3	2	1	0
0	Message Code = 11 hex							

When the Target receives this message, it is supposed to stop the I/O Process as soon as possible. It is also supposed to stop the I/O Process without "breaking" anything: No ***Logical Blocks*** or any other internal data structures are to be left in a damaged state by a TERMINATE I/O PROCESS Message.

This ***Message*** is different from the ***ABORT Message*** and ***ABORT TAG Message***; those Messages require the Target to stop immediately. An ABORT can leave a Logical Block only partially written, or some MODE SELECT data structure corrupted.

Table 25 shows how the Target should respond to the TERMINATE I/O PROCESS Message. The table is in the order that the conditions should be evaluated. Note that some of the responses call for "Sense Data" to be returned. The meaning of these Sense Data terms (such as "residue") is given in the ***SCSI-2*** standard, and in later Volumes of this Encyclopedia.

TABLE 25: TERMINATE I/O PROCESS RESPONSES

The Target receives TERMINATE I/O PROCESS and:	Then the Target response is:
The Nexus for the Current I/O Process is not "active" or Queued; i.e., no Command has been transferred for the Nexus.	Return COMMAND TERMINATED Status and then a COMMAND COMPLETE Message. SENSE Data is set as follows: Valid = 0 Sense Key = NO SENSE Sense Code = I/O PROCESS TERMINATED
It hasn't started executing the I/O Process and/or the I/O Process is Queued.	Return COMMAND TERMINATED Status and then a COMMAND COMPLETE Message. SENSE Data is set as follows: Valid = 0 Sense Key = NO SENSE Sense Code = I/O PROCESS TERMINATED
It can't stop the I/O Process.	Return the MESSAGE REJECT Message and continue.
The I/O Process is already done; e.g., all Logical Blocks have been transferred.	Ignore the TERMINATE I/O PROCESS Message.
The I/O Process has an error condition.	Ignore the TERMINATE I/O PROCESS Message.
The Command has started executing and there is a DATA Phase required to execute it.	Return COMMAND TERMINATED Status and then a COMMAND COMPLETE Message. SENSE Data is set as follows: Valid = 1 Information Field = "residue" (see text) Sense Key = NO SENSE Sense Code = I/O PROCESS TERMINATED

Summary of Use: The TERMINATE I/O PROCESS Message is sent only by an Initiator to cause the Target to end an I/O Process in an orderly manner that does not corrupt the storage medium.

Termination. Any transmission line, as we all learned in school (or somewhere...), must be terminated to prevent reflections from degrading the signal. Well, a SCSI ***Cable*** is a transmission line, and therefore, for proper operation, it must be terminated at both ends.

The design of the termination for the ***Single-Ended Interface*** is different than that for the ***Differential Interface***. However, they both require the Thevenin equivalent resistance looking into the terminator to be matched to the characteristic impedance of the Cable (see ***Cable*** for the definition of a good "match"). Diagram 45 shows the Thevenin model of SCSI Bus Termination. When $Z_t = Z_o$, then reflections back from the terminator are zero. For each interface and terminator type, V_t and Z_t are defined as shown in Table 26.

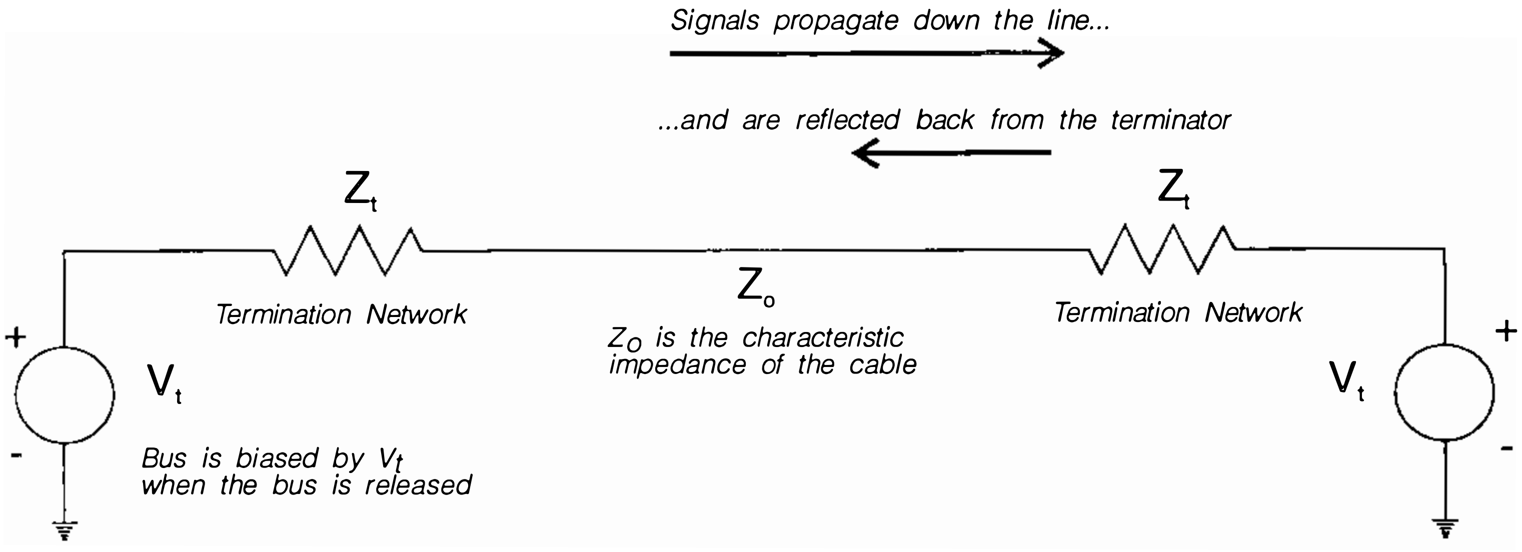
TABLE 26: TERMINATOR THEVENIN VOLTAGE AND IMPEDANCE

Interface Type	Thevenin Voltage (Open Circuit) (Terminator Voltage varies from 4.00 V to 5.25 V)	Thevenin Impedance
Single-Ended "Preferred"	2.85 V	110 Ω
Single-Ended "Old Style"	2.40 V to 3.15 V	132 Ω
Differential +Signal	1.64 V to 2.14 V	122 Ω
Differential -Signal	2.36 V to 3.10 V	122 Ω

Note that the impedance will vary dependent on the tolerance of the resistors used to create it. The voltage variation is function both of the resistor tolerance and the level of the ***Terminator Power*** voltage available at the terminator. The table shows the voltage variation for nominal resistor values that can occur due to possible variations in Terminator Power.

Note also that the table shows Terminator Power as low as 4.0 Volts for the Single-Ended termination. While ***SCSI-2*** increases the minimum Terminator Power voltage to 4.25 Volts, there is the potential for a new device to encounter the ***SCSI-1*** minimum of 4.0 Volts.

DIAGRAM 45: THEVENIN EQUIVALENT TERMINATION


Signals propagate down the line...
...and are reflected back from the terminator
Z_t
Z_t
Termination Network
Z_o
Termination Network
Z_O is the characteristic impedance of the cable
+
V_t
-
Bus is biased by V_t when the bus is released
+
V_t
-

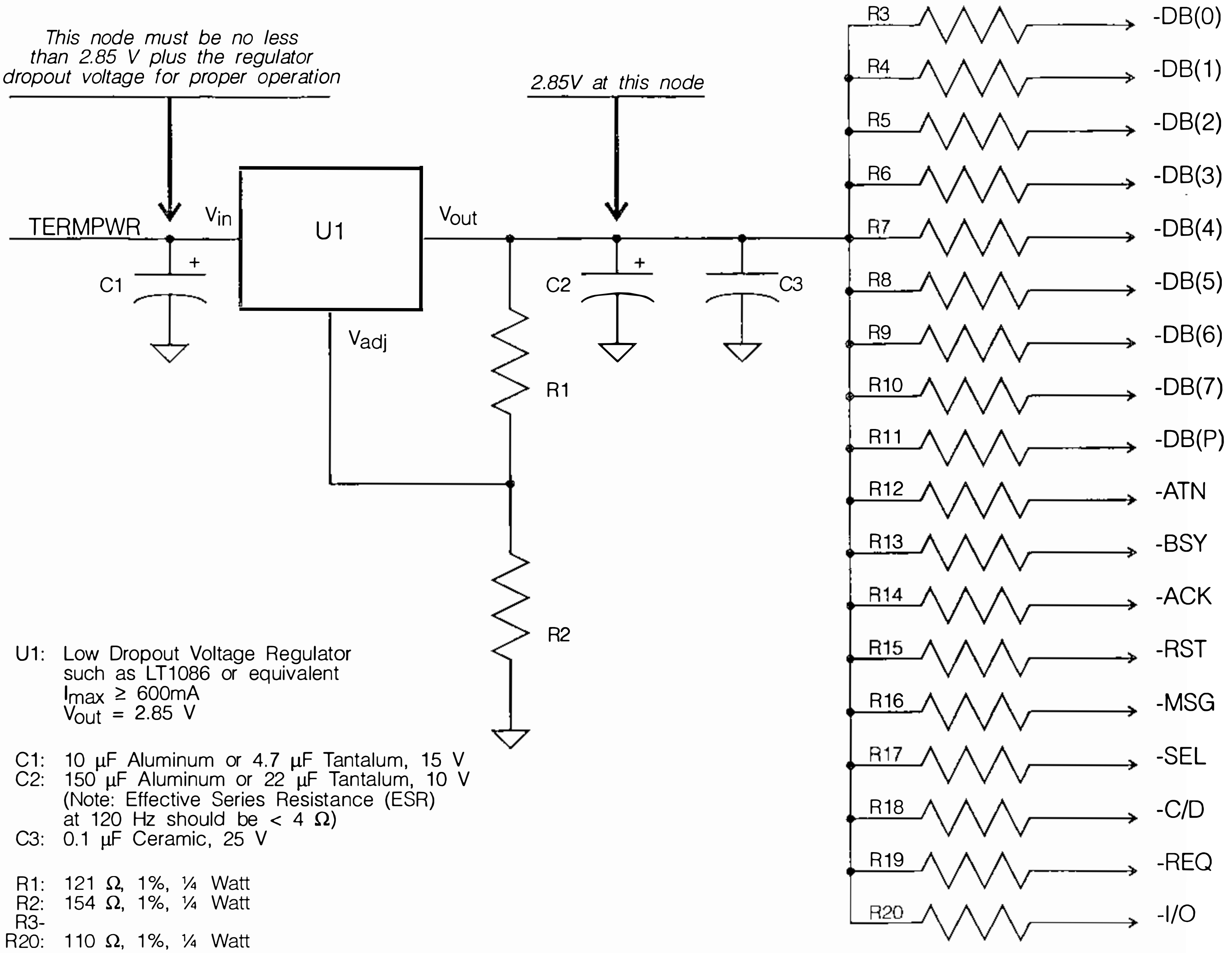
The "Preferred" Single-Ended Termination is shown in Diagram 46. This terminator directly models the Thevenin circuit. A "low dropout voltage" regulator is used to create the bias voltage for the bus signal. This bias voltage is applied across each termination resistor to bias each bus signal.

Why "Preferred"? A little history will explain. The original ***SASI*** terminator is shown in Diagram 47; this is the "Old Style" Terminator. This terminator worked just fine for the data transfer rates used on a SASI Bus. The problem with the old terminator is that it has very little worst case margin for the bias voltage, and the impedance is too high for the high density Cables and ***Connectors*** typically used today. The new "Preferred" terminator was devised to stabilize the bias voltage high enough and also to present a lower termination impedance to the Cable.

The bias voltage is a function of R1 and R2 and the characteristics of the voltage regulator; typically 2.85 V. The termination impedance is equal to the value of resistors R3-R20; 110 Ω.

Using the "Preferred" terminator at both ends of the cable is best, but it turns out that using a "Preferred" terminator at one end of the cable, and an "Old Style" terminator at the other end, is far better than using an "Old Style" at both ends (yes, they can be mixed).

DIAGRAM 46: SINGLE-ENDED "PREFERRED" TERMINATION



U1: Low Dropout Voltage Regulator
such as LT1086 or equivalent
$I_{max} \geq$ 600mA
V_{out} = 2.85 V

C1: 10 µF Aluminum or 4.7 µF Tantalum, 15 V
C2: 150 µF Aluminum or 22 µF Tantalum, 10 V
(Note: Effective Series Resistance (ESR)
at 120 Hz should be < 4 Ω)
C3: 0.1 µF Ceramic, 25 V

R1: 121 Ω, 1%, ¼ Watt
R2: 154 Ω, 1%, ¼ Watt
R3-
R20: 110 Ω, 1%, ¼ Watt

The "Old Style" terminator in Diagram 47 is built from a resistor divider. The bias voltage is the divider voltage given by the equation:

$$V_t = V_{term} \frac{R_{330}}{R_{330} + R_{220}} = 0.6\,V_{term}(nominal)$$

As you can see, the actual bias voltage can vary based on the Terminator Power voltage (V_{term}) and the resistor tolerance of the two resistors. For example, with V_{term} = 4.25 V and nominal resistor values, V_t = 2.55 V.

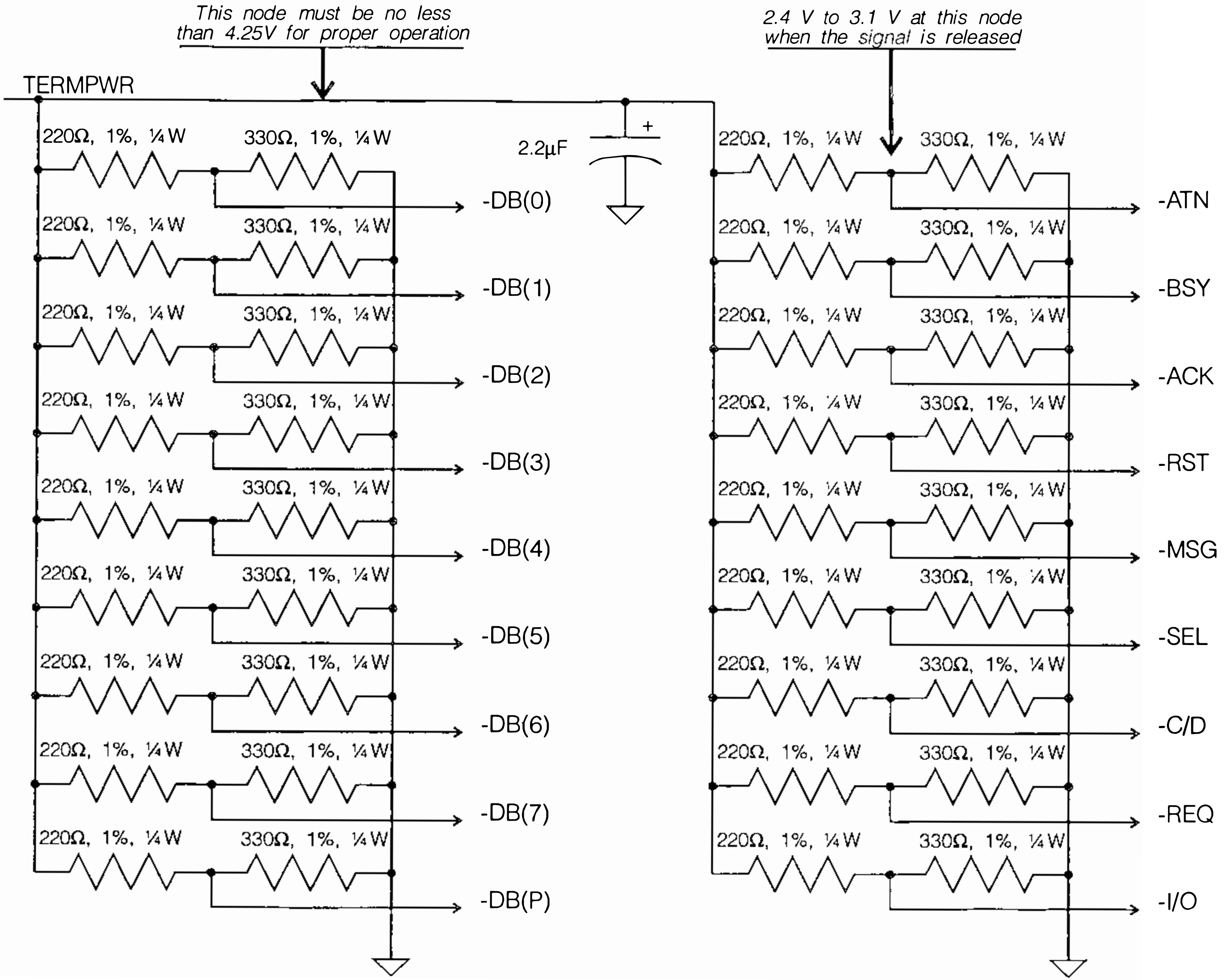
The termination impedance is given by:

$$Z_t = \frac{1}{\left(\frac{1}{R_{220}} + \frac{1}{R_{330}}\right)} = 132\Omega(nominal)$$

Like the bias voltage, the impedance can vary as the resistor tolerance varies. We recommend using 1% tolerance resistors if you must use this terminator.

An additional termination method was presented to the ***X3T9.2 Committee*** just before our publication deadline. Please see ***Forced Perfect Termination (FPT)*** for a quick discussion of another alternative.

DIAGRAM 47: SINGLE-ENDED "OLD STYLE" TERMINATION



The termination for the Differential Interface is shown in Diagram 48. Note that it is similar in design to the "Old Style" Single Ended Terminator. In principle, it is. It is adequate for the Differential Interface because this interface is less sensitive to the actual bias voltage on the bus. It works as long as the -Signal is greater than the +Signal. Also, the cable has a higher impedance when used for the Differential Interface, because all those grounds in the Single-Ended Interface cause the cable to have a lower impedance.

The terminator is built from a resistor divider. The bias voltage on each signal (+Signal and -Signal) is the divider voltage given by the equations:

$$V_{t-} = V_{term}\frac{R_{330}+R_{150}}{R_{330}+R_{150}+R_{330}} = 0.59\,V_{term}(nominal)$$

$$V_{t+} = V_{term}\frac{R_{330}}{R_{330}+R_{150}+R_{330}} = 0.41\,V_{term}(nominal)$$

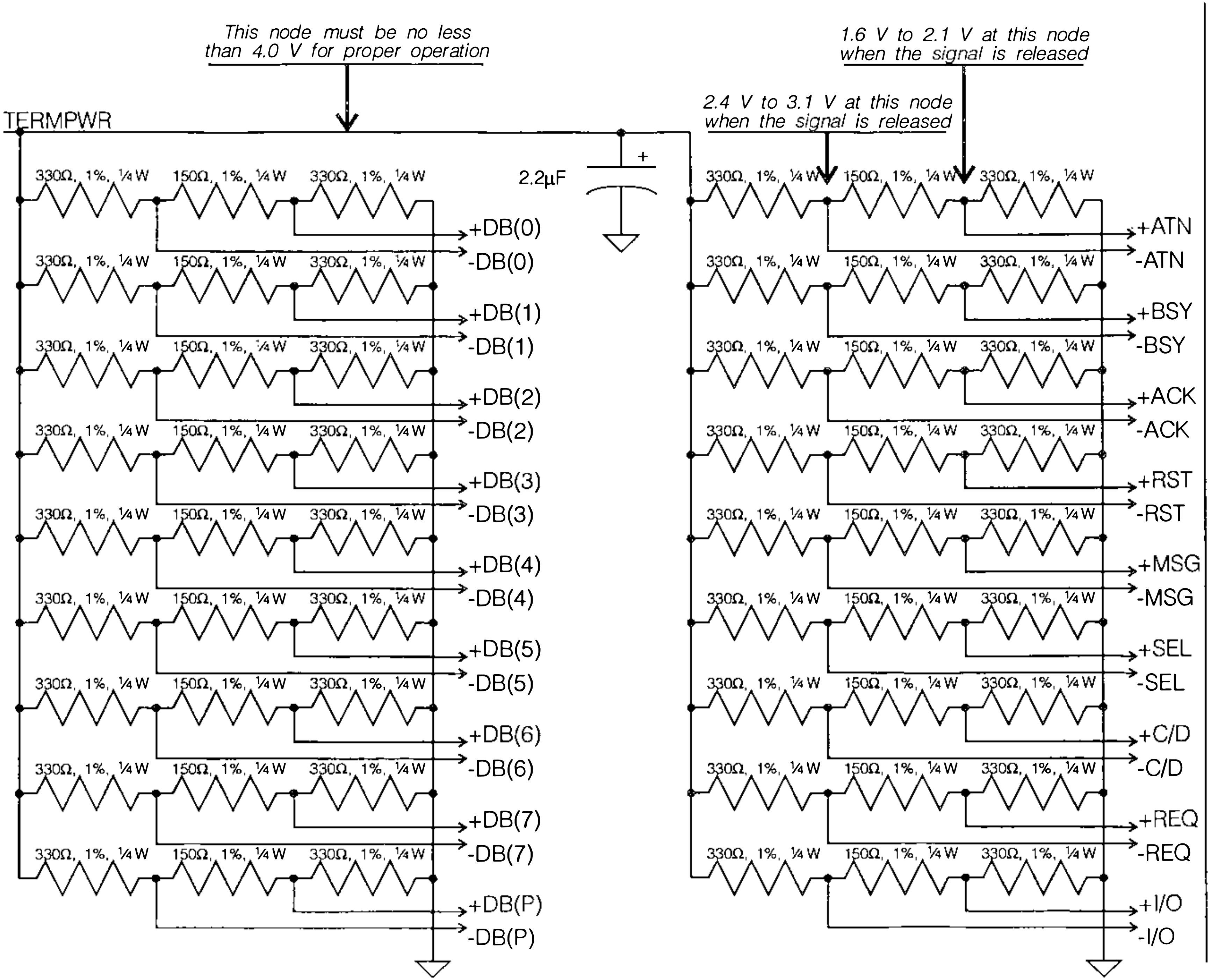
As before, the actual bias voltage can vary based on the Terminator Power voltage (V_{term}) and the resistor tolerance of the three resistors. For example, with V_{term} = 4.0 V and nominal resistor values, V_{t+} = 2.4 V and V_{t-} = 1.6 V.

The termination impedance is actually measured <u>across</u> the +Signal and -Signal, and is given by:

$$Z_t = \frac{1}{\left(\frac{1}{R_{150}}+\frac{1}{R_{330}+R_{330}}\right)} = 122\Omega(nominal)$$

Like the bias voltage, the impedance can vary as the resistor tolerance varies. We recommend using 1% tolerance resistors.

DIAGRAM 48: DIFFERENTIAL TERMINATION



Terminator Power (TERMPWR). The TERMPWR signal (and it is a signal!) supplies the power necessary for the active bus ***Termination*** at each end of the ***Cable***. Depending on the interface option (***Single-Ended Interface*** or ***Differential Interface***) and the Cable type (***A Cable***, ***B Cable***, or ***P Cable***) the TERMPWR signal is carried on one or more physical conductors. Terminator Power on the B Cable is called TERMPWRB.

