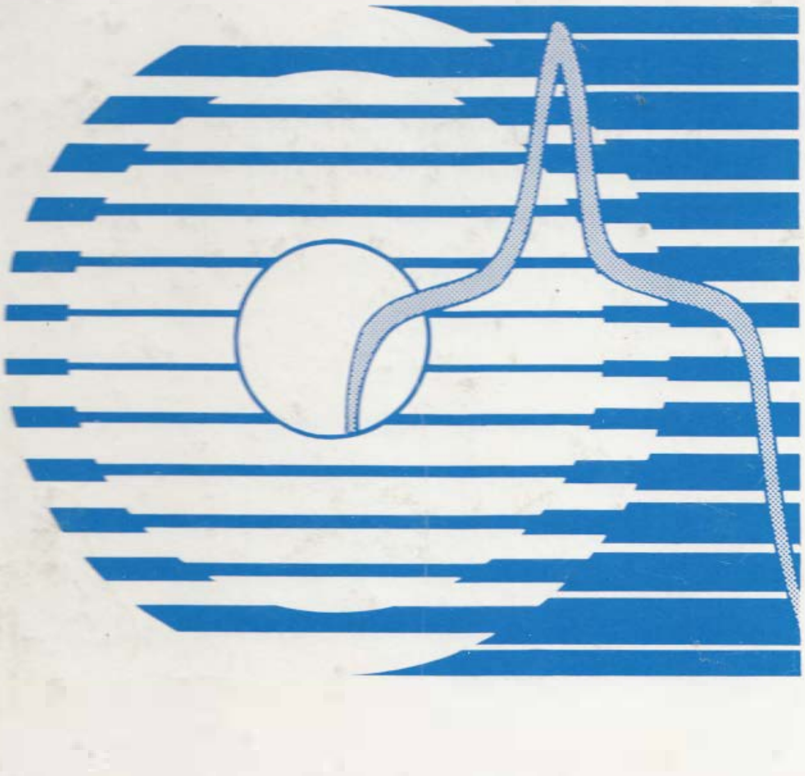
The SCSI Encyclopedia

Volume I
Phases and Protocol
(A–M)



ENDL
PUBLICATIONS

The SCSI Encyclopedia

Volume I
Phases and Protocols
(A-M)

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

Throughout this book, the trade names and trademarks of many companies and products have been used, and no such uses are intended to convey endorsement of or other affiliations with the book. Any brand names or products names used within this book are trademarks or registered trademarks of their respective holders.

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

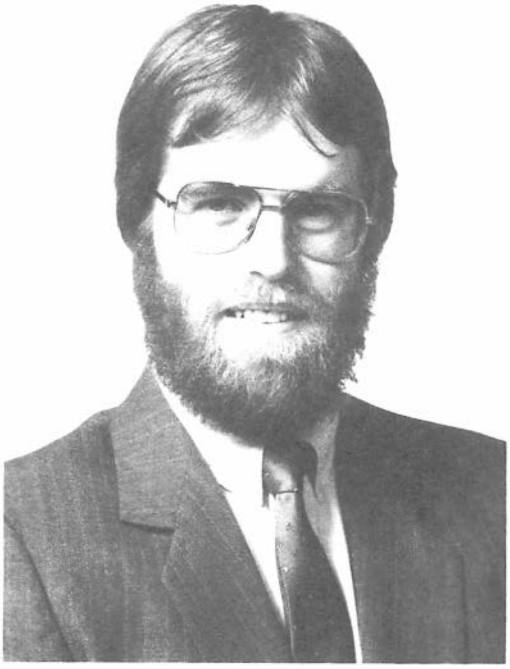
Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

ISBN 1-879936-11-9

ENDL Publications, Saratoga, California

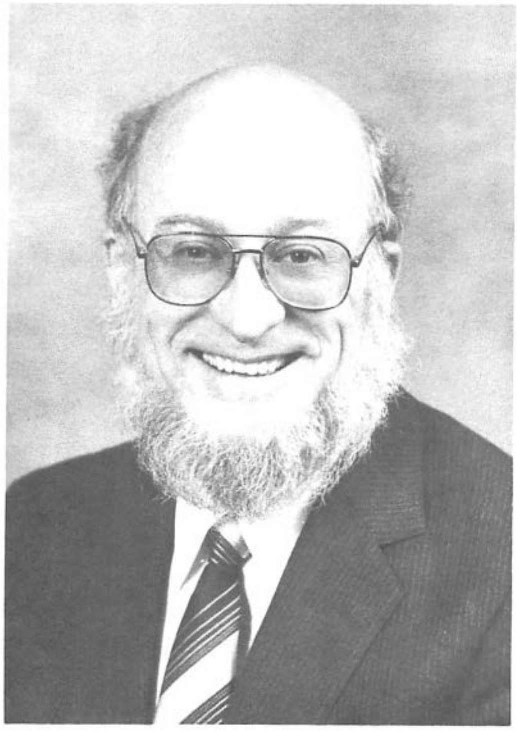
For Mary,
who put up with it.

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 (N-Z) of the SCSI Encyclopedia:

Negate Signal.
Negation Period.
Nexus.
NO OPERATION Message.
ORDERED QUEUE TAG Message.
P Cable.
Parity.
Path Control.
Peripheral Device.
Phase.
Pointers.
Power-On to Selection Time.
Protocol Chips.
Pull-Up.
Queue.
Queue Tag.
Queue Tag Messages.
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.
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.
Tagged Queuing.
Target.
Target Routine.
Target Routine Number (TRN).
TERMINATE I/O PROCESS Message.
Termination.
Terminator Power (TERMPWR).
Timing.
Transfer Period.
True.
Unexpected BUS FREE Phase.
Unit Attention Condition.
Untagged Queuing.
Vendor Specific or Unique.
Wide Data Transfer.
Wide Data Transfer Negotiation.
WIDE DATA TRANSFER REQUEST Message.
Wire-OR Glitch.
X3T9.2 Committee.

Introduction *by Dal Allan*

The size of this encyclopedia may be daunting to many readers, but it is a reflection of the complexity in what has been almost a fairy tale success in the saga of the computer industry.

Who would have thought that an idea which germinated in a company that is no longer in business would rise to become the dominant interface of choice about a decade after the first specification was printed?

During the 1970s, Shugart Associates became extremely successful after it introduced the 8" floppy disk as a storage device. The floppy had originally been conceived at IBM as a way to store microcode and diagnostics information in controllers and CPUs. Shugart was not the first company to clone a look-alike but it was the first to promote the floppy as a stand-alone storage device, and the floppy spawned an entire industry of data entry and word processing.

Shugart Associates dominated the floppy business, and it was a natural growth path to expand into manufacturing non-removable rigid disks, or Winchesters. The first Winchester was a single platter 14" disk which integrated the data separator functionality into the disk in a similar fashion to the de facto industry standard known as the SMD (Storage Module Drive) interface.

The next generation Winchester developed was smaller with an 8" diameter and fitted the same physical envelope as the 8" floppy disk. Being smaller, the new 8" disk was to be priced less than the 14" disk, and marketing faced a dilemma - it was expected that customers would flock to the smaller drive and revenues would fall.

Instead, the interface was changed. By taking out the data separator, the cost could be reduced and customers would be unable to plug the small disk in place of the large disk. Revenue was protected, but nobody realized that the time to integrate a disk without separated data would take considerably longer than previous disks.

Having to include data separation into the disk controller was a burden that most small integrators were technically incapable of achieving, and represented a burden that many integrators were unprepared to accept. The 8" disk, even though it was cheaper, smaller, and used less power, faced a long and difficult integration period.

Other 8" disk manufacturers had already faced the problem and developed their own solutions. Amongst them were Century Data, Kennedy, Micropolis, and Priam, all of which had introduced easy-to-use interfaces under various names such as Intelligent, Picobus, Parallel, and Smart. Each of these companies locked their customers in by treating these as proprietary advantages, and they did not want anybody else manufacturing to the same interface.

Shugart Associates may have been late to realize that it needed an intelligent interface to help in integrating Winchesters, but it was the first to publish a specification and make it publicly available. There were a large number of reasons for this, one of which was the need to develop controllers quickly.

The original contractor was Data Technology Corporation, which has survived the intervening years, and is still a controller manufacturer today.

Thus it was that the late entrant into the "make it easy philosophy" of attaching disk drives was actually the first to publish a public specification. In the absence of alternatives, it soon became a de facto standard and a number of independent controller companies, including then well known names such as OMTI (acquired by SMS), Xebec (now defunct), and ADES (acquired by Western Digital) adopted it.

The original SASI (Shugart Associates Systems Interface) specification was only 20 pages. It left out a great deal, but represented a valuable working nucleus. Although SASI was still a fledgling, there was a desire to broaden its scope and application of SASI. Controllers started to appear in a variety of forms, not just to attach 8" disk drives but also floppies and QIC (Quarter Inch Cartridge) tape drives.

Shugart Associates liked the idea of making SASI into an ANSI standard and approached ASC (Accredited Standards Committee) X3T9.3 in mid-1981.

The previous year, X3T9.3 had spent several months drawing up a list of functional requirements for an intelligent interface that would cover the spectrum from inexpensive products to high performance systems. The result of this was that X3T9.3 was actively developing the IPI (Intelligent Peripheral Interface).

The driving thrust behind IPI was system software simplification and future growth for mainframe and minicomputer systems. There was thorough attention to detail and extensive debate among the committee members to ensure that the definition covered the needs of a large installed base of existing products.

In December 1981, a large number of SASI supporters attended the X3T9.3 meeting and attempted to replace the IPI development with SASI. The effort failed for a number of reasons, but the most important one was that most of the SASI supporters were not voting members. In this effort, Shugart Associates had picked up an important ally in NCR, and representing these companies were Hank Meyer and John Lohmeyer, respectively.

However, this was not the end. It was obvious that there was sufficient industry interest in SASI to justify a committee activity which would not replace, but complement, IPI. At the next meeting of X3T9.3, SASI supporters met as Group B while arrangements were being made to resurrect the X3T9.2 committee. X3T9.2 met in April 1982, under the Chairmanship of Bill Burr of the National Bureau of Standards.

The first task of X3T9.2 was to change the name of the interface, because standards cannot include the name of a company. The acronym chosen was SCSI for Small Computer System Interface, and the first reaction when this was announced was that it would be pronounced as "sexy," "sucksy," or "scuzzy." It was the latter designation that was to stick.

In an interesting paradox, many months later, Shugart Associates changed its name to become Shugart Corporation, thus aligning the company initials again with the name of the interface.

In the short time before SASI was brought to the standards process, a number of suppliers had developed variations to suit different market applications and each had implemented its own extensions in its own way.

Compatibility was poor. The leading de facto implementations came from DTC, OMTI, and Xebec. At one time, a set of the specifications for the products from these companies were included in the SCSI draft as appendices.

The desire to protect the implementations of companies which were already shipping SASI products meant that compromises had to be made between the best technical solution and the least disruptive. Many of the anomalies that bother SCSI implementors today can be traced back to these early beginnings, where it was necessary to protect the fledgling industry members.

Inconsistent architectural characteristics crept in because the committee members wanted to ensure that everybody could claim compliance with the developing SCSI standard.

NCR took a strong leadership role in the committee definition, adding differential logic and shielded connectors, to provide a longer cable length so that SCSI could move outside the cabinet. NCR was the first to do a thorough analysis on the timing characteristics of arbitration, and realized in 1983 that what was to become known as the "wired-OR glitch" required a timing change. NCR discovered this during the design of the first chip to integrate the SCSI protocol.

In April 1984, SCSI was felt to be far enough along to justify its first public review. The bureaucracy requires a 4 month period for the review, but between delays in the process and discoveries made during public review period, it was not until a year later that a new draft was thought to be ready to distribute.

In June 1985, there was a joint meeting with ECMA (European Computer Manufacturers Association); the European companies interested in standardizing SCSI were working in parallel, but slightly behind the United States ANSI (American National Standards Institute) effort.

The idea of the joint meeting was to share information so that there would be no deviation between the two standards. It was at this meeting that NCR dropped a bombshell in the form of proposing an alternative shielded ribbon connector which was of a more rugged design. Connectors have always been a problem in SCSI and the shielded ribbon style caused quite a stir.

Not until November was a compromise on how to include the new connector beaten out between ECMA and X3T9.2. This resulted in the shielded connectors winding up in the appendix rather than in the body of the standard. There is a subtle difference between information in the body of a standard and that in the appendix. Under ANSI guidelines, information in an appendix is considered to be guidelines, not a requirement.

The June 1985 meeting was memorable for another event as well, in that it authorized an ad hoc group to discuss the problem of diverging implementations, an effort which was to become the Common Command Set (CCS).

The wide diversity and implementations of SCSI meant that lack of compatibility was a major issue for system integrators who were buying products from a number of controller manufacturers and disk suppliers.

On the supplier side, the vendors had their own troubles as they had to face the problem of attempting to design all of the variations that were required by OEMs. CCS was the industry's way of recognizing that these variations somehow had to be pulled into line.

To some, the CCS activities were seen as those of a vigilante group operating outside the accredited standards process. This was not true.

From June 1985 to December 1985, X3T9.2 was totally involved in the effort of getting the standard completed, absorbing the new shielded ribbon connector issue and beginning the cycle towards ISO approval. The CCS ad hoc group was an authorized activity being conducted in parallel with the completion of the standard.

In December 1985, Revision 17B was forwarded for public review and several months later, in June 1986, it passed all of the bureaucratic hurdles and formally began life as X3.131-1986. This event lagged by one month the formal approval of a new project to begin the SCSI-2 standard.

SCSI-2 was intended to be a quick effort to absorb the CCS activities which had been isolated to disk, and extend them to other devices, such as tape.

The amount of industry activity was vastly underestimated. It seemed that within only a few weeks, a large volume of enhancements were proposed. X3T9.2 is a vendor-driven committee because the majority of members are suppliers, and suppliers have a problem.

If everything is created equal, the lowest priced supplier wins.

For this reason, vendors like to add capabilities and features which suit their own company personality and add value. The implementors which can get to market first with the latest features attain a marketing edge over their competitors. If a feature becomes popular and accepted by industry, then everybody else follows but there is a benefit to being early to market.

The SCSI market is segmented vertically, with several strata of performance, functionality, and cost characteristics, and the years of SCSI-2 development matched an era of explosive growth. From its humble beginnings as an industry standardization effort in 1981, SCSI has spread far and wide.

The secret to its success was the NCR 5380 chip. SCSI is a very complicated interface and one has to admire the early implementors who managed to make it work using MSI (Medium Scale Integration) components; it took a great deal of skill even when the implementation was kept simplistic.

Shugart Associates kept promising silicon and failing to deliver, whereas NCR went to work and produced the first useful chip. Although this could have been kept proprietary and used to establish a competitive advantage, NCR took the opposite approach.

NCR created an industry by virtually giving away the chip. The price was set low. A large education effort as well as application development support was put in place so that integrators had somewhere to turn for advice. This strategy paid off in a way that NCR could not have possibly anticipated when Apple selected SCSI as its interface of choice to attach devices.

It would be a great story to say that this was a strategic decision by Apple, but it turned out to be more a case of serendipity. The original goal for the Macintosh had been to attach peripherals via Apple Talk. Rather late in the development cycle it was realized that Apple Talk was too slow to support disk drives. A mad scramble occurred as Apple engineers went hunting for some kind of alternative. There was SCSI, and there was a chip, and so it was that SCSI became the peripheral interface for the Macintosh.

Apple was off and running.

During development, Apple used an OMTI controller as the straw horse to implement disk support. Many complaints have been made about Mac SCSI not being fully compatible with the standard. Much of this can be attributed to the fact that the boot code was developed against the OMTI controller. The problem lay in the controller having originally been designed for SASI and then warmed over to become more like SCSI. Bottom line, the OMTI controller was not fully compliant with the standard.

Apple created a market.

Apple was the first systems integrator to define an interface attachment for SCSI. Not just hardware, but also software. Non-standard maybe, but it was good enough for a flourishing third party industry to develop that attached SCSI devices to Apple systems. A few deviations from the standard were not enough to slow this growth.

Apple was slow to correct the incompatibility problems because the microcode had been burned into ROMs, and would require a major recall. It was easier to document the differences and let the third parties adjust their microcode. Thus it was that a whole generation of MacSCSI peripherals came to exist in the retail channel.

With each new Macintosh, Apple has come closer to the standard, but for one glaring exception: the connector. Despite the influence SCSI has had on the market, it has never proved strong enough to influence systems integrators into using the connectors defined in the standard. IBM recently introduced SCSI for the PS/2, and IBM too showed no reluctance to using a unique connector on the host adapter.

If one were to buy a complete array of SCSI host adapters one might find that the SCSI connector chosen has been honored more in the breach. Fortunately, peripheral manufacturers have chosen to follow the standard more closely, and all it takes is a special cable to get from a non-compliant host to compliant peripherals.

As a standard, SCSI has always lagged industry usage. Being embraced by such wide segments meant there were innovations constantly occurring in different market segments, and these were very often implemented before they were brought to the X3T9.2 committee as suggestions, recommendations, ideas and proposals.

SCSI has been likened to UNIX in that it is an open interface which is favored for open computing. SCSI was right for the times, it was easy to comprehend and flexible enough to be adapted. Unfortunately, as happens to so many people as they approach middle age, SCSI has grown in girth over the years, having achieved 600 pages in SCSI-2, X3.131-199x.

Trying to absorb so much information is overwhelming, and this makes SCSI difficult to comprehend for the neophyte, but it is still flexible enough to be adapted to new applications. SCSI is a living interface.

By way of explanation for the variations in SCSI, one only needs to look at the original standard in which there were few rules of "what not to do" and as a result there were variations which suited the implementor. There has always been a wide variety in SCSI implementations as competitors vied for market share and OEM opportunities.

Even though one might expect vendors to be possessive about their variations, competitors have shown a remarkable propensity to agree on what should be done in the best interests of the standard. Even when committee decisions have not necessarily favored companies which had implemented in a different manner, their representatives voted in the best interests of the standard.

SCSI suppliers take the very positive attitude that they have an obligation to be backwards compatible with their existing customer base because they chose to implement early. As a result, succeeding generations which are compliant with the standard include the non-compatible variations to support existing customers. This is one factor that has been a major force in expanding the SCSI market, customers get at least one generation of products in which to migrate their implementations.

NCR may have started the chip race, but they were by no means alone in the running. The race in pursuit of design wins was, and continues to be, a brutal one. Chip suppliers are constantly unveiling new solutions which reduce overhead and add functionality to chips. The 5380 requires lots of microcode to support it, and for that reason performance is pretty slow.

Every generation of chips has included more capability in gates and required less microcode and microprocessor intervention. The result is that the time required for overhead processing has been continually shrinking.

The competitive drive to win chip sales pushed the envelope of performance hard, with design cycle times as little as 9 months between silicon revisions. The result was that product introductions accelerated as the market kept choosing the latest, best design available to them. One result has been a dramatic improvement in SCSI performance every few months.

Originally, SCSI was the interface of choice on small systems where performance was not so important. As overhead shrank because of added silicon functions and efficient microcode, SCSI moved upscale to become the choice on mid-range systems as well, and initiated a second surge of life in the SCSI market.

SCSI is an immature interface.

No one can tell what SCSI will look like next year. It may be possible to predict the features likely to be included in next year's SCSI, but it takes a brave man to suggest he knows how it will be defined. As one example, take Asynchronous Event Notification. SCSI originally had no means for a target to advise the initiator(s) of unanticipated events such as media being loaded into a tape drive.

Removable media manufacturers campaigned to add some kind of asynchronous capability late in the days of SCSI-1, and it was agreed to address the need in SCSI-2. If bets had been taken, a lot of money would have been placed on adding a new signal line called Attention In but such was not to be. As anybody who has ever looked up Asynchronous Event Notification knows, the definition is a lot more complicated.

Much of SCSI's success cannot be attributed to the capabilities of the interface so much as the very deliberate campaign by SCSI suppliers to convince OEMs that implementing SCSI was easy. You might say that this was a slight exaggeration because SCSI is unlike existing device interfaces.

For all other interfaces, the point of management and control is the originator *of a* request; i.e., the host. Not so SCSI, the host makes requests and it is the target which decides when it will act upon them and when it will service the initiator.

This change in philosophy can have a brutal impact upon some operating system software drivers, but this discovery usually came too late to change plans which had been put into action. By the time the companies which had wanted to believe SCSI was easy had learned differently, they were already involved in the project and committed to completing the project.

After wrestling through the first learning experience, the second time was straightforward and easy, so SCSI delivered on its promises the second time around.

In the early days, compatibility was a promise that SCSI failed to deliver, but despite those who still claim otherwise, there is a high degree of plug and play compatibility between peripherals today. A number of reasons account for this, but the most important is that the controller is no longer a separate element provided by a separate supplier from the peripheral manufacturer.

Embedded SCSI may be important, but it is not new.

The first implementation was by Xebec in a disk drive called the Owl, and it was pre-SCSI. It was actually embedded SASI. The disk drive was only adequate, the performance was low and the implementation was slow. The Owl met with a poor reception in the United States, but met with an extraordinary degree of success in Europe.

Europeans like performance as much as Americans so there had to be another explanation, and it was to be found not in technology, but in business issues. European integrators deal with two countries for their hi-tech products, America and Japan. Take the case of a disk drive being supplied from Japan and a controller being supplied from the US.

When a problem arises, an engineer in Germany is faced with making a call to Japan at the crack of dawn to explain the situation only to be told that it is not the disk drive, it must be the controller. Shortly before dinner, the engineer calls the controller supplier in America who claims the problem is not in the controller, it must be in the disk drive.

When fingers are being pointed a day apart, Europeans have a much more difficult time in integrating their systems. Embedding the controller in the peripheral solved this problem, because instead of having to yell at two vendors a day apart, one was completely responsible for the integration process. If an embedded peripheral does not work, there is only one call to make to one vendor; a much cleaner and easy-to-work with situation.

Many of the early embedded devices were done on a turnkey basis by controller companies who were given their "own end of the board to play with." Buried on that board was often an external interface, such as ESDI or ST412, and embedded SCSI caught on in the United States only after SCSI overhead had been reduced.

The success of the early implementations led to disk drive manufacturers taking over SCSI integration and making it a native interface. This really led to a reduction in overhead. The majority of implementations today are native SCSI to reduce overhead and maximize performance. Controller manufacturers have become chip suppliers for disk companies. One consequence of this trend has been that the market for bridge controllers has shrunk dramatically.

Major OEMs who committed to SCSI, did so at different points in time, and when they did so, they carved a specification out of the revision of the standard then current. As the standard moved on, the specification didn't. As a result, OEMs are still buying based on specifications that may have been written 3, 4 or even 5 years ago. When you ask why they haven't changed you learn all about the limitations of changing software. A by-product of this situation is that OEM suppliers of disks and tape (disks especially) face the problem of having to supply 30-40 different variations of micro-code for each drive.

Since its first introduction, SCSI has been the preferred interface for tape and optical devices because they are difficult to integrate and performance is not a critical issue. On the other hand, disk is an integral element to system performance and many OEMs would not accept the penalty on every SCSI request.

As overhead shrank with improved silicon, disk performance became acceptable and SCSI became an alternative to device interfaces. Although disks lagged the trend towards embedded SCSI, they have surged to dominate the dollars in the sale of embedded SCSI devices.

When talking about SCSI it is very important to recognize that it is not a single entity but a market segmented into several layers. At the lowest rung on the ladder is Kentucky Fried SCSI (cheap, cheap, cheap). Performance is not critical on the lowest rung, but as one climbs the ladder, performance becomes the critical factor separating the rungs.

In the broadest of terms, there have been five de facto industry standards:

o SASI
o SCSI-1
o MacSCSI
o CCS (Common Command Set)
o SCSI-2

In recent months there have been many who have questioned why SCSI-2 has so many choices and added such a proliferation of features to the original SCSI. As noted before, the committee efforts on SCSI-2 coincided with an explosion in SCSI applications in industry and a large number of companies wanted to add features and capabilities to the interface.

The problem faced by the committee was not what to add, but where to stop adding features. Originally, the goal of SCSI-2 was to "legitimize" CCS (Common Command Set) and extend it to device types other than disk but it quickly grew (kind of like a zucchini plant).

You may even find some things in the Encyclopedia which are not in the SCSI-2 standard!

SCSI is a living entity.

SCSI-3 is now well under way. In fact, SCSI-3 started before SCSI-2 had finished, in SSWGs (Specific Subject Working Groups).

X3T9.2 froze its definition of SCSI in February of 1989, but here it is 1991, and SCSI-2 is still not published as a standard. Nor is it likely to make publication until the end of 1991 or early 1992.

You may find it amazing that the bureaucratic procedures are still not finished. You can contribute this to the painful review of the document in excruciating detail, trying to eliminate errors and improve consistency.

While SCSI-2 is being refined and clarified for publication as a standard, X3T9.2 is continuing to move forward on SCSI-3 by focusing on some features of SCSI-2 that were lacking, and extending them. Some of this material has been included in the Encyclopedia as it stood in January, 1991.

In some cases, it is likely that some SCSI-3 features will become more popular in SCSI-2 implementations than the definitions in the SCSI-2 standard. One case in particular that comes to mind is that of the 16-bit single cable attachment popularly known as the P-Cable. SCSI-2 defined a second B-Cable for transferring data but did not extend the number of devices that may arbitrate.

The P-Cable provides an extra 8 data bits and permits selection of an extra 8 devices during arbitration, so it is a natural to become popular in an era of embedded SCSI devices. The P-Cable is included in the SCSI Encyclopedia because although it is not in SCSI-2, it will be introduced in products long before SCSI-3 becomes a standard.

To give some idea of the magnitude of the task of SCSI-2, at one time there were 9 editors, and the document had grown to be unwieldy. Each of the editors made a major contribution to the development of the standard, but not being of the same mind and English skills, the styles were inconsistent. It fell to Larry Lamers as general editor and John Lohmeyer to bear the brunt of weaving a cohesive document out of the ungainly set of chapters that composed SCSI-2.

Small things which escape initial reading can cause implementation differences; e.g., a slight variation in wording of the same action can be interpreted to imply that there is a difference. If the same action is described in similar, but not identical, words then undoubtedly there will be implementations that differ.

The problem of different interpretations usually comes to light when some OEM, during integration, discovers devices have inconsistent behavior. The committee then faces the problem of trying to figure out the intended definition, and at least one of the parties is going to be hurt. There have even been some cases when both parties involved have been hurt because the clarification wound up affecting all implementations.

Sometimes, it may take careful reading of several parts of the standard to derive enough information to make a decision on how to implement some feature. Even then, it can seem that there are nuances which make an interpretation less than obvious.

It is here that the SCSI Encyclopedia is attempting to bridge a gap between the language of a standard and the microcode of an implementation. The standard is extremely difficult to read, especially for the uninitiated.

Being faced with having to learn SCSI for the first time is daunting, as woven throughout the 600 pages are directions, implementation notes, and warnings. Despite the best efforts of all the committee members, there are bound to be some errors in 600 pages of densely packed material, and this represents an overwhelming learning curve for every design engineer.

The purpose of the SCSI Encyclopedia is to provide a way for an engineer to knowledge as needed. It is much easier to learn by looking up an entry on something that one does not understand, or has not come across before, than to hunt through the standard hoping to find it mentioned.

The final word on interpretation is in the standard. The SCSI Encyclopedia is more than an interpretation because it binds together the information, and it cross references other material which should be understood in conjunction with the specific subject being investigated.

It is to be hoped that readers will find this a useful text and we appreciate criticism, preferably constructive. Where something might be better done, we want to know.

SCSI is a living standard, and we want this to be a living SCSI Encyclopedia that will grow and expand to cover all of SCSI. Your input to future editions is solicited. Send along your ideas on what is missing, what is wrong, and what should be added. No guarantees on their being included, but they will definitely be read. Send them to:

ENDL Publications
14426 Black Walnut Court
Saratoga, CA 95070

Study Guide

I think we'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

<u>Lesson #1: Bus and Devices</u>

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

<u>Lesson #2: Processes and Phases</u>

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

<u>Lesson #3: Bus Control</u>

- 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.

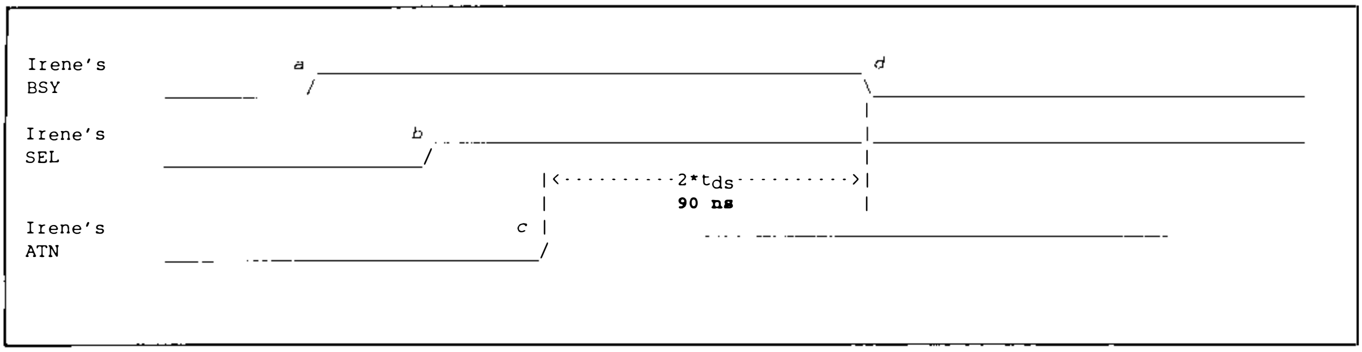


FIGURE 1: EXAMPLE TIMING DIAGRAM

Timing diagram conventions:

- The "owner" of the signal (either as a driver or receiver) is shown on the left hand side.

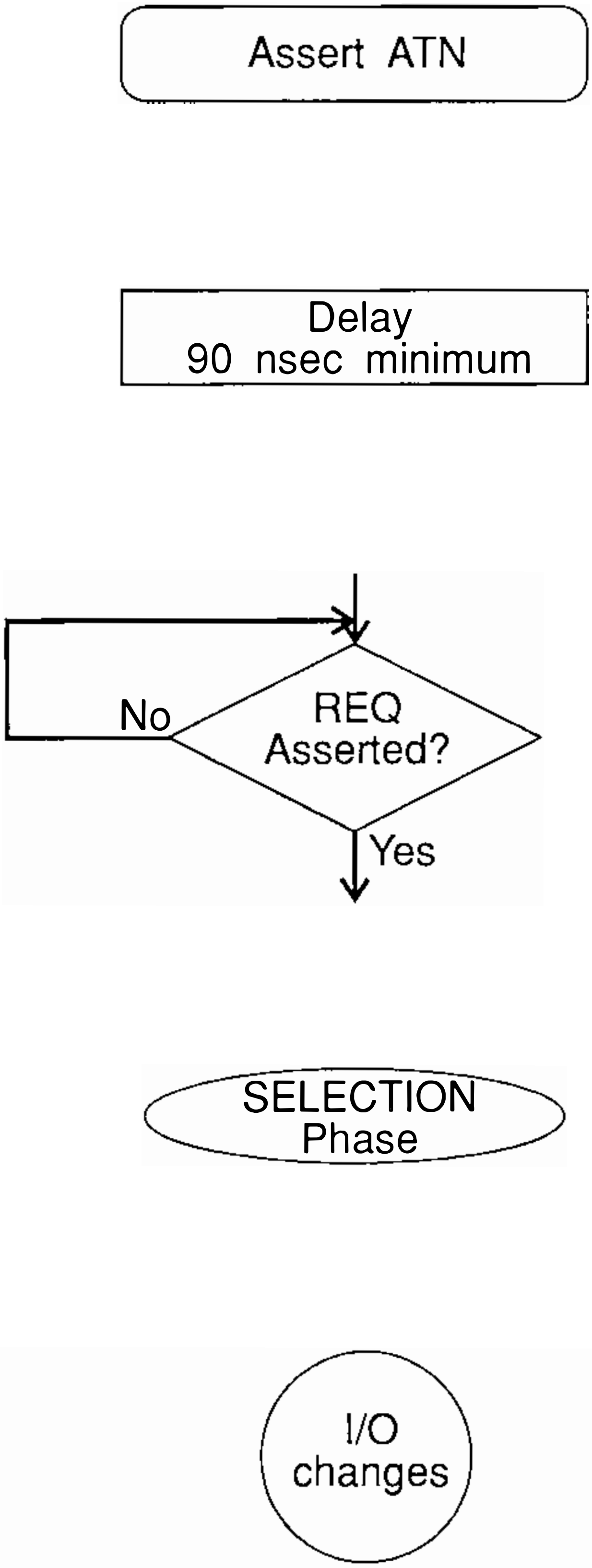
- All signals are shown as "high-true/low-false", independent of the electrical interface type (***Single Ended Interface*** or ***Differential Interface***). For instance, Single Ended Interface signals are electrically low when true, but the timing diagram shows them high when true. While this may 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

A Cable. The A Cable is the ***SCSI-2*** name of the 50-conductor cable used in all 8-bit SCSI systems. The A Cable carries the ***Control Signals*** and the first eight bits of the ***Data Bus Signals***. See ***Cables***.

ABORT Message. The ABORT Message is used by the Initiator to tell a Target to stop what it is doing for the current ***I/O Process***; typically, commands are aborted by this ***Message***. ABORT is a single-byte message:

Bit Byte	7	6	5	4	3	2	1	0
0	Message Code = 06 hex							

Some things to know about ABORT:

- ABORT is for changing your mind: it is an "Oh No!" function. An example of the proper use of abort is to stop a disk format you didn't mean to do. The system format utility would issue the ABORT message to stop the operation, with luck before your data was lost.

- ABORT is not for everyday use: don't expect sane behavior regarding the current command from a Target when it receives the ABORT message. The Target may pull the plug halfway through an operation. It may forget about the command entirely: no Sense Data or history is retained. Issuing ABORT is usually a sign that something went WRONG at the Initiator end of the system.

  In fact, the ABORT Message is used in response to "insane" behavior by the Target. For example, if the Target ***Reconnects*** an Initiator for a ***Nexus*** that does not exist (from the Initiator's point of view), then the Initiator should respond with the ABORT Message. Of course, sometimes an Initiator may use the ABORT Message when it thinks the Target is misbehaving, but in fact the Initiator is messed up.

- ABORT is not supposed to affect anything on the Target associated with other Initiators for the same ***Logical Unit***, or anything associated with other Logical Units on the Target. Of course, if a disk Target aborts a write halfway through a sector (not recommended behavior, admittedly), this can affect other Initiators. Again, don't make a habit of using ABORT.

- After receiving the ABORT, the Target goes to ***BUS FREE***. No status is sent; no other phase occurs before BUS FREE. There is no time specified between the "receipt" of the message (presumably the trailing edge of ACK) and when

BUS FREE occurs. It is possible that a Target will hold the bus until it has done what it has to do to abort the I/O Process.

- So how does the Initiator send this message? The same way as any other: by asserting the ***ATN signal*** (creating the ***Attention Condition*** on the bus) and waiting for the Target to get around to responding. But what if the Target is disconnected from the Initiator? Simple: despite the fact that it has an I/O Process currently executing, the Initiator is allowed to select the Target just for the purpose of sending this message. The Initiator first must send an ***IDENTIFY*** message (see ***Nexus***) or the Target won't know which Logical Unit's I/O Process to abort (Targets won't try to read Initiator's minds). If the Initiator doesn't send the ***IDENTIFY Message***, the Target will just go to BUS FREE phase without doing anything else.

- ABORT has additional meaning when ***Queuing*** is implemented. In this case, all queued I/O Processes for the Initiator (not other Initiators!) are aborted for the current Logical Unit. This might be used, for example, after a fatal write error to stop subsequent I/Os to the same area of the media. It isn't likely that the Target will create sense data for all of the aborted I/O Processes in the queue. Note that ABORT with no queuing aborts one I/O Process, but with queuing it potentially aborts several I/O Processes. To abort a single queued I/O Process, use the ***ABORT TAG Message***.

Summary of Use: The ABORT message is sent only by the Initiator at any time to cause the Target to abort the current I/O Process, or when Queuing is used, to abort all I/O Processes for the current Logical Unit for the currently connected Initiator.

ABORT TAG message. This message is used to abort a single queued ***I/O Process***. ABORT TAG is a single-byte ***Message***; the ***Nexus*** must have been established by a previous ***IDENTIFY Message*** and ***Queue Tag Message***:

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

All of the cautions and uses for the ***ABORT Message*** also apply to this message. In fact, if the queued I/O Process has already started processing, the behavior is exactly the same as ABORT: stop the I/O Process and go to ***BUS FREE Phase***. ABORT TAG is part of a hierarchy of messages that can be used to remove I/O Processes from the command queue:

- ABORT TAG removes one I/O Process for the connected Initiator from the ***Queue*** or stops it if the I/O Process is currently executing. This applies only to one queued I/O Process that belongs to the connected Initiator for the LUN identified.

- ABORT removes I/O Processes from the Queue and stops an I/O Process if it is currently executing. This applies only to I/O Processes that belong to the connected Initiator for the LUN identified.

- The ***CLEAR QUEUE Message*** removes I/O Processes from the Queue and stops an I/O Process if it is currently executing. This applies to all I/O Processes that belong to all Initiators for the LUN identified.

- The ***BUS DEVICE RESET Message*** removes all I/O Processes from the Queue and stops an I/O Process if it is currently executing. This applies to all I/O Processes that belong to all Initiators for all LUNs.

Summary of Use: The ABORT TAG message is sent only by an Initiator at any time to abort a single I/O Process belonging to the Initiator associated with a Queue Tag.

ACK Signal. The ACK Signal is used by the Initiator to handshake data bytes with the Target. For transfers from the Target to the Initiator, the ACK signal is an indicator that the data was received. For transfers from the Initiator to the Target, the ACK signal indicates that the data.is available to the Target. See ***Asynchronous Data Transfer*** and ***Synchronous Data Transfer*** for details on the use of this signal.

ACKB Signal. The ACKB Signal is used in systems that use the ***B Cable*** to do ***Wide Data Transfers***. The ACKB Signal is independent of the ACK signal: it only reacts to the ***REQB Signal***; it is only related to the ***Data Bus Signals*** on the B Cable. The use of the ACKB Signal is identical to the use of the ACK Signal.

Active I/O Process. An Active I/O Process is an ***I/O Process*** that exists in a Target that is not ***Queued***. To clarify, if an I/O Process is not queued, then the Target must be executing it. An Active I/O Process may or may not be ***Connected***; if it is, then it is also the ***Current I/O Process***.

You will need to understand the terms "Active I/O Process", "Current I/O Process", and "Queued I/O Process" to fully understand the ***SCSI-2*** standard document. For now, just remember that an I/O Process on the SCSI Bus can be queued, active (executing), and/or currently connected to the bus.

Active Pointers. The Active Pointers (as opposed to the ***Saved Pointers***) are the three pointers that refer to the command, status, and data locations within the Initiator for the ***Current I/O Process***. An Active Pointer indicates the current byte to be accessed. There are three Active Pointers:

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

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

Active Pull-Up. This is one of the latest SCSI "buzz-words". Until fairly recently, most SCSI Protocol ***Chips*** for the ***Single-Ended Interface*** implemented "open-collector" drivers. In other words, the driver actively "pulls" the signal down to ***Assert*** it ***True***, but it then ***Releases*** the signal to ***Negate*** it to the ***False*** state.

The net effect is that the signal is asserted very quickly, but is negated more slowly. This is because the driver is using the bus ***Termination*** to return the signal to False. The Termination does not have the "oomph" to overcome the capacitance of the ***Cable*** and the receivers. This is okay for ***SCSI-1*** transfer rates, but the new demands of ***SCSI-2*** (e.g., ***Fast Data Transfer***) require a different solution.

New chips are being designed which use an Active Pull-Up in the driver circuit; in other words, a "totem-pole" driver. This allows the driver to pull the signal up to negate it as fast as it can assert it. The pull-up also improves the overall noise margin of the signal, and reduces reflections caused by allowing the Termination to bias the signal False.

Arbitration Delay. *tarbd = 2400 nsec.* The Arbitration Delay is the minimum settling time that a device must delay between asserting the ***BSY Signal*** to begin ***ARBITRATION Phase***, and testing the SCSI bus to see if it won. This value is as large as it is to ensure that the bus has settled before it is read by any device. The settling time is long because different devices will assert BSY for arbitration at slightly different times.

ARBITRATION Phase. The Arbitration Phase is the mechanism that a SCSI device uses to secure control of the bus. Once a device has control of the bus, it may then proceed to initiate an ***I/O Process*** with a Target (***SELECTION Phase***) or continue an I/O Process with an Initiator (***RESELECTION Phase***). Diagram 2 shows the steps a device goes through to arbitrate for the bus.

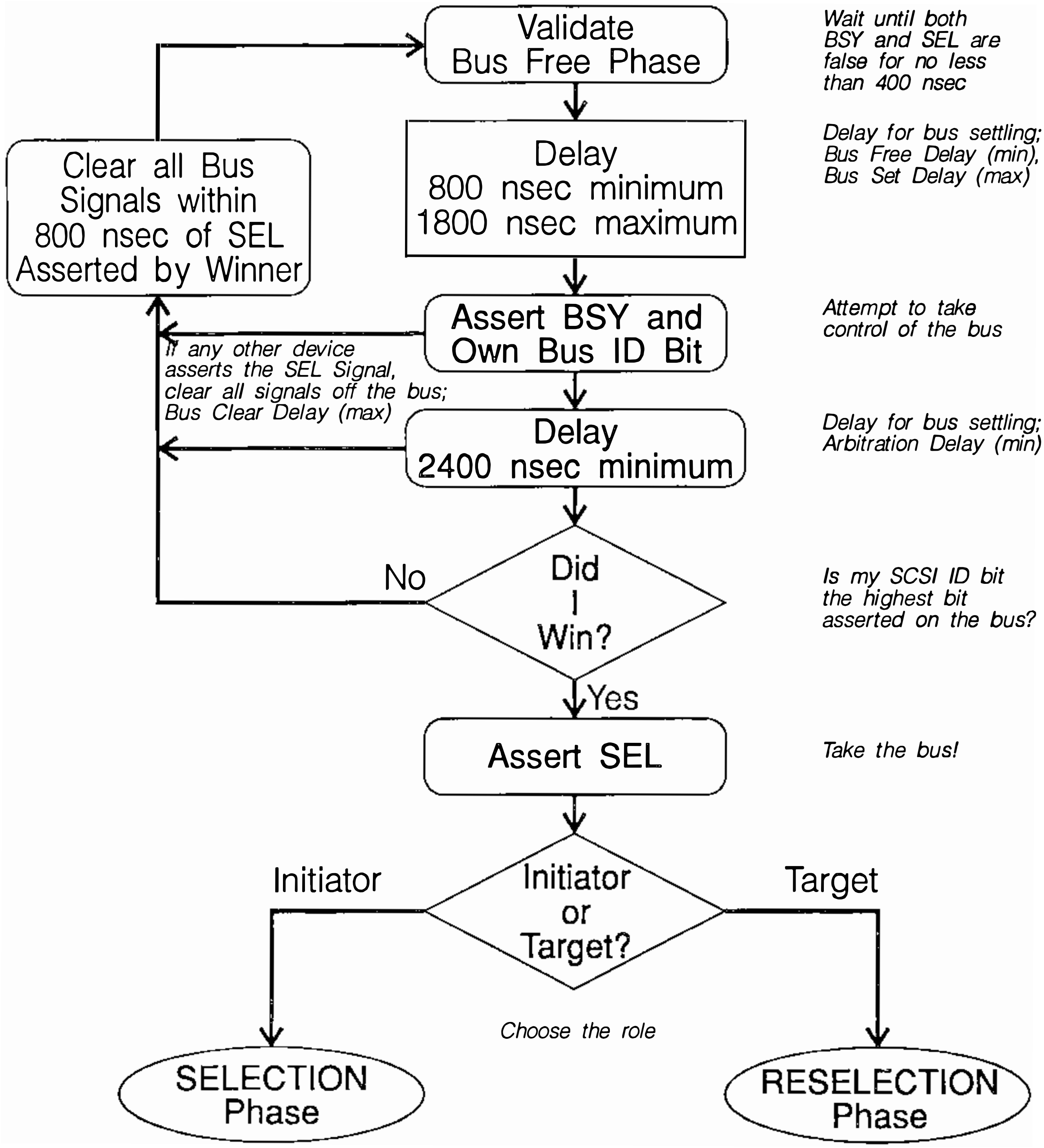
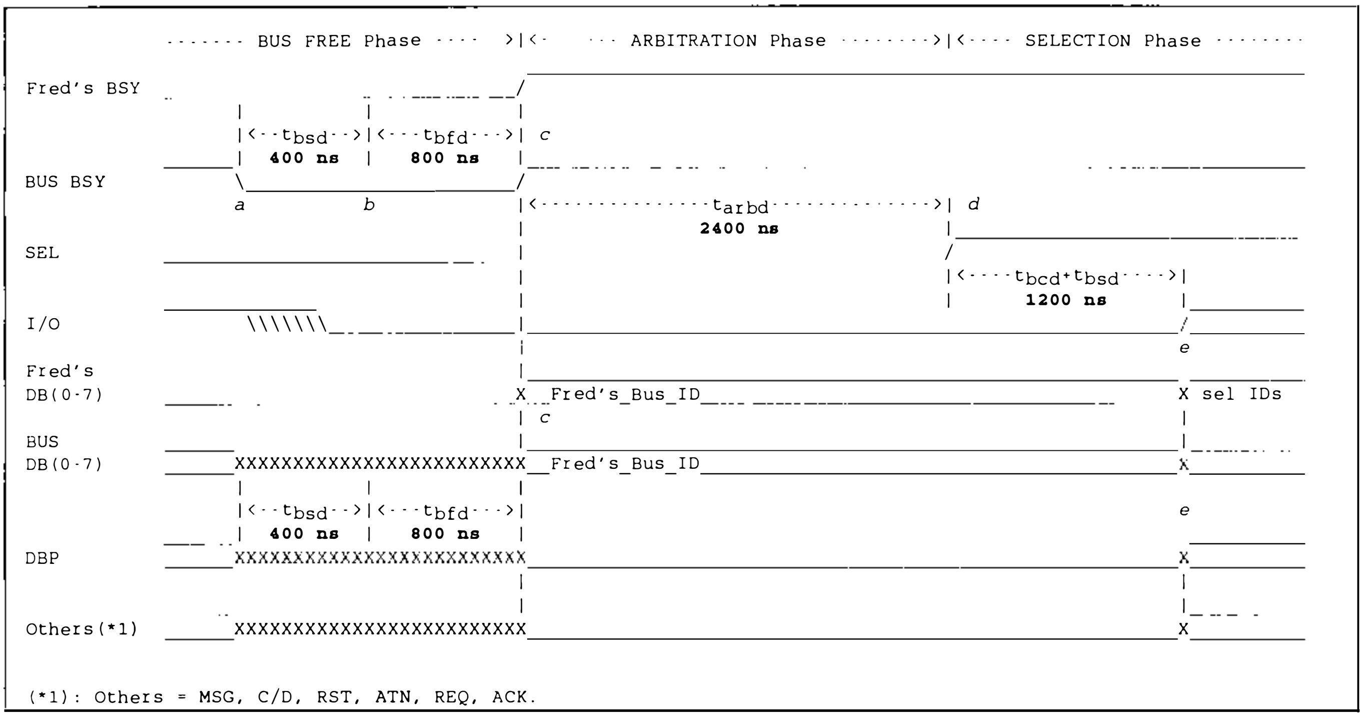