TERMPWR has the characteristics shown in Table 27. The table shows the allowable voltage range for TERMPWR, and the minimum source drive current requirement (which is actually the maximum the device is expected to support). Note that these are measured at the device supplying TERMPWR.

TABLE 27: TERMINATOR POWER VOLTAGE AND CURRENT

		Interface Option	
		Single-Ended Interface	Differential Interface
Cable Type	A Cable	4.25 V to 5.25 V DC 900 mA (minimum)	4.00 V to 5.25 V DC 600 mA (minimum)
	B Cable	4.25 V to 5.25 V DC 1500 mA (minimum)	4.00 V to 5.25 V DC 1000 mA (minimum)
	P Cable	4.25 V to 5.25 V DC 1500 mA (minimum)	4.00 V to 5.25 V DC 1000 mA (minimum)

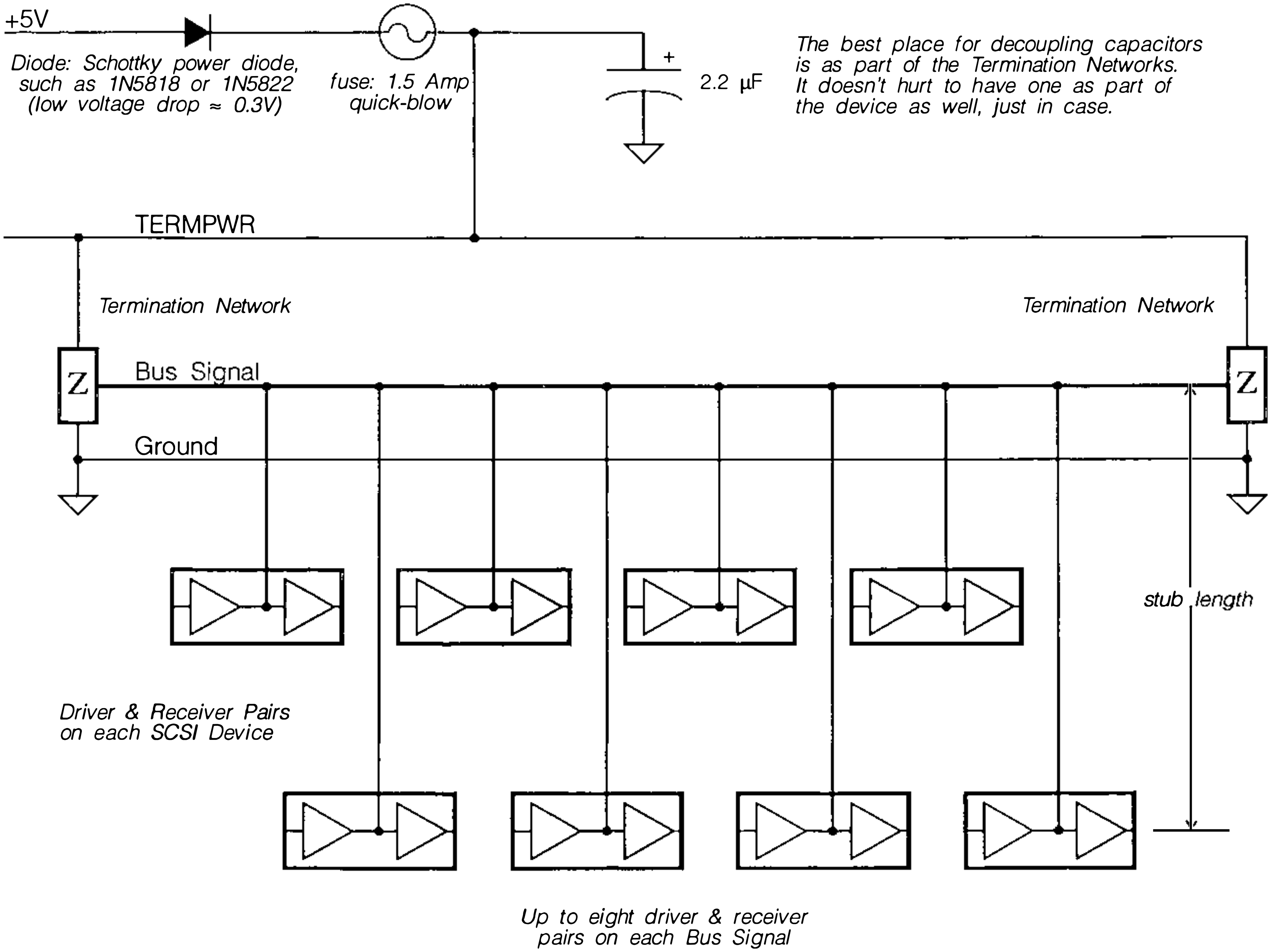
TERMPWR is supplied by any or all ***SCSI Devices*** on the Cable. Initiators are required to be able to supply TERMPWR; Targets may or may not be able to supply it. See Diagram 49 for an example schematic. In general, the following guidelines apply when supplying TERMPWR to the Cable:

- TERMPWR should be supplied through some circuit which protects the power supply of the SCSI Device from backflow current from other devices also supplying TERMPWR. Happily, a diode meets this criteria. The best diode to use is a diode that has a minimum voltage drop when forward biased, such as a Schottky power diode capable of handling the minimum current requirement. A low voltage drop is desirable because it improves noise margin on each signal line (particularly when using the "Old Style" Termination). The diode circuit is shown as part of Diagram 49.

- It is also a good idea to locate a fuse after the diode. A 1.5 Amp quick-blow fuse will do the trick (use 2 Amp for TERMPWRB).

(...more...)

Diagram 49: Termination Power Schematic


+5V
Diode: Schottky power diode, such as 1N5818 or 1N5822 (low voltage drop ≈ 0.3V)
fuse: 1.5 Amp quick-blow
+
2.2 µF
TERMPWR
Termination Network
Termination Network
Z
Z
Bus Signal
Ground
stub length
Driver & Receiver Pairs on each SCSI Device
Up to eight driver & receiver pairs on each Bus Signal

The best place for decoupling capacitors is as part of the Termination Networks. It doesn't hurt to have one as part of the device as well, just in case.

- The TERMPWR signal can fluctuate due to instantaneous changes in the current drawn through the Termination as signals are asserted, negated, and released. To smooth out these fluctuations, a 2.2 µF tantalum capacitor should be installed on TERMPWR at each Terminator. If you can't do that (for instance, you have old encapsulated terminators), install the capacitor at the TERMPWR pin on the connector on each SCSI Device.

- If you can, supply TERMPWR as high as possible. The Cable resistance can drop a lot of voltage from TERMPWR by the time it reaches the Terminator. If you can afford the extravagance of power and money, use a 3-terminal regulator to supply 5.0V or 5.25V from a higher voltage source (like 12V). This improves voltage margins greatly, as described in the section on Termination.

- Don't have too many devices supplying TERMPWR to the same wire. You might end up exceeding local safety regulations. On a heavily loaded bus, disable the TERMPWR supply on some devices if regulations are exceeded.

- The best placement for TERMPWR sources is as close to the Terminators as possible. This can minimize the voltage drops that occur as TERMPWR propagates the length of the Cable (can approach one volt!).

Why Terminator Power? Why not just power terminators directly from the devices which contain the Terminators or are located closest to them? Using local power works just fine, actually, in most cases. The reason for the more complicated system described above goes back to the SCSI philosophy of shared peripherals: The bus should be able to operate no matter which devices are powered up. For example, consider a three device system, with two PCs sharing a single SCSI printer. If one of the PCs was the only device that could power the Termination, and it was powered off, then the other PC could not use the printer.

Timing. See ***Bus Timing***.

Transfer Period. *ttp = set by SYNCHRONOUS DATA TRANSFER REQUEST Message.* See ***Synchronous Transfer Period***.

True. "Not ***False***", of course! In SCSI, a signal that is ***Asserted*** is in the True state. The definition of Asserted is specific to the Interface type, either ***Single-Ended*** or ***Differential***. The following are equivalent from a logical point of view in SCSI:

- "1"
- One
- True
- Asserted

Unexpected BUS FREE Phase.
Unit Attention Condition.
Untagged Queuing.

Unexpected BUS FREE Phase. The Unexpected ***BUS FREE Phase*** is the Target's "last way" to indicate an error to the Initiator. This method is to be used only if the Target has attempted to perform ***Error Recovery*** without success, or it cannot complete a ***STATUS Phase*** properly. In those cases, the only recourse for the Target is to "drop the bus"; in other words, ***Release*** the ***BSY Signal***.

Targets should not use Unexpected BUS FREE Phase to indicate an error if there is another way, such as STATUS Phase or the ***Message System***. If Unexpected BUS FREE Phase is necessary, something may be broken.

When the Target does an Unexpected BUS FREE Phase, it has the option of creating SENSE Data to describe the failure; in other words, the ***Contingent Allegiance Condition***. It is entirely up to the Target whether to do this:

- The Target can decide to be "good" and always create SENSE Data just in case the Initiator wants to know why the Target went away.

- On the other hand, if something so awful happened that the Target couldn't use normal ***Status*** and/or ***Messages***, what's the point of creating data that can't be transferred?

It's your call!

Unit Attention Condition. The Unit Attention ***Condition*** is used by the Target to inform the Initiator of an important change in the operating environment of the Target and the associated ***Logical Unit***. Let's look at the operating environment of the Target:

- There is a particular storage medium installed.
- There is a set of parameters that the Initiator can control via the MODE SELECT Command. These parameters tell the Target, among other things, how to handle errors, when to ***Disconnect*** and ***Reconnect***, and how to handle various other high level SCSI functions.
- The Target is executing a particular microcode program in its control microprocessor.
- The Target has a ***Queue*** of commands, or has commands pending with several Initiators.
- There is data that the Initiator may read (via an INQUIRY Command), but may not change.
- The mechanical or electrical aspects of the Target can change.
- Anything else you can think of that seems appropriate.

The Target is expected to create the Unit Attention Condition whenever any of the above aspects of the operating environments changes:

- The installed storage medium changes. This can be caused by an operator popping in a new cartridge, or a change effected by a "jukebox".

- The MODE SELECT Parameters change. This can be caused by a MODE SELECT Command from another Initiator, or by a ***Hard Reset***.

- The Target loads a different set of microcode for execution. This does not include swapping in an overlay to perform a particular function. This can be caused by a CHANGE DEFINITION Command or a WRITE BUFFER Command to download the microcode. It can also be caused by a Hard Reset.

- One ***SCSI Device*** causes the Commands of another device to be lost by the Target. This can be caused by a Hard Reset, by a ***CLEAR QUEUE Message***, or by a ***BUS DEVICE RESET Message***.

- The INQUIRY Command data can be changed by anything that can change the microcode (see above). It can also be changed when a Logical Unit becomes "ready". Many Targets load their full set of INQUIRY data from the storage medium.

- The types of mechanical aspects that can change include a disk spindle that has changed its synchronization state relative to other spindles in a system. Or, the medium is reformatted.

The Target creates the Unit Attention Condition for each Logical Unit that the change affects. For example, when a Hard Reset occurs, the Target should create the Unit Attention for all valid Logical Units on the Target. The Target should not create the Unit Attention Condition for Logical Units that are not valid.

The Target also creates the Unit Attention Condition for each SCSI Device that should be informed of the change. In theory, all of the other possible devices could be Initiators, and therefore can be devices that would be informed of the Unit Attention Condition. The Target does <u>not</u> create the Unit Attention for an Initiator if it directly and provably caused the condition. For instance, if an Initiator issues a CLEAR QUEUE Message to the Target, the Target would create the Unit Attention Condition for all of the <u>other</u> Initiators, but not that one. On the other hand, if that Initiator ***Asserted*** the ***RST Signal*** to create the ***Reset Condition***, the Target creates the Unit Attention Condition for all Initiators, since it cannot know which Initiator asserted the RST Signal.

A typical method for implementing this is a bit map for each Logical Unit. After a Hard Reset, the bit map is cleared to zero. Bit 0 of the bit map corresponds to the Initiator with a ***SCSI Address*** of 0. If a bit in the bit map is zero, then a Unit Attention Condition exists. If the bit is one, then there is no Unit Attention Condition for that Initiator. Anything that requires reporting the Unit Attention Condition clears the appropriate bits in the bit map.

So how does the Target report Unit Attention? Diagram 50 shows how Targets handle Unit Attention once it is created. Unit Attention is treated something like an error condition in the Logical Unit. The Unit Attention Condition is reported as a ***Contingent Allegiance Condition*** (kind of figures, doesn't it?). In other words, as an error on the Logical Unit. It is more involved as you can see in the flow chart.

Bear with us in this discussion, because the Commands involved (REQUEST SENSE and INQUIRY) are fully described in later Volumes of this Encyclopedia. This is one of those difficult gray areas between the basic SCSI protocols and commands.

After the Unit Attention Condition is created, the Target starts to receive Commands from Initiators. If the Command is not REQUEST SENSE or INQUIRY, the Target returns an error via ***STATUS Phase*** (CHECK CONDITION ***Status***). The SENSE Data includes a SENSE Key that indicates that a Unit Attention Condition occurred. Once the SENSE Data is transferred, the Unit Attention (and Contingent Allegiance) Condition is cleared.

If the Command was REQUEST SENSE, the Target can report and clear the Unit Attention Condition directly.

If the Command is INQUIRY, the Unit Attention Condition is put on hold to execute the INQUIRY Command. In other words, never report a Unit Attention Condition in response to an INQUIRY Command. The reasons are somewhat historical: Unit Attention is most often reported in response to a Hard Reset. The first thing an Initiator often does after a Hard Reset is issue an INQUIRY Command. It was felt that the Initiator, which is in the process of configuring its drivers, should not get CHECK CONDITION Status during this process. This is why INQUIRY is special.

Note at the bottom of the flow chart are tests to see if there are "more" Unit Attention Conditions pending. The Target is allowed to "stack" Unit Attentions if there is more than one change. In general, this ability should not be used at all, or only used sparingly:

- If the Target gets a Hard Reset, it may get its microcode changed, its MODE SELECT Parameters changed, and its Queue cleared. Only the Reset itself is of any interest.

- On the other hand, it may be interesting to know about both a spindle sync change <u>and</u> a MODE SELECT Parameter change, and the only way to report both is to create two Unit Attention Conditions.

Use your own discretion...

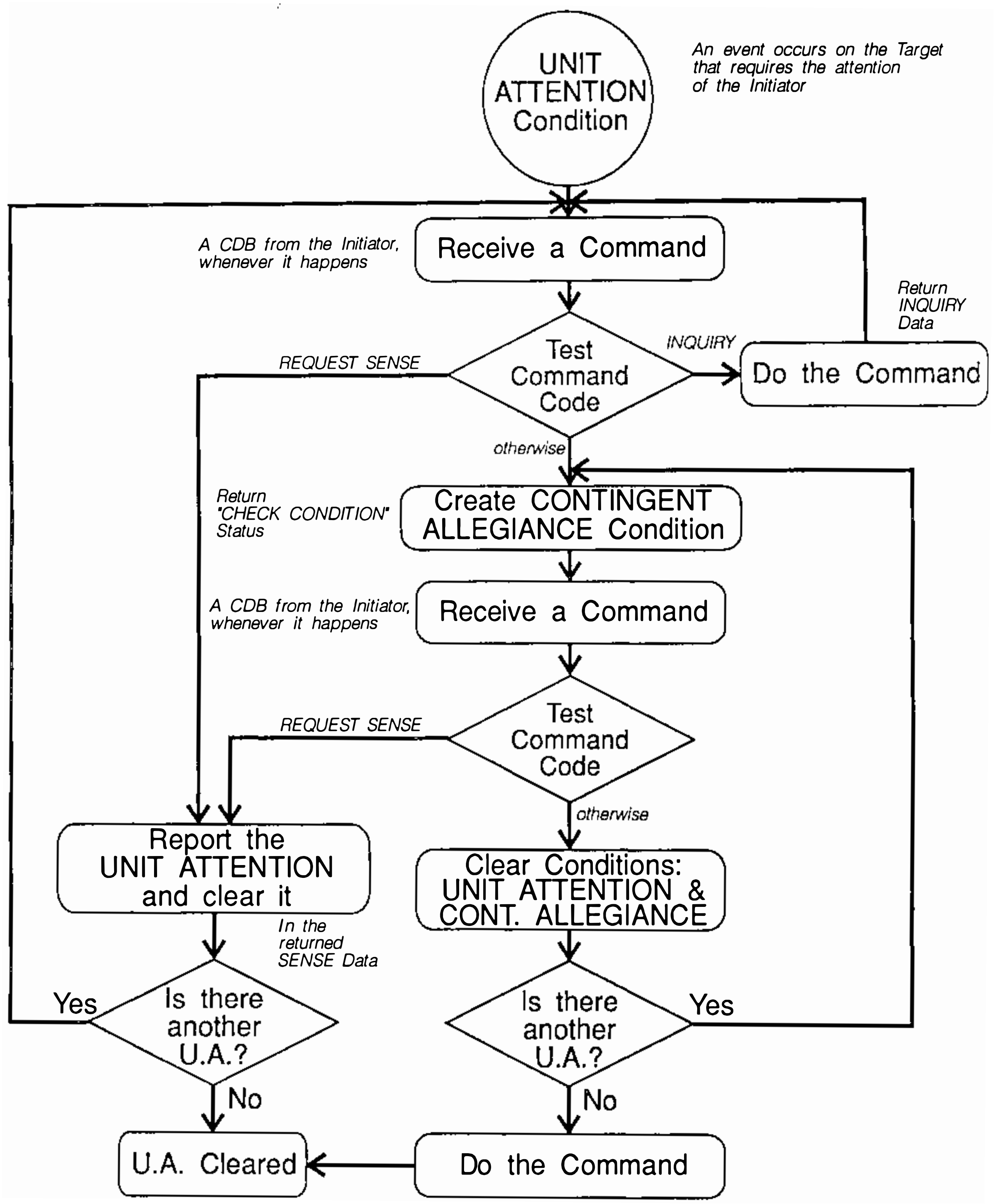

UNIT ATTENTION Condition
An event occurs on the Target that requires the attention of the Initiator
Receive a Command
A CDB from the Initiator, whenever it happens
Test Command Code
REQUEST SENSE
INQUIRY
Do the Command
Return INQUIRY Data
otherwise
Create CONTINGENT ALLEGIANCE Condition
Return "CHECK CONDITION" Status
Receive a Command
A CDB from the Initiator, whenever it happens
Test Command Code
REQUEST SENSE
otherwise
Report the UNIT ATTENTION and clear it
In the returned SENSE Data
Clear Conditions: UNIT ATTENTION & CONT. ALLEGIANCE
Is there another U.A.?
Yes
No
Is there another U.A.?
Yes
No
U.A. Cleared
Do the Command

DIAGRAM 50: UNIT ATTENTION FLOW DIAGRAM FOR TARGETS

Untagged Queuing. Untagged Queuing is a ***Queue*** for ***I/O Processes*** for a ***Logical Unit*** which does not use ***Queue Tags***. Contrast to ***Tagged Queuing***. Untagged Queuing only allows one I/O Process to be active or queued from a single Initiator for that Logical Unit. This was the only form of Queuing available under ***SCSI-1***. See ***Queue*** for the big description....

Vendor Specific or Unique.

Vendor Specific or Unique. These words describe any feature added by a Target or an Initiator that is not covered by ***SCSI***. Typically, it is a new Command or Command feature, but there is also the potential for Vendor Specific ***Messages*** (see ***Extended Messages*** and ***Etiquette***).

As an example, vendors will often add Vendor Unique "Pages" to the MODE SELECT Command to control special features of their product. See later Volumes of this Encyclopedia for details.

As a rule, don't invent a Vendor Specific feature if a standard one will do. It just makes things harder on your users since they have to have special drivers or utilities to deal with it.

As a user, try not to use a Vendor Specific feature if you can use a similar standard feature (many vendors actually offer two ways of doing things!). A better way is to ignore the feature completely, if possible. Also, do yourself a favor and resist the urge to demand Vendor Specific functions from your vendors. It just reduces the availability of alternate sources.

Wide Data Transfer.
Wide Data Transfer Negotiation.
WIDE DATA TRANSFER REQUEST Message.
Wire-OR Glitch.

Wide Data Transfer. "Wider! Faster! Deeper! (Deeper?)" was the rally cry of ***SCSI-2***, wherein the data transfer performance of the SCSI Bus was improved. Like ***Fast Data Transfer***, Wide Data Transfer is a method that can be used to go beyond the SCSI-1 data transfer rates. Wide Data Transfer, like Synchronous Data Transfer, is only used during a ***DATA Phase***. All other ***Information Transfer Phases*** use only 8-bit Asynchronous Data Transfer. Table 28 summarizes all combinations of Wide Data Transfer with the other modes: ***Asynchronous Data Transfer***, ***Synchronous Data Transfer***, and Fast Data Transfer.