Validate
Bus Free Phase
Wait until both BSY and SEL are false for no less than 400 nsec
Delay
800 nsec minimum
1800 nsec maximum
Delay for bus settling; Bus Free Delay (min), Bus Set Delay (max)
Clear all Bus Signals within 800 nsec of SEL Asserted by Winner
Assert BSY and Own Bus ID Bit
Attempt to take control of the bus
If any other device asserts the SEL Signal, clear all signals off the bus; Bus Clear Delay (max)
Delay
2400 nsec minimum
Delay for bus settling; Arbitration Delay (min)
No
Did I Win?
Is my SCSI ID bit the highest bit asserted on the bus?
Yes
Assert SEL
Take the bus!
Initiator
Initiator or Target?
Target
Choose the role
SELECTION Phase
RESELECTION Phase

DIAGRAM 2: ARBITRATION FLOW DIAGRAM

In summary, the steps involved in Arbitration are:

- The device makes sure the bus is available.
- The device attempts to take the bus.
- If the device succeeds in the attempt, it takes the bus and proceeds to the next phase.
- If another device "wins", the device clears itself off the bus and tries again later.

Now we'll show some examples of Arbitration. Figure 2 below shows a simple arbitration bus sequence in which one device (Fred) arbitrates for the bus and wins by default.



(*1): Others = MSG, C/D, RST, ATN, REQ, ACK.

FIGURE 2: SIMPLE ARBITRATION SEQUENCE (FRED WINS)

Here's how events proceed in Figure 2:

(a) Fred has decided that he wants to arbitrate for the bus to do something useful (we hope). The first step is to establish that the bus is free; in other words, determine that the current phase is ***BUS FREE Phase***. Fred determines this by monitoring the ***BSY Signal*** and ***SEL Signal***; if they both stay false for 400 nsec, then the phase is BUS FREE. If either goes true for even a short glitch, then Fred has to begin the 400 nsec waiting period again. One can think of this as a 'cooling-off period' for the bus to let everything settle. Note that if the bus is in the ***Reset Condition***, the bus is not in the

BUS FREE Phase. See ***Wire-OR Glitch*** for the reasons why there is a 400 nsec waiting period.

(b) The BUS FREE phase has been established. Just when Fred thought it was safe to arbitrate, he has another delay. This delay exists to ensure that all of the other Devices have the opportunity to see the BUS FREE Phase. *This* time Fred waits 800 nsec. The good news is that Fred has established BUS FREE phase; there is nothing that can kick him back to the start during this waiting period.

(c) Time to arbitrate! Fred asserts BSY true and also asserts his ***SCSI Bus ID*** bit on the data bus. Fred asserts no other signals, data bits, or parity bits. Parity is not valid during ARBITRATION Phase, nor is it checked. Once this occurs, yet another waiting period ensues for 2400 nsec to let the dust settle.

(d) Since Fred is the only device arbitrating in this example, he wins. Fred knows he won because he looked at bus after the 2400 nsec delay and saw no other bits asserted. Once Fred has determined that he won, he may assert SEL true to establish himself as the device with control of the bus.

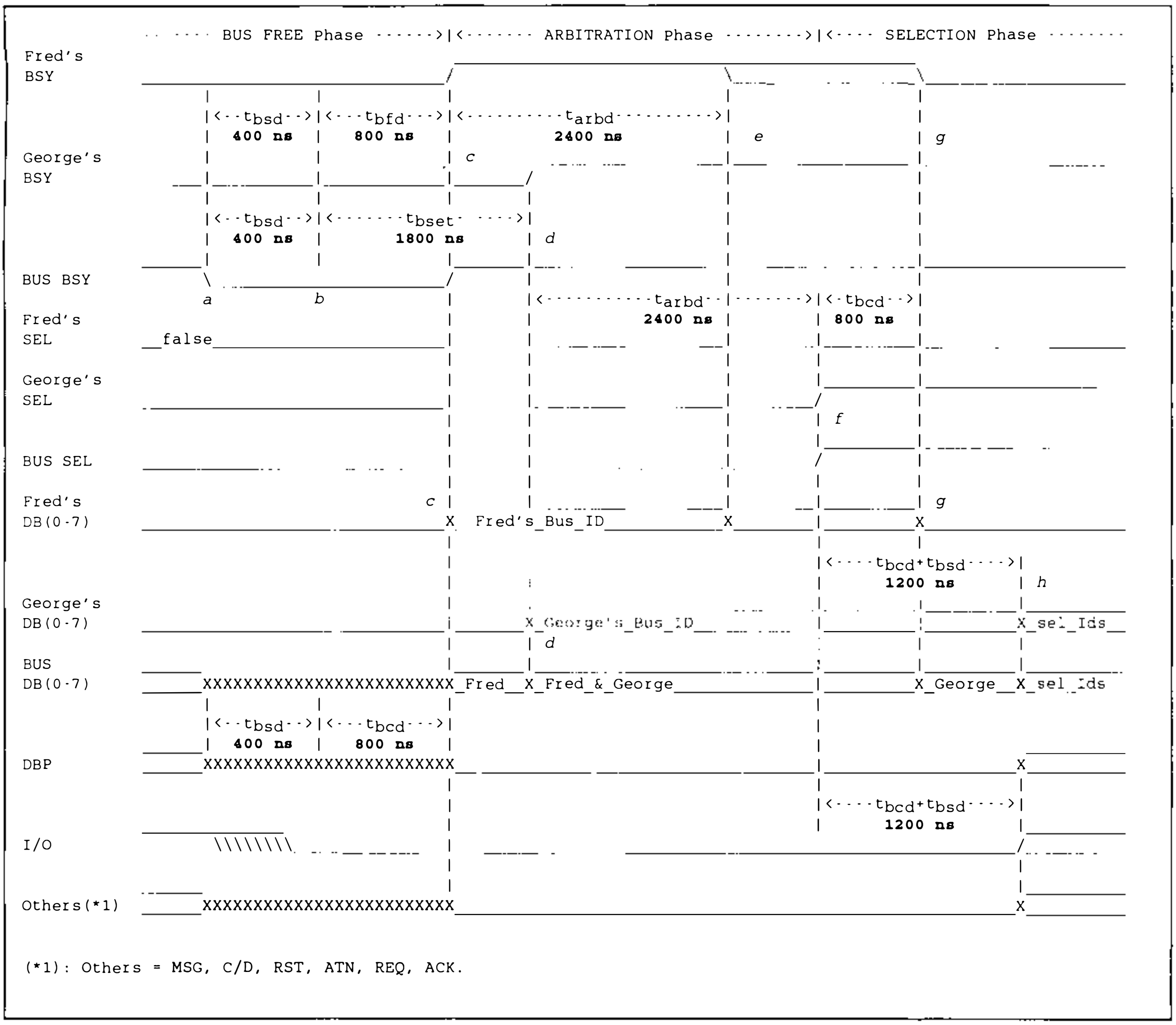
(e) But Fred's not quite through! After he asserts SEL, he must wait an additional 1200 nsec before he can do anything else; i.e., ***SELECTION Phase***. After the delay, he may then assert the data bus, parity bit, and so on to begin the selection process.

Observations on ARBITRATION Phase:

- Once a device decides that it is time to arbitrate, it must validate the BUS FREE phase before going any further, even if the bus has been free for some time. Clever circuits can be implemented which constantly validate BUS FREE in anticipation of beginning the arbitration process.

- The 800 nsec Bus Free Delay that Fred had to endure after timepoint (b) was the minimum; the maximum time is called the Bus Set Delay, which is 1800 nsec. In other words, there is a 1000 nsec window after BUS FREE is established to actually do something about it. This lets devices that depend on a clock for timing (most if not all) to have some uncertainty in its clock and still meet the spec. We will see this delay in the next example.

- What happens when Fred loses? Let's look at the next example:



(*1): Others = MSG, C/D, RST, ATN, REQ, ACK.

FIGURE 3: FAST ARBITRATION SEQUENCE (FRED LOSES, GEORGE WINS)

Here's how events proceed in Figure 3:

(a) Both Fred (SCSI Bus ID #2) and George (SCSI Bus ID #4) have been eagerly waiting for their chance at the bus, but someone else had control (probably Sam). At last the bus goes free, but, to be sure, both Fred and George wait 400 nsec, just like in the first example.

(b) The BUS FREE phase has been established, just like the first example. But there is more waiting...

(c) Time to arbitrate! Fred is first out of the gate and he asserts BSY true and also asserts his SCSI bus ID bit on the data bus (Bus value is now **00000100**). Fred asserts no other signals, data bits, or parity bits. But what happened to George? George can still arbitrate as long as he asserts BSY and his SCSI bus ID bit no more than 1800 nsec after last detecting BUS FREE phase. The fact that Fred has asserted BSY first does not invalidate George's attempt at the bus.

(d) Just in the nick of time, George asserts BSY and his SCSI bus ID on the data bus (Bus value is now **00010100**). The data bus now has two bits asserted: Fred's and George's.

(e) Note that Fred's 2400 nsec waiting period began at timepoint (c), while George's began at timepoint (e). In this example, the actual "dust settling" period (the time to wait before looking to see who won) is 1400 nsec (it can be shorter, see note below). Fred looks now and sees that he lost (oh well), because George's SCSI Bus ID bit is in a higher bit number/position. At this point, he can clear himself from the bus, or wait for George to do it for him. Stay tuned to see how...

(f) Since George started late, he must also wait longer to look at the bus. When he does, he sees that he won. As soon as he determines that he won, he is entitled to assert the SEL signal and take what is his. Of course, he can wait longer if he so chooses (see note below).

(g) George asserted SEL, so now Fred had better clear off the bus. He has 800 nsec after SEL becomes true at his input to release all signals he is asserting, which he (of course) does, being a law abiding SCSI device.

(h) Just like Fred did before, after George asserts SEL, he must wait an additional 1200 nsec before he can do anything else; i.e., ***SELECTION Phase***. After the delay, he may then assert the data bus, parity bit, and so on to begin the selection process.

More Observations on ARBITRATION phase:

- Fred has the option (via the standard) to get off when he sees that he lost, or to let George kick him off. Most devices (in our experience) will wait to be kicked off; this is the preferred method. After being kicked off, most devices will re-arm (or can be re-armed) to attempt another arbitration when George is done.

- Note that the 2400 nsec delay is a minimum. George could have waited much longer to look to see if he won (but, it's kind of like waiting for morning to check the paper for Lotto results). Some SCSI chips require the user to perform this step in software; the chip will do the timing and assertion, but the user evaluates the result. A simple software algorithm is (in 8085 code):

```
TEST_WIN: IN   SCSI_DATA       ; Read the current bus state
                               ; into a register
          ANI  NOT MY_ID_BIT   ; Zero the bit in the register that
                               ; corresponds to the device's own
                               ; Bus ID bit
          CPI  MY_ID_BIT       ; Compare the result to the
                               ; device's own Bus ID bit
          JC   I_WON           ; If the result is less than the
                               ; device's own Bus ID bit, I won!
          JMP  TRY_AGAIN       ; Otherwise, try again
```

- This is not the "worst case" timing. Note that George could have started validating BUS FREE phase even later, 400 nsec before timepoint (c). If he had, the "dust settling" period would have been only 600 nsec. If we take into account the Wire-OR Glitch, it turns out that the 2400 nsec specified is just right!

- If you haven't already noticed, there is no "fairness" on the SCSI bus. A device with the lowest bus ID will always lose when arbitrating against any of the other devices. There is an implicit assumption here that devices will tend to get on the bus, do their business as quickly as possible, and then get off. It is expected that there is enough spare bandwidth on the bus to provide the "lulls" necessary for the lowest bus ID device.

- Table 3 shows the standard SCSI ***Bus Timing*** values used during ARBITRATION Phase.

TABLE 3: TIMING VALUES USED DURING ARBITRATION PHASE

Symbol	Timing Name	MIN or MAX?	Time
t_{bsd}	Bus Settle Delay	MINIMUM	400 nsec
t_{bfd}	Bus Free Delay	MINIMUM	800 nsec
t_{bset}	Bus Set Delay	MAXIMUM	1800 nsec
t_{arbd}	Arbitration Delay	MINIMUM	2400 nsec
t_{bcd}	Bus Clear Delay	MAXIMUM	800 nsec

Assertion Period. *Tast = 90 nsec.* The Assertion Period is the minimum period of time that the ***REQ Signal*** or ***ACK Signal*** is held asserted during a synchronous data transfer. See ***Synchronous Data Transfer*** and ***Negation Period.*** More precisely:

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

Note that because of differences in ***Cable*** skew, the received pulse width can be narrower by a few nanoseconds. See also ***Fast Assertion Period.***

Assert Signal. To drive a signal to the ***True*** state. "Asserted" is used in the SCSI standard and in the SCSI Encyclopedia to indicate that a signal or signal pair is driven to the "true" or "one" state. Compare to "negate" and "release". See ***Signal Levels*** for a comparison of these different states.

Asynchronous Data Transfer. If there was ever a "bread and butter" part of SCSI, this is it. The asynchronous data transfer began with ***SASI*** as the way that data is transferred between two devices. For this topic, flow and timing diagrams will show how data is transferred in and out from both the Initiator and Target perspective.

Diagram 3 shows how the Target handles an Asynchronous Transfer Out. The steps involved include:

(1) Establish the Information Transfer ***Bus Phase***.
(2) Request the transfer.
(3) Wait for the response and take the data.
(4) Respond to the response (complete the handshake).
(5) Repeat (2) through (4) as needed to complete the Bus Phase.

Diagram 4 shows how the Initiator handles an Asynchronous Transfer Out. The steps involved include:

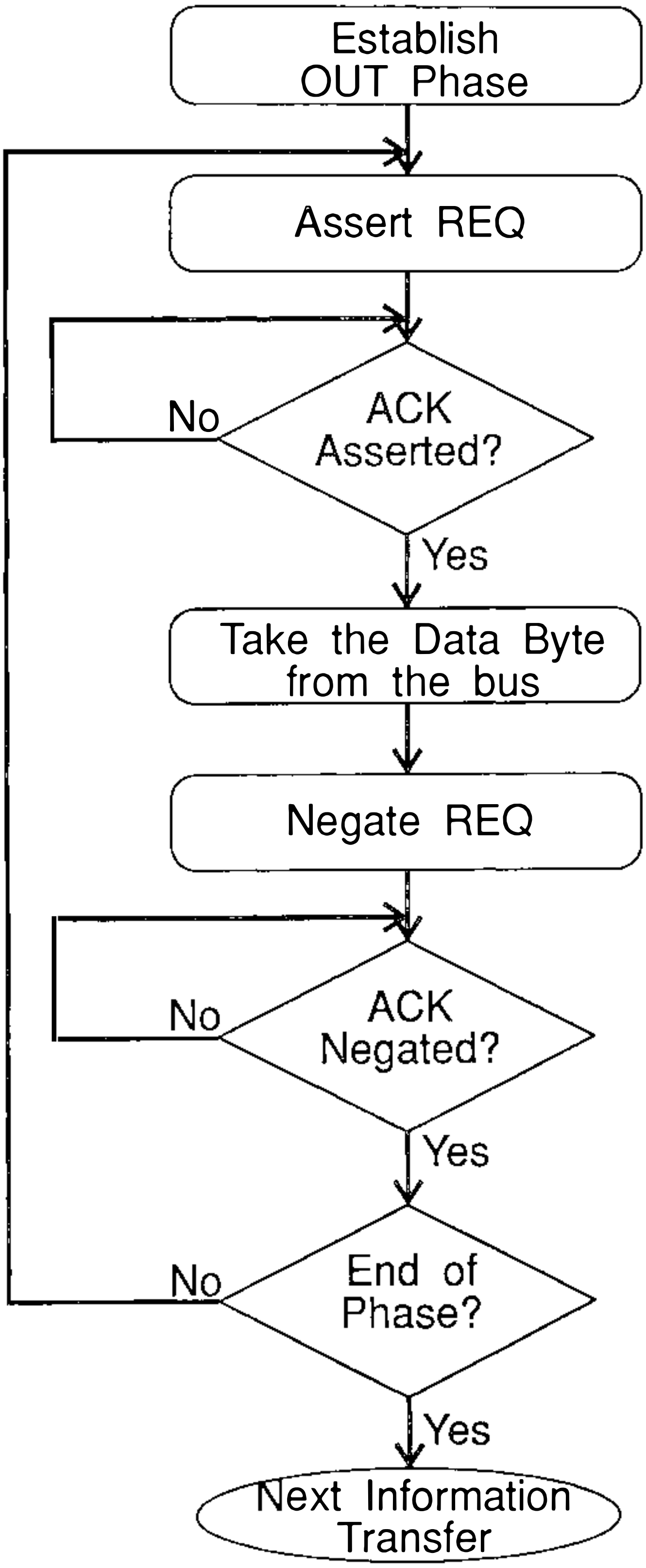
(1) Wait for the Target request to establish the Information Transfer Bus Phase.
(2) Place the data on the bus and respond to the request.
(3) Respond to the response (complete the handshake).
(4) Repeat (2) and (3) as needed until the next Bus Phase.

Diagram 5 shows how the Target handles an Asynchronous Transfer In. The steps involved are almost the same as for a transfer out:

(1) Establish the Information Transfer Bus Phase.
(2) Place the data on the bus and request the transfer.
(3) Wait for the response.
(4) Respond to the response (complete the handshake).
(5) Repeat (2) through (4) as needed to complete the Bus Phase.

Diagram 6 shows how the Initiator handles an Asynchronous Transfer In. The steps involved are almost the same as for a transfer out:

(1) Wait for the Target request to establish the Information Transfer Bus Phase.
(2) Take the data from the bus and respond to the request.
(3) Respond to the response (complete the handshake).
(4) Repeat (2) and (3) as needed until the next Bus Phase.



MSG and C/D set the Bus Phase; I/O is negated

Tells the Initiator to transfer the byte

Wait for the initiator to respond

Valid data may be latched now by the Target

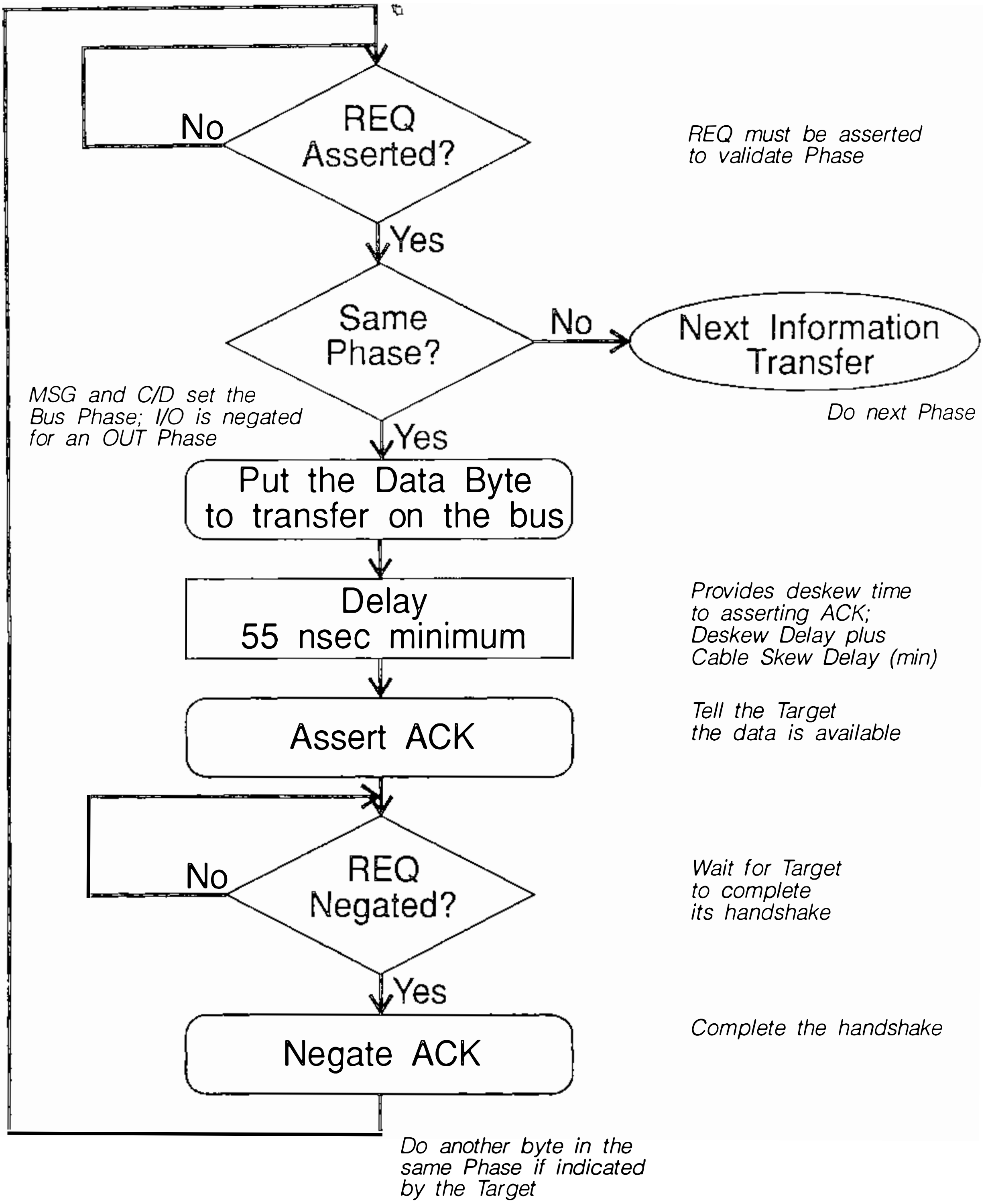
Complete the handshake

Wait for the Initiator to complete the handshake

Do another byte in the same Phase

Go to another Phase

DIAGRAM 3: ASYNC TRANSFER OUT FOR TARGETS



REQ must be asserted to validate Phase

MSG and C/D set the Bus Phase; I/O is negated for an OUT Phase

Do next Phase

Provides deskew time to asserting ACK; Deskew Delay plus Cable Skew Delay (min)

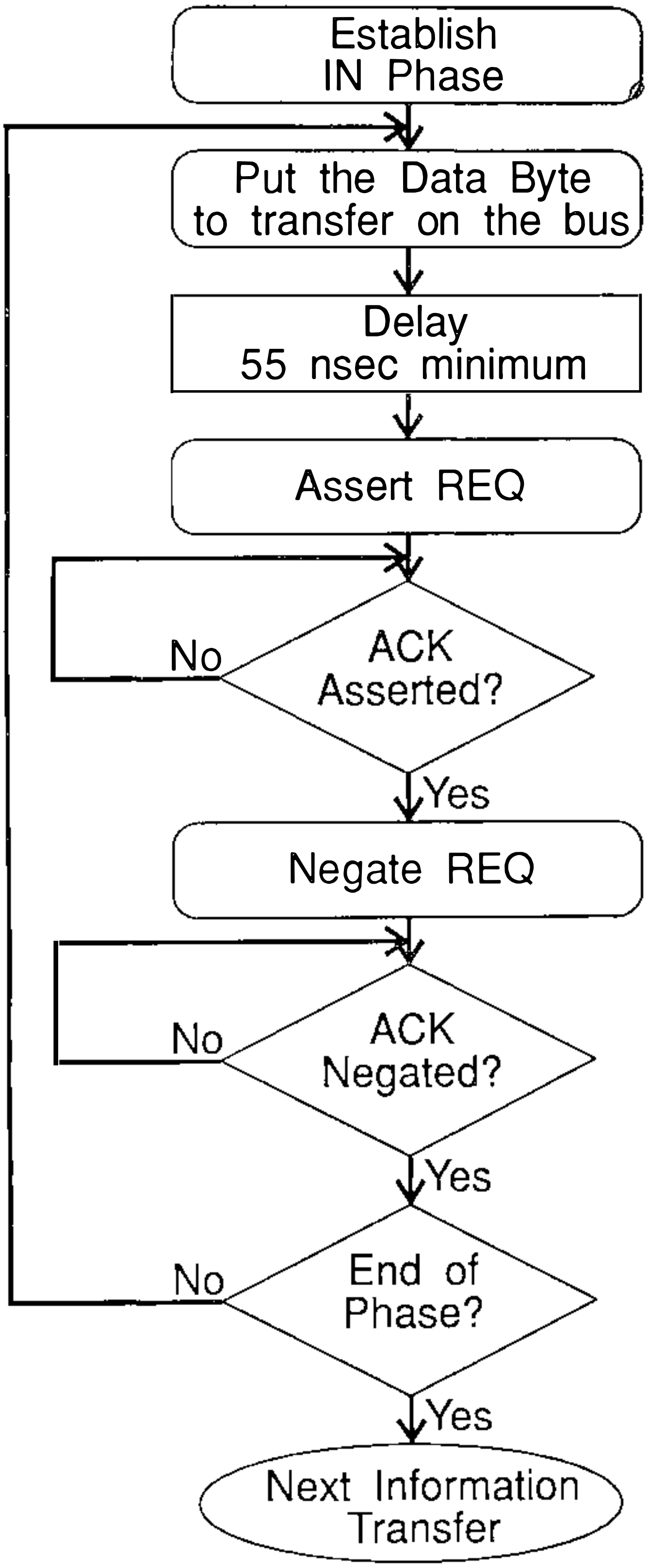
Tell the Target the data is available

Wait for Target to complete its handshake

Complete the handshake

Do another byte in the same Phase if indicated by the Target

DIAGRAM 4: ASYNC TRANSFER OUT FOR INITIATORS



MSG and C/D set the Bus Phase; I/O is asserted

Provides deskew time prior to asserting REQ; Deskew Delay plus Cable Skew Delay (min)

Tells the Initiator to take the byte

Wait for the initiator to respond

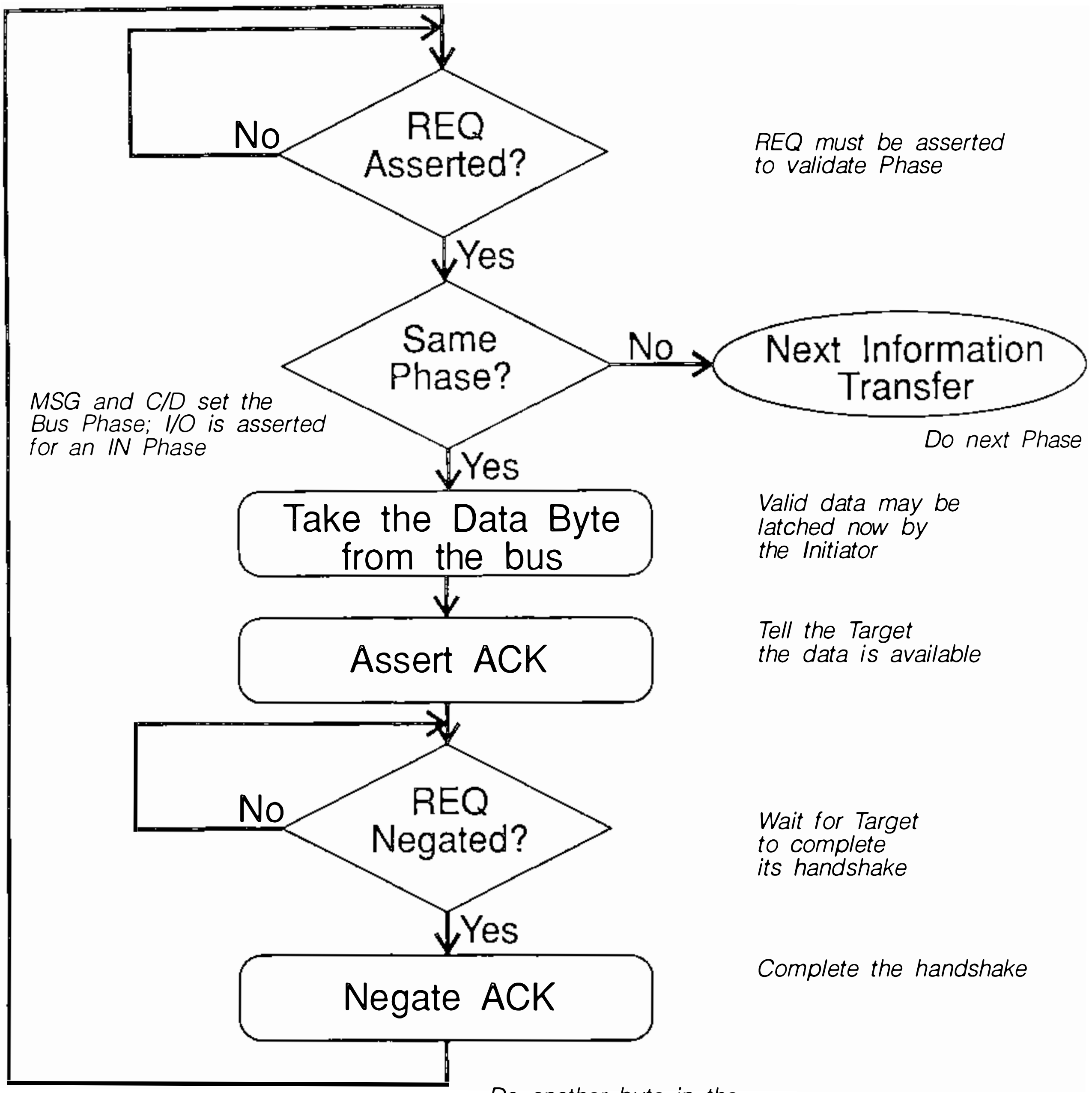
Complete the handshake

Wait for the Initiator to complete the handshake

Do another byte in the same Phase

Go to another Phase

DIAGRAM 5: ASYNC TRANSFER IN FOR TARGETS



REQ must be asserted to validate Phase

MSG and C/D set the Bus Phase; I/O is asserted for an IN Phase

Do next Phase

Valid data may be latched now by the Initiator

Tell the Target the data is available

Wait for Target to complete its handshake

Complete the handshake

Do another byte in the same Phase if indicated by the Target

DIAGRAM 6: ASYNC TRANSFER IN FOR INITIATORS

Now for some examples. The first example (Figure 4) shows how to get data **out to** a Target:

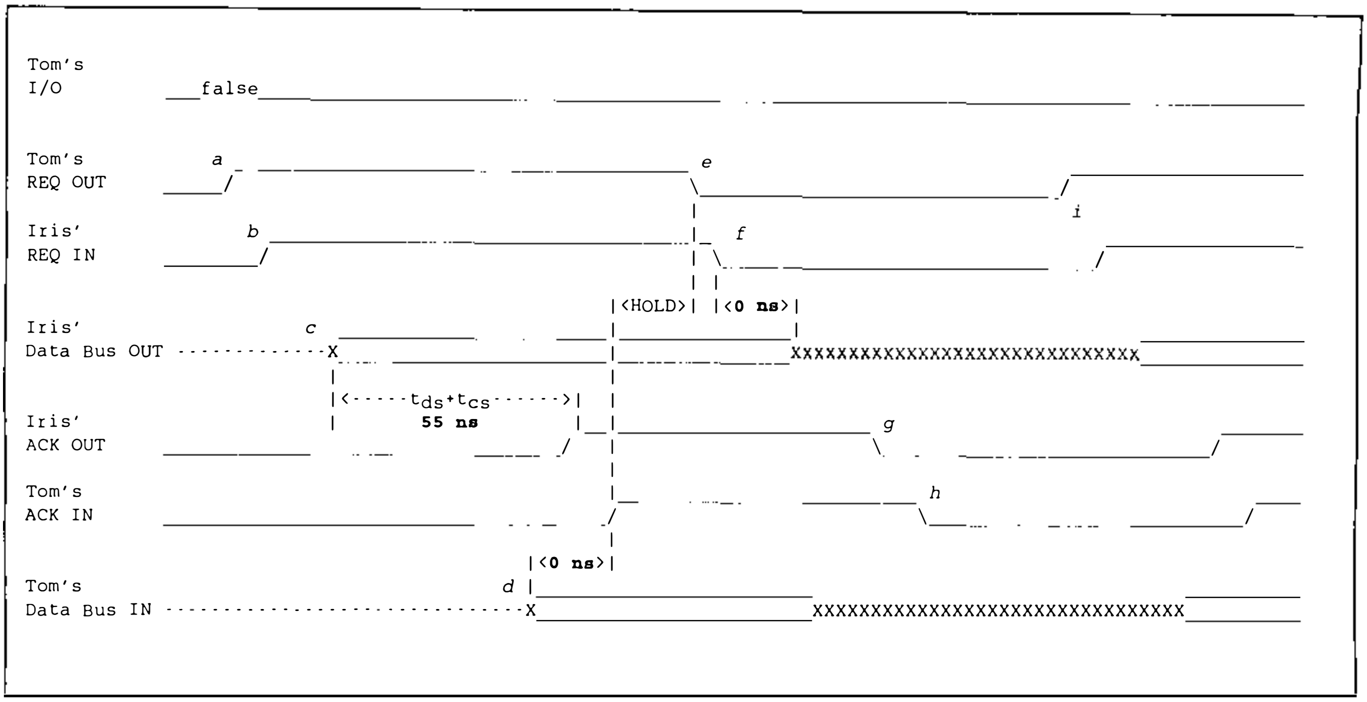


FIGURE 4: ASYNCHRONOUS DATA TRANSFER OUT - INITIATOR TO TARGET

The Details on Figure 4:

(a) Tom asserts his ***REQ Signal*** ("request data, please") true to begin the first transfer of the phase.

(b) After the signal propagates down the cable for a time, Iris detects the REQ signal going true at her input and gets ready to begin the transfer. Depending on the chip and other implementation details, Iris 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.

(c) Now Iris is ready to start sending data to Tom. The first thing she does is to set the data bus to the value of the first data byte to send. After a 55 nsec delay, she asserts her ***ACK Signal*** true ("acknowledging your request, sir; here's your data"). The delay provides at least zero setup time for Tom's receiving latch.

(d) After the data and ACK signal propagate down the cable for a time, Tom 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 latches it from the bus and eventually stores it to its final destination.

Note what happened to the set up time for the data. When it left Iris, there was 55 nsec of lead time between the data and the ACK signal. But when it reached Tom, there was no more lead time. The reason: the 55 nsec is designed to at least provide zero setup time. The 55 nsec time 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 the 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.

(e) After Tom detects the ACK signal going true he may take the data from the data bus. When he has completed taking the data, by whatever means (see next paragraph), he indicates this to Iris by negating his REQ signal ("I got it, thank you"). Note that this implies a "hold" time: By saying "I got it", Tom is saying that he doesn't care if the data bus changes. Therefore, the time between detecting ACK and negating REQ exactly defines the hold time of the data. Tom may establish his hold time to any value simply by delaying the negation of the REQ signal.

Another reason Tom may have for delaying the negation of the REQ signal is to hold off Iris until he is ready to receive more data. This usually happens when Tom's data path stalls; he can't take data until an internal resource becomes available. Some Targets will pause here (i.e., delay negating REQ) until the data path is no longer stalled, and others will delay asserting the next REQ, depending on the implementation of the data path.

(f) However long it takes Tom to negate the REQ signal, Iris detects the event after a propagation down the cable. At the instant that she detects the negated REQ signal, Iris is entitled to remove the data from the data bus and get ready to send the next data byte. Asserting the data early (before ACK is negated) can give Iris a little "head start" on the next transfer, and may improve the transfer rate.

(g) Iris is also entitled to negate the ACK signal as soon as she detects the negated REQ signal. Iris will usually do this as soon as possible. In some cases Iris will hold off because she doesn't have data ready yet; other Initiators will delay asserting ACK at the beginning of the cycle when this occurs. Another reason to hold off negating ACK is that some internal problem occurred and Iris wants to create the ***Attention Condition*** and tell Tom all about it.

(h) After Iris negates the ACK signal, Tom detects it after ACK propagates down the cable. Tom may now assert the REQ signal to start the next cycle.

Some additional observations on asynchronous 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. Bus direction is indicated by the ***I/O Signal***.

- Cable propagation time is a function of cable length and cable design. One type of cable may be faster than another type. A designer of a SCSI device or system should keep in mind that when a device is integrated with other devices, the cable propagation time may be very fast or slow, depending on cable design and how far away the other device is. The SCSI standard assumed a worst case propagation rate of 2.0 nsec per foot (6.6 nsec per meter), and a maximum cable length of 100 feet (30.5 meters, longer than 25 meters), giving a worst case propagation time of 200 nsec.

- When a phase change occurs, the ***MSG, C/D, and I/O Signals*** change to a new state. After 400 nsec (or more), the Target may assert the REQ signal to begin the transfer. The phase does not begin until the REQ signal is asserted. 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 wrong 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 signal is asserted. See ***Between Phases*** for more on this.

- Different Target implementations capture data in different ways. One method is to use a transparent latch: when ACK is low, the latch is open. When ACK goes high, the data is captured. A fast data latch allows the Target to directly reset the REQ signal with the ACK signal, giving a fairly efficient and speedy implementation. Another good method detects and synchronizes the ACK signal to an internal clock. The circuit then generates a write signal that strobes it directly into a data buffer. The trailing edge of the write signal corresponds to the time that REQ can be negated.

- One way to think about asynchronous data transfer OUT is to imagine two friends exchanging things:

  - assert REQ: "May I have the next thing?"
  - assert ACK: "Yes, you may. Here it is."
  - negate REQ: "Thank you, I have it now."
  - negate ACK: "You are most welcome!"

  One might think of this as a "friendly handshake", which is why this is often called the "REQ/ACK handshake". This exchange should remind fans of old Warner Bros. shorts of the Mac and Tosh cartoons...

Now we'll look at how to get data **in from** a Target (see Figure 5):

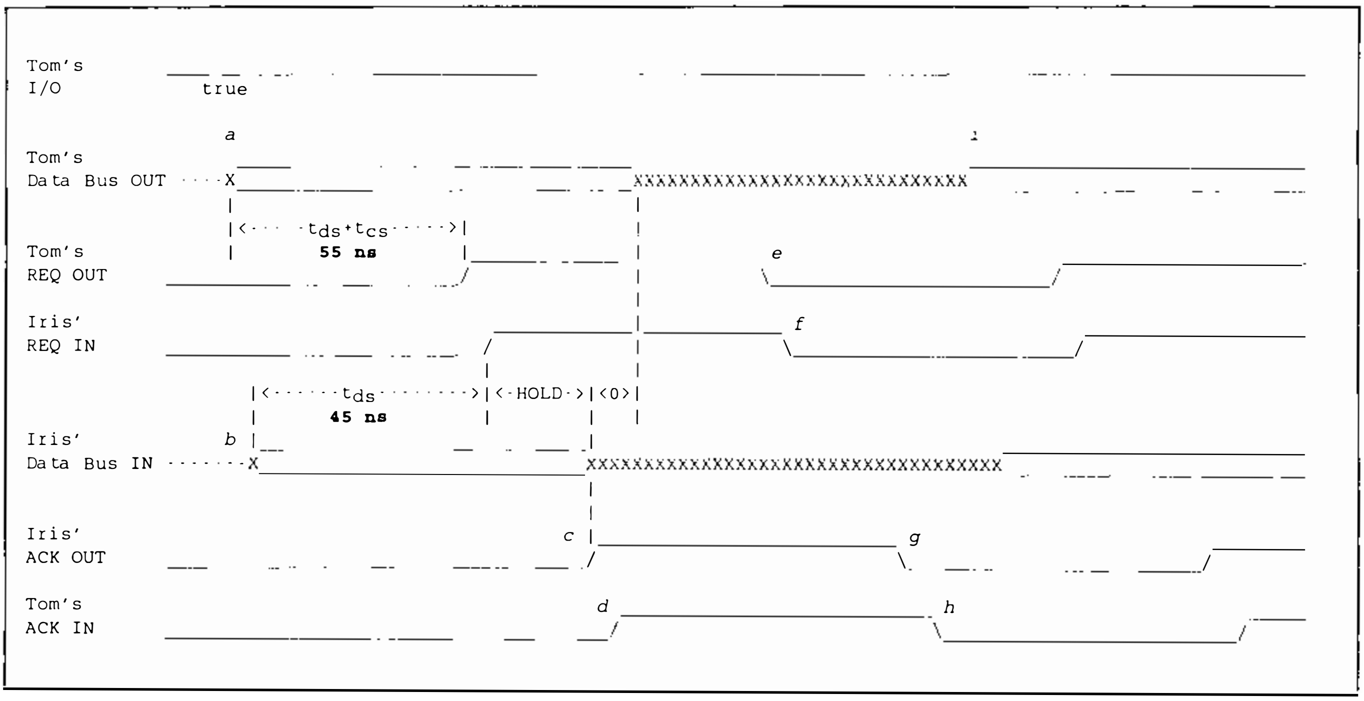


FIGURE 5: ASYNCHRONOUS DATA TRANSFER IN - TARGET TO INITIATOR

The Details on Figure 5:

(a) Tom puts the first data byte onto the data bus and, after a delay of 55 nsec, asserts his REQ signal true ("here is your data, ma'am") to begin the first transfer of the phase.

(b) After the REQ signal and data bus propagate down the cable for a time, Iris detects the REQ signal going true at her input. As with the transfer OUT, she must first get ready to begin the transfer. Depending on the chip and other implementation details, Iris 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 and bookkeeping before she can get started.

Anyway, when she sees the REQ signal, she knows the data is valid on the bus, so, she latches it from the bus and eventually stores it to its final destination.