TABLE 28: POSSIBLE DATA TRANSFER RATES

	Transfer Width		
Transfer Mode	8-bit (A Cable)	16-bit (P or A/B Cable)	32-bit (A/B Cable)
Asynchronous Data Transfer	set by device and cable delays	≈2 times the 8-bit rate	≈4 times the 8-bit rate
Synchronous Data Transfer	5 Megabytes per second	10 Megabytes per second	20 Megabytes per second
Fast Data Transfer	10 Megabytes per second	20 Megabytes per second	40 Megabytes per second

There are two ways to implement Wide Data Transfer:

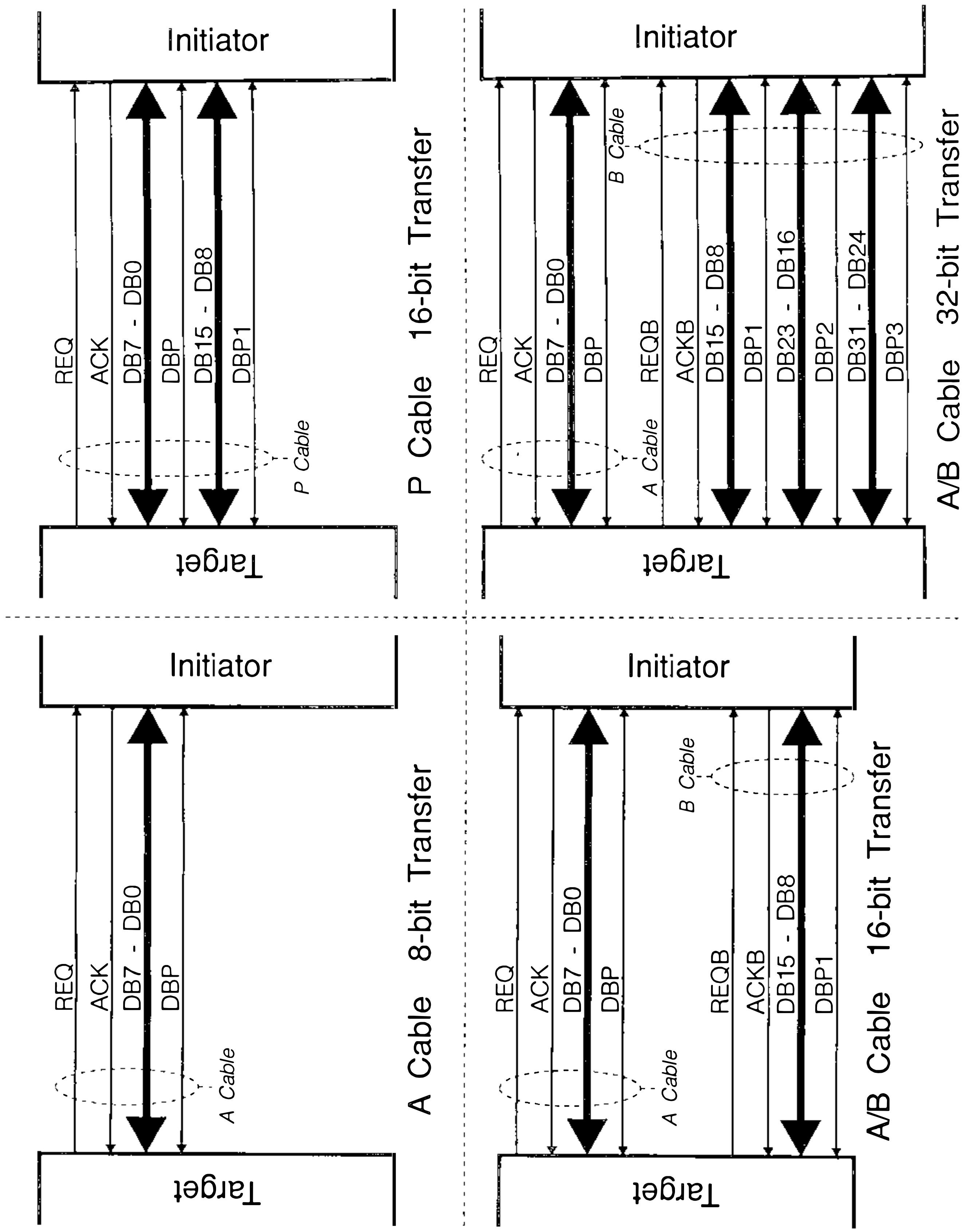
- Technically (according to the SCSI-2 Standard), there is only one way to achieve Wide Data Transfer, which is by adding a second cable to each ***SCSI Device***. This second cable is known as the ***B Cable*** (the original 8-bit cable is known as the ***A Cable***). The B Cable can provide a 16-bit data path or a 32-bit data path. The problem is that, well, you need two cables.

- Since SCSI-2 was finished, a new proposal for Wide Data Transfer has been developed by the ***X3T9.2 Committee*** as part of the ***SCSI-3*** effort. Since this proposal has received a lot of attention and work, it is included here. This is a single cable, 16-bit data path known as the ***P Cable***. When using the P Cable, it replaces the A Cable as the only cable between each SCSI Device.

The different methods of achieving Wide Data Transfer are shown in Diagram 51.

We recommend the P Cable for Wide Data Transfers (in fact, it could be useful for new 8-bit designs; see ***P Cable***). You will find that many of the newest SCSI ***Chips*** will only support the P Cable method, since it is much easier to implement in silicon. As we cover both methods, the disadvantages of the B Cable method will become clear.

If you have a 32-bit requirement, contact the X3T9.2 Committee first. They are also working on something called the "Q Cable", which extends the P Cable to 32-bits.



A Cable 8-bit Transfer

P Cable 16-bit Transfer

A/B Cable 16-bit Transfer

A/B Cable 32-bit Transfer

DIAGRAM 51: WIDE DATA TRANSFER CONFIGURATIONS

The B Cable Method. As shown in Diagram 51, the B Cable adds the necessary additional data signals to the existing 8-bit A Cable signals as a second cable. Because of the danger of skew between unequal length cables, the B Cable also has its own ***REQB Signal*** and ***ACKB Signal***. This ensures that the data on the B Cable is clocked properly (setup and hold times for the B Cable data are met).

Diagram 52 shows the data path for a B Cable system. Note that there are two unique and independent data paths on the Initiator and the Target. Each data path transfers data independently of the other: ACK responds only to REQ, and ACKB responds only to REQB. Data moves through each data path from one device to the other with no interaction with the other path. Note again that the B Cable is used only during a DATA Phase.

Data for each path is split at the entrance to the data paths. The data is recombined at the exit from the data path in the receiving device.

The behavior of each cable follows the rules for either:

- Asynchronous Data Transfer; or,

- Synchronous Data Transfer, if it has been negotiated via ***Synchronous Data Transfer Negotiation (SDTN)***.

Note that both data paths must use the same transfer modes and parameters (***Synchronous Offset*** and ***Synchronous Transfer Period***).

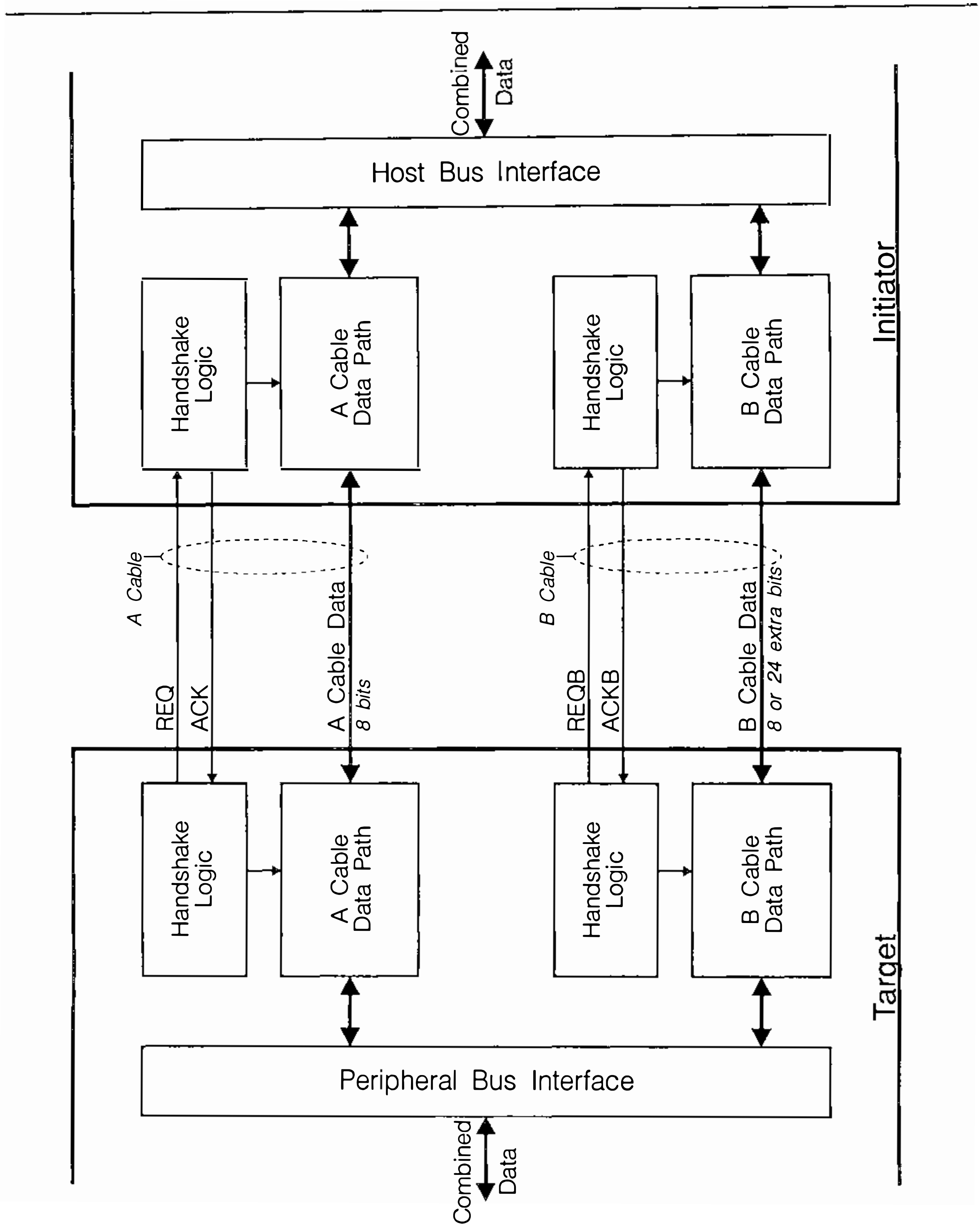


DIAGRAM 52: B CABLE WIDE DATA TRANSFER DATA PATH

The P Cable Method. As shown in Diagram 51, the P Cable adds 8 additional data signals to the existing 8-bit A Cable signals within one cable. Since only one cable is used, only one REQ Signal and ACK Signal are required.

Diagram 53 shows the data path for a P Cable system. As we can see, this is simpler than the B Cable method. Only one data path is required. In fact, this is as simple as a basic SCSI data path; only the number of data bits is increased.

This shows why the P Cable is gaining popularity, even though it is not technically standardized. It is easy for chip manufacturers to create a 16-bit device from an existing architecture just by increasing the data path width (and they are doing it). Creating a B Cable 16-bit device requires two data paths to be included in the architecture, which is much more difficult. Also, in this age of smaller and smaller devices, it is difficult to conceive of mounting two cable connectors on a single device.

Byte Ordering. When doing Wide Data Transfer, it's important to get the byte order right. The following diagrams give an example of an eight byte transfer, and how each of the bytes are handled in each Wide Data Transfer method. Diagram 54 shows how the eight bytes are transferred over the 8-bit A Cable. Eight bytes (designated **Byte a** through **Byte f**) are transferred using eight REQ/ACK handshake cycles. No surprises there... (I hope!)

Diagram 55 shows the byte ordering for a Wide Data Transfer on the P Cable. Eight bytes are transferred in four 16-bit handshake cycles. The first handshake transfers **Byte a** on the ***Data Bus Signals*** DB7-DB0, and **Byte b** on DB15-DB8.

Diagram 56 shows the byte ordering for a 16-bit Wide Data Transfer using the B Cable to supplement the A Cable. Eight bytes are transferred in eight 8-bit handshake cycles on two cables. The first handshake on the A Cable transfers **Byte a** on DB7-DB0, and the first handshake on the B Cable transfers **Byte b** on DB15-DB8.

Diagram 57 shows the byte ordering for a 32-bit Wide Data Transfer using the B Cable to supplement the A Cable. Eight bytes are transferred in two 8-bit handshake cycles on the A Cable, and two 24-bit handshake cycles on the B Cable. The first handshake on the A Cable transfers **Byte a** on DB7-DB0, and the first handshake on the B Cable transfers **Byte b** on DB15-DB8, **Byte c** on DB23-DB16, and **Byte d** on DB31-DB24.

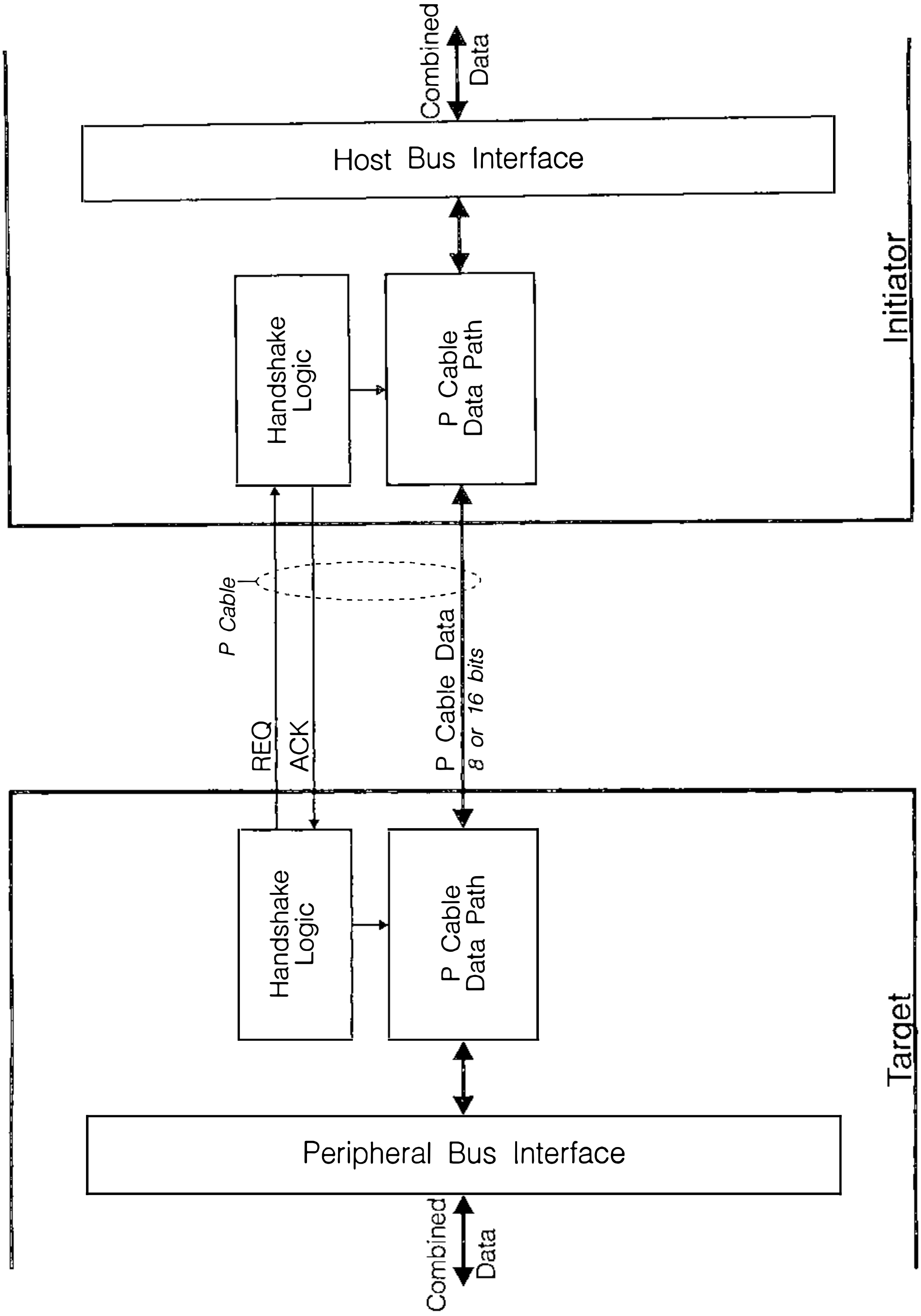


DIAGRAM 53: P CABLE WIDE DATA TRANSFER DATA PATH

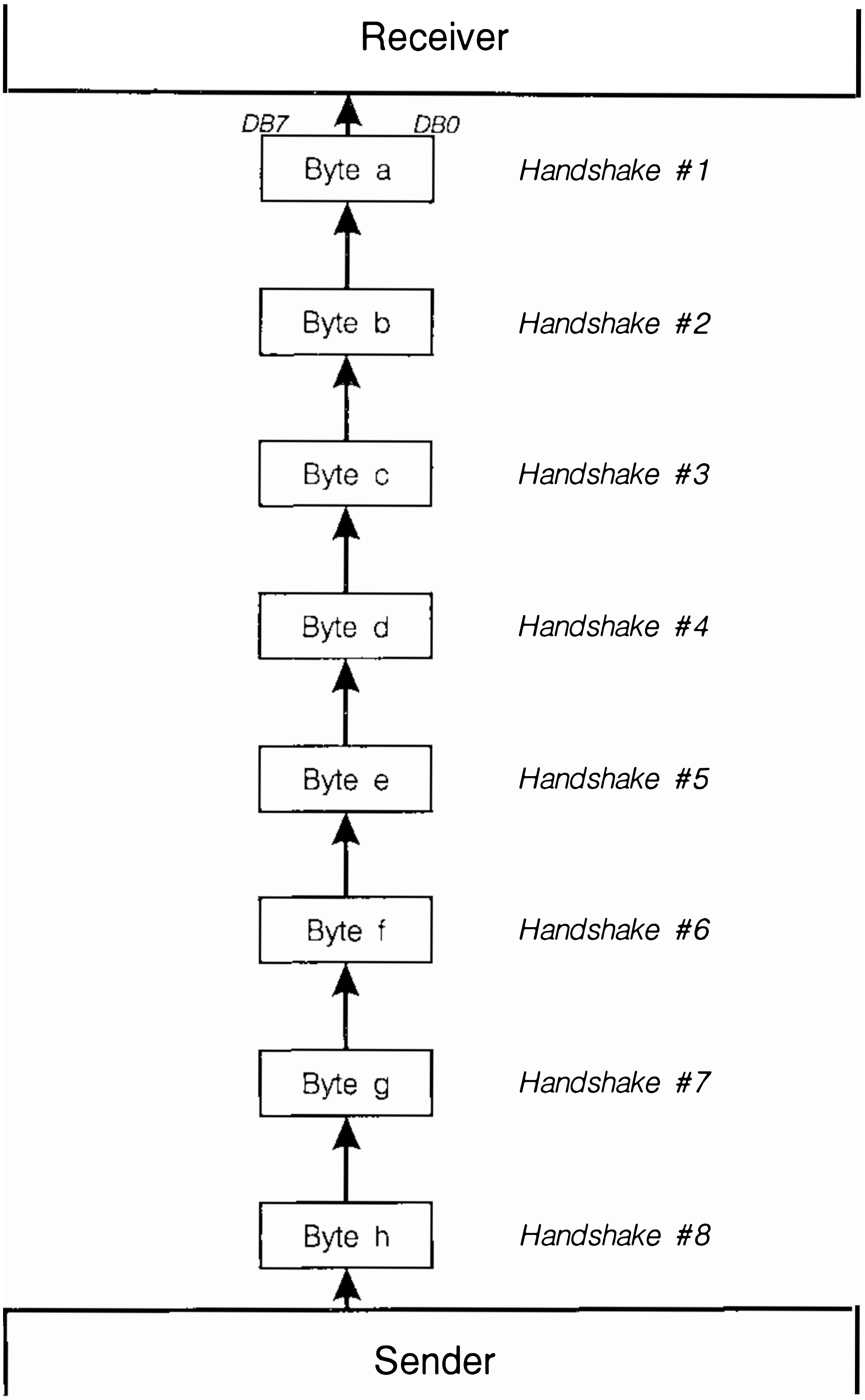


DIAGRAM 54: A CABLE DATA TRANSFER BYTE ORDERING

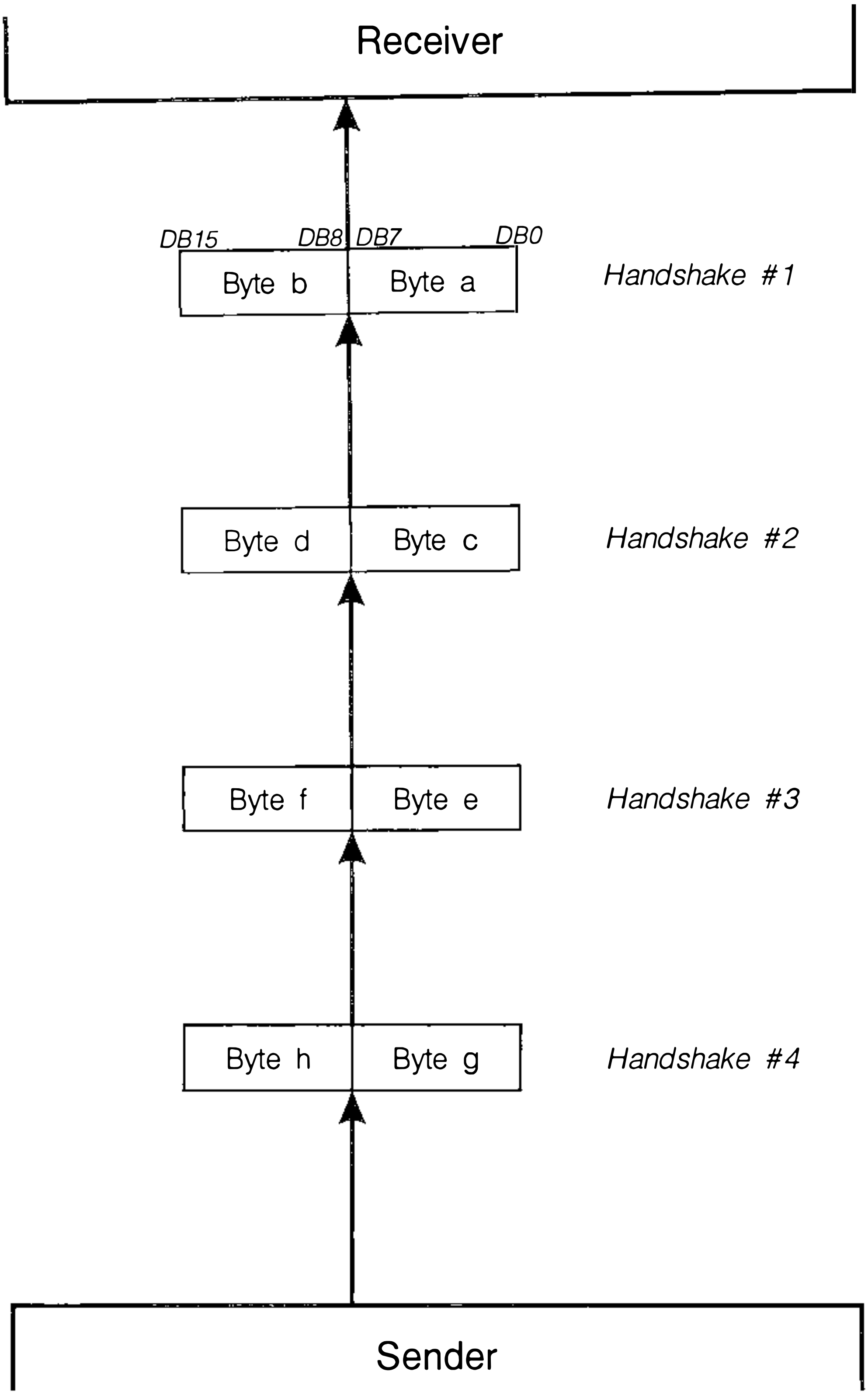


DIAGRAM 55: P CABLE 16-BIT DATA TRANSFER BYTE ORDERING

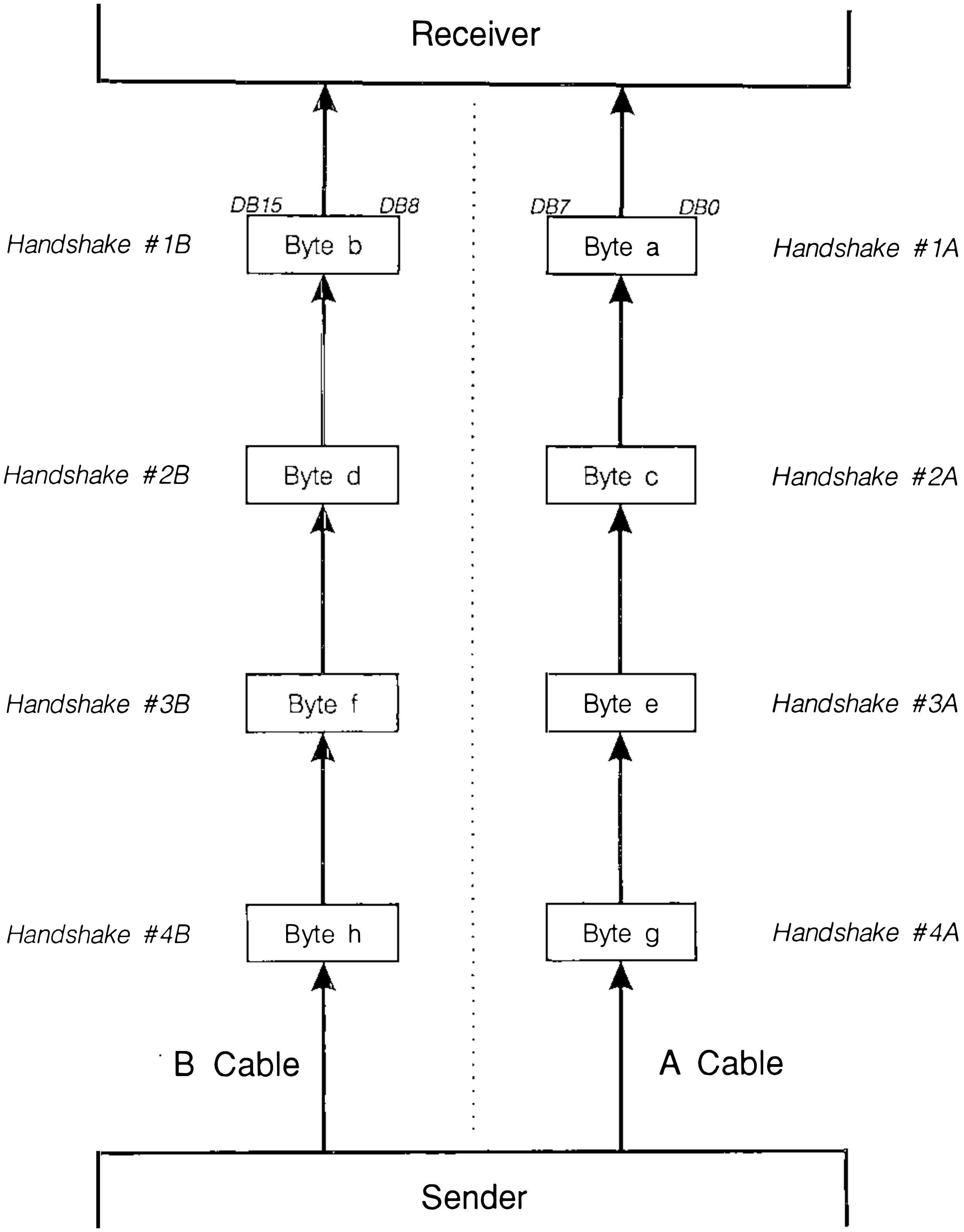


DIAGRAM 56: A/B CABLE 16-BIT TRANSFER BYTE ORDERING

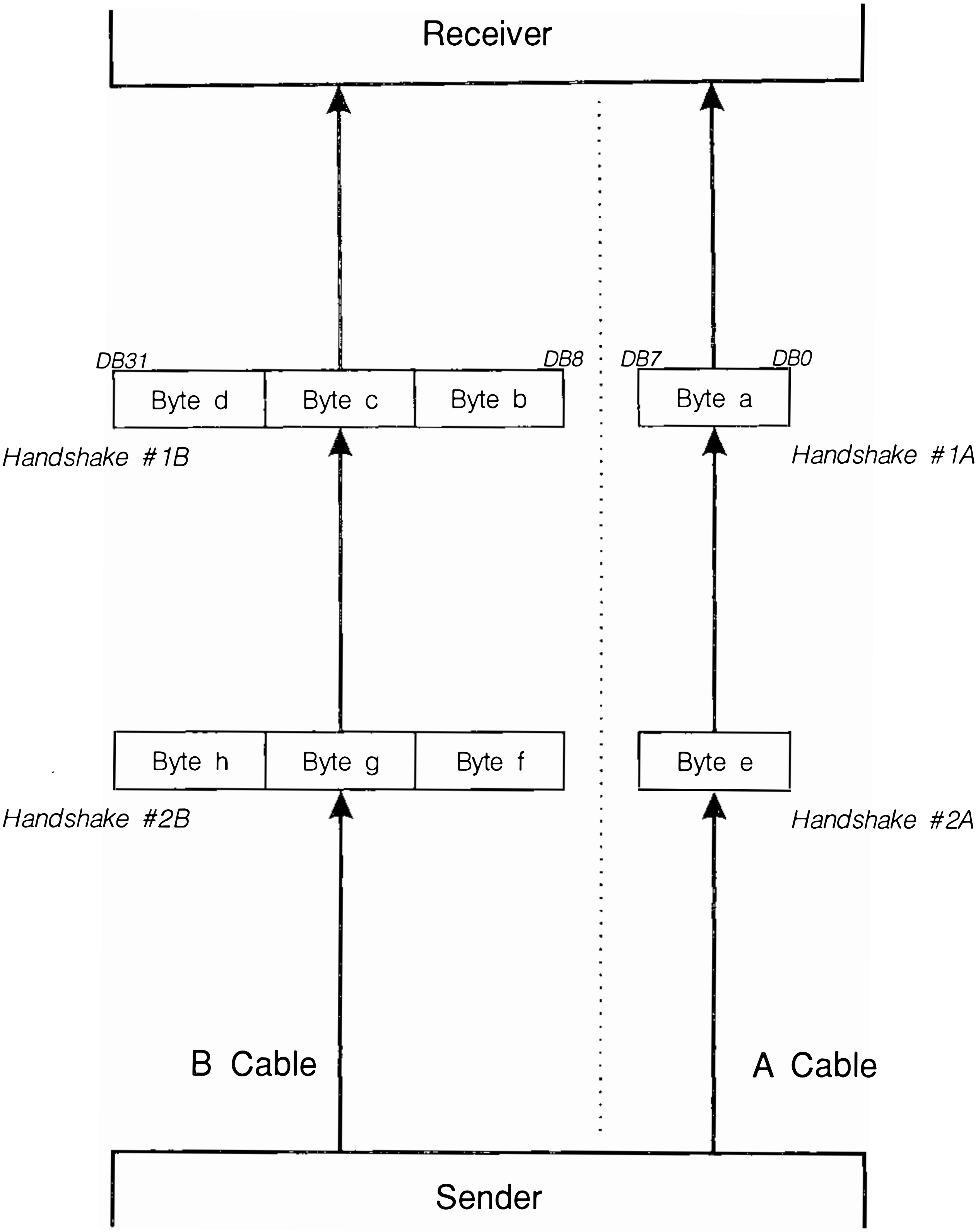


DIAGRAM 57: A/B CABLE 32-BIT TRANSFER BYTE ORDERING

Wide Data Transfer Negotiation. Wide Data Transfer Negotiation (WDTN) is used to establish the data transfer width to be used between two ***SCSI Devices***. The process is almost the same as ***Synchronous Data Transfer Negotiation***. If two devices never engage in a WDTN, then the default data transfer width is 8 bits. After the WDTN process has completed, the two devices will (hopefully) have agreed to a bus width that results in a ***Wide Data Transfer***.

The specific purpose of WDTN is to provide a procedure where two devices can agree on a bus width for Wide Data Transfer. The intent is to arrive at agreement such that the maximum performance is achieved.

Note that the agreement is between two SCSI devices. This agreement is independent of Initiator and Target role, and is also independent of ***Logical Unit*** on each device. In other words, the same agreement is used for all ***I/O Processes*** in all of the following situations:

- Device A takes the Initiator role and Selects Device B (which then takes the Target role) and transfers data with Logical Unit #X.
- Device B takes the Target role and Reselects Device A (which then takes the Initiator role) and transfers data with Logical Unit #Y.
- Device B takes the Initiator role and Selects Device A (which then takes the Target role) and transfers data with Logical Unit #Y.
- Device A takes the Target role and Reselects Device B (which then takes the Initiator role) and transfers data with Logical Unit #X.

If there is a Device C on the bus, both Device A and Device B must each reach their own agreement with Device C.

Any device may begin the WDTN process at any time. Typically, the process is begun on the first ***Connection*** following a ***Hard Reset*** on either device. For example:

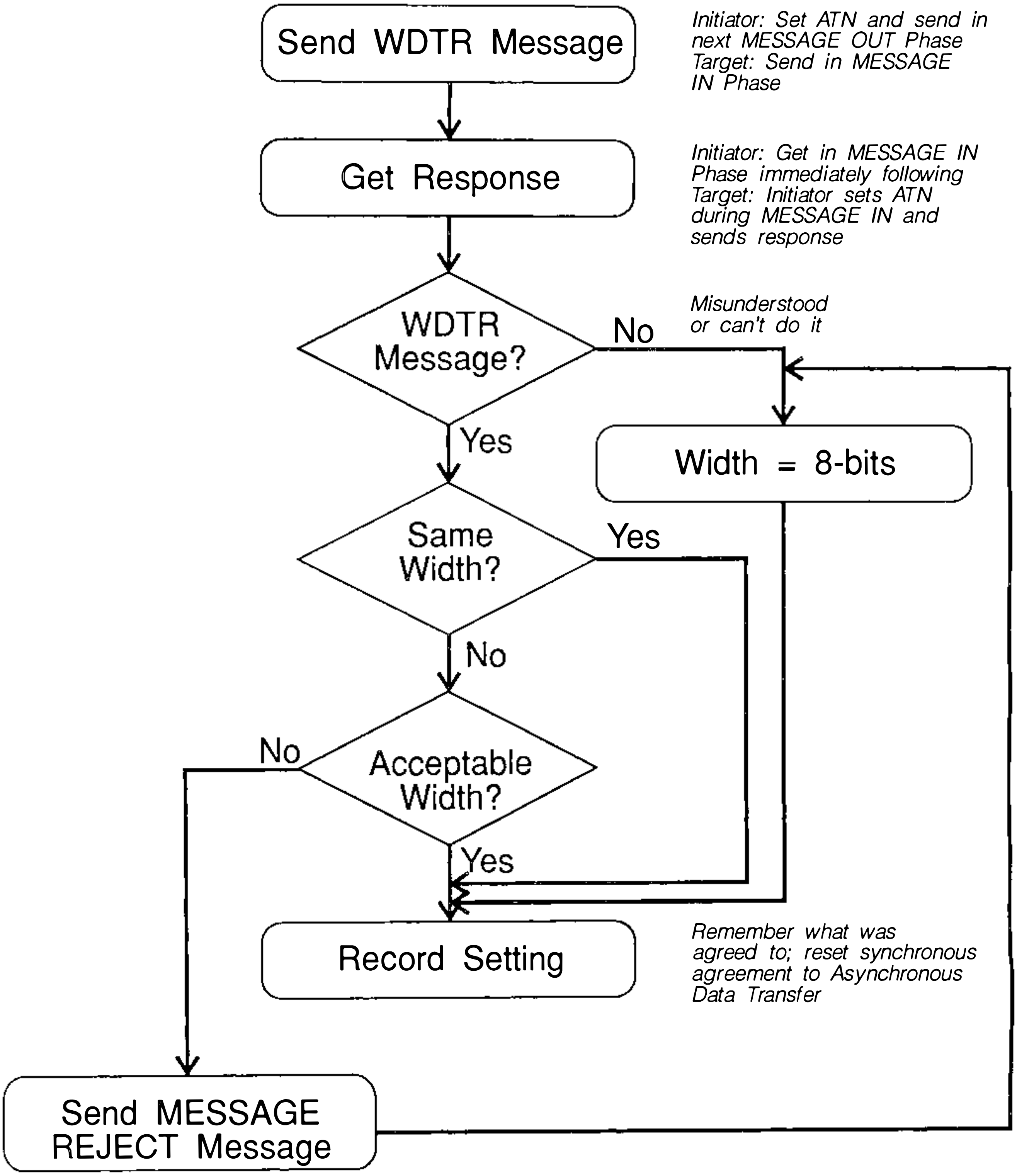
- An Initiator ***Connects*** to a Target. After the ***MESSAGE OUT Phase*** in which the ***IDENTIFY Message*** is sent by the Initiator, the Target changes to ***MESSAGE IN Phase*** and sends the ***Wide Data Transfer Request (WDTR)*** to begin the WDTN process.

- A Target ***Reconnects*** to an Initiator. During the MESSAGE IN Phase in which the IDENTIFY Message is sent by the Target, the Initiator creates the ***Attention Condition***. In response, the Target changes to MESSAGE OUT Phase, and the Initiator sends the WDTR message to begin the WDTN process.

You might also see the WDTN process begin just prior to the ***DATA Phase***, after the ***COMMAND Phase***, or after a ***Queue Tag Message***.

A WDTN must always precede an SDTN. After a WDTN is completed, the data transfer method is always reset to ***Asynchronous Data Transfer***. The two devices may begin an SDTN process after completing the WDTN process.

Diagram 58 and Diagram 59 illustrate how SCSI devices negotiate Wide Data Transfer bus width. Note that the diagrams do not refer to Initiators and Targets, rather, they refer to the Originator and Responder.



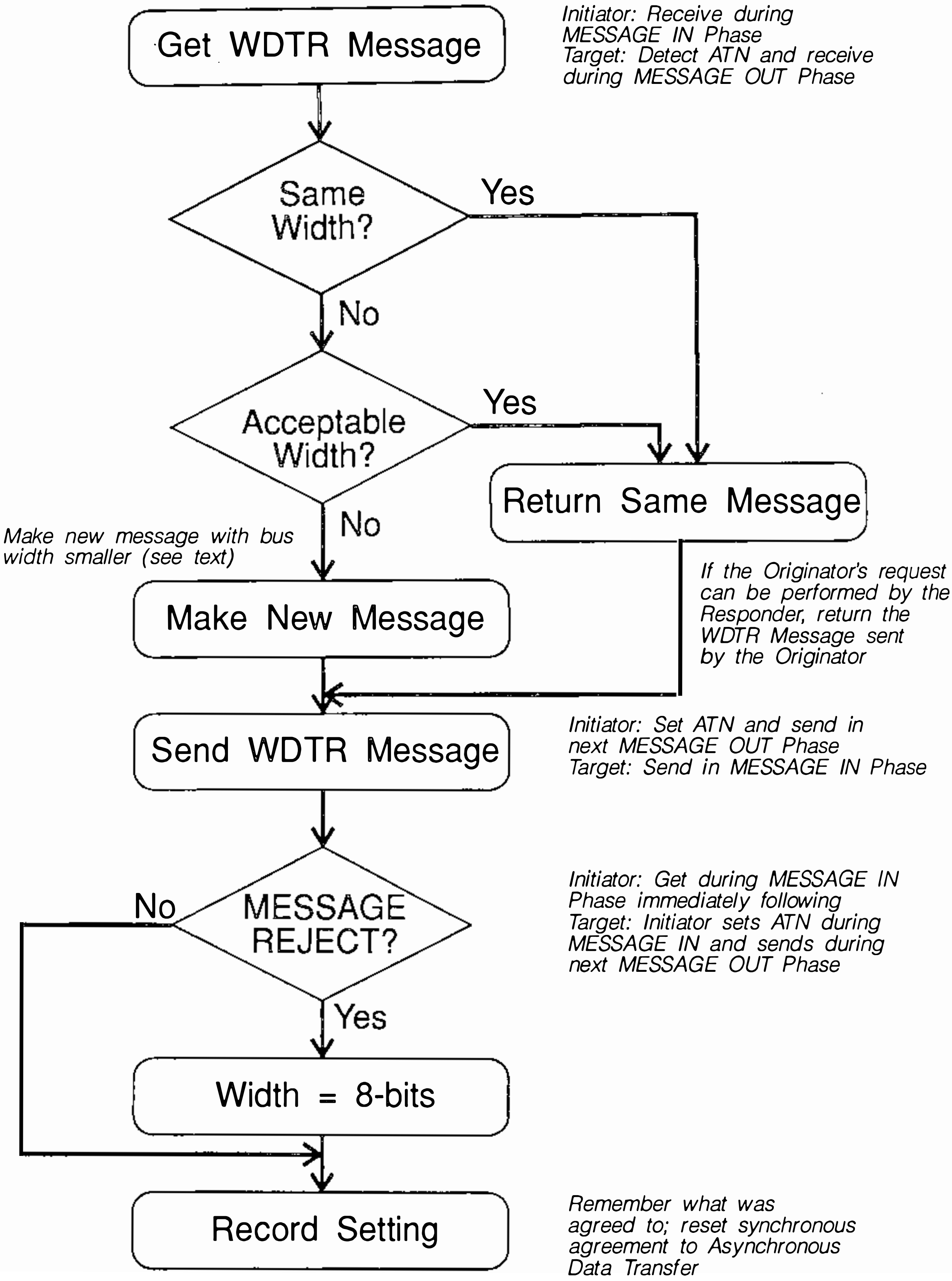
Initiator: Set ATN and send in next MESSAGE OUT Phase Target: Send in MESSAGE IN Phase

Initiator: Get in MESSAGE IN Phase immediately following Target: Initiator sets ATN during MESSAGE IN and sends response

Misunderstood or can't do it

Remember what was agreed to; reset synchronous agreement to Asynchronous Data Transfer

DIAGRAM 58: WDTR FLOW DIAGRAM FOR ORIGINATORS



Initiator: Receive during MESSAGE IN Phase
Target: Detect ATN and receive during MESSAGE OUT Phase

Make new message with bus width smaller (see text)

If the Originator's request can be performed by the Responder, return the WDTR Message sent by the Originator

Initiator: Set ATN and send in next MESSAGE OUT Phase
Target: Send in MESSAGE IN Phase

Initiator: Get during MESSAGE IN Phase immediately following
Target: Initiator sets ATN during MESSAGE IN and sends during next MESSAGE OUT Phase

Remember what was agreed to; reset synchronous agreement to Asynchronous Data Transfer

DIAGRAM 59: WDTR FLOW DIAGRAM FOR RESPONDERS

When a device decides it needs to begin the WDTN process (more on why it might decide that later), it first requests the maximum bus width it can perform, and sets up the WDTR message accordingly. The bus width is sent as part of the WDTR message. The device has now taken the role of the Originator in the WDTN process.

The other device, upon receiving the WDTR message from the Originator, becomes the Responder. In response, it can do one the following:

- If the bus width is acceptable, it repeats the same WDTR message back to the Originator. The bus width is acceptable when the Responder is able to perform a data transfer with the Originator at that bus width.

- If the bus width is too big, it sends a WDTR message back to the Originator set for the desired bus width.

- If the Responder cannot perform Wide Data Transfer, the Responder does one of the following:

  - It creates a WDTR message with the Transfer Width set to zero (indicates 8-bit transfer), and sends it to the Originator (this is preferred);

  - It returns a ***MESSAGE REJECT Message***.

  With either response, the Originator and Responder are agreeing that an 8-bit bus width is the data transfer width that will be used.

- If the Responder hasn't a clue about Wide Data Transfer, it will probably respond to the WDTR Message with a MESSAGE REJECT Message. In this case the Originator should get the hint and stick to 8-bit data transfers with the Responder from now on. Note that Synchronous Data Transfer is still a possibility.

After the first or second response, the Originator may have to respond to the response if it can't support the bus width requested by the Responder. For example: The Originator can handle 32-bit and 8-bit transfers, but not 16-bit transfers. The Responder can handle 16-bit and 8-bit transfers, but not 32-bit transfers:

- The Originator issues a WDTR Message requesting 32-bit transfers.

- The Responder returns a WDTR Message requesting 16-bit transfers.

The Originator can't handle 16-bit transfers, so it must respond with a MESSAGE REJECT Message. This tells the responder that its response was unacceptable. In this case, the two devices must transfer at an 8-bit bus width. Call your System Integrator right away!

When is Negotiation Required? A device should originate a WDTN whenever it has no record of a previous negotiation. (Notice this is the same rule as for ***Synchronous Data Transfer Negotiation***) This, of course, implies that a device that completes a WDTN should keep a record of it. This record is maintained for each SCSI Device on the bus, and could be represented as a two bit code:

- 00 = No WDTN completed with that device.
- 01 = WDTN completed with that device; 8-bit transfers.
- 10 = WDTN completed with that device; 16-bit transfers.
- 11 = WDTN completed with that device; 32-bit transfers.

The ways that a device could lose the value stored in this code are somewhat unique to each device, but we think we can all agree that at least the following events would clear these two bits to zero:

- A power-on reset.

- A ***Hard Reset***.

- A ***BUS DEVICE RESET Message***.

WIDE DATA TRANSFER REQUEST Message. The WIDE DATA TRANSFER REQUEST (WDTR) Message is used to establish the parameters that will be used during a ***Wide Data Transfer***. This is an ***Extended Message***:

Bit Byte	7	6	5	4	3	2	1	0
0	Message Code = 01 hex							
1	Additional Message Length = 02 hex							
2	Extended Message Code = 03 hex							
3	Transfer Width							

Byte 3 of this message carries the Transfer Width. Table 30 shows the codes for each Transfer Width.

TABLE 30: WIDE DATA TRANSFER WIDTH CODES

Transfer Width Code	Bus Width
00 hex	8 bits
01 hex	16 bits
02 hex	32 bits
03-FF hex	Reserved for future use

See ***Wide Data Transfer Negotiation*** for a complete description of the use of this message.

Summary of Use: The WDTR Message is sent by any SCSI Device to establish the parameters to be used during Wide Data Transfer.

Wire-OR Glitch. "Yow! What's that? A feature?" No, it's a physical fact. A Wire-OR glitch can occur on any signal path where more than one devices can ***Assert*** and ***Release*** the signal. The glitch occurs when two devices are Asserting the signal (pulling it low) and then one of them Releases it.

Before delving into this topic, we expect you have already studied ***Cables***, ***Termination***, ***BUS FREE Phase***, ***ARBITRATION Phase***, and ***RESELECTION Phase***.
In SCSI, a glitch may occur during RESELECTION Phase. After the Initiator has responded to the Reselection by the Target, and the Target has Released the ***SEL Signal***, both devices are Asserting the ***BSY Signal***. The next step is for the Initiator to Release the BSY Signal. This is where the glitch may occur. (The glitch may also occur when the SEL Signal is asserted during ARBITRATION Phase, and all losing devices Arbitrating for the bus release their BSY Signals. We will examine the RESELECTION Phase Wire-OR Glitch here because it is more predictable.)