Note what happened to the set up time for the data. When it left Tom, there was 55 nsec of lead time between the data and the REQ signal. But when it reached Iris, there was no more lead time. As with the transfer out, the 55 nsec is designed to at least provide zero setup time. The 55 nsec time is made up of the same two parts. The first part is called the ***Deskew Delay*** (45 nsec), which is intended to compensate for internal skews in the Initiator. 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 (we cannot stress this too much) therefore, the user ought to use a ***Cable*** that at least meets this requirement.

(c) After Iris detects the REQ signal going true, she may take the data from the data bus. When she has completed taking the data, by whatever means (see note above), she indicates this to Tom by asserting her ACK signal true ("I am acknowledging that I got it, thank you"). Just like for the OUT transfer, this implies a "hold" time. By saying "I got it", Iris is saying that she doesn't care if the data bus changes. Therefore, the time between detecting the REQ signal and asserting the ACK signal exactly defines the hold time of the data. Iris may establish her hold time to any value simply by delaying the ACK signal.

As before, another reason Iris may have for delaying the assertion of the ACK signal is to hold off Tom until she is ready to receive more data. This usually happens when Iris' data path stalls; she can't take data until an internal resource becomes available. Some Initiators will pause here (i.e., delay asserting ACK) until the data path is no longer stalled, and others will delay negating the ACK, depending on the implementation of the data path.

(d) However long it takes Iris to assert the ACK signal, Tom detects the event after a propagation down the cable. At the instant that he detects the asserted ACK signal, Tom is entitled to remove the data from the data bus and get ready to send the next data byte. Asserting the data early (before ACK is negated) can give Tom a little "head start" on the next transfer, and may improve the transfer rate.

(e) Tom is also entitled to negate the REQ signal as soon as he detects the asserted ACK signal. Tom will usually do this as soon as possible. In some cases Tom will hold off because he doesn't have data ready yet; other Targets will delay asserting the REQ signal at the beginning of the cycle when this occurs.

(f) However long it takes Tom to negate the REQ signal, Iris detects the event after a propagation down the cable. At the instant that she detects the negated REQ signal, Iris may negate her ACK signal to complete the transfer, (g). As before, Iris may decide to hold off negating ACK because some internal problem occurred and Iris wants to create the Attention Condition and tell Tom all about it.

(h) After Iris negates the ACK signal, Tom detects it after ACK propagates down the cable. Tom may now put the next data on the data bus and assert the REQ signal to start the next cycle (i).

Some additional observations on asynchronous data transfer IN:

- Different Initiator implementations capture data in different ways, but the ways are similar to the ways Targets can do it. One method is to use a transparent latch: when REQ is low, the latch is open. When REQ goes high, the data is captured. A fast data latch allows the Initiator to directly set the ACK signal with the REQ signal, giving a fairly efficient and speedy implementation. Another good method detects and synchronizes the REQ signal to an internal clock. The circuit then generates a write signal that strobes it directly into a data buffer. The trailing edge of the write signal corresponds to the time that ACK can be asserted.

- One way to think about asynchronous data transfer IN is to imagine two friends exchanging things:

  - assert REQ: "Here is the next thing!"
  - assert ACK: "Thank you, I have it now."
  - negate REQ: "You are most welcome!"
  - negate ACK: "The pleasure is all mine."

  All this politeness can be a bit nauseating, however... Maybe that's the real reason ***Synchronous Data Transfer*** was invented...?

- By now the reader should be thinking that the discussion on transfer IN looks like a word processor massage of the transfer OUT discussion, and maybe the author is trying to pad pages with lots of words? Well, maybe. But it IS important to note how similar the two are. The similarity of the descriptions was intended to make the reader aware of how similar Initiator and Target transfer modes are.

Ever wonder just how fast asynchronous data transfer can be? You've probably heard anything from 1.5 MBytes/second to 4 MBytes/second as vendor claims. Let's see just how fast we could go, if only... if only...

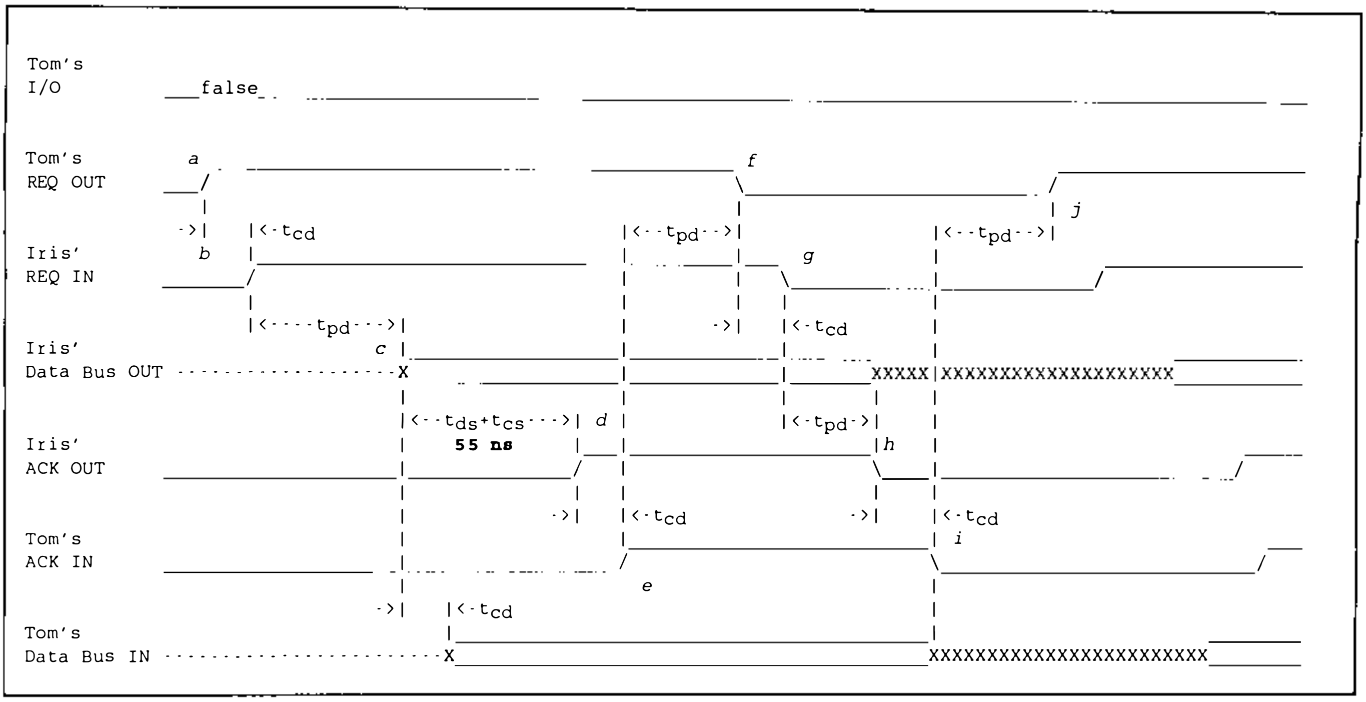


FIGURE 6: ASYNCHRONOUS DATA TRANSFER RATE (OUT)

Figure 6 shows the specified and unspecified delays in a single transfer from the Initiator to the Target. Let's look at the steps:

(a) Tom, the Target, asserts REQ.

(b) One cable propagation delay (t_{cd}) later, the REQ appears at the input of Iris, the Initiator.

(c) One internal propagation delay (t_{pd}) later, Iris asserts her data.

(d) After the standard delay (t_{ds} + t_{cs} = 55 nsec), Iris asserts her ACK.

(e) One cable delay later, the ACK appears at Tom's input.

(f) One internal delay later, Tom has latched the data and negates his REQ.

(g) One cable delay later, the REQ goes false at Iris' input.

(h) One internal delay later, Iris negates her ACK.

(i) One cable delay later, the ACK goes false at Tom's input.

(j) One internal delay later, Tom asserts his REQ for the next transfer.

Adding up the delays:

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

Of course, we are assuming that cable and internal delays are consistent. Anyway, let's try some fairly normal numbers first. The devices are far apart on a 20 foot (6 meter) single ended bus, so cable delay is about 40 nsec. The devices are built from some fairly decent CMOS parts, so a delay time of 50 nsec seems reasonable. Filling in:

t_{xfer} = 4 * 40 + 4 * 50 + 45 + 10 = 415 nsec

So, the transfer rate = 1/415 = 2.4 MBytes/second. Not too bad...

Now let's try some reasonable, but gonzo numbers. The devices are real close together (like in a laptop computer), so cable delay is small, say 1 nsec. The chips are built from the latest technology, and can take data way past when most devices would stall, say 20 nsec delay time. Filling in:

t_{xfer} = 4 * 1 + 4 * 20 + 45 + 10 = 139 nsec

So, the transfer rate = 1/139 = 7.2 MBytes/second!

But, if the devices were REAL close, cable delay would be nearly zero. And if the chips were made from the latest QuadBiTriCMOSpolarRISCwidget process and had virtually zero delay, then the transfer rate would have to be:

t_{xfer} = 45 + 10 = 55 nsec; transfer rate = 1/55 = **18.2** MBytes/second!

There's your upper limit on asynchronous SCSI!

Table 4 shows the ***Bus Timing*** values used during Asynchronous Data Transfer.

TABLE 4: TIMING VALUES USED DURING ASYNCHRONOUS DATA TRANSFER

Symbol	Timing Name	MIN or MAX?	Time
t_{bsd}	Bus Settle Delay	MINIMUM	400 nsec
t_{ds}	Deskew Delay	MINIMUM	45 nsec
t_{cs}	Cable Skew Delay	MAX/MIN	10 nsec
t_{drd}	Data Release Delay	MAX/MIN	400 nsec

Asynchronous Event Notification (AEN). AEN is used by one ***SCSI Device*** to report an error or other condition to one or more other SCSI Devices. AEN takes advantage of the "Peer-to-Peer" nature of SCSI; any SCSI Device may assume either the Target or Initiator role. AEN is optional: a device need not implement it. And now....

*** A NOTE: The complete method of reporting error and other conditions is not documented in this volume of the SCSI Encyclopedia since it is a command level function and the Sense Keys and Codes can have different meaning for each device type. On the other hand, AEN is a special protocol which is appropriate to the ***Path Control*** level functions described in this volume. This makes it difficult to describe these topics in a coherent manner. So, here's what we're going to do:

(1) In this volume, we will cover the basics of the ***Contingent Allegiance Condition***, the ***Extended Contingent Allegiance Condition (ECA)***, and AEN as protocols for managing error handling. Within the description of AEN, we will review the use of ***Status*** and the REQUEST SENSE Command for normal error reporting and for Deferred Error Reporting.

(2) In the other, device specific volumes of the SCSI Encyclopedia, we will review the Contingent Allegiance Condition, ECA, and AEN. In those volumes, device specific error handling will be covered in detail.

A review of normal error reporting: Errors and other conditions (such as the ***Unit Attention Condition***) are indicated to the Initiator by the Target at the end of an ***I/O Process*** during the ***STATUS Phase***. A Status byte is sent by the Target to the Initiator that indicates the general result of the ***SCSI Command***, and in some cases that the Target has Sense Data for the Initiator. This establishes the Contingent Allegiance Condition. The Target then may or may not create the ECA Condition by sending the ***INITIATE RECOVERY Message***. The Target may then return the Sense Data to the Initiator after the Initiator sends a subsequent REQUEST SENSE Command.

The Target may also report a "Deferred Error" to the Initiator via the REQUEST SENSE Command. A Deferred Error is an error that actually applied to a previous I/O Process. In an effort to gain some performance, some Targets may actually "lie" about the completion of an I/O Process. For instance, the Target may take some data to write on a tape, and then indicate that the write has been completed even though it has only bufferred the data. Under normal circumstances, this works nicely; the Target performs the slow write to tape while the Initiator can move on to new tasks.

Unfortunately, this creates a problem when something goes awry. The command associated with the error has already been completed and its ***Pointers*** discarded. In this case, the Target would wait indefinitely for a subsequent command from the Initiator to the same ***Nexus*** that had the error. The Target would then report an error for the new command, and then return Sense Data that indicates a Deferred Error for the previous command.

Okay, so what is AEN? AEN is a method whereby a Target can take a more proactive approach to error reporting. Usually, the Target must wait for an Initiator to select it before it can report one of the following conditions:

- An error condition that occurred after the I/O Process associated with it has completed; in other words, a Deferred Error. Note that errors which occur during the command execution are not reported by AEN. They must be reported as normal errors.

- A ***Unit Attention Condition***. Normally, the Target waits until the Initiator selects it to report a Unit Attention Condition. By using AEN, the Target can let the Initiator know about changes on a ***Logical Unit*** without any waiting. It can be useful to report the condition immediately if, for instance, a delay in reporting leaves the Initiator's data in jeopardy.

- A special case of Unit Attention Condition is when a device goes online or offline. Using AEN helps keep the status of a system current to the user.

- Any other asynchronous event you can think of, such as a change of media.

Two elements are required for AEN to be used:

- A SCSI Device which normally assumes the Initiator role that is also capable of assuming the Target role. An example is a ***Host Adapter*** that can assume both roles. Note that we are referring to a SCSI Device that is normally an Initiator, not a Target. When the SCSI Device assumes the Target role, it has a Logical Unit 0, which must be a Processor Device.

- A SCSI Device which normally assumes the Target role that is also capable of assuming the Initiator role. An example is a ***Controller*** that can assume both roles. Note that we are referring to a SCSI Device that is normally a Target, not an Initiator. When the SCSI Device assumes the Initiator role, it does so only to implement AEN, and possibly the COPY Command.

A SCSI Device determines that another Device supports AEN by ***Selecting*** the other Device and issuing an INQUIRY Command. If the other Device responds to the Select, and returns the proper INQUIRY data, then the first Device knows that the other Device supports AEN. Usually, a Device attempts this exchange only with Initiators that have previously Selected it as a Target.

In a nutshell, the following steps occur:

(1) A Controller detects a condition that must be immediately reported to the Host System. The Target takes the Initiator role and selects the Host Adapter.

(2) The Host Adapter responds to the ***SELECTION Phase*** by assuming the Target role and asserting the ***BSY Signal***.

(3) The Controller (as an Initiator) issues a Processor Device SEND Command to the Host Adapter.

(4) The Host Adapter takes the SEND Command and enters a ***DATA OUT Phase***. The Controller sends a Sense Data block during the phase.

(5) The Host Adapter completes the command and ***Disconnects*** from the bus.

Of course, we will illustrate the above steps in detail with the help of our friends, Ian and Tanya, and a little cross dressing...

To set the stage, Ian is a Host Adapter attached to a file server system. Ian's usual role is Initiator. Tanya is an optical disk drive, and usually takes the Target role.

(1) Someone goes to Tanya's drive and changes the mounted disk. We'll assume for this example that this happened without the permission of the system operator. As a result, the change occurred without the proper procedure being executed. Tanya detects the unauthorized change and decides to report it immediately via AEN to the host system.

(2) Tanya waits for a ***BUS FREE Phase***, and enters the ***ARBITRATION Phase***. After winning the bus, she asserts the ***SEL Signal***, but she does not assert the ***I/O Signal***, which indicates that she is taking the Initiator role to select another SCSI Device which is expected take the Target role. She asserts her ***SCSI Bus ID*** and Ian's SCSI Bus ID and enters the SELECTION Phase.

(3) Ian sees the SELECTION Phase and his SCSI Bus ID, and realizes that he is being selected. He responds by asserting the BSY Signal. Tanya then releases the SEL Signal.

At this instant, Tanya, who is now an Initiator, is ***Connected*** to Ian, who is now a Target. To try to keep things clear while they are in their opposite roles, we will refer to them as Tanya^{I} (for Tanya the Initiator) and Ian^{t} (for Ian the Target).

(4) Via the ***MESSAGE OUT Phase***, a Nexus is established between Tanya^i and Ian^t. Note that the ***Logical Unit Number (LUN)*** portion of the Nexus is always for Logical Unit 0. In other words, the ***IDENTIFY Message*** sent by Tanya^i contains an LUN of zero, not the LUN of her disk drive.

(5) Via the ***COMMAND Phase***, the SEND Command is issued by Tanya^i to Ian^t. There is a bit in the SEND ***Command Descriptor Block (CDB)*** that indicates that AEN data is to be transferred.

(6) Ian^t switches to DATA OUT Phase, and Tanya^i sends the AEN data block. Within the data block (which is described in the SCSI standard and also in another volume of this Encyclopedia) is a field which describes which Logical Unit the data refers to. The rest of the data block contains data identical to a Sense Data block that describes the unauthorized disk change.

(7) Ian^t completes the command by sending GOOD ***Status*** during the ***STATUS Phase***, and then sends a ***COMMAND COMPLETE Message*** during a ***MESSAGE IN Phase***. After the message, Ian^t goes to ***BUS FREE Phase*** by releasing the ***BSY Signal***.

What has happened is that by using AEN Tanya was able to inform Ian about an unauthorized disk change. In this case the system can inform a user about the change immediately, so he/she can run to the server and tackle the fool who changed disks. Without using AEN, the user wouldn't know about the change until his/her next access of the disk. An early warning may be able to give the user more options as to what to do about a problem.

Okay, but is AEN worth it? In reality, for the majority of systems, probably not. Software that could handle an early warning as in the above example could also probably give the user a host of options to react to the problem if it was reported on a subsequent command. Software that isn't smart enough to give those options probably can't process an unexpected warning either. AEN will probably find a place in high-end fault tolerant systems and disk arrays, where a switch to a redundant element can be triggered by an AEN.

AEN is costly to implement as well. While most SCSI ***Chips*** are able to do both Initiator and Target roles, the software to handle those chips in two roles is very complicated. Many man months are involved with developing Initiator software for a Target, or Target software for an Initiator. Even though only a handful of commands are involved, all of the software to deal with ***Error Handling***, ***Path Control***, ***Message System***, and ***Pointers*** must be written. It should be noted that SCSI Devices that already must support the COPY command as Targets will more easily support AEN.

Some companies have implemented a non-standard version of AEN where it is not necessary for a device to take the opposite role. In this method, a Target that wants to send AEN data to an Initiator ***Reconnects*** to the Initiator. If the Initiator accepts the Reconnect, the Target sends the AEN data and completes the Connection as if it were a regular command. In this situation, both the Initiator and Target behave in a non-standard manner:

- The Target reconnects to a Nexus that is not active.

- The Initiator accepts the reconnect.

As a result, both devices have to be built to operate in this manner, and by extension, the system must be configured to operate this way. While we don't condone non-standard implementations (see ***Etiquette***), we offer this information in case you might encounter it. Because this method requires less effort to implement for both the Initiator and Target, it could become a de facto standard.

ATN Signal. The ATN Signal is asserted by an Initiator to indicate to a Target that it really wants to "talk". The Target should respond to the Initiator's request as soon as possible via a ***MESSAGE OUT Phase***. The ATN signal is the only non-destructive way that the Initiator can tell the Target to "hold on a minute". The ATN signal should only be asserted at certain times (relative to the ***ACK Signal*** or ***BSY Signal***) so that the intent of the Initiator is not misunderstood by the Target. The ATN signal may never be legally asserted during ***ARBITRATION Phase*** or during ***BUS FREE Phase***. See ***Attention Condition***.

Attention Condition. The Attention Condition is created by an Initiator during a ***Connection*** when it wants the Target to "pay attention" to it. The Initiator does this by asserting the ***ATN signal*** on the bus. When the Target notices the ATN signal, it is supposed to go to ***MESSAGE OUT Phase***. This lets the Initiator send a ***Message*** to the Target to tell it why it needed the Target's attention. What could be simpler? "Hey you! I've got a message for you!" "OK, hang on.... OK go ahead!" " OK, here's my message....!" See the ***Message System*** for details on how the Attention Condition relates to message handling.

Let's look at some timing in Figure 7.

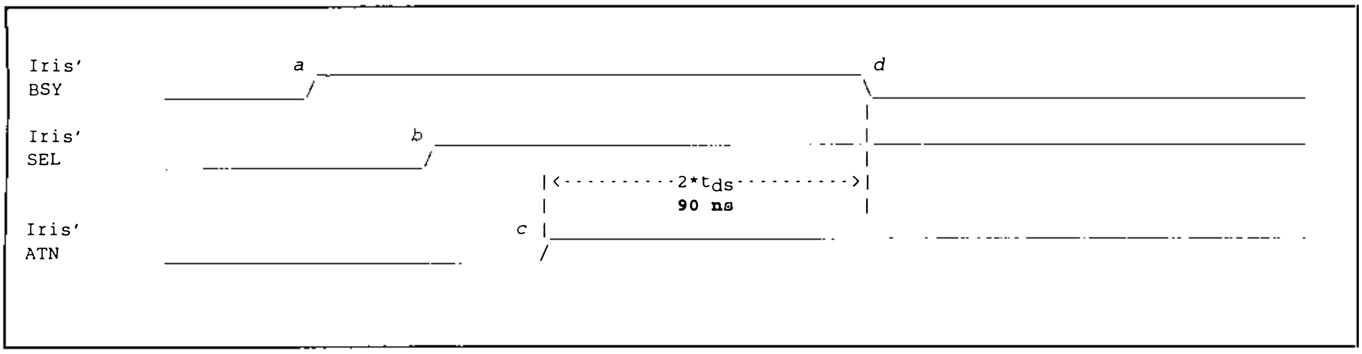


FIGURE 7: ATN ASSERTED DURING SELECTION

The Details on Figure 7:

(a) The Initiator (our friend Iris) arbitrates for the bus in the usual way (see ***ARBITRATION Phase***).

(b) Iris wins arbitration and takes control by asserting the ***SEL Signal***.

(c) Before Iris negates the ***BSY Signal*** to begin ***SELECTION Phase***, she asserts the ATN signal to indicate to Tom (the Target she wants to select) that she wants to start with a MESSAGE OUT phase. Of course, Tom is not aware of this because he doesn't know he is being selected yet...

(d) After a delay of two ***Deskew Delays*** (90 nsec) from asserting the ATN signal, Iris negates the BSY signal and the ***SELECTION Phase*** begins. The purpose of the delay is to ensure that Tom does not miss it: if the ATN signal was asserted after the BSY signal was negated, a fast Target like Tom might miss it.

Other observations:

- Iris is not allowed by the standard to assert ATN during the ***BUS FREE Phase*** or during ARBITRATION Phase. BUS FREE is not allowed because the bus is not hers when it is free. Arbitration Phase is not allowed because she has not yet won the bus; it is not hers. Note that ARBITRATION Phase ends when Iris asserts the SEL signal.

- The usual reason for asserting the ATN signal during Selection is to send the ***IDENTIFY Message*** to the Target. Other reasons might be to send an ***ABORT Message*** or a ***BUS DEVICE RESET Message***.

- Iris might also want to assert the ATN signal in response to a ***RESELECTION Phase*** by Tom. One reason to do this would be to send an ABORT message. There is no special protocol for RESELECTION Phase because Tom is not required (by the standard) to respond to the ATN signal until some point after he completes the RESELECTION Phase. NOTE: Unfortunately, as of this writing, the exact response time relative to RESELECTION Phase has not been nailed down. Contact the ***X3T9.2 Committee*** for the current definition.

Now, we will look at asserting the ATN signal during data transfers (Figure 8):

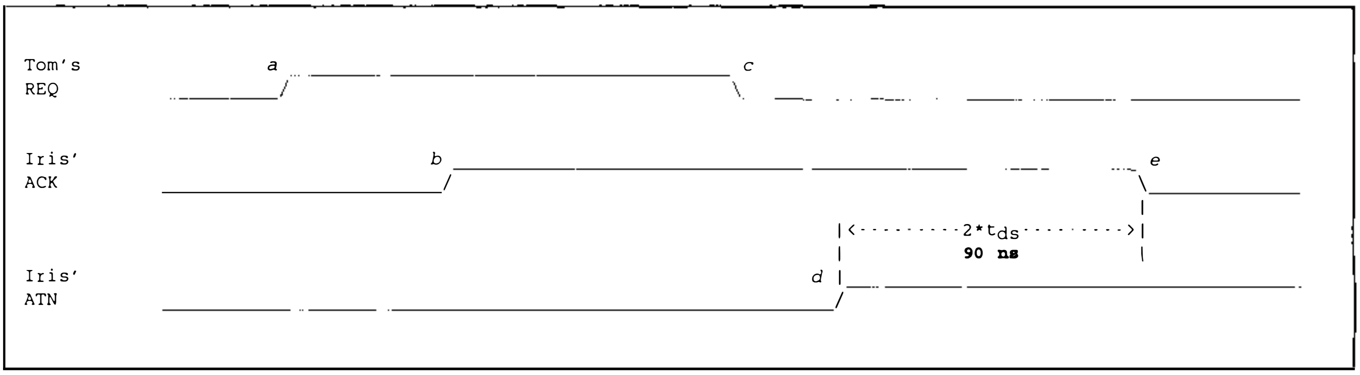


FIGURE 8: ATN ASSERTED DURING ASYNCHRONOUS DATA TRANSFER

The Details on Figure 8:

(a) Tom asserts his ***REQ Signal*** in the usual manner.

(b) Iris asserts her ***ACK Signal*** in the usual manner.

(c) Tom negates his REQ in response to Iris' ACK, in, again, the usual manner. Note at this point that the data exchange has been completed, no matter which direction the transfer was going (see ***Asynchronous Data Transfer***). All that's left is to complete the handshake....

(d) Iris decides she's got a Message for Tom, so she asserts ATN at this time, before negating ACK.

(e) After two ***Deskew Delays*** (90 nsec), Iris negates ACK to complete the handshake. Once again, asserting ATN before negating ACK provides an *interlock* that enables Tom to establish that the ATN signal is related to the current byte handshake and phase.

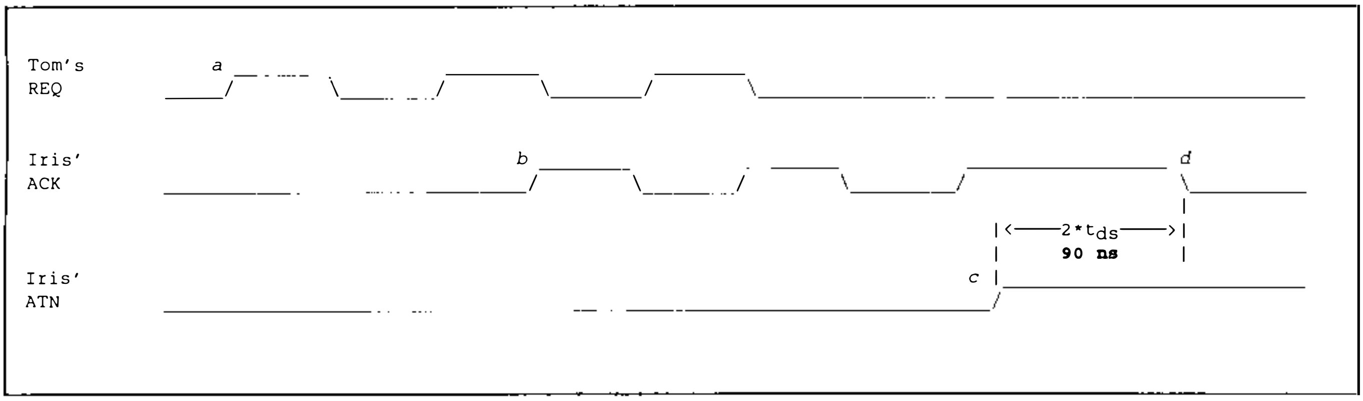


FIGURE 9: ATN ASSERTED DURING SYNCHRONOUS DATA TRANSFER

The Details on Figure 9:

(a) Tom asserts his stream of REQ pulses in the usual manner.

(b) Iris asserts her stream of ACK pulses in the usual manner. (see ***Synchronous Data Transfer***).

(c) Iris decides she simply MUST speak with Tom, so she asserts ATN at this time, before negating ACK for the last pulse. This behavior is typical of many devices: the ACK pulse is sustained to allow the Initiator time to decide to assert ATN. Other devices will hold off asserting the ACK pulse until ATN is asserted.

(d) After two ***Deskew Delays*** (90 nsec), Iris negates ACK to complete the ACK pulse. Once again, asserting ATN before negating ACK provides an *interlock* that enables the target to establish that the ATN signal is related to the current phase. As a practical matter, the Target will not be able to determine exactly when ATN was asserted in the middle of a transfer due to the high speed of data transfer.

Other observations:

- The **MOST IMPORTANT THING** to remember about the Attention Condition is that the Target can respond to the ATN signal **whenever it wants to** during the current phase, but if it is going to change to another phase from the current phase, it must change to MESSAGE OUT Phase if the Attention Condition exists (some SCSI-1 Targets did not do this, see next note). The Target does not have to respond until it is convenient for the Target.

- Starting with SCSI-2, the Target **must** respond before going to some other phase, since it is likely that the message is related to the current phase. If the Target (a SCSI-1 Target) does not respond to the ATN signal with MESSAGE OUT Phase right after the current phase, then the Initiator may have to decide whether the message is still appropriate. See "Target Response to Attention" below.

Figure 10 shows how the Attention Condition ends:

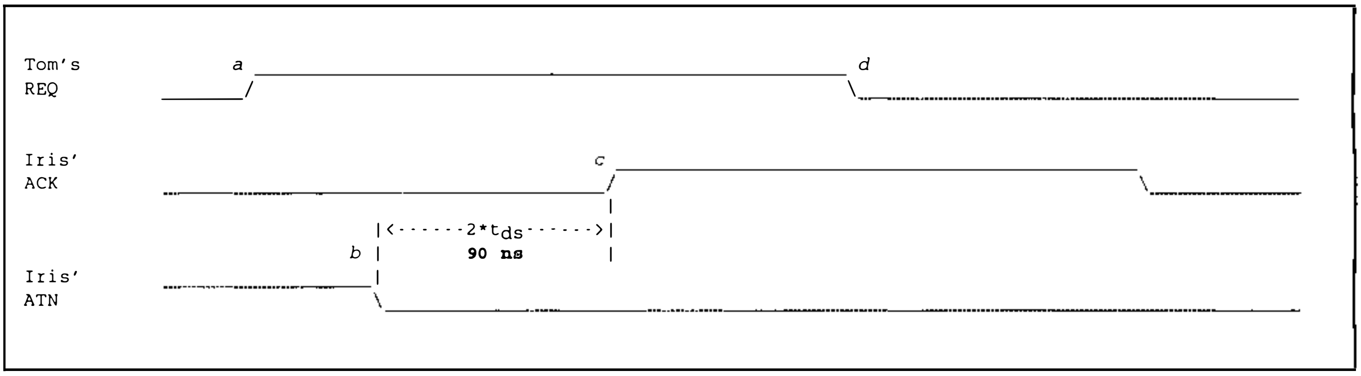


FIGURE 10: ATN NEGATED AFTER ASYNCHRONOUS MESSAGE OUT TO TARGET

The Details on Figure 10:

(a) Tom asserts his REQ signal to begin a MESSAGE OUT Phase transfer.

(b) Before transferring the last byte of the MESSAGE OUT, Iris negates her ATN signal. This lets Tom know that this is the last byte of the MESSAGE OUT Phase.

(c) Iris then waits two ***Deskew Delays*** (90 nsec) and asserts the ACK signal.

(d) The rest of the MESSAGE OUT handshake completes normally.

Other observations:

- If more than one MESSAGE OUT byte is to be sent (for example, a ***Synchronous Data Transfer Request*** message), then the ATN signal is held true through all bytes until the last byte is transferred, as described under ***MESSAGE OUT Phase***.

- The ATN signal can become asserted again during a MESSAGE OUT phase as part of the error recovery process. See ***Message System***.

Now, we will put it all together into two typical sequences in Figure 11 and Figure 12:

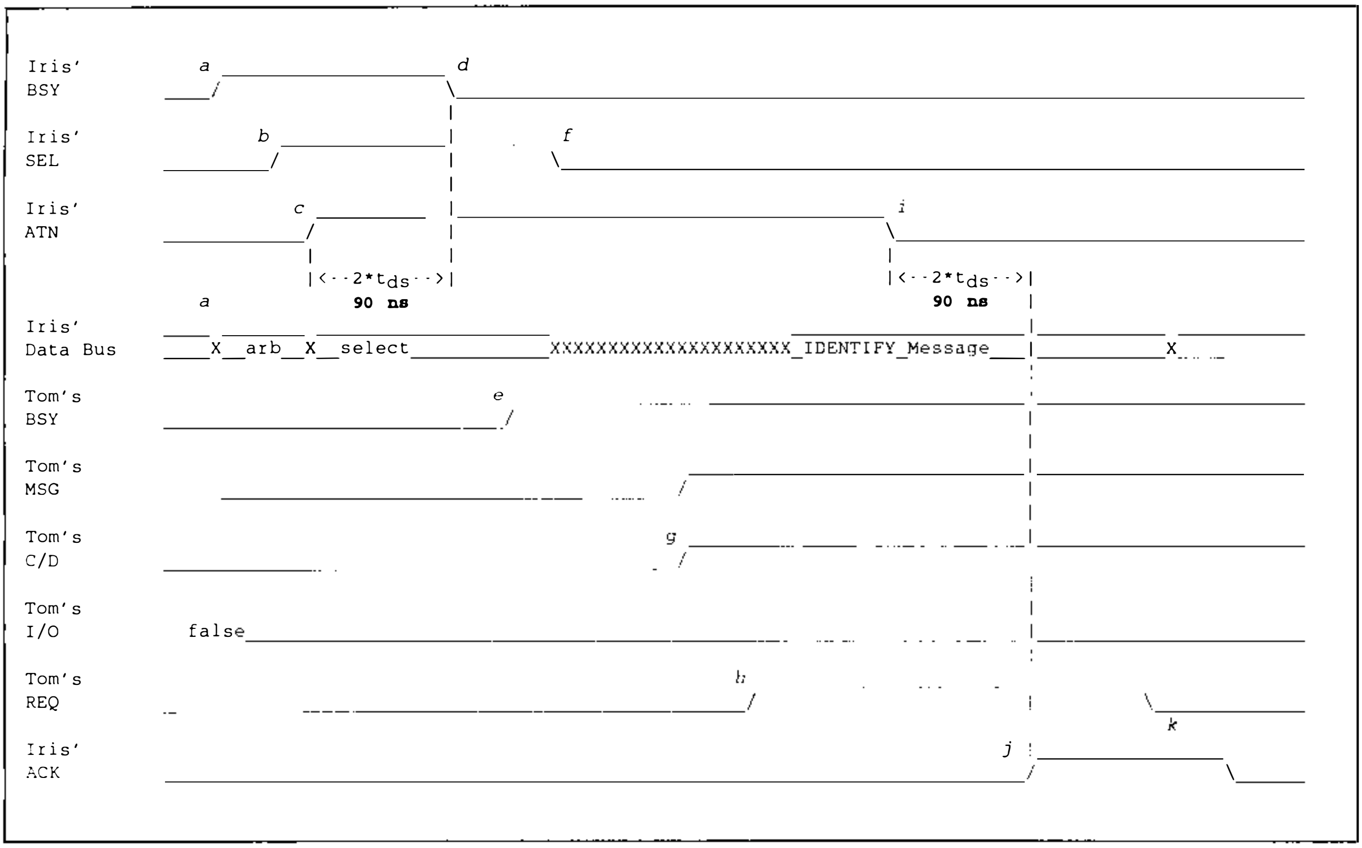


FIGURE 11: PUT IT ALL TOGETHER: IDENTIFY MESSAGE OUT AFTER SELECTION

The Details on Figure 11:

(a) Iris asserts the ***BSY Signal*** and her ***SCSI Bus ID*** on the ***Data Bus*** during ***BUS FREE Phase*** to begin ***ARBITRATION Phase***.

(b) Iris wins ARBITRATION and asserts her ***SEL Signal*** to take the bus.

(c) Iris asserts the ATN Signal and hers and Tom's SCSI Bus ID on the Data Bus.

(d) After two Deskew Delays, Iris releases the BSY Signal to begin ***SELECTION Phase***.

(e) Tom (remember him?) asserts his BSY Signal to respond to the selection from Iris.

(f) Iris releases her SEL Signal to complete SELECTION Phase.

(g) Tom sees Iris' ATN Signal after SELECTION Phase is completed. To respond to the ATN, he asserts the ***MSG Signal*** and the ***C/D Signal*** to set up for a MESSAGE OUT Phase.

(h) Tom asserts the ***REQ Signal*** to begin the MESSAGE OUT Phase.

(i) Iris sees the REQ Signal. Since the phase is now MESSAGE OUT, she places the ***IDENTIFY Message*** on the Data Bus, and she negates the ATN Signal.

(j) After two Deskew Delays, Iris then asserts the ACK Signal to send the IDENTIFY Message.

(k) Tom and Iris complete the handshake.

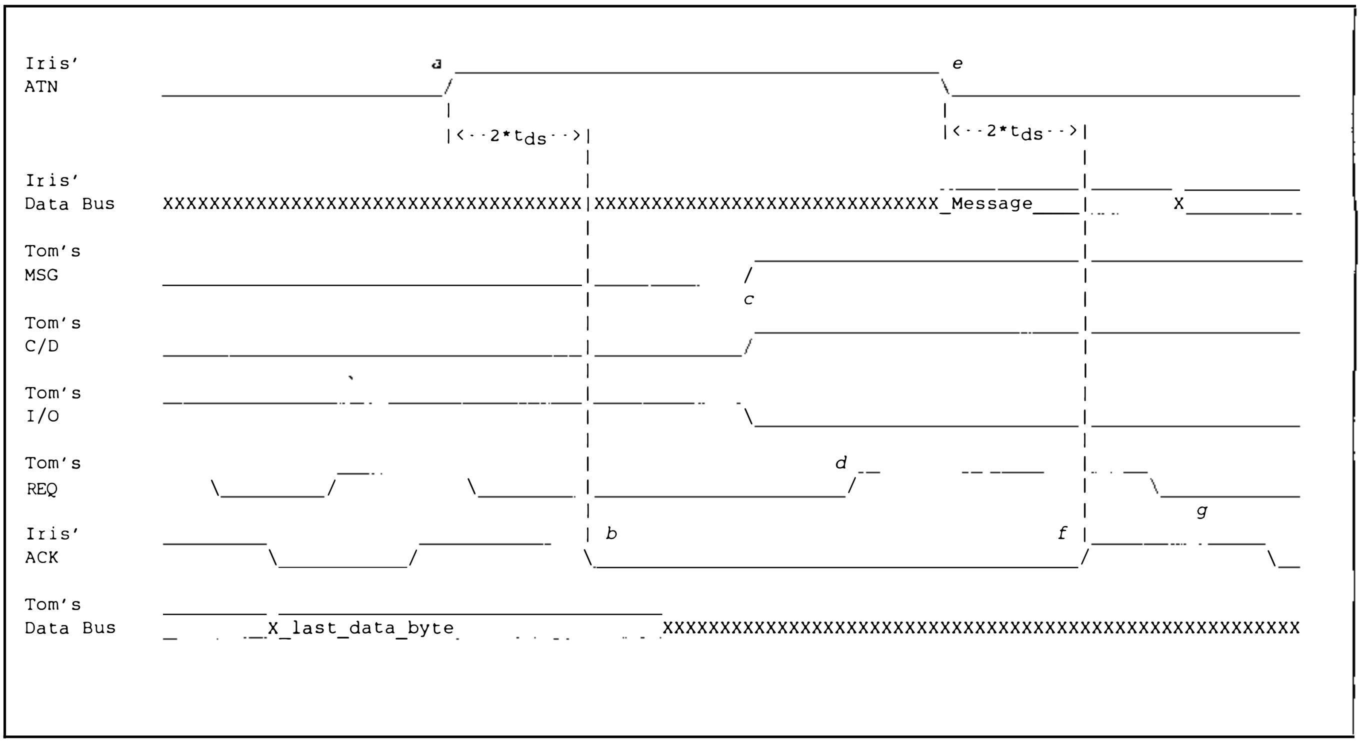


FIGURE 12: PUT IT ALL TOGETHER: MESSAGE OUT AFTER DATA TRANSFER

The Details on Figure 12:

(a) Before completing the ACK of a ***DATA IN Phase*** transfer, Iris asserts her ATN Signal.

(b) Two Deskew Delays later, Iris negates ACK to complete a data transfer.

(c) Tom sees the ATN Signal and asserts MSG and C/D to set up for a MESSAGE OUT Phase.

(d) Tom asserts his REQ Signal to begin the MESSAGE OUT Phase.

(e) Iris sees the REQ Signal. Since the phase is now MESSAGE OUT, she places the Message on the Data Bus, and she negates the ATN Signal.

(f) After two Deskew Delays, Iris then asserts the ACK Signal to send the Message.

(g) Tom and Iris complete the handshake.

Target Response to Attention. As of SCSI-2, the response of a Target to the Attention Condition was nailed down a little. Since these rules are little more than "common sense", you will find that most Targets behave in the manner shown in Table 5.

TABLE 5: TARGET RESPONSE TO THE ATTENTION CONDITION

If the Attention Condition occurs during:	Then the Target must enter MESSAGE OUT Phase:
COMMAND Phase	After all or part of the ***Command Descriptor Block (CDB)*** has been transferred
DATA IN or DATA OUT Phase	After all or part of the Data Transfer intended by the Target. With luck, this will be a ***Logical Block*** boundary. The standard says "Target's earliest convenience", a non-operative statement if we ever heard one....
STATUS Phase	After the ***Status*** byte has been transferred.
MESSAGE IN Phase	After the Target has sent one complete ***Message***, and before sending another one. In other words, don't send a partial Message (if it is a multiple-byte Message) before responding to the Attention Condition.
SELECTION Phase, before the Initiator releases BSY	After the SELECTION Phase is completed.
RESELECTION Phase, after the Target releases BSY	After the RESELECTION Phase is completed. The exact time was not established at publication time. Contact the ***X3T9.2 Committee*** for the latest information.

If the Attention Condition occurs at any other time, the Initiator is violating the Standard!

Table 6 shows the ***Bus Timing*** values used during the Attention Condition.

TABLE 6: TIMING VALUES USED DURING THE ATTENTION CONDITION

Symbol	Timing Name	MIN or MAX?	Time
t_{ds}	Deskew Delay	MINIMUM	45 nsec

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.

B Cable. The B Cable is the ***SCSI-2*** name of the 68-conductor cable used in all 16-bit and 32-bit SCSI systems (see ***Wide Data Transfer***). The B Cable carries 24 more bits of ***Data Bus Signals***, as well as the ***REQB Signal*** and the ***ACKB Signal***. See ***Cables***. See ***P Cable*** for a "better" way to do 16-bit and 32-bit SCSI systems....

Between Phases. "The New SciFi Block Buster...." Actually, the SCSI committee has found it very easy to define what a phase **is** (see ***Bus Phases***), but has had quite a time defining what a phase **isn't**. If you find the definitions regarding phase changes and the states between phases confusing, you aren't alone.

The confusion arises because SCSI is not a pure state machine. You must know the bus history to interpret the current state the bus. You might think that when both the ***REQ Signal*** and ***ACK Signal*** are negated after ***SELECTION Phase*** or ***RESELECTION Phase*** can be considered as "between phases", because the SCSI Standard states that a phase is not defined until REQ is asserted. In other words, when REQ is asserted it qualifies the ***Bus Phase Signals*** (again see ***Bus Phases***). However, if a Target is halfway through getting the command from the Initiator, it seems silly to think of the transfer as anything but a continuous ***COMMAND Phase***. Indeed, many other definitions (particularly the ***Message System***) rely on the fact that a transfer of several bytes is a single Bus Phase.

Technically, an ***Information Transfer Phase*** ends when the Bus Phase Signals change after the Negation of ACK at the end of a bus transfer. In general, if you assume that a phase exists until the REQ signal qualifies a different bus phase, you will be correct. In practice, most modern devices will change the phase cleanly, so there is little chance for ambiguity. Some older devices did not change phase cleanly, which caused some grief with other devices that didn't wait for REQ to qualify the phase.

Note that as there are no "between phase" states for the ***Connection Phases***; there are no gray areas between those phases. The transition from SELECTION Phase or RESELECTION Phase to an Information Transfer Phase does follow the rules described here.

The SCSI standard defines the following rules for behavior "between phases":

- Certain signals must not change when the intent of the Target is to change the three ***Bus Phase Signals*** (MSG, C/D, I/O). It must sound impossible for the Initiator to know the Target's intent; let's look at each signal and see how it works:

  BSY Signal: The Target must hold BSY true while it continues performing bus phases. Of course, the standard provides exceptions to this:

  - BSY goes false after a Message; i.e., a transition to ***BUS FREE Phase***.