Note that when two devices are asserting a signal, both devices are taking current from the Terminators. Usually, one device has a better transistor than the other, and so takes more current. This discussion assumes that the Initiator is taking all of the current. If the Target were taking all of the current, there would be no glitch. The worst-case magnitude of the glitch occurs when the Initiator is taking all of the current from both Terminators.

Diagram 60 shows the bus topology and the cause of a Wire-OR glitch during the RESELECTION Phase:

(1) The Initiator releases its BSY output, represented by a transistor pull-down. When that occurs, the Initiator BSY output is no longer providing a ground to the BSY Signal. This means the Initiator is no longer taking current from the both Termination networks. The problem is that at the instant when BSY is released by the Initiator, the terminators are still supplying current to the Cable. This causes an instantaneous voltage to appear on the BSY Signal, and to propagate away from the Initiator in both directions. The magnitude of the voltage for each glitch (in each direction) is given by the equation (remember Ohm's Law?):

$$V_{glitch}=I_{term}\times Z_o$$

Where V_{glitch} is the voltage magnitude of the glitch; I_{term} is the current supplied by the terminator to the Initiator BSY output; and Z_o is the characteristic impedance of the cable.

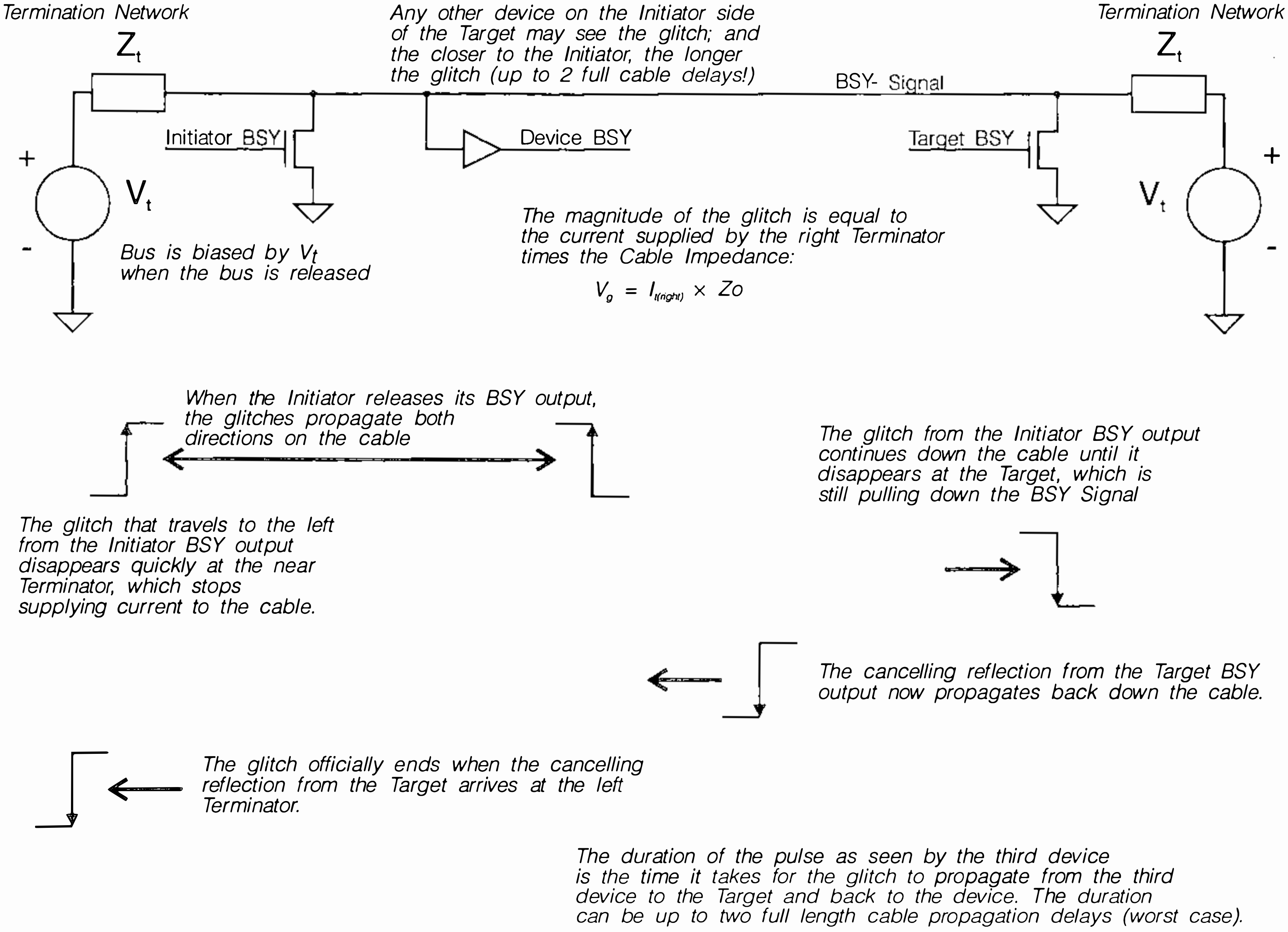
(2) The instantaneous voltage transition propagates to the left Terminator and quickly dissipates. Since the Initiator is no longer sinking the terminator bias voltage, the terminator is now allowed to pull the BSY Signal up to the bias voltage. The other voltage transition travels to the right down the cable. As it passes the third device, that device sees the transition in the level of the BSY Signal, At this point the third device might begin to assume that this is the start of a BUS FREE Phase.

(3) The glitch arrives at the Target output and dissipates on the ground there. This causes a high to low transition to propagate back to the left from the Target.

(4) When the high to low transition arrives at the third device, it sees the end of the glitch, and knows that it isn't really a BUS FREE Phase (we hope). The glitch actually ends when it arrives back at the left Terminator. Worst case, the glitch can last for the time it takes to propagate twice the length of the cable.

This explains why the ***Bus Free Delay*** is as long as it is. This is the time it takes to propagate down a worst-case slow ***Cable*** for a ***Differential Interface*** (yes, it can happen on Differential too!) and back, plus some extra margin.

DIAGRAM 60: WIRE-OR GLITCH SCHEMATIC



Any other device on the Initiator side of the Target may see the glitch; and the closer to the Initiator, the longer the glitch (up to 2 full cable delays!)

Bus is biased by V_t when the bus is released

The magnitude of the glitch is equal to the current supplied by the right Terminator times the Cable Impedance:

$$V_g = I_{t(right)} \times Zo$$

When the Initiator releases its BSY output, the glitches propagate both directions on the cable

The glitch that travels to the left from the Initiator BSY output disappears quickly at the near Terminator, which stops supplying current to the cable.

The glitch from the Initiator BSY output continues down the cable until it disappears at the Target, which is still pulling down the BSY Signal

The cancelling reflection from the Target BSY output now propagates back down the cable.

The glitch officially ends when the cancelling reflection from the Target arrives at the left Terminator.

The duration of the pulse as seen by the third device is the time it takes for the glitch to propagate from the third device to the Target and back to the device. The duration can be up to two full length cable propagation delays (worst case).

This page is nearly blank!

X3T9.2 Committee.

X3T9.2 Committee. The X3T9.2 Committee is the "Keeper of the Flame" for SCSI. That's the group that created the original ***SCSI-1*** standard (X3.131-1986) and then created the update to the standard which is called ***SCSI-2***. X3T9.2 is a "task group" of the X3T9 Committee, which oversees "I/O Interfaces", such as SCSI, IPI (Intelligent Peripheral Interface), and Fiber Channel. X3T9 is a "sub-committee" of X3, which oversees ANSI (American National Standards Institute) standards activity in the USA related to "Information Processing Systems".

If you want to get more information regarding membership in X3T9.2, call the X3 Secretariat at (202)737-8888.

At the time of this writing, the Chairperson of X3T9.2 was John Lohmeyer of NCR Corp. You can reach John at:

John Lohmeyer
NCR Corp.
3718 North Rock Road
Wichita, KS 67226

John also operates a computer "Bulletin Board System" (BBS) dedicated to SCSI. The phone number is (316) 636-8700. Set your modem to 2400 baud, 8 data bits, 1 stop bit, no parity. The SCSI BBS is the place to find the latest information on what's happening today in ***SCSI-3***.

Examples of SCSI Behavior.
Example #1 (Disk Read).
Example #2 (Disk Write).
Example #3 (Tape Read).
Example #4 (Tape Write).

This page is nearly blank!
We use the space to improve Readability.

Examples of SCSI Behavior. This section gives several examples of SCSI I/O Processes. Each example has the following elements:

- A description of the "scenario".
- A Block Diagram showing the model for the system described in the scenario.
- Text describing the steps in the scenario in more detail.
- Any other diagrams and tables needed to describe details of the scenario.

Example #1: The first example shows a simple disk read operation. Diagram 61 shows the system for this example. The Host System has a system bus, such as the VME Bus or the EISA Bus, which connects to a ***Host Adapter***. The Host Adapter performs the ***Initiator*** function for the Host System. The Host Adapter communicates with the Host System via Direct Memory Access (DMA) with the Host Memory on the system bus. This DMA Channel is used to transfer all Commands, Data, and Status between the Host System and the ***SCSI Bus***. In other words, this is a classic Host I/O Channel.

The ***Target*** is a simple (!) "Embedded" SCSI Disk Drive with a single ***Logical Unit*** that corresponds to the physical hard disk mechanism. The sectors on the hard disk are mapped to SCSI ***Logical Blocks***. The Target contains a "Data Buffer" consisting of a local memory block that holds sectors during a transfer:

- When writing, the Data Buffer holds the data from the Host System prior to writing the data to the hard disk.

- When reading, the Data Buffer holds the data read from the hard disk prior to transfer to the Host System.

The hard disk has a mechanical head positioning system which is located by the Target on the disk track which contains the desired Logical Blocks. Moving the head takes a relatively long period of time to complete. In other words, this is a classic Intelligent Disk Drive. As we will see in the other volumes of this Encyclopedia, an Intelligent Disk Drive is called a Direct Access Device in SCSI.

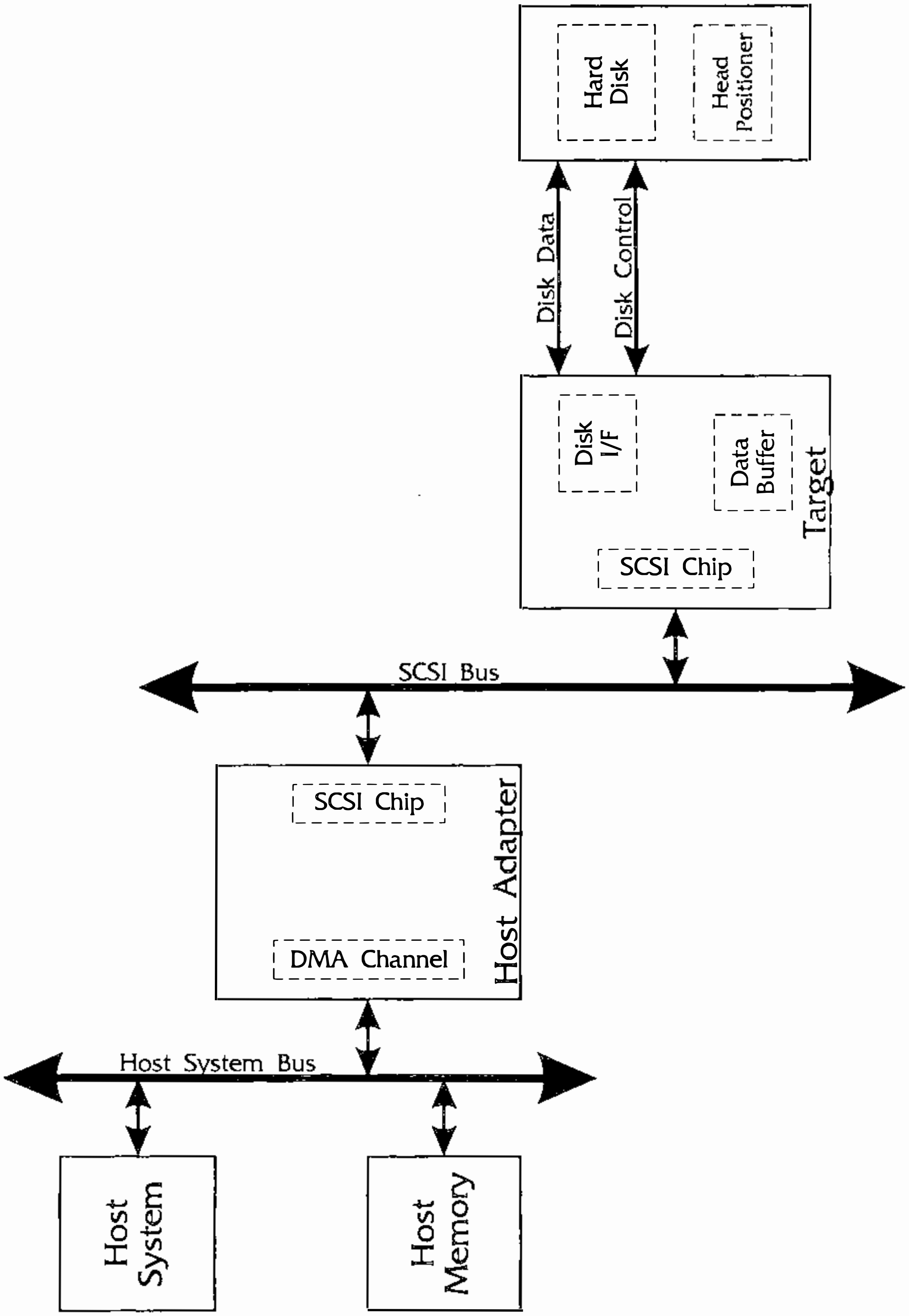


DIAGRAM 61: DISK EXAMPLE SYSTEM ARCHITECTURE

The file read request must filter down through the different layers of the Host System, as shown in Diagram 62. At each level, the request is translated into a standard form understood by the next lowest level. Note that the diagram shows the flow of information between levels; it does not show the time order of that flow.

- An application program on the Host System makes a request to the Operating System to read a file; for example, a spreadsheet program loads a user spreadsheet. The application specifies to the Operating System which file, how much of the file to read, and where to put the data.

- The Operating System takes the file read request from the application. Using its internal file system data structures, it determines which file system blocks make up the requested file. From this information, the Operating System creates a read disk block request and issues it to the SCSI Driver program.

- The SCSI Driver program takes the block read request from the Operating System. It converts the file system block read request into a SCSI ***Command Descriptor Block (CDB)***. It then incorporates the CDB into a Host Adapter Control Block (see ***Host Adapter***). This Control Block is set up in Host Memory. The Host System then tells the Host Adapter to begin executing the Command specified in the Control Block.

- When the SCSI ***I/O Process*** is completed, ***Status*** has been returned and stored in the Host Adapter Control Block. The data has been transferred directly to the application data area. If the Command caused "CHECK CONDITION" Status, the Host Adapter may also have fetched Sense Data from the Target disk drive.

- The SCSI Driver translates the returned Status and Sense Data (if any) to driver completion codes understood by the Operating System and passes them back up.

- The Operating System passes OS completion codes back to the application. The operation is complete, and the application has its data.

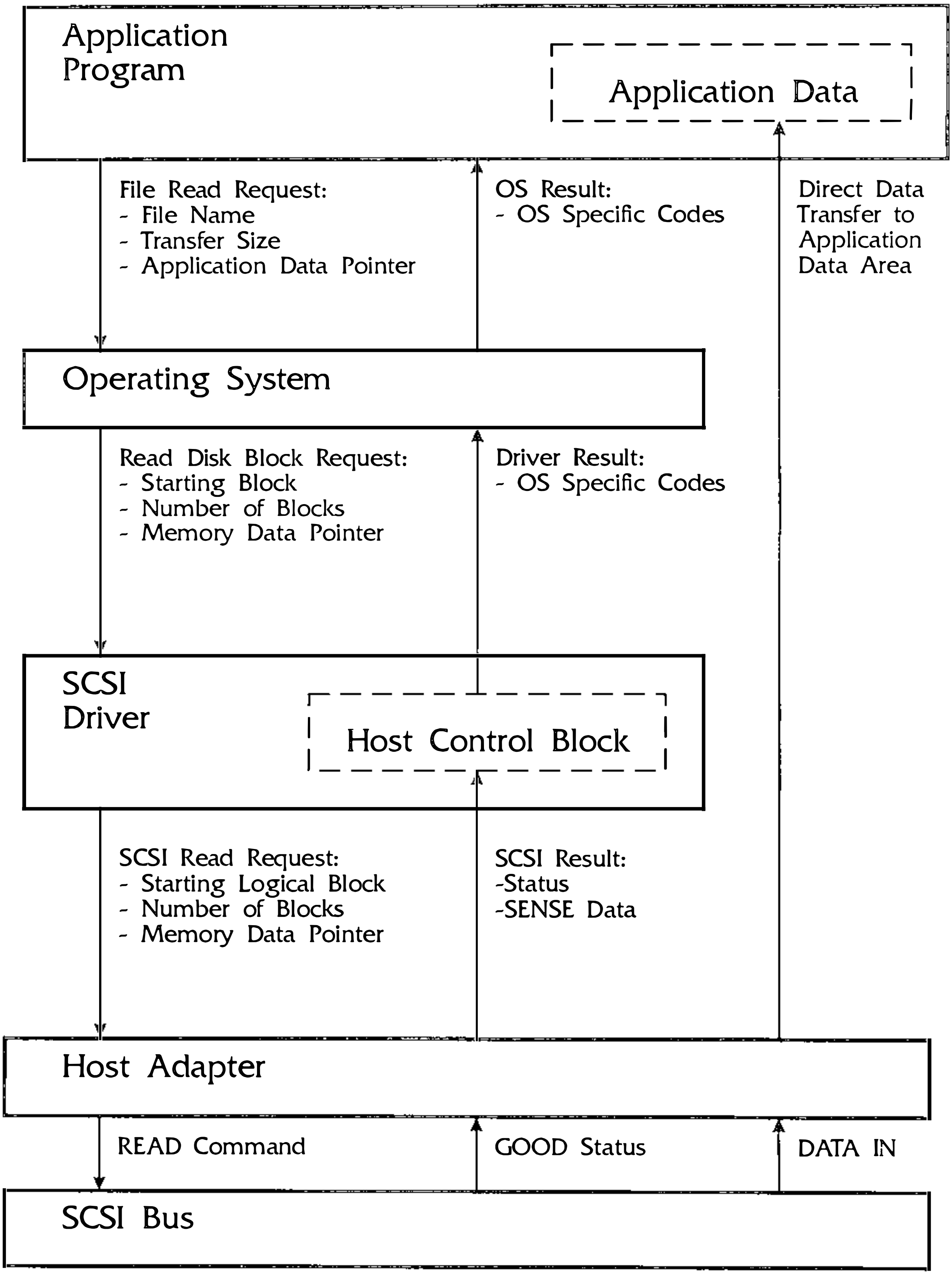


DIAGRAM 62: DISK READ EXAMPLE SOFTWARE LEVELS

The Disk Read Command example begins with the request for a file by the application program, as described above. We'll pick it up after the Host Adapter receives the Host Adapter Control Block from the SCSI Driver:

- The Host Adapter performs the Initiator function for the Host System. The Initiator enters the ***ARBITRATION Phase*** (after validating the ***BUS FREE Phase***) to get control of the bus. It asserts the ***BSY Signal*** and its own ***SCSI Bus ID***.

- The Initiator wins Arbitration by having the highest SCSI Bus ID asserted. The Initiator then takes control of the bus by asserting the ***SEL Signal***. It then asserts the ***ATN Signal*** (to create the ***Attention Condition***) and the SCSI Bus ID of the Target. It releases the BSY Signal to begin the ***SELECTION Phase***.

- The Target recognizes the Selection by the Initiator and asserts the BSY Signal in response. The Initiator releases the SEL Signal in response. This completes the SELECTION Phase.

- The Target now takes charge of ***Bus Phase*** selection. Since the Initiator asserted the ATN Signal, the Target goes to ***MESSAGE OUT Phase*** (see ***Message System***). The Initiator sends the ***IDENTIFY Message*** to establish the ***Nexus*** for the ***I/O Process***. The IDENTIFY Message indicates which ***Logical Unit*** is going to receive a Command from the Initiator.

- The Target then changes to ***COMMAND Phase*** to fetch the CDB from the Initiator. The Initiator sends the CDB via DMA from Host Memory in response. The Target examines the first byte to determine how many bytes of CDB to transfer.

- After receiving and decoding the CDB, the Target determines that a seek must be performed on the Logical Unit to the location of the Logical Blocks. In other words, it has to seek to the track with the requested sectors. Since this takes some time, the Target decides to ***Disconnect*** from the SCSI Bus. To do this, it changes to the ***MESSAGE IN Phase*** and sends the ***DISCONNECT Message***. The Initiator receives the Message and clears the Active Pointers. The Target then releases the BSY Signal to go to BUS FREE Phase.

Host System	Initiator	Target	Logical Unit
An Application Program generates a Read File Request to the Operating System.			
The Operating System translates the Read File Request to a Read Blocks Request, and issues it to the SCSI Driver.			
The SCSI Driver translates the Read Blocks Request to a Host Adapter Control Block, which includes a SCSI Command Descriptor Block (CDB), and issues the Control Block to the Host Adapter.	The Initiator Arbitrates for control of the SCSI Bus.		
	The Initiator wins control of the SCSI Bus and Asserts the SEL Signal. The Initiator Asserts the ATN Signal and begins the SELECTION Phase.	The Target Asserts BSY to respond to the Selection by the Initiator.	
	The Initiator Releases the SEL Signal to complete the SELECTION Phase.	The Target changes to the MESSAGE OUT Phase in response to the Attention Condition.	
	The Initiator sends the IDENTIFY Message to establish the Nexus and Negates the ATN Signal. The Initiator copies the Saved Pointers to the Active Pointers.		
		The Target changes to COMMAND Phase to receive the CDB from the Initiator.	
The CDB is DMA transferred from Host Memory.	The Initiator sends the CDB to the Target.		
		The Target changes to MESSAGE IN Phase and sends the DISCONNECT Message.	The Logical Unit begins a Seek to the requested Blocks.
	The Initiator receives the DISCONNECT Message and suspends the I/O Process.		
		The Target Releases BSY and the Bus goes to BUS FREE Phase.	

DISK READ COMMAND EXAMPLE (1 OF 2)

The read request continues after the disk seek has completed:

- The seek completes on the Logical Unit, and the first sector of the read request begins transfer into the Target Data Buffer.

- Sometime before the end of the transfer of the first sector into the Target Data Buffer, the Target enters the ARBITRATION Phase (after validating the BUS FREE Phase) to get control of the bus. It asserts the BSY Signal and its own SCSI Bus ID.

- The Target wins Arbitration by having the highest SCSI Bus ID asserted. The Target then takes control of the bus by asserting the SEL Signal. It then asserts the I/O Signal (to choose the ***RESELECTION Phase***) and the SCSI Bus ID of the Initiator. It releases the BSY Signal to begin the RESELECTION Phase.