  - BSY goes false because of a fatal error; i.e., like a power failure. See ***Unexpected BUS FREE Phase***.

  SEL Signal: SEL must be held false by both the Initiator and the Target once the SELECTION Phase (or RESELECTION Phase) is completed. Neither Initiator or Target has any business asserting SEL during any Information Transfer Phase.

  REQ Signal and ***REQB Signal***: Yes, this sounds a little funny: "How can I change the phase if I can't change REQ or REQB?" This is on the list only to indicate that the REQ and REQB signal can change only under certain conditions (see Diagram 7).

  ACK Signal and ***ACKB Signal***: This is easy: The Initiator has no business asserting ACK or ACKB without REQ or REQB asserted (or without a REQ or REQB pulse outstanding). Also, the Target can't change phase if ACK or ACKB are asserted, so there is no way for ACK or ACKB to change between phases. **NOTE:** This is a good way for the Initiator to prevent the Target from changing phases before the Initiator is ready. A use of this would be to hold the phase until the Initiator can create the ***Attention Condition***.

- The ***ATN Signal*** is allowed to change as described under Attention Condition. The ***RST Signal*** is allowed to change as described under ***Reset Condition***.

- The ***MSG Signal***, ***C/D Signal***, ***I/O Signal***, and the ***Data Bus Signals*** may change between phases. There are very specific rules for changing the direction of Data Bus drive and for establishing the Bus Phase Signals prior to REQ and REQB. The rest of this topic describes the timing for bus direction changes.

Diagram 7 and Diagram 8 (respectively) show how the Target and Initiator handle switching between phases. As you can see, the rules attempt to prevent contention on the Data Bus Signals.

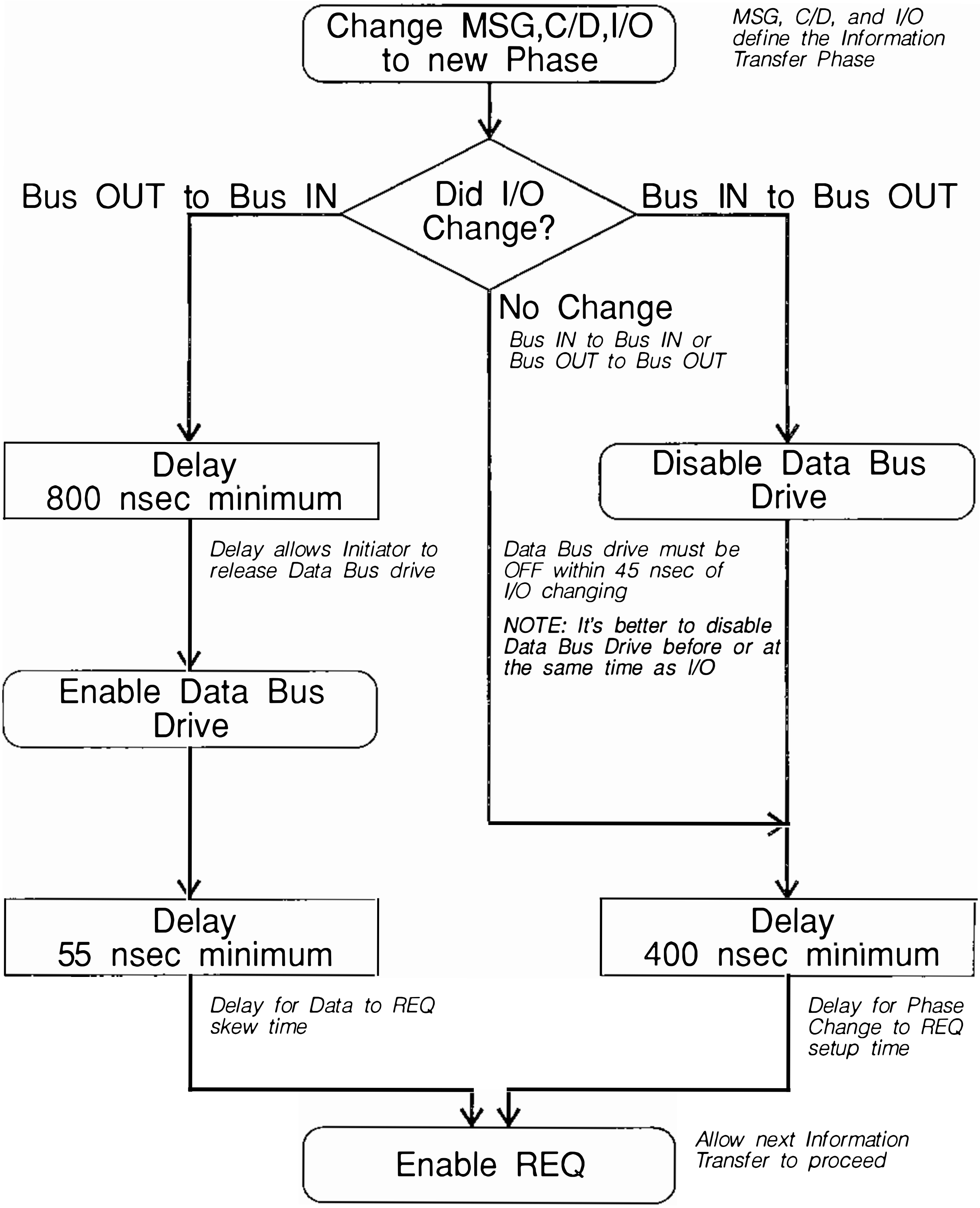

Change MSG,C/D,I/O to new Phase
MSG, C/D, and I/O define the Information Transfer Phase
Did I/O Change?
Bus OUT to Bus IN
Bus IN to Bus OUT
No Change
Bus IN to Bus IN or Bus OUT to Bus OUT
Delay 800 nsec minimum
Delay allows Initiator to release Data Bus drive
Enable Data Bus Drive
Disable Data Bus Drive
Data Bus drive must be OFF within 45 nsec of I/O changing
NOTE: It's better to disable Data Bus Drive before or at the same time as I/O
Delay 55 nsec minimum
Delay for Data to REQ skew time
Delay 400 nsec minimum
Delay for Phase Change to REQ setup time
Enable REQ
Allow next Information Transfer to proceed

DIAGRAM 7: BUS PHASE SWITCHING FOR TARGETS

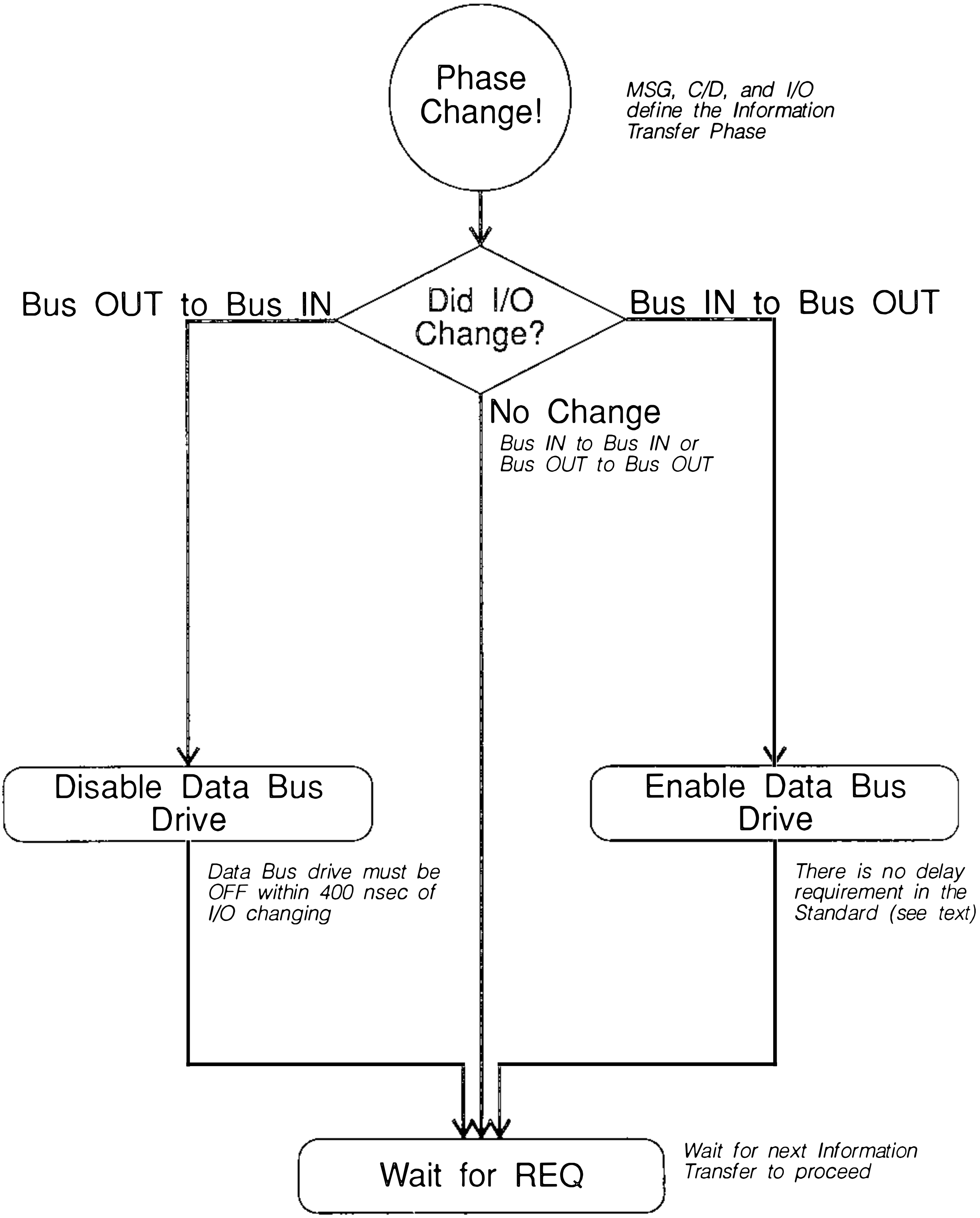

Phase Change!
MSG, C/D, and I/O define the Information Transfer Phase
Did I/O Change?
Bus OUT to Bus IN
Bus IN to Bus OUT
No Change
Bus IN to Bus IN or Bus OUT to Bus OUT
Disable Data Bus Drive
Data Bus drive must be OFF within 400 nsec of I/O changing
Enable Data Bus Drive
There is no delay requirement in the Standard (see text)
Wait for REQ
Wait for next Information Transfer to proceed

DIAGRAM 8: BUS PHASE SWITCHING FOR INITIATORS

The most interesting transitions between Information Transfer Phases are an "OUT" Phase to an "IN" Phase (such as MESSAGE OUT Phase to DATA IN Phase), and an "IN" Phase to an "OUT" Phase (such as MESSAGE IN Phase to DATA OUT Phase), and we will look at these first.

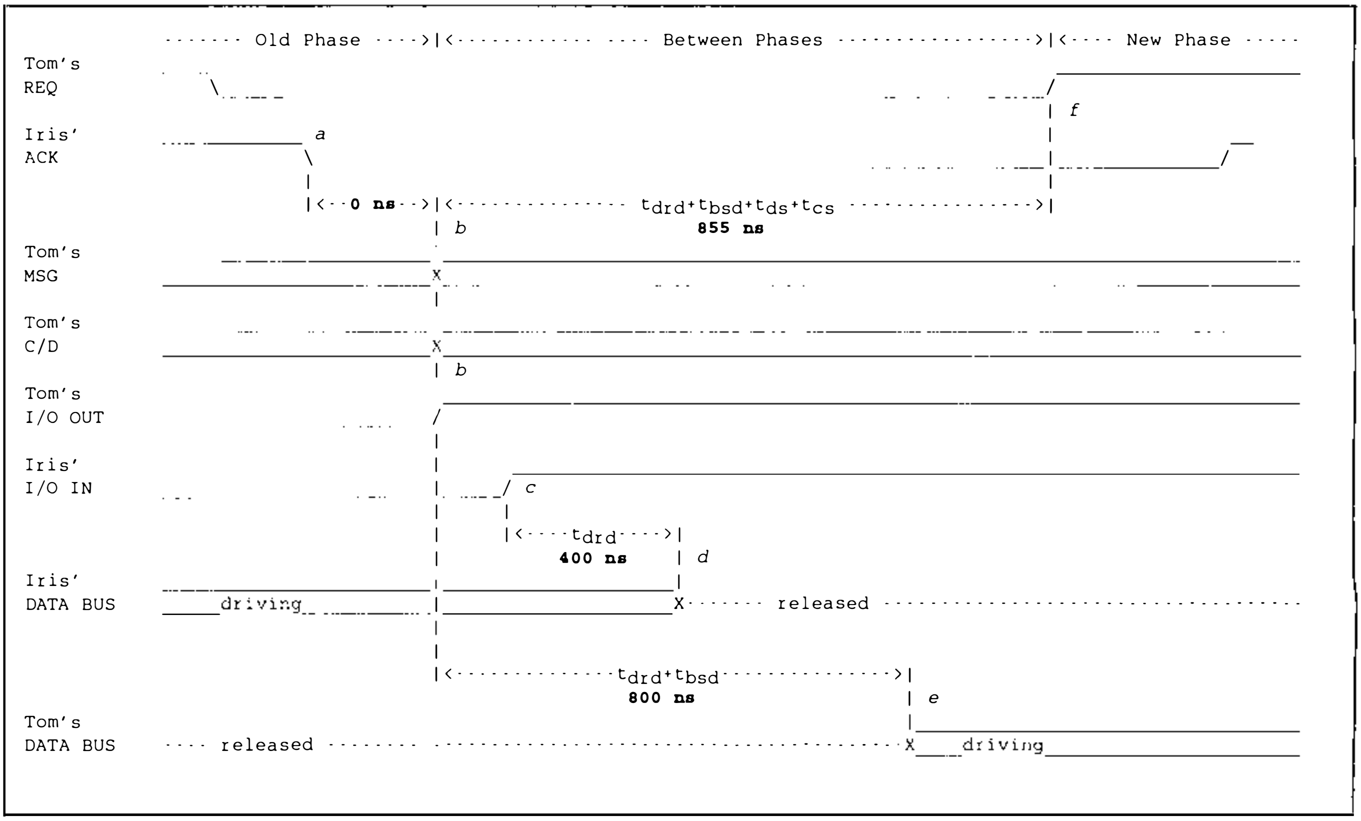


FIGURE 13: DATA TRANSFER OUT PHASE SWITCH TO DATA TRANSFER IN PHASE (INITIATOR DRIVING SWITCH TO TARGET DRIVING)

Here's how events proceed in Figure 13:

(a) Iris releases or negates ACK to end the last data transfer. Note that this transition of ACK can also be the falling edge of the last ACK pulse of a ***Synchronous Data Transfer***. This transition "ends" the Information Transfer Phase and now Tom may change the bus phase.

(b) Tom detects the transition of ACK to false, ending the last transfer of the phase. Tom may now (with no delay) change any of three bus phase signals (MSG, C/D, and I/O). In this case Tom asserts the I/O signal to the true state, causing the bus direction to change from OUT to IN. (Remember: Direction is relative to the Initiator, so the direction is now IN to Iris from Tom.)

(c) Iris detects the transition of I/O from false to true. As soon as I/O is true at Iris' connector, she has 400 nsec to stop driving the data bus signals. In practice, this should be easy for any implementation. Most of the time is used to let the I/O signal propagate and settle.

(d) As she is supposed to do, Iris releases her data bus drivers within 400 nsec of I/O going true. At this point, neither device is driving the data bus.

(e) After asserting the I/O signal true, Tom must delay 800 nsec before driving the data bus. This ensures that Tom and Iris do not drive the bus at the same time. (In practice, many Targets will not drive the data bus until the data for the new phase is available.)

(f) Tom asserts the first REQ of the new phase. This validates the new phase to Iris, who may then begin transferring data to the appropriate destination. Total time from I/O asserted true to REQ asserted is 855 nsec.

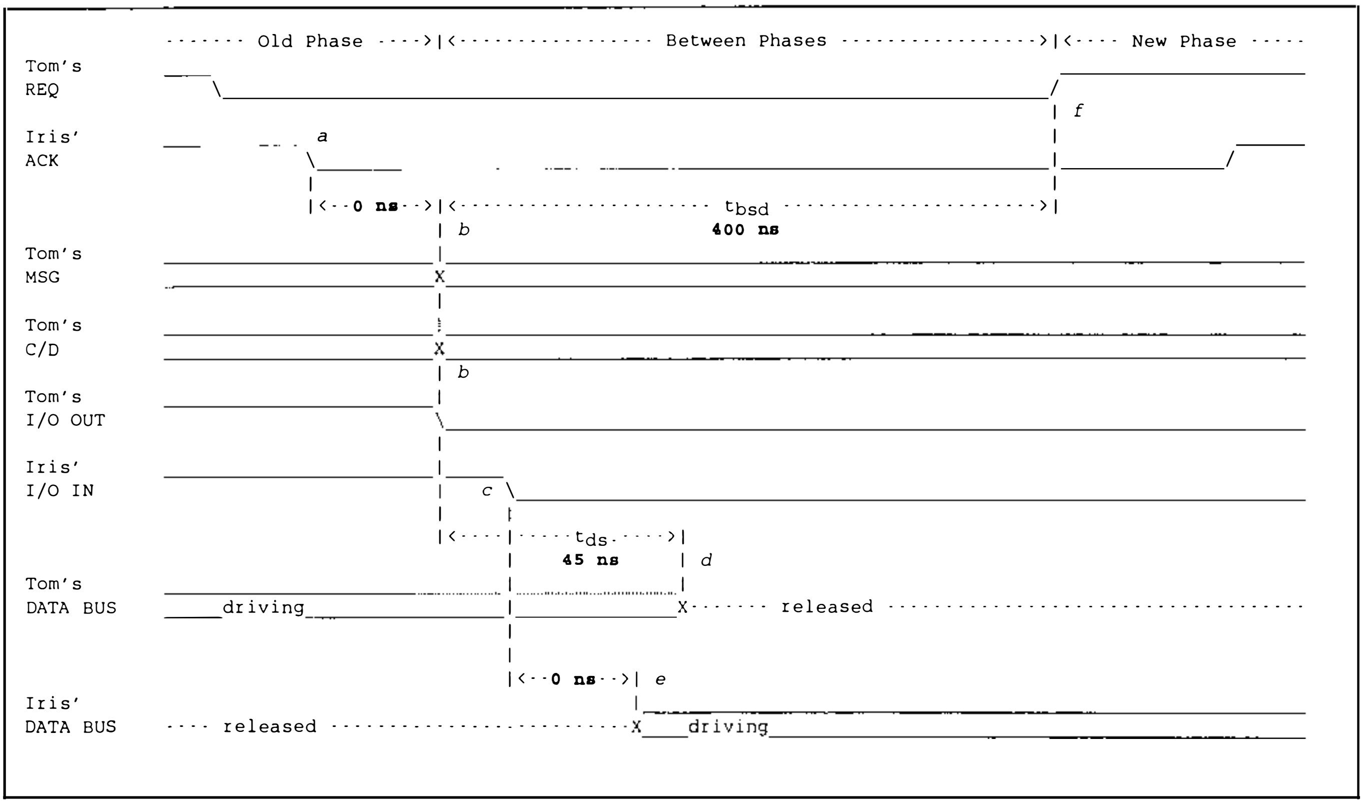


FIGURE 14: DATA TRANSFER IN PHASE SWITCH TO DATA TRANSFER OUT PHASE
(TARGET DRIVING SWITCH TO INITIATOR DRIVING)

Here's how events proceed in Figure 14:

(a) Iris releases or negates ACK to end the last data transfer. Note that this transition of ACK can also be the falling edge of the last ACK pulse of a ***Synchronous Data Transfer***. This transition "ends" the Information Transfer Phase and now Tom may change the bus phase.

(b) Tom detects the transition of ACK to false, ending the last transfer of the phase. Tom may now (with no delay) change any of three bus phase signals (MSG, C/D, and I/O). In this case Tom negates or releases the I/O signal to the false state, causing the bus direction to change from IN to OUT. (Remember: Direction is relative to the Initiator, so the direction is now OUT to Tom from Iris.)

(c) Iris detects the transition of I/O from true to false, indicating the bus direction change.

(d) After negating or releasing the I/O signal to the false state, Tom has 45 nsec to stop driving the data bus. In practice, Tom should release the data bus before setting I/O false to ensure this timing. The 45 nsec allows for variations in Tom's internal delays.

(e) After detecting the I/O signal false, Iris may immediately drive the data bus. This allows Iris to directly control bus drive with the I/O signal. Note that Tom and Iris might drive the bus at the same time. In practice, bus propagation delays and Iris' internal delays will cause this delay to be longer.

(f) Tom asserts the first REQ of the new phase. This validates the new phase to Iris, who may then begin transferring data to the appropriate destination. Total time from I/O negated or released false to REQ asserted is 400 nsec.