- The Initiator recognizes the Reselection by the Target and asserts the BSY Signal in response. The Target asserts the BSY Signal and releases the SEL Signal in response. When the Initiator sees the SEL Signal go ***False***, it releases the BSY Signal. This completes the RESELECTION Phase.

- The Target again takes charge of Bus Phase selection. The first Phase after Reselection is always the MESSAGE IN Phase (see ***Message System***). The Target sends the IDENTIFY Message to re-establish the Nexus with the Initiator for the I/O Process. The IDENTIFY Message indicates which Logical Unit is going to continue a Command from the Initiator.

- The Target then changes to ***DATA IN Phase*** to begin sending the requested data to the Initiator. The Initiator passes the data via DMA to the location in Host Memory requested by the Host System. The Target continues until all data is transferred.

- After completing the data transfer, the Target changes to ***STATUS Phase*** to return completion ***Status*** to the Initiator. The Initiator passes the Status via DMA to the Host System.

- The last task of the Target for this I/O Process is to change to MESSAGE IN Phase and transfer the ***COMMAND COMPLETE Message*** to complete the I/O Process. The Initiator receives the Message and gives the Host System some indication that I/O Process has been completed. This indication is usually a system interrupt, although it may also be indicated by setting a bit in a status register on the Host Adapter.

- The SCSI Driver takes the SCSI Status, converts it into an Operating System completion code (SCSI and OS codes will seldom coincide), and returns control to the Operating System. The Operating System returns completion to the application program, which can then start using the requested file.

Host System	Initiator	Target	Logical Unit
The Host System is waiting for completion and may be doing other processing.	The Initiator is waiting for the Target to Reconnect and may be handling other SCSI I/O Processes.	The Target is waiting for the Logical Unit to be ready and may be handling other I/O Processes.	The Logical Unit Seek Completes. The Logical Unit starts transferring data into the Target Data Buffer.
		The Target Arbitrates for control of the SCSI Bus.	
		The Target wins control of the SCSI Bus and Asserts the SEL Signal. The Target Asserts the I/O Signal and begins the RESELECTION Phase.	
	The Initiator Asserts BSY to respond to the RESELECTION Phase. It Releases BSY when the Target Releases SEL.	The Target Asserts BSY and Releases SEL to end the RESELECTION Phase. The Target switches to MESSAGE IN Phase to send the IDENTIFY Message to re-establish the Nexus.	
	The Initiator receives the IDENTIFY Message and copies the appropriate Saved Pointers to the Active Pointers.		The Logical Unit completes the Data transfer to the Target Data Buffer.
		The Target changes to DATA IN Phase to send the Read Data to the Initiator.	
The Data is DMA transferred into Host Memory.	The Initiator receives the DATA IN from the Target and passes it on to the Host.		
		The Target changes to STATUS Phase and sends Completion Status to the Initiator.	
The Status is DMA transferred into Host Memory.	The Initiator receives the Status and passes it on to the Host.		
		The Target changes to MESSAGE IN Phase and sends the COMMAND COMPLETE Message.	
	The Initiator receives the COMMAND COMPLETE Message and closes the Nexus. The Initiator indicates to the Host System that the Command is completed.	The Target Releases BSY and the Bus goes to BUS FREE Phase.	
The Host System SCSI Driver receives the indication and returns the Data and Status back to the Operating System.			
The Operating System passes the Data and Status back to the Application.			

Disk Read Command Example (2 of 2)

Table 31 shows the SCSI Bus Phases used during the Disk Read example. The table shows the Bus ***Control Signals*** that define each phase, but does not include any REQ/ACK handshakes for clarity. The Data Bus contents are shown when appropriate; "--" indicates several bytes are transferred in the Phase.

The Initiator ***SCSI Address*** is assumed to be 7, and the Target SCSI Address is assumed to be 0.

TABLE 31: DISK READ EXAMPLE BUS PHASES

BSY	SEL	ATN	MSG	C/D	I/O	RST	Data	Phase
0	0	0	0	0	0	0	00	BUS FREE
1	0	0	0	0	0	0	80	ARBITRATION Phase
1	1	1	0	0	0	0	81	Initiator takes Bus after winning
0	1	1	0	0	0	0	81	SELECTION Phase
1	1	1	0	0	0	0	81	Target responds to Selection
1	0	1	0	0	0	0	xx	Initiator releases SEL to end SELECTION Phase
1	0	0	1	1	0	0	C0	MESSAGE OUT Phase - IDENTIFY Message (Logical Unit 0, Disconnect OK)
1	0	0	0	1	0	0	--	COMMAND Phase - Target receives CDB
1	0	0	1	1	1	0	04	MESSAGE IN Phase - DISCONNECT Message
0	0	0	0	0	0	0	00	BUS FREE Phase
1	0	0	0	0	0	0	01	ARBITRATION Phase
1	1	0	0	0	1	0	81	Target takes Bus after winning
0	1	0	0	0	1	0	81	RESELECTION Phase
1	1	0	0	0	1	0	81	Initiator responds to Reselection
1	0	0	0	0	1	0	xx	Target asserts BSY and releases SEL to end RESELECTION Phase
1	0	0	1	1	1	0	80	MESSAGE IN Phase - IDENTIFY Message (Logical Unit 0)
1	0	0	0	0	1	0	--	DATA IN Phase - Initiator receives read data
1	0	0	0	1	1	0	00	STATUS Phase - GOOD Status
1	0	0	1	1	1	0	00	MESSAGE IN Phase - COMMAND COMPLETE Message
0	0	0	0	0	0	0	00	BUS FREE Phase

This page is nearly blank!
We use the space to improve Readability.

Example #2: The second example shows a simple disk write operation. The system for Example #2 is the same as that for Example #1, as shown in Diagram 61.

Like the read request, the file write request must filter down through the different layers of the Host System, as shown in Diagram 63. At each level, the request is translated into a standard form understood by the next lowest level. Note that the diagram shows the flow of information between levels; it does not show the time order of that flow.

- An application program on the Host System makes a request to the Operating System to write a file; for example, a word processing program saves a user document. The application specifies to the Operating System which file, how much of the file to write, and where to get the data.

- The Operating System takes the file write request from the application. Using its internal file system data structures, it determines which file system blocks are free for writing in a file. From this information, the Operating System creates a write disk block request and issues it to the SCSI Driver program.

- The SCSI Driver program takes the block write request from the Operating System. It converts the file system block write request into a SCSI ***Command Descriptor Block (CDB)***. It then incorporates the CDB into a Host Adapter Control Block (see ***Host Adapter***). This Control Block is set up in Host Memory. The Host System then tells the Host Adapter to begin executing the Command specified in the Control Block.

- When the SCSI ***I/O Process*** is completed, ***Status*** has been returned and stored in the Host Adapter Control Block. The data has been transferred directly to the application data area. If the Command caused "CHECK CONDITION" Status, the Host Adapter may also have fetched Sense Data from the Target disk drive.

- The SCSI Driver translates the returned Status and Sense Data (if any) to driver completion codes understood by the Operating System and passes them back up.

- The Operating System passes OS completion codes back to the application. The operation is complete, and the application has saved its data.

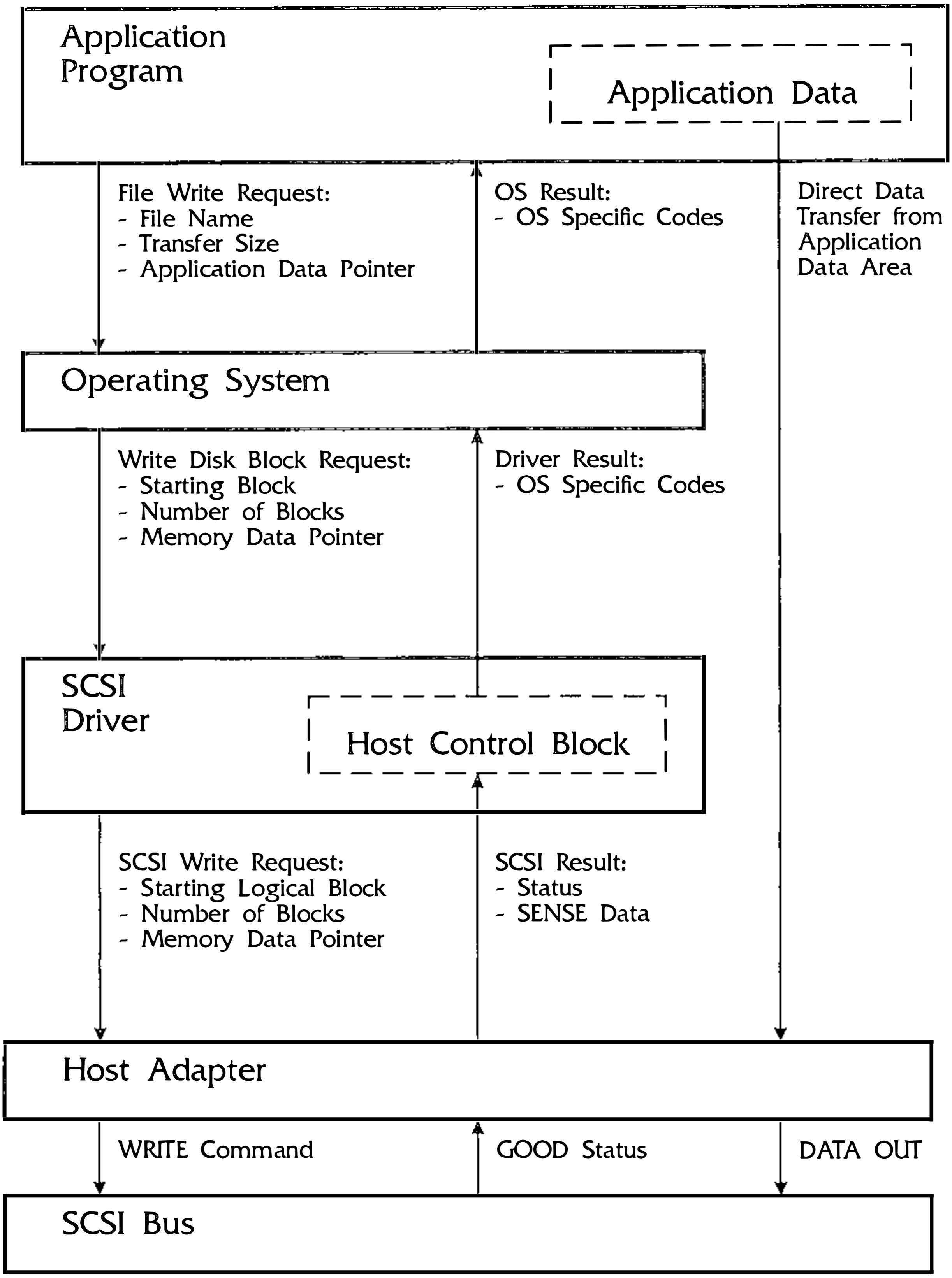


DIAGRAM 63: DISK WRITE EXAMPLE SOFTWARE LAYERS

The Disk Write Command example begins with the request for saving a file by the application program, as described above. We'll pick it up after the Host Adapter receives the Host Adapter Control Block from the SCSI Driver:

- The Host Adapter performs the Initiator function for the Host System. The Initiator enters the ***ARBITRATION Phase*** (after validating the ***BUS FREE Phase***) to get control of the bus. It asserts the ***BSY Signal*** and its own ***SCSI Bus ID***.

- The Initiator wins Arbitration by having the highest SCSI Bus ID asserted. The Initiator then takes control of the bus by asserting the ***SEL Signal***. It then asserts the ***ATN Signal*** (to create the ***Attention Condition***) and the SCSI Bus ID of the Target. It releases the BSY Signal to begin the ***SELECTION Phase***.

- The Target recognizes the Selection by the Initiator and asserts the BSY Signal in response. The Initiator releases the SEL Signal in response. This completes the SELECTION Phase.

- The Target now takes charge of ***Bus Phase*** selection. Since the Initiator asserted the ATN Signal, the Target goes to ***MESSAGE OUT Phase*** (see ***Message System***). The Initiator sends the ***IDENTIFY Message*** to establish the ***Nexus*** for the ***I/O Process***. The IDENTIFY Message indicates which ***Logical Unit*** is going to receive a Command from the Initiator.

- The Target then changes to ***COMMAND Phase*** to fetch the CDB from the Initiator. The Initiator sends the CDB via DMA from Host Memory in response. The Target examines the first byte to determine how many bytes of CDB to transfer.

- After receiving and decoding the CDB, the Target determines that a seek must be performed on the Logical Unit to the location of the Logical Blocks. In other words, it has to seek to the track with the requested sectors. While the seek is proceeding, and since this is a Write Command, the Target then changes to ***DATA OUT Phase*** to begin sending the requested data to the Initiator. The Initiator passes the data via DMA to the location in Host Memory requested by the Host System. The Target continues until all data is transferred. By overlapping the seek with the data transfer the Target saves command processing time.

Host System	Initiator	Target	Logical Unit
An Application Program generates a Write File Request to the Operating System.			
The Operating System translates the Write File Request to a Write Blocks Request, and issues it to the SCSI Driver.			
The SCSI Driver translates the Write Blocks Request to a Host Adapter Control Block, which includes a SCSI Command Descriptor Block (CDB), and issues the Control Block to the Host Adapter.	The Initiator Arbitrates for control of the SCSI Bus.		
	The Initiator wins control of the SCSI Bus and Asserts the SEL Signal. The Initiator Asserts the ATN Signal and begins the SELECTION Phase.	The Target Asserts BSY to respond to the Selection by the Initiator.	
	The Initiator Releases the SEL Signal to complete the SELECTION Phase.	The Target changes to the MESSAGE OUT Phase in response to the Attention Condition.	
	The Initiator sends the IDENTIFY Message to establish the Nexus and Negates the ATN Signal. The Initiator copies the Saved Pointers to the Active Pointers.		
		The Target changes to COMMAND Phase to receive the CDB from the Initiator.	
The CDB is DMA transferred from Host Memory.	The Initiator sends the CDB to the Target.		
		The Target changes to DATA OUT Phase and receives the write data from the Initiator.	The Logical Unit begins a Seek to the requested Blocks.
The write data is DMA transferred from Host Memory.	The Initiator sends the write data to the Target.		

Disk Write Command Example (1 of 2)

The write request continues after the DATA OUT Phase:

- The seek completes on the Logical Unit, and the first sector of the write request begins transfer from the Target Data Buffer to the disk.

- Since there is still data to write to the disk, and this takes some time, the Target decides to ***Disconnect*** from the SCSI Bus. To do this, it changes to the ***MESSAGE IN Phase*** and sends the ***DISCONNECT Message***. The Initiator receives the Message and clears the Active Pointers. The Target then releases the BSY Signal to go to BUS FREE Phase.

- Sometime before the end of the transfer of the last sector from the Target Data Buffer to the disk, the Target enters the ARBITRATION Phase (after validating the BUS FREE Phase) to get control of the bus. It asserts the BSY Signal and its own SCSI Bus ID.

- The Target wins Arbitration by having the highest SCSI Bus ID asserted. The Target then takes control of the bus by asserting the SEL Signal. It then asserts the I/O Signal (to choose the ***RESELECTION Phase***) and the SCSI Bus ID of the Initiator. It releases the BSY Signal to begin the RESELECTION Phase.

- The Initiator recognizes the Reselection by the Target and asserts the BSY Signal in response. The Target asserts the BSY Signal and releases the SEL Signal in response. When the Initiator sees the SEL Signal go ***False***, it releases the BSY Signal. This completes the RESELECTION Phase.

- The Target again takes charge of Bus Phase selection. The first Phase after Reselection is always the MESSAGE IN Phase (see ***Message System***). The Target sends the IDENTIFY Message to re-establish the Nexus with the Initiator for the I/O Process. The IDENTIFY Message indicates which Logical Unit is going to continue a Command from the Initiator.

- After completing the Message transfer, the Target changes to ***STATUS Phase*** to return completion ***Status*** to the Initiator. The Initiator passes the Status via DMA to the Host System.

- The last task of the Target for this I/O Process is to change to MESSAGE IN Phase and transfer the ***COMMAND COMPLETE Message*** to complete the I/O Process. The Initiator receives the Message and gives the Host System some indication that the I/O Process has been completed. This indication is usually a system interrupt, although it may also be indicated by setting a bit in a status register on the Host Adapter.

- The SCSI Driver takes the SCSI Status, converts it into an Operating System completion code (SCSI and OS codes will seldom coincide), and returns control to the Operating System. The Operating System returns completion to the application program, which then knows the data has been saved.

Host System	Initiator	Target	Logical Unit
		The Target changes to MESSAGE IN Phase and sends the DISCONNECT Message.	
	The Initiator receives the DISCONNECT Message and suspends the I/O Process.	The Target Releases BSY and the Bus goes to BUS FREE Phase.	The Logical Unit Seek Completes. The Logical Unit starts transferring data from the Target Data Buffer to the hard disk.
The Host System is waiting for completion and may be doing other processing.	The Initiator is waiting for the Target to Reconnect and may be handling other SCSI I/O Processes.	The Target is waiting for the Logical Unit to finish and may be handling other I/O Processes.	The Logical Unit finishes writing data on the hard disk.
		The Target Arbitrates for control of the SCSI Bus.	
		The Target wins control of the SCSI Bus and Asserts the SEL Signal. The Target Asserts the I/O Signal and begins the RESELECTION Phase.	
	The Initiator Asserts BSY to respond to the RESELECTION Phase. It Releases BSY when the Target Releases SEL.	The Target Asserts BSY and Releases SEL to end the RESELECTION Phase. The Target switches to MESSAGE IN Phase to send the IDENTIFY Message to re-establish the Nexus.	
	The Initiator receives the IDENTIFY Message and copies the Saved Pointers to the Active Pointers.		The Logical Unit completes the Data transfer to the Target Data Buffer.
		The Target changes to STATUS Phase and sends Completion Status to the Initiator.	
The Status is DMA transferred into Host Memory.	The Initiator receives the Status and passes it on to the Host.		
		The Target changes to MESSAGE IN Phase and sends the COMMAND COMPLETE Message.	
	The Initiator receives the COMMAND COMPLETE Message and closes the Nexus. The Initiator indicates to the Host System that the Command is completed.	The Target Releases BSY and the Bus goes to BUS FREE Phase.	
The Host System SCSI Driver receives the indication and returns the Status back to the Operating System.			
The Operating System passes the Data and Status back to the Application.			

DISK WRITE COMMAND EXAMPLE (2 OF 2)

Table 32 shows the SCSI Bus Phases used during the Disk Write example. The table shows the Bus ***Control Signals*** that define each phase, but does not include any REQ/ACK handshakes for clarity. The Data Bus contents are shown when appropriate; "--" indicates several bytes are transferred in the Phase.

The Initiator ***SCSI Address*** is assumed to be 5, and the Target SCSI Address is assumed to be 2.

TABLE 32: DISK WRITE EXAMPLE BUS PHASES

BSY	SEL	ATN	MSG	C/D	I/O	RST	Data	Phase
0	0	0	0	0	0	0	00	BUS FREE
1	0	0	0	0	0	0	20	ARBITRATION Phase
1	1	1	0	0	0	0	24	Initiator takes Bus after winning
0	1	1	0	0	0	0	24	SELECTION Phase
1	1	1	0	0	0	0	24	Target responds to Selection
1	0	1	0	0	0	0	xx	Initiator releases SEL to end SELECTION Phase
1	0	0	1	1	0	0	C0	MESSAGE OUT Phase - IDENTIFY Message (Logical Unit 0, Disconnect OK)
1	0	0	0	1	0	0	--	COMMAND Phase - Target receives CDB
1	0	0	0	0	0	0	--	DATA OUT Phase - Initiator sends write data
1	0	0	1	1	1	0	04	MESSAGE IN Phase - DISCONNECT Message
0	0	0	0	0	0	0	00	BUS FREE Phase
1	0	0	0	0	0	0	04	ARBITRATION Phase
1	1	0	0	0	1	0	24	Target takes Bus after winning
0	1	0	0	0	1	0	24	RESELECTION Phase
1	1	0	0	0	1	0	24	Initiator responds to Reselection
1	0	0	0	0	1	0	xx	Target asserts BSY and releases SEL to end RESELECTION Phase
1	0	0	1	1	1	0	80	MESSAGE IN Phase - IDENTIFY Message (Logical Unit 0)
1	0	0	0	1	1	0	00	STATUS Phase - GOOD Status
1	0	0	1	1	1	0	00	MESSAGE IN Phase - COMMAND COMPLETE Message
0	0	0	0	0	0	0	00	BUS FREE Phase

This page is nearly blank!
We use the space to improve Readability.

Example #3: For a change of pace, the third example shows a tape restore operation. Diagram 64 shows the system for this example. Like the previous examples, the Host System has a system bus, such as the VME Bus or the EISA Bus, which connects to a ***Host Adapter***. The Host Adapter performs the ***Initiator*** function for the Host System. Unlike the previous examples, this Host Adapter communicates with the Host System strictly via I/O Port access on the system bus. These I/O Ports are used to transfer all Commands, Data, and Status between the Host System and the ***SCSI Bus***. In other words, this is a classic simple Peripheral I/O Adapter.

The ***Target*** is a simple (!) "Embedded" SCSI Tape Drive with a single ***Logical Unit*** that corresponds to the physical tape transport and head mechanism. The physical blocks recorded on the tape are mapped to SCSI ***Logical Blocks***. The Target contains a large "Data Buffer" consisting of a local memory block that holds blocks during a transfer:

- When writing, the Data Buffer holds the data from the Host System prior to writing the data to the tape.

- When reading, the Data Buffer holds the data read from the tape prior to transfer to the Host System.

The tape transport is a relatively slow mechanism that advances the tape past the heads. The heads record data on the tape and read it back. The physical tape blocks correspond directly to the desired Logical Blocks. Moving the tape at all takes a relatively long period of time to complete. As a result, the physical transfer rate is very slow relative to the capability of the SCSI Bus. Therefore, it is desirable to use the Data Buffer to make use of the SCSI Bus more efficient:

- When writing, the Data Buffer is filled by data from the Host System. The Target then ***Disconnects*** from the Bus to perform the actual write to tape. The Target ***Reconnects*** to the Bus when the Data Buffer is (nearly) empty.

- When reading, the Target Disconnects from the Bus after receiving the read request. Offline, the Data Buffer is filled by data from the tape. When the buffer is (nearly) full, the Target Reconnects to the Bus to send the data.

In other words, this is a classic Intelligent Tape Drive. As we will see in the other volumes of this Encyclopedia, an Intelligent Tape Drive is called a Sequential Access Device in SCSI.

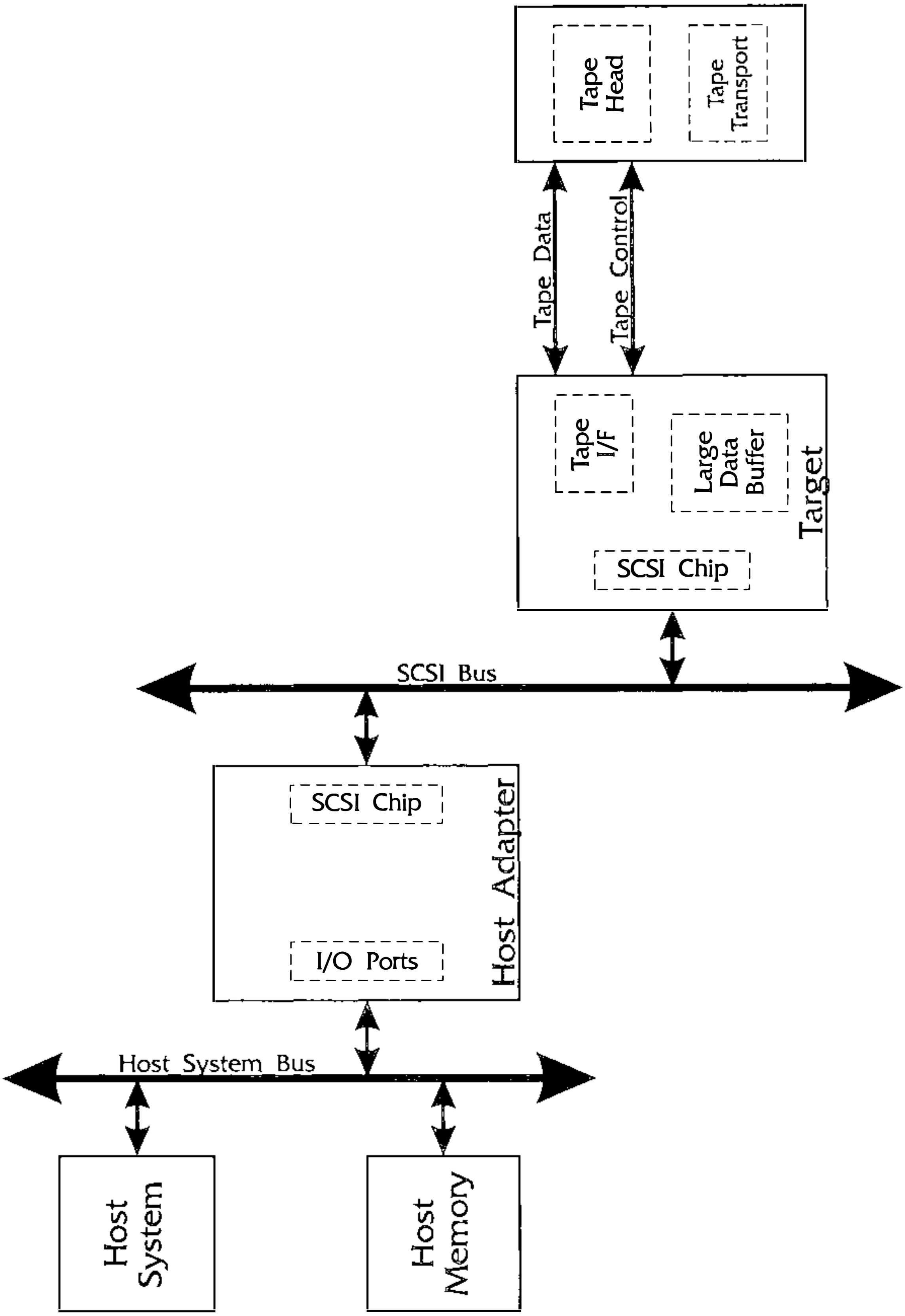


DIAGRAM 64: TAPE EXAMPLE SYSTEM ARCHITECTURE

The tape restore request must filter down through the different layers of the Host System, as shown in Diagram 65. In this example, the application program bypasses the Operating System because the Operating System does not support tape. At each level, the request is translated into a standard form understood by the next lowest level. Note that the diagram shows the flow of information between levels; it does not show the time order of that flow.

- An application program on the Host System makes a request directly to the SCSI Driver (bypassing the Operating System) to read the tape; for example, a backup utility is going to restore a system file from the tape. The first step is to read the file from the tape. The application specifies to the SCSI Driver how many blocks to read from the tape, and where to put the data.

- The SCSI Driver program takes the tape read request from the application. It converts the tape block read request into a SCSI ***Command Descriptor Block (CDB)***. It then issues a command to the Host Adapter to Select the Target and send the CDB. More commands are issued to the Host Adapter until the operation is complete. Note that the SCSI Driver manages the ***I/O Process***: it responds to Phases and maintains the ***Pointers***.

- When the SCSI ***I/O Process*** is completed, ***Status*** has been returned to the SCSI Driver. The data has been transferred directly to the application data area. If the Command caused "CHECK CONDITION" Status, the SCSI Driver may also have fetched Sense Data from the Target disk drive. The SCSI Driver translates the returned Status and Sense Data (if any) to driver completion codes understood by the application and passes them back up.

- The operation is complete, and the application has its data.

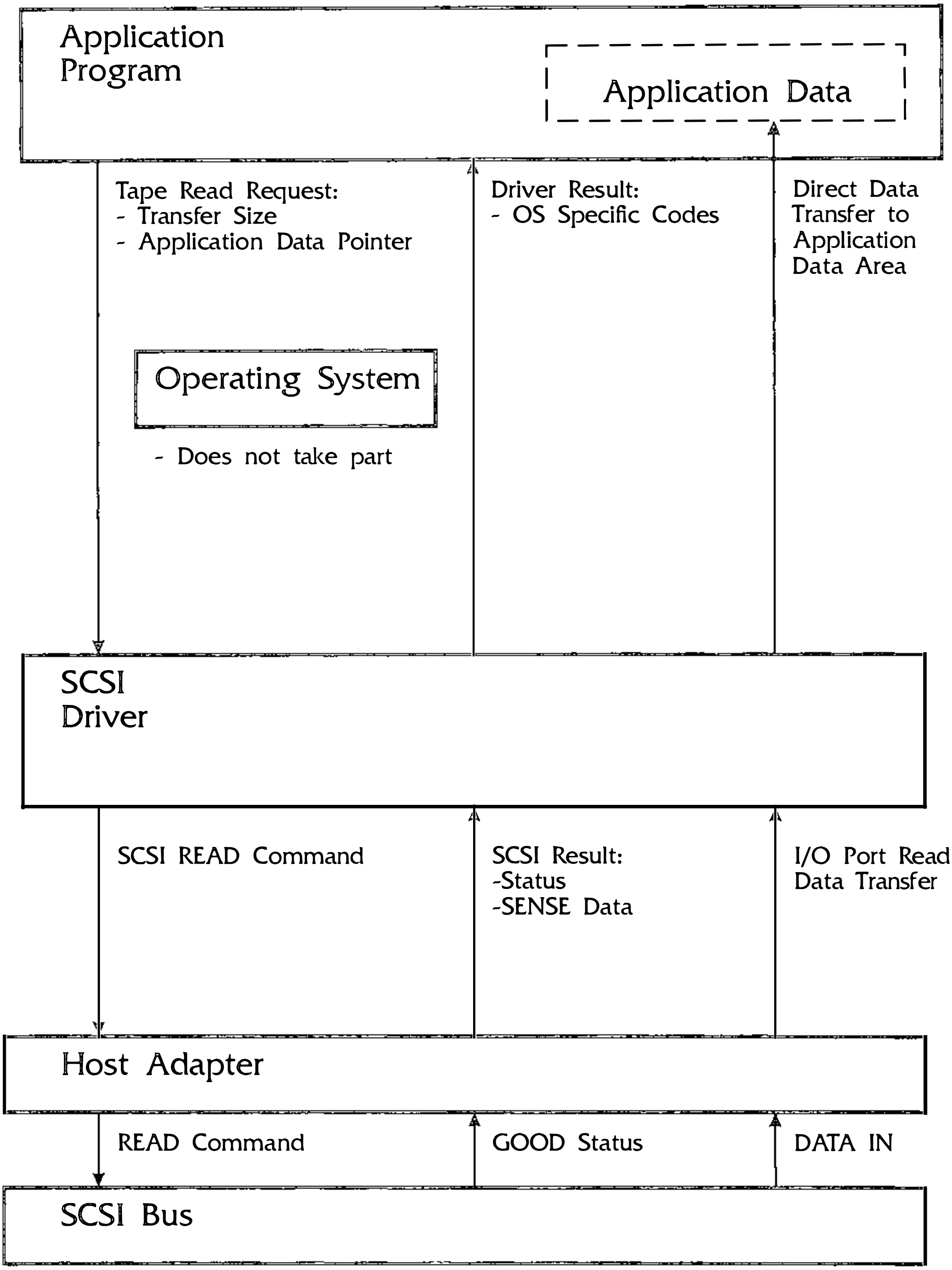


DIAGRAM 65: TAPE READ EXAMPLE SOFTWARE LEVELS

The Tape Read Command example begins with the request for a file by the application program, as described above. We'll pick it up after the SCSI Driver receives request from the application:

- The Host Adapter under control of the SCSI Driver performs the Initiator function for the Host System. The Initiator enters the ***ARBITRATION Phase*** (after validating the ***BUS FREE Phase***) to get control of the bus. It asserts the ***BSY Signal*** and its own ***SCSI Bus ID***.

- The Initiator wins Arbitration by having the highest SCSI Bus ID asserted. The Initiator then takes control of the bus by asserting the ***SEL Signal***. It then asserts the ***ATN Signal*** (to create the ***Attention Condition***) and the SCSI Bus ID of the Target. It releases the BSY Signal to begin the ***SELECTION Phase***.

- The Target recognizes the Selection by the Initiator and asserts the BSY Signal in response. The Initiator releases the SEL Signal in response. This completes the SELECTION Phase.

- The Target now takes charge of ***Bus Phase*** selection. Since the Initiator asserted the ATN Signal, the Target goes to ***MESSAGE OUT Phase*** (see ***Message System***). The Initiator sends the ***IDENTIFY Message*** to establish the ***Nexus*** for the ***I/O Process***. The IDENTIFY Message indicates which ***Logical Unit*** is going to receive a Command from the Initiator.

- The Target then changes to ***COMMAND Phase*** to fetch the CDB from the Initiator. The Initiator sends the CDB via an I/O Port transfer with the SCSI Driver in response. The Target examines the first byte to determine how many bytes of CDB to transfer.

- After receiving and decoding the CDB, the Target begins reading data from the tape. Since this takes some time, the Target decides to ***Disconnect*** from the SCSI Bus. To do this, it changes to the ***MESSAGE IN Phase*** and sends the ***DISCONNECT Message***. The Initiator receives the Message and clears the Active Pointers. The Target then releases the BSY Signal to go to BUS FREE Phase.

Host System	Initiator	Target	Logical Unit
An Application Program generates a Read File Request to the SCSI Driver.			The Logical Unit (i.e., the tape) is currently positioned where the Application wants it.
The SCSI Driver translates the Read Blocks Request to a SCSI Command Descriptor Block (CDB), and issues a Selection command to the Host Adapter.	The Initiator Arbitrates for control of the SCSI Bus.		
	The Initiator wins control of the SCSI Bus and Asserts the SEL Signal. The Initiator Asserts the ATN Signal and begins the SELECTION Phase.	The Target Asserts BSY to respond to the Selection by the Initiator.	
	The Initiator Releases the SEL Signal to complete the SELECTION Phase.	The Target changes to the MESSAGE OUT Phase in response to the Attention Condition.	
	The Initiator sends the IDENTIFY Message to establish the Nexus and Negates the ATN Signal. The Initiator copies the Saved Pointers to the Active Pointers.		
		The Target changes to COMMAND Phase to receive the CDB from the Initiator.	
The CDB is transferred to the Host Adapter by the SCSI Driver.	The Initiator sends the CDB to the Target.		
		The Target changes to MESSAGE IN Phase and sends the DISCONNECT Message.	The Logical Unit begins transferring data from the tape to the Target Data Buffer.
	The Initiator receives the DISCONNECT Message and suspends the I/O Process.		
		The Target Releases BSY and the Bus goes to BUS FREE Phase.	

TAPE READ COMMAND EXAMPLE (1 OF 3)

The read request continues after the Target Data Buffer is (nearly) full of data from the tape:

- Sometime before the Target Data Buffer is actually full, the Target enters the ARBITRATION Phase (after validating the BUS FREE Phase) to get control of the bus. It asserts the BSY Signal and its own SCSI Bus ID.

- The Target wins Arbitration by having the highest SCSI Bus ID asserted. The Target then takes control of the bus by asserting the SEL Signal. It then asserts the I/O Signal (to choose the ***RESELECTION Phase***) and the SCSI Bus ID of the Initiator. It releases the BSY Signal to begin the RESELECTION Phase.

- The Initiator recognizes the Reselection by the Target and asserts the BSY Signal in response. The Target asserts the BSY Signal and releases the SEL Signal in response. When the Initiator sees the SEL Signal go ***False***, it releases the BSY Signal. This completes the RESELECTION Phase.

- The Target again takes charge of Bus Phase selection. The first Phase after Reselection is always the MESSAGE IN Phase (see ***Message System***). The Target sends the IDENTIFY Message to re-establish the Nexus with the Initiator for the I/O Process. The IDENTIFY Message indicates which Logical Unit is going to continue a Command from the Initiator.

- The Target then changes to ***DATA IN Phase*** to begin sending the requested data to the Initiator. The Initiator passes the data via I/O Port access by the SCSI Driver, which then writes it to the location in Host Memory requested by the Host System. The Target continues until all data is transferred.

- Only half of the data has been transferred to the Initiator, and data is still coming off the tape. Since this will take a while longer, the Target decides to Disconnect from the SCSI Bus again. This time is a little different than the first time: the Target changes to the ***MESSAGE IN Phase*** and sends the ***SAVE DATA POINTER Message***. The Initiator receives the Message and copies the Active Data Pointers to the Saved Data Pointer.

- The Target then sends the DISCONNECT Message. The Initiator receives the Message and clears the Active Pointers. The Target then releases the BSY Signal to go to BUS FREE Phase. The transfer of data from tape to Data Buffer continues.

Host System	Initiator	Target	Logical Unit
The Host System SCSI Driver is waiting for the next operation and may be doing other processing.	The Initiator is waiting for the Target to Reconnect and may be handling other SCSI I/O Processes.	The Target is waiting for the Logical Unit to be ready and may be handling other I/O Processes.	The Logical Unit continues transferring data from the tape into the Target Data Buffer.
		The Target Arbitrates for control of the SCSI Bus.	
		The Target wins control of the SCSI Bus and Asserts the SEL Signal. The Target Asserts the I/O Signal and begins the RESELECTION Phase.	
	The Initiator Asserts BSY to respond to the RESELECTION Phase. It Releases BSY when the Target Releases SEL.	The Target Asserts BSY and Releases SEL to end the RESELECTION Phase. The Target switches to MESSAGE IN Phase to send the IDENTIFY Message to re-establish the Nexus.	
	The Initiator receives the IDENTIFY Message and copies the appropriate Saved Pointers to the Active Pointers.		
		The Target changes to DATA IN Phase to send the Read Data to the Initiator.	
The Data is I/O Port transferred from the Host Adapter into Host Memory.	The Initiator receives the DATA IN from the Target and passes it on to the Host.		
		The Target changes to MESSAGE IN Phase and sends the SAVE DATA POINTER Message.	
	The Initiator receives the SAVE DATA POINTER Message and copies the Active Data Pointer to the Saved Data Pointer.		
		The Target continues in MESSAGE IN Phase and sends the DISCONNECT Message.	
	The Initiator receives the DISCONNECT Message and suspends the I/O Process.		
		The Target Releases BSY and the Bus goes to BUS FREE Phase.	The Logical Unit continues transferring data into the Target Data Buffer.

TAPE READ COMMAND EXAMPLE (2 OF 3)

The read request again continues after the Target Data Buffer is (nearly) full of data from the tape:

- Sometime before the Target Data Buffer is actually full or before the transfer from tape is completed (whichever happens first), the Target enters the ARBITRATION Phase (after validating the BUS FREE Phase) to get control of the bus. It asserts the BSY Signal and its own SCSI Bus ID.

- The Target wins Arbitration by having the highest SCSI Bus ID asserted. The Target then takes control of the bus by asserting the SEL Signal. It then asserts the I/O Signal (to choose the RESELECTION Phase) and the SCSI Bus ID of the Initiator. It releases the BSY Signal to begin the RESELECTION Phase.

- The Initiator recognizes the Reselection by the Target and asserts the BSY Signal in response. The Target asserts the BSY Signal and releases the SEL Signal in response. When the Initiator sees the SEL Signal go False, it releases the BSY Signal. This completes the RESELECTION Phase.

- The Target again takes charge of Bus Phase selection. The first Phase after Reselection is always the MESSAGE IN Phase (see ***Message System***). The Target sends the IDENTIFY Message to re-establish the Nexus with the Initiator for the I/O Process. The IDENTIFY Message indicates which Logical Unit is going to continue a Command from the Initiator.

- The Target then changes to ***DATA IN Phase*** to begin sending the requested data to the Initiator. The Initiator passes the data via I/O Port access by the SCSI Driver, which then writes it to the location in Host Memory requested by the Host System. The Target continues until all data is transferred.

- After completing the data transfer, the Target changes to ***STATUS Phase*** to return completion ***Status*** to the Initiator. The Initiator passes the Status via DMA to the Host System.

- The last task of the Target for this I/O Process is to change to MESSAGE IN Phase and transfer the ***COMMAND COMPLETE Message*** to complete the I/O Process. The Initiator receives the Message and gives the Host System some indication that I/O Process has been completed. This indication is usually a system interrupt, although it may also be indicated by setting a bit in a status register on the Host Adapter.

- The SCSI Driver takes the SCSI Status, converts it into a Driver or application completion code (SCSI and application completion codes will seldom coincide), and returns control to the application. The application program can then start using the requested file.

Host System	Initiator	Target	Logical Unit
The Host System SCSI Driver is waiting for the next operation and may be doing other processing.	The Initiator is waiting for the Target to Reconnect and may be handling other SCSI I/O Processes.	The Target is waiting for the Logical Unit to be ready and may be handling other I/O Processes.	The Logical Unit continues transferring data into the Target Data Buffer.
		The Target Arbitrates for control of the SCSI Bus.	
		The Target wins control of the SCSI Bus and Asserts the SEL Signal. The Target Asserts the I/O Signal and begins the RESELECTION Phase.	
	The Initiator Asserts BSY to respond to the RESELECTION Phase. It Releases BSY when the Target Releases SEL.	The Target Asserts BSY and Releases SEL to end the RESELECTION Phase. The Target switches to MESSAGE IN Phase to send the IDENTIFY Message to re-establish the Nexus.	
	The Initiator receives the IDENTIFY Message and copies the appropriate Saved Pointers to the Active Pointers.		
		The Target changes to DATA IN Phase to send the Read Data to the Initiator.	The Logical Unit completes the Data transfer to the Target Data Buffer.
The Data is I/O Port transferred from the Host Adapter into Host Memory.	The Initiator receives the DATA IN from the Target and passes it on to the Host.		
		The Target changes to STATUS Phase and sends Completion Status.	
The Status is received by the SCSI Driver via I/O Port access.	The Initiator receives the Status from the Target.		
		The Target changes to MESSAGE IN Phase and sends the COMMAND COMPLETE Message.	
	The Initiator receives the COMMAND COMPLETE Message and closes the Nexus. The Initiator indicates to the Host System that the Command is completed.	The Target Releases BSY and the Bus goes to BUS FREE Phase.	
The Host System SCSI Driver receives the indication via I/O Port access and returns the Status back to the Application.			

TAPE READ COMMAND EXAMPLE (3 OF 3)

Table 33 shows the SCSI Bus Phases used during the Disk Read example. The table shows the Bus ***Control Signals*** that define each phase, but does not include any REQ/ACK handshakes for clarity. The Data Bus contents are shown when appropriate; "--" indicates several bytes are transferred in the Phase.

The Initiator ***SCSI Address*** is assumed to be 1, and the Target SCSI Address is assumed to be 0.

TABLE 33: TAPE READ EXAMPLE BUS PHASES

BSY	SEL	ATN	MSG	C/D	I/O	RST	Data	Phase
0	0	0	0	0	0	0	00	BUS FREE
1	0	0	0	0	0	0	02	ARBITRATION Phase
1	1	1	0	0	0	0	03	Initiator takes Bus after winning
0	1	1	0	0	0	0	03	SELECTION Phase
1	1	1	0	0	0	0	03	Target responds to Selection
1	0	1	0	0	0	0	xx	Initiator releases SEL to end SELECTION Phase
1	0	0	1	1	0	0	C0	MESSAGE OUT Phase - IDENTIFY Message (Logical Unit 0, Disconnect OK)
1	0	0	0	1	0	0	--	COMMAND Phase - Target receives CDB
1	0	0	1	1	1	0	04	MESSAGE IN Phase - DISCONNECT Message
0	0	0	0	0	0	0	00	BUS FREE Phase
1	0	0	0	0	0	0	01	ARBITRATION Phase
1	1	0	0	0	1	0	03	Target takes Bus after winning
0	1	0	0	0	1	0	03	RESELECTION Phase
1	1	0	0	0	1	0	03	Initiator responds to Reselection
1	0	0	0	0	1	0	xx	Target asserts BSY and releases SEL to end RESELECTION Phase
1	0	0	1	1	1	0	80	MESSAGE IN Phase - IDENTIFY Message (Logical Unit 0)
1	0	0	0	0	1	0	--	DATA IN Phase - Initiator receives read data
1	0	0	1	1	1	0	02	MESSAGE IN Phase - SAVE DATA POINTER Message
1	0	0	1	1	1	0	04	MESSAGE IN Phase - DISCONNECT Message
0	0	0	0	0	0	0	00	BUS FREE Phase
1	0	0	0	0	0	0	01	ARBITRATION Phase
1	1	0	0	0	1	0	03	Target takes Bus after winning
0	1	0	0	0	1	0	03	RESELECTION Phase
1	1	0	0	0	1	0	03	Initiator responds to Reselection
1	0	0	0	0	1	0	xx	Target asserts BSY and releases SEL to end RESELECTION Phase
1	0	0	1	1	1	0	80	MESSAGE IN Phase - IDENTIFY Message (Logical Unit 0)
1	0	0	0	0	1	0	--	DATA IN Phase - Initiator receives read data
1	0	0	0	1	1	0	00	STATUS Phase - GOOD Status
1	0	0	1	1	1	0	00	MESSAGE IN Phase - COMMAND COMPLETE Message
0	0	0	0	0	0	0	00	BUS FREE Phase

This page is nearly blank!
We use the space to improve Readability.

Example #4: The fourth example shows a tape backup operation (WRITE to tape), but this time the SCSI Bus is real flaky, so we have to go through some ***Error Recovery***. Diagram 64 from the third example shows the system for this example.

Let's recall from the third example that the Target is a "slow" tape drive that contains a large "Data Buffer" consisting of a local memory block that holds blocks during a transfer. When writing, the Data Buffer is filled by data from the Host System. The Target then ***Disconnects*** from the Bus to perform the actual write to tape. The Target ***Reconnects*** to the Bus when the Data Buffer is (nearly) empty.

As with the tape restore, the tape backup request must filter down through the different layers of the Host System, as shown in Diagram 66. In this example, the application program bypasses the Operating System because the Operating System does not support tape. At each level, the request is translated into a standard form understood by the next lowest level. Note that the diagram shows the flow of information between levels; it does not show the time order of that flow.

- An application program on the Host System makes a request directly to the SCSI Driver (bypassing the Operating System) to write the tape; for example, a backup utility is going to save a system file to the tape. Prior to this step the file, or the first part of the file, is read into Host System memory. The next step is to write the file to the tape. The application specifies to the SCSI Driver how many blocks to write to the tape, and where to get the data.

- The SCSI Driver program takes the tape write request from the application. It converts the tape block write request into a SCSI ***Command Descriptor Block (CDB)***. It then issues a command to the Host Adapter to Select the Target and send the CDB. More commands are issued to the Host Adapter until the operation is complete. Note that the SCSI Driver manages the ***I/O Process***: it responds to Phases and maintains the ***Pointers***.

- When the SCSI ***I/O Process*** is completed, ***Status*** has been returned to the SCSI Driver. The data has been transferred directly from the application data area to the tape. If the Command caused "CHECK CONDITION" Status, the SCSI Driver may also have fetched Sense Data from the Target disk drive. The SCSI Driver translates the returned Status and Sense Data (if any) to driver completion codes understood by the application and passes them back up.

- The operation is complete, and the application has saved the data.

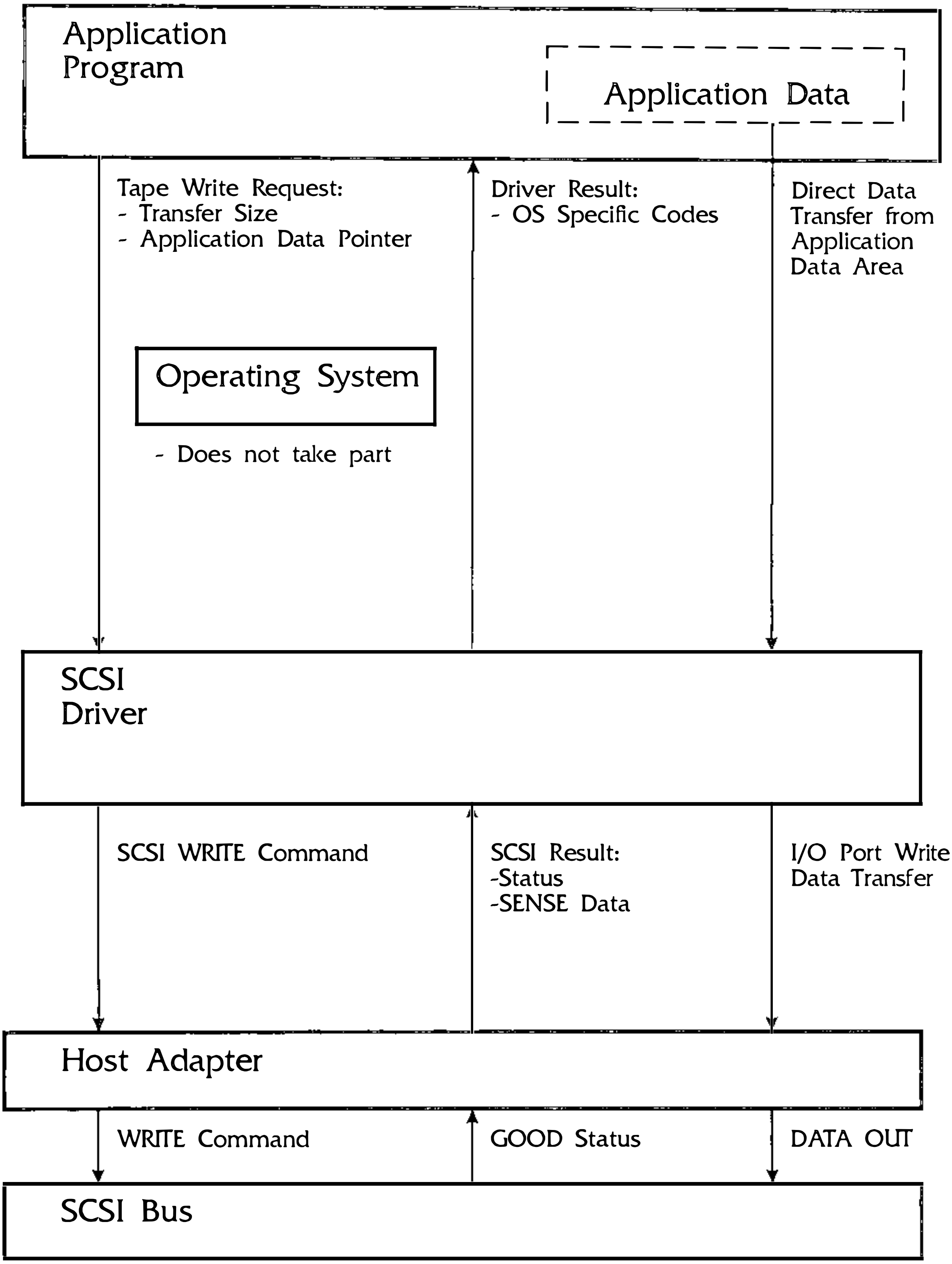


DIAGRAM 66: TAPE WRITE EXAMPLE SOFTWARE LEVELS

The Tape Write Command example begins with the request to write a file by the application program, as described above. We'll pick it up after the SCSI Driver receives request from the application:

- The Host Adapter under control of the SCSI Driver performs the Initiator function for the Host System. The Initiator enters the ***ARBITRATION Phase*** (after validating the ***BUS FREE Phase***) to get control of the bus. It asserts the ***BSY Signal*** and its own ***SCSI Bus ID***.

- The Initiator wins Arbitration by having the highest SCSI Bus ID asserted. The Initiator then takes control of the bus by asserting the ***SEL Signal***. It then asserts the ***ATN Signal*** (to create the ***Attention Condition***) and the SCSI Bus ID of the Target. It releases the BSY Signal to begin the ***SELECTION Phase***.

- The Target recognizes the Selection by the Initiator and asserts the BSY Signal in response. The Initiator releases the SEL Signal in response. This completes the SELECTION Phase.

- The Target now takes charge of ***Bus Phase*** selection. Since the Initiator asserted the ATN Signal, the Target goes to ***MESSAGE OUT Phase*** (see ***Message System***). The Initiator sends the ***IDENTIFY Message*** to establish the ***Nexus*** for the ***I/O Process***. The IDENTIFY Message indicates which ***Logical Unit*** is going to receive a Command from the Initiator.

- The Target then changes to ***COMMAND Phase*** to fetch the CDB from the Initiator. The Initiator sends the CDB via an I/O Port transfer with the SCSI Driver in response. The Target examines the first byte to determine how many bytes of CDB to transfer.

- The Target then changes to ***DATA OUT Phase*** to begin receiving the write data from the Initiator. The Initiator sends the data via I/O Port access by the SCSI Driver, which it got from the location in Host Memory requested by the Host System. The Target continues until its Data Buffer is full.

- When the Target gets enough data from the Initiator, it begins writing data to the tape. Since this takes some time, the Target decides to ***Disconnect*** from the SCSI Bus. To do this, it changes to the ***MESSAGE IN Phase*** and sends the ***SAVE DATA POINTER Message***, because it hasn't transferred all the data yet. The Initiator receives the Message, copies the Active Data Pointers to the Saved Data Pointer, and sends the ***DISCONNECT Message***. The Initiator receives the Message and clears the Active Pointers. The Target then releases the BSY Signal to go to BUS FREE Phase.

Host System	Initiator	Target	Logical Unit
An Application Program generates a Write File Request to the SCSI Driver.			The Logical Unit (i.e., the tape) is currently positioned where the Application wants it.
The SCSI Driver translates the Write Blocks Request to a SCSI Command Descriptor Block (CDB), and issues a Selection command to the Host Adapter.	The Initiator Arbitrates for control of the SCSI Bus.		
	The Initiator wins control of the SCSI Bus and Asserts the SEL Signal. The Initiator Asserts the ATN Signal and begins the SELECTION Phase.	The Target Asserts BSY to respond to the Selection by the Initiator.	
	The Initiator Releases the SEL Signal to complete the SELECTION Phase.	The Target changes to the MESSAGE OUT Phase in response to the Attention Condition.	
	The Initiator sends the IDENTIFY Message to establish the Nexus and Negates the ATN Signal. The Initiator copies the Saved Pointers to the Active Pointers.		
		The Target changes to COMMAND Phase to receive the CDB from the Initiator.	
The CDB is transferred to the Host Adapter by the SCSI Driver.	The Initiator sends the CDB to the Target.		
		The Target changes to DATA OUT Phase to get the Write Data from the Initiator.	
The Data is I/O Port transferred from Host Memory to the Host Adapter.	The Initiator sends the DATA OUT from the Host to the Target.		
		The Target changes to MESSAGE IN Phase and sends the SAVE DATA POINTER and the DISCONNECT Message.	The Logical Unit begins transferring data to the tape from the Target Data Buffer.
	The Initiator receives the SAVE DATA POINTER and DISCONNECT Message and suspends the I/O Process.		
		The Target Releases BSY and the Bus goes to BUS FREE Phase.	

TAPE WRITE COMMAND EXAMPLE (1 OF 3)

The write request continues after the Target Data Buffer is (nearly) empty from writing data to the tape:

- Sometime before the Target Data Buffer is actually empty, the Target enters the ARBITRATION Phase (after validating the BUS FREE Phase) to get control of the bus. It asserts the BSY Signal and its own SCSI Bus ID.

- The Target wins Arbitration by having the highest SCSI Bus ID asserted. The Target then takes control of the bus by asserting the SEL Signal. It then asserts the I/O Signal (to choose the ***RESELECTION Phase***) and the SCSI Bus ID of the Initiator. It releases the BSY Signal to begin the RESELECTION Phase.

- The Initiator recognizes the Reselection by the Target and asserts the BSY Signal in response. The Target asserts the BSY Signal and releases the SEL Signal in response. When the Initiator sees the SEL Signal go ***False***, it releases the BSY Signal. This completes the RESELECTION Phase.

- The Target again takes charge of Bus Phase selection. The first Phase after Reselection is always the MESSAGE IN Phase (see ***Message System***). The Target sends the IDENTIFY Message to re-establish the Nexus with the Initiator for the I/O Process. The IDENTIFY Message indicates which Logical Unit is going to continue a Command from the Initiator. The Initiator copies the Saved Pointers for that Nexus to the Active Pointers.

- The Target then changes to DATA OUT Phase to continue receiving the data from the Initiator. The Initiator sends the data via I/O Port access by the SCSI Driver, which it got from the location in Host Memory requested by the Host System. Normally, the Target continues until its Data Buffer is full. Unfortunately, a ***Parity*** Error occurs during the transfer.

- To recover from the Parity Error, the Target changes to MESSAGE IN Phase. The Target then sends the ***RESTORE POINTERS Message*** to restart the DATA OUT Phase that had the error. The Initiator receives the Message and copies the Saved Pointers for that Nexus, which define the state of things at the start of this Connection, to the Active Pointers.

- The Target then changes to DATA OUT Phase to retry the data transfer from the Initiator. The Target continues until its Data Buffer is full, or until all data requested by the Initiator has been transferred. In this case, the latter occurs.

- The Target changes to MESSAGE IN Phase and sends the SAVE DATA POINTER Message, and then sends the DISCONNECT Message. The Initiator receives the Message and clears the Active Pointers. The Target then releases the BSY Signal to go to BUS FREE Phase. The transfer of data to tape from the Data Buffer continues.

Host System	Initiator	Target	Logical Unit
The Host System SCSI Driver is waiting for the next operation and may be doing other processing.	The Initiator is waiting for the Target to Reconnect and may be handling other SCSI I/O Processes.	The Target is waiting for the Logical Unit to be ready and may be handling other I/O Processes.	The Logical Unit continues transferring data from the Target Data Buffer to the tape.
		The Target Arbitrates for control of the SCSI Bus.	
		The Target wins control of the SCSI Bus and Asserts the SEL Signal. The Target Asserts the I/O Signal and begins the RESELECTION Phase.	
	The Initiator Asserts BSY to respond to the RESELECTION Phase. It Releases BSY when the Target Releases SEL.	The Target Asserts BSY and Releases SEL to end the RESELECTION Phase. The Target switches to MESSAGE IN Phase to send the IDENTIFY Message to re-establish the Nexus.	
	The Initiator receives the IDENTIFY Message and copies the appropriate Saved Pointers to the Active Pointers.		
		The Target changes to DATA OUT Phase to get more Write Data from the Initiator.	
The Data is I/O Port transferred from Host Memory.	The Initiator sends the DATA OUT to the Target.	The Target detects a Parity Error during the DATA OUT Phase.	
		The Target changes to MESSAGE IN Phase and sends the RESTORE POINTERS Message.	
	The Initiator receives the RESTORE POINTERS Message and copies the Saved Pointers to the Active Pointers.		
		The Target changes to DATA OUT Phase to try again.	
The Data is I/O Port transferred again.	The Initiator sends the DATA OUT to the Target.		
		The Target changes to MESSAGE IN Phase and sends the SAVE DATA POINTER and DISCONNECT Messages.	
	The Initiator receives the Messages, copies the Active Pointers to the Saved Pointers, and suspends the I/O Process.		
		The Target Releases BSY and the Bus goes to BUS FREE Phase.	The Logical Unit continues transferring data to tape from the Target Data Buffer.

TAPE WRITE COMMAND EXAMPLE (2 OF 3)

The write request again continues after the transfer from the Target Data Buffer to the tape is complete:

- When the tape write is complete, the Target enters the ARBITRATION Phase (after validating the BUS FREE Phase) to get control of the bus. It asserts the BSY Signal and its own SCSI Bus ID.

- The Target wins Arbitration by having the highest SCSI Bus ID asserted. The Target then takes control of the bus by asserting the SEL Signal. It then asserts the I/O Signal (to choose the RESELECTION Phase) and the SCSI Bus ID of the Initiator. It releases the BSY Signal to begin the RESELECTION Phase.

- The Initiator recognizes the Reselection by the Target and asserts the BSY Signal in response. The Target asserts the BSY Signal and releases the SEL Signal in response. When the Initiator sees the SEL Signal go False, it releases the BSY Signal. This completes the RESELECTION Phase.

- The Target again takes charge of Bus Phase selection. The first Phase after Reselection is always the MESSAGE IN Phase (see ***Message System***). The Target sends the IDENTIFY Message to re-establish the Nexus with the Initiator for the I/O Process. The IDENTIFY Message indicates which Logical Unit is going to continue a Command from the Initiator.

- The Target changes to ***STATUS Phase*** to return completion ***Status*** to the Initiator. Unfortunately, the Initiator detected a Parity Error during the Status transfer. Before ***Negating*** the ***ACK Signal*** of the Status transfer, the Initiator asserts the ***ATN Signal*** to create the ***Attention Condition***.

- The Target sees the ATN Signal asserted and changes to MESSAGE OUT Phase. The Initiator sends the ***INITIATOR DETECTED ERROR Message*** to the Target to indicate that it saw an error during the STATUS Phase.

- The Target changes to MESSAGE IN Phase to send the RESTORE POINTERS Message. This facilitates the retry requested by the Initiator. The Initiator copies its Saved Pointers for this Nexus to its Active Pointers.

- The Target changes to STATUS Phase again to return completion Status to the Initiator. This time it works. The Initiator passes the Status via I/O port transfer to the Host System.

....continued after the diagram...

Host System	Initiator	Target	Logical Unit
The Host System SCSI Driver is waiting for the next operation and may be doing other processing.	The Initiator is waiting for the Target to Reconnect and may be handling other SCSI I/O Processes.	The Target is waiting for the Logical Unit to be ready and may be handling other I/O Processes.	The Logical Unit completes the transfer of data from the Target Data Buffer to the tape.
		The Target Arbitrates for and wins control of the SCSI Bus. The Target Selects the Initiator and re-establishes the Nexus with the IDENTIFY Message.	
	The Initiator receives the IDENTIFY Message and copies the appropriate Saved Pointers to the Active Pointers.		
		The Target changes to STATUS Phase and sends Completion Status.	The Logical Unit completes the Data transfer to the Target Data Buffer.
The Data is I/O Port transferred from the Host Adapter into Host Memory.	The Initiator receives the Status from the Target, but detects a Parity Error, so it asserts ATN before negating ACK.		
		The Target sees the ATN signal and changes to MESSAGE OUT Phase.	
The Status is received by the SCSI Driver via I/O Port access.	The Initiator sends the INITIATOR DETECTED ERROR Message to the Target.	The Target receives the Message, switches to MESSAGE OUT Phase, and sends the RESTORE POINTERS Message.	
	The Initiator receives the Message and copies the Saved Pointers to the Active Pointers.	The Target changes to STATUS Phase attempts to send Completion Status again.	
	The Initiator receives the Status from the Target, successfully this time.		
		The Target changes to MESSAGE IN Phase and sends the COMMAND COMPLETE Message.	
	The Initiator receives the COMMAND COMPLETE Message and closes the Nexus. The Initiator indicates to the Host System that the Command is completed.	The Target Releases BSY and the Bus goes to BUS FREE Phase.	
The Host System SCSI Driver receives the indication via I/O Port access and returns the Status back to the Application.			

TAPE WRITE COMMAND EXAMPLE (3 OF 3)

- The last task of the Target for this I/O Process is to change to MESSAGE IN Phase and transfer the ***COMMAND COMPLETE Message*** to complete the I/O Process. The Initiator receives the Message and gives the Host System some indication that I/O Process has been completed. This indication is usually a system interrupt, although it may also be indicated by setting a bit in a status register on the Host Adapter.

- The SCSI Driver takes the SCSI Status, converts it into a Driver or application completion code (SCSI and application completion codes will seldom coincide), and returns control to the application.

Table 34 shows the SCSI Bus Phases used during the Tape Backup example. The table shows the Bus ***Control Signals*** that define each phase, but does not include any REQ/ACK handshakes for clarity. The Data Bus contents are shown when appropriate; "--" indicates several bytes are transferred in the Phase.

The Initiator ***SCSI Address*** is assumed to be 4, and the Target SCSI Address is assumed to be 2.

TABLE 34: TAPE WRITE EXAMPLE BUS PHASES

BSY	SEL	ATN	MSG	C/D	I/O	RST	Data	Phase
0	0	0	0	0	0	0	00	BUS FREE
1	0	0	0	0	0	0	04	ARBITRATION Phase
1	1	1	0	0	0	0	06	Initiator takes Bus after winning
0	1	1	0	0	0	0	06	SELECTION Phase
1	1	1	0	0	0	0	06	Target responds to Selection
1	0	1	0	0	0	0	xx	Initiator releases SEL to end SELECTION Phase
1	0	0	1	1	0	0	C0	MESSAGE OUT Phase - IDENTIFY Message (Logical Unit 0, Disconnect OK)
1	0	0	0	1	0	0	--	COMMAND Phase - Target receives CDB
1	0	0	0	0	0	0	--	DATA OUT Phase - Initiator sends write data
1	0	0	1	1	1	0	02	MESSAGE IN Phase - SAVE DATA POINTERS Message
1	0	0	1	1	1	0	04	MESSAGE IN Phase - DISCONNECT Message
0	0	0	0	0	0	0	00	BUS FREE Phase
1	0	0	0	0	0	0	02	ARBITRATION Phase
1	1	0	0	0	1	0	06	Target takes Bus after winning
0	1	0	0	0	1	0	06	RESELECTION Phase
1	1	0	0	0	1	0	06	Initiator responds to Reselection
1	0	0	0	0	1	0	xx	Target asserts BSY and releases SEL to end RESELECTION Phase
1	0	0	1	1	1	0	80	MESSAGE IN Phase - IDENTIFY Message (Logical Unit 0)
1	0	0	0	0	0	0	--	DATA OUT Phase - Parity Error Occurs!
1	0	0	1	1	1	0	03	MESSAGE IN Phase - RESTORE POINTERS Message to retry the Phase
1	0	0	0	0	0	0	--	DATA OUT Phase - Initiator sends the rest of the write data
1	0	0	1	1	1	0	02	MESSAGE IN Phase - SAVE DATA POINTER Message
1	0	0	1	1	1	0	04	MESSAGE IN Phase - DISCONNECT Message
0	0	0	0	0	0	0	00	BUS FREE Phase
1	0	0	0	0	0	0	02	ARBITRATION Phase
1	1	0	0	0	1	0	06	Target takes Bus after winning
0	1	0	0	0	1	0	06	RESELECTION Phase
1	1	0	0	0	1	0	06	Initiator responds to Reselection
1	0	0	0	0	1	0	xx	Target asserts BSY and releases SEL to end RESELECTION Phase
1	0	0	1	1	1	0	80	MESSAGE IN Phase - IDENTIFY Message (Logical Unit 0)
1	0	1	0	1	1	0	00	STATUS Phase - Parity Error
1	0	1	1	1	0	0	05	MESSAGE OUT Phase - INITIATOR DETECTED ERROR Message
1	0	0	1	1	1	0	03	MESSAGE IN Phase - RESTORE POINTERS Message to retry the Phase
1	0	0	0	1	1	0	00	STATUS Phase - GOOD Status
1	0	0	1	1	1	0	00	MESSAGE IN Phase - COMMAND COMPLETE Message
0	0	0	0	0	0	0	00	BUS FREE Phase

This page is nearly blank!

A Hitchhiker's Guide to the Small Computer Systems Interface

Hitchhikers know where they want to go, they just can't control how they get there. Neither can those trying to learn SCSI.

SCSI is a complex interface and you have to learn little bits and pieces about a whole lot of things before you learn the subject you originally started trying to grasp.

The 600 pages in the SCSI-2 standard were not written as an aid for engineers and programmers, but to define the requirements for conformance. Learning is difficult when references to a particular subject may be scattered anywhere throughout all those pages.

Enter the SCSI Encyclopedia

Interested in a subject?

Just look it up.

Not only will you find a description of the subject itself, but all the other information you need in order to learn the subject is referenced.

The SCSI Encyclopedia is no heavy, hard-to-follow tome: the information you need is conveyed in a light and informal style.

Both the beginner and the experienced designer will find the SCSI Encyclopedia to be an invaluable tool to understanding all the new SCSI-2 features.

ISBN 1-879936-12-7

90000

9 781879 936126

Volume I (A-M):	Bus Phases and Protocols
Volume I (N-Z):	Bus Phases and Protocols
Volume II:	Disk Operations
Volume III:	Tape Operations
Volume IV:	Optical Disk Operations

SCSI Encyclopedia Volume I (N-Z)