The following two figures show how events proceed when the bus direction does not change:

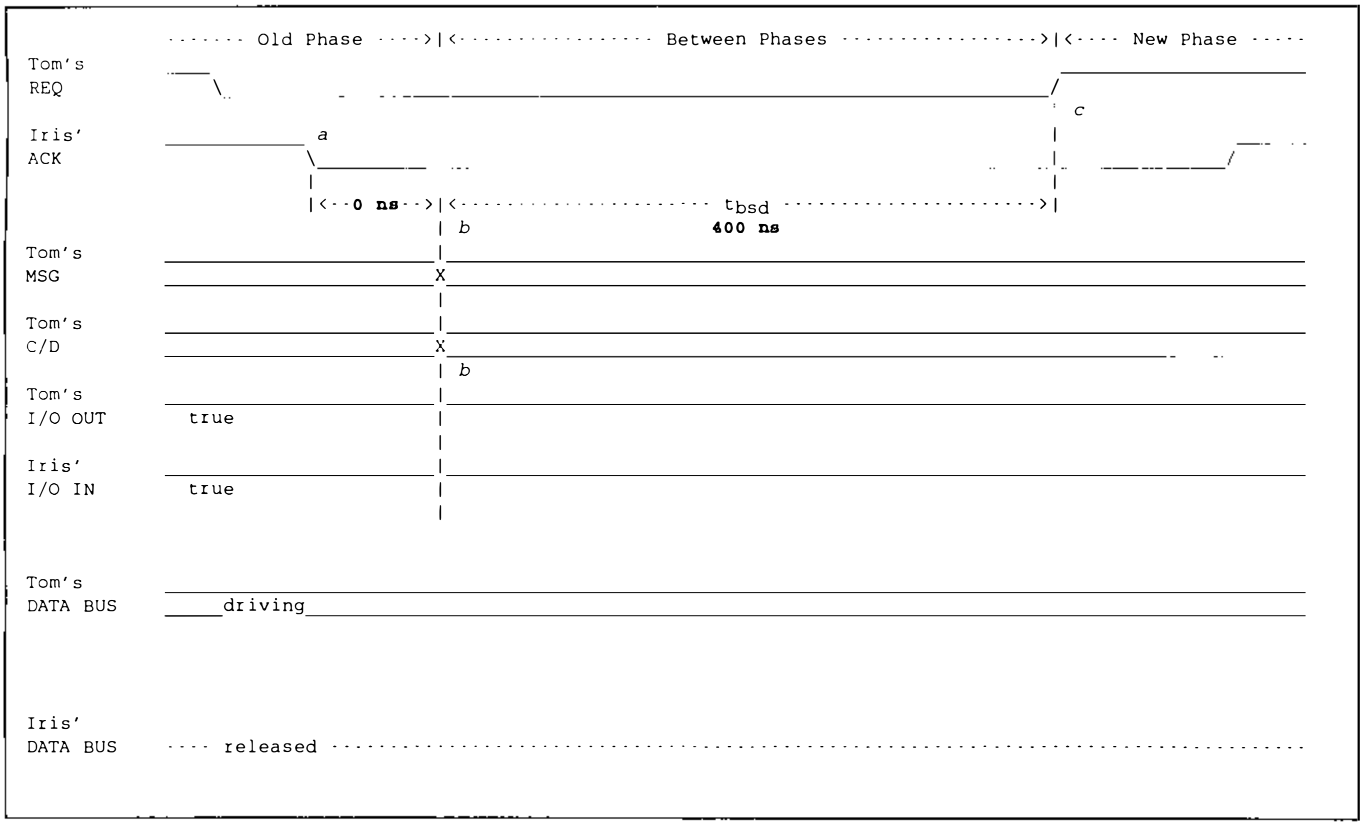


FIGURE 15: DATA TRANSFER IN PHASE SWITCH TO DATA TRANSFER IN PHASE (TARGET DRIVING CONSTANTLY)

Here's how events proceed in Figure 15:

(a) Iris releases or negates ACK to end the last data transfer. Note that this transition of ACK can also be the falling edge of the last ACK pulse of a ***Synchronous Data Transfer***. This transition "ends" the Information Transfer Phase and now Tom may change the bus phase.

(b) Tom detects the transition of ACK to false, ending the last transfer of the phase. Tom may now (with no minimum delay) change any of three bus phase signals (MSG, C/D, and I/O). In this case Tom does not change the state of the I/O signal, causing no change in the bus direction; the direction stays IN. Tom continues driving the Data Bus, and Iris does not drive it.

(c) Tom asserts the first REQ of the new phase. This validates the new phase to Iris, who may then begin transferring data to the appropriate destination. Total time from bus phase change to REQ asserted is 400 nsec.

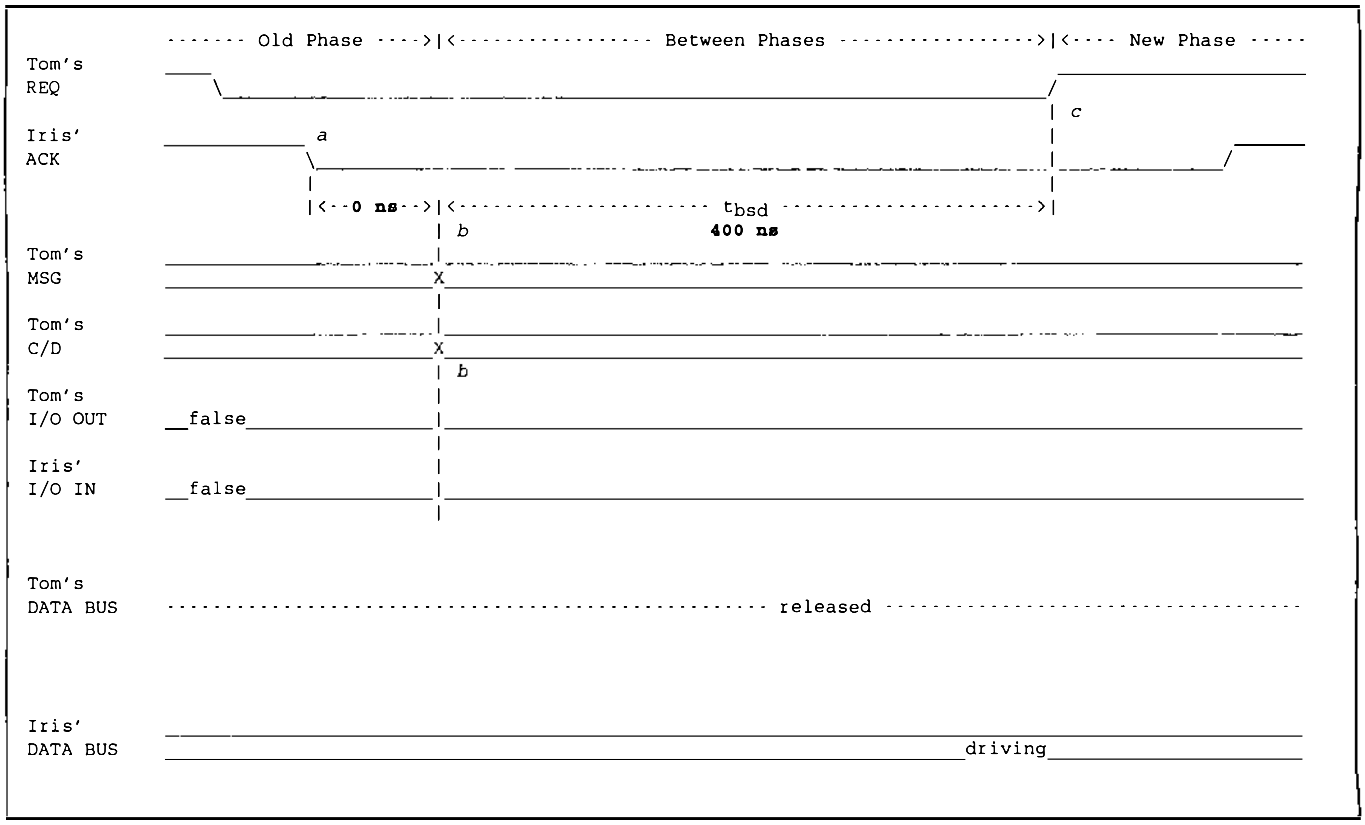


FIGURE 16: DATA TRANSFER OUT PHASE SWITCH TO DATA TRANSFER OUT PHASE (INITIATOR DRIVING CONSTANTLY)

Here's how events proceed in Figure 16:

(a) Iris releases or negates ACK to end the last data transfer. Note that this transition of ACK can also be the falling edge of the last ACK pulse of a ***Synchronous Data Transfer***. This transition "ends" the Information Transfer Phase and now Tom may change the bus phase.

(b) Tom detects the transition of ACK to false, ending the last transfer of the phase. Tom may now (with no minimum delay) change any of three bus phase signals (MSG, C/D, and I/O). In this case Tom does not change the state of the I/O signal, causing no change in the bus direction; the direction stays OUT. Iris continues driving the Data Bus, and Tom does not drive it.

(c) Tom asserts the first REQ of the new phase. This validates the new phase to Iris, who may then begin transferring data to the appropriate destination. Total time from bus phase change to REQ asserted is 400 nsec.

Table 7 shows the ***Bus Timing*** values used during Information Transfer Phase changes.

TABLE 7: TIMING VALUES USED BETWEEN PHASES

Symbol	Timing Name	MIN or MAX?	Time
t_{bsd}	Bus Settle Delay	MINIMUM	400 nsec
t_{ds}	Deskew Delay	MINIMUM	45 nsec
t_{cs}	Cable Skew Delay	MAX/MIN	10 nsec
t_{drd}	Data Release Delay	MAX/MIN	400 nsec

BSY Signal. The BSY signal is used for several different SCSI conditions and phases, and none of them have to do with the "busy" state of a device. Rather, the BSY signal indicates that the SCSI bus itself is busy; one or two ***SCSI Devices*** are using it now, call back later....

The BSY Signal is "OR-tied". No device may ***Negate*** the BSY signal. It may only be ***Asserted*** to the ***True*** state or ***Released*** to the ***False*** state. This allows more than one SCSI Device to assert the signal. See ***Signal Levels***.

BSY is used for the following purposes:

- When BSY is false, and SEL is also false, the current bus phase is ***BUS FREE Phase***.
- When the current bus phase is BUS FREE, and one or more SCSI devices assert BSY true, the ***ARBITRATION Phase*** begins.
- When SEL is true, and BSY goes false, the ***SELECTION Phase*** begins.
- As long as a Target holds BSY true, the bus is held for use by that Target to communicate with an Initiator. The only way any other device not involved in the ***Current I/O Process*** can take the bus away, as long as BSY is true, is to create the ***Reset Condition***.

Bus Clear Delay. *tbcd = 800 nsec.* The Bus Clear Delay is the time allowed a device to Get-Off-The-Bus after a change of state or phase on the bus. These changes are:

- A transition to the ***BUS FREE Phase*** is detected. Actually, the device has 1200 nsec total to clear off the bus (Bus Clear Delay plus Bus Settle Delay), but this includes the time required by the device to detect the BUS FREE Phase. If it takes the device longer than a Bus Settle Delay to detect the BUS FREE Phase, then only the remaining time is available for clearing off the bus.

- The device is arbitrating during an ***ARBITRATION Phase***, and another ***SCSI Device*** asserts the ***SEL Signal***.

- The ***RST Signal*** goes true.

BUS DEVICE RESET Message. The BUS DEVICE RESET Message is used by the Initiator to **completely** reset a Target. BUS DEVICE RESET is a single byte message:

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

After receiving the message, the Target begins executing its "***Hard Reset***" sequence; for most Targets, this is the same as a "power on" reset. Some other things to know about BUS DEVICE RESET:

- BUS DEVICE RESET may also be thought of as a "selective reset". You can cause the equivalent of a hardware reset of any individual SCSI Device without directly affecting any other device since you are not asserting RST. In practice, you will likely affect other Initiators that use that device, but other SCSI Devices are not affected. If the offending device is holding the bus, won't respond to the ***Attention Condition***, or cannot be selected, the ***Reset Condition*** is the only alternative.

- Don't go using BUS DEVICE RESET for every little thing! BUS DEVICE RESET is the second-to-last resort for when a device is severely misbehaving (last resort is RST). Don't use it for "grabbing" the Target away from other initiators; there are better ways that won't mess up system performance. Exception: BUS DEVICE RESET is appropriate for when an Initiator has failed with outstanding reservations or other processes active on the Target. In that case, the Target could be reset to release it from the failed Initiator.

- After a BUS DEVICE RESET, the Target **must** go to the ***Hard Reset*** state, even if it would normally perform a ***Soft Reset*** in response to the RST signal. This ensures that there is always a way to cause Hard Reset in SCSI Devices that implement Soft Reset. The Target creates all conditions that should occur after a Hard Reset (e.g., it creates the ***Unit Attention Condition***).

- After receiving the BUS DEVICE RESET message, the Target goes to ***BUS FREE Phase***. No ***Status*** is sent; no other phase occurs before BUS FREE. There is no time specified between the "receipt" of the message (presumably the trailing edge of ACK) and when BUS FREE occurs. A Target may hold the bus until it has done what it has to do to prepare to reset.

- The Initiator sends this message by asserting the ***ATN signal*** (creating the ***Attention Condition*** on the bus) and waiting for the Target to get around to responding. If the Target is disconnected from the Initiator, the Initiator is allowed to select the Target for the purpose of sending this message, even if a command is currently active. The Initiator need not send an ***IDENTIFY Message*** prior to sending BUS DEVICE RESET; it doesn't matter either way.

Summary of Use: The BUS DEVICE RESET message is sent only by an Initiator to cause the Target to be completely cleared to the "power-on" condition.

Bus Free Delay. *tbfd = 800 nsec*. The **minimum** time between detecting ***BUS FREE Phase*** and asserting BSY to begin ***ARBITRATION Phase***. The maximum time is the ***Bus Set Delay***. See ARBITRATION Phase.

BUS FREE Phase. The BUS FREE Phase describes the state of the ***SCSI Bus*** when no devices are using the bus. We can say that the bus is in an idle state. All signals are in the ***False*** (***Released***) state. When the bus is free, any device may enter the ***ARBITRATION Phase*** and begin using the bus to talk to another device.

The BUS FREE Phase is entered (under normal operating situations) when the ***BSY Signal***, ***SEL Signal***, and ***RST Signal*** become CONTINUOUSLY false for a ***Bus Settle Delay*** (400 nsec). Soon after this occurs (a ***Bus Clear Delay*** of 800 nsec) all other bus signals are expected to be released, if they are not already released.

In the following example, the BUS FREE phase is entered when the Target (Tom) releases the BSY signal while the Initiator (Iris) has ATN asserted. This can occur for several reasons which are enumerated in the notes that follow. After Tom releases BSY, the signal must settle out before BUS FREE Phase can be fully validated. A device should implement a "validation timer" (to ignore a ***Wire-OR Glitch***) which begins counting up to 400 nsec whenever BSY and SEL are both false, and resets whenever either signal goes true.

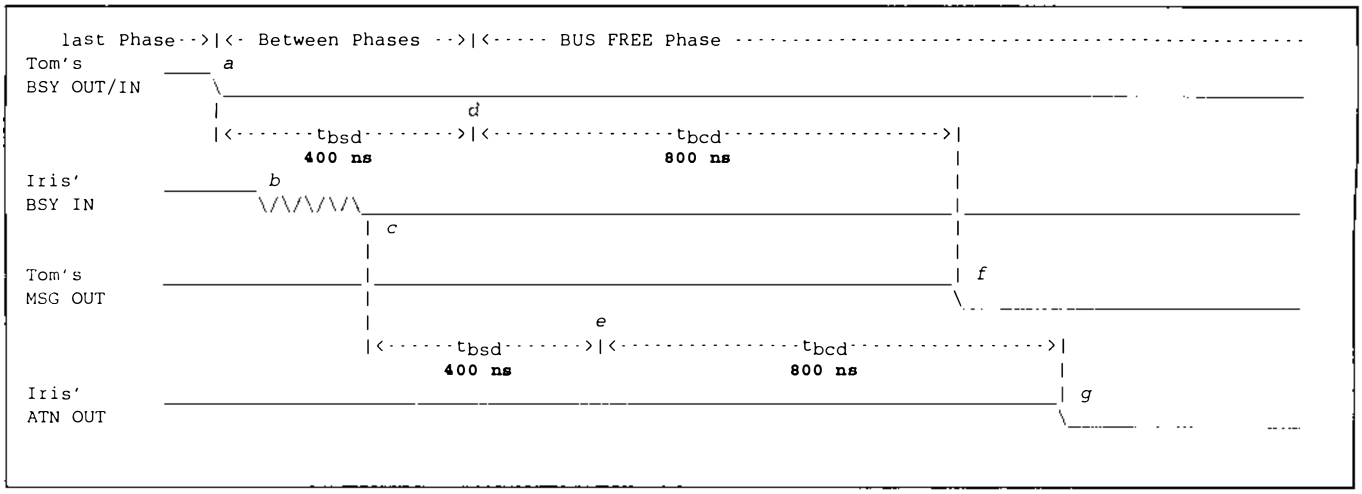

last Phase-->|<- Between Phases -->|<----- BUS FREE Phase
Tom's BSY OUT/IN
a
d
|<--------t_{bsd}-------->|<------------------t_{bcd}------------------>|
400 ns
800 ns
Iris' BSY IN
b
c
Tom's MSG OUT
f
e
|<-------t_{bsd}------->|<------------------t_{bcd}------------------>|
400 ns
800 ns
Iris' ATN OUT
g

FIGURE 17: BUS FREE DUE TO BSY SIGNAL RELEASE

Here's how events proceed in Figure 17:

(a) Tom releases BSY to begin the transition to BUS FREE phase. Tom's BUS FREE validation timer begins here since Tom was the device that negated BSY, and because the BSY signal happens to make a clean transition at Tom's input.

(b) Iris first "sees" BSY beginning to change here, but let's assume that due to some arcane laws of ***Cable*** physics, Iris' signal has reflections and glitches in it (see also ***Termination***). Since BSY is not **continuously** false at this time, the Iris' BUS FREE validation timer does not start yet; it is being reset by the glitches.

(c) Iris' BSY input has now settled and the validation timer begins counting its way up to 400 nsec.

(d) Tom's validation timer has reached 400 nsec. At this point the BUS FREE phase is validated for Tom. He now has 800 nsec to get all other signals cleared off the bus.

(e) Iris' validation timer has reached 400 nsec. At this point the BUS FREE phase is also validated for Iris. She now has 800 nsec to get all other signals cleared off the bus.

(f) Tom's time has come to clear all other signals off the bus. As a Target, he must remove BSY, MSG, C/D, I/O, REQ, and the DB0-n plus parity signals if he is asserting them. ACK and ATN are asserted only by Initiators. RST is not involved in this, and BSY and SEL are released by whichever device is asserting them at the time BUS FREE occurs.

Important note: A Target may actually clear all of its other signals off the bus prior to releasing BSY when it knows that it will be going to BUS FREE phase. This eliminates the need to meet the 800 nsec requirement in this case. Other Targets will let their regular BUS FREE detection circuit release all signals. In either case a Target must be able to clear off in the time allotted in cases where the Initiator causes the BUS FREE phase (e.g., during ***SELECTION Phase***).

(g) Now Iris' time has come to clear all of her signals off the bus. As an Initiator, she must remove ACK, ATN, and DB0-n plus parity signals if she is asserting them. REQ, MSG, C/D, and I/O are asserted only by Targets. RST is not involved in this, and BSY and SEL are released by whichever device is asserting them at the time BUS FREE occurs.

Some other notes about BUS FREE Phase:

- BUS FREE Phase is also the "initial" state of the bus; all devices are powered on but no Selections have occurred yet.
- BUS FREE Phase can be entered AFTER the RST signal is asserted and then negated. Note that when RST is asserted we are in the ***RESET Condition*** and not BUS FREE Phase, even though BSY is negated during this time.
- The Target must release BSY and enter BUS FREE phase after it completes sending any of the following messages:
    - ***DISCONNECT Message***
    - ***COMMAND COMPLETE Message***
- The Target must also release BSY and enter BUS FREE phase after it receives any of the following messages:
    - ***ABORT Message***
    - ***ABORT TAG Message***
    - ***BUS DEVICE RESET Message***
    - ***CLEAR QUEUE Message***
    - ***RELEASE RECOVERY Message***

- The Target may also release BSY to cause a BUS FREE phase as an error reporting mechanism. Normally, BSY remains continuously asserted by the Target:

  - Beginning with when BSY is asserted in response to the SELECTION Phase, or when BSY is asserted at the end of the RESELECTION Phase;
  - Ending with one of the conditions listed in (2), (3), and (4) above.

  If the Target releases BSY any time between these two time points, the Initiator must then assume that a catastrophic failure has occurred on the Target. After this occurs the Target may or may not be able to accept Selection, and may or may not have any ***Sense Data*** to describe the failure (in general, it is worth trying to get the sense data anyway). Some examples of what can cause BSY to be released include:

  - Power failure on the Target
  - Circuit failure internal to the Target
  - Transfer retry failed to resolve bus parity errors (see ***RESTORE POINTERS***)

- Either Initiators or Targets may also enter BUS FREE phase by releasing the SEL signal as a result of a timeout of the ***SELECTION*** or ***RESELECTION*** Phase. The timing for the release of SEL is the same as that for BSY as shown in Figure 17.

Table 9 shows the ***Bus Timing*** values used during BUS FREE Phase.

TABLE 9: TIMING VALUES USED DURING BUS FREE PHASE

Symbol	Timing Name	MIN or MAX?	Time
t_{bsd}	Bus Settle Delay	MINIMUM	400 nsec
t_{bcd}	Bus Clear Delay	MAXIMUM	800 nsec

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

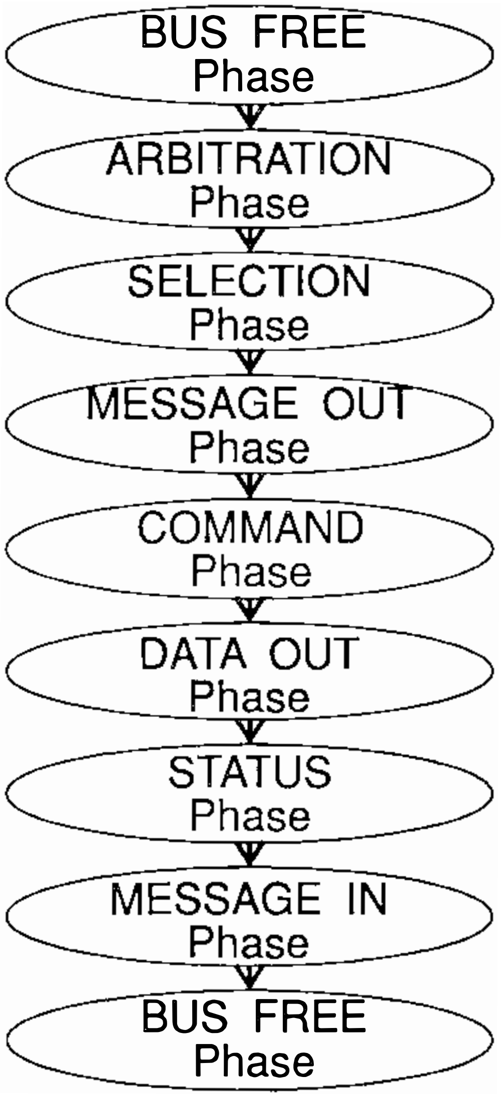
Bus Phases. All activity on the ***SCSI Bus*** takes place within Bus Phases. The dictionary definition of phase is "any stage in a series or cycle of changes". In SCSI, each Bus Phase is one stage of a ***Connection***, and each type of Bus Phase is designed to exchange different types of information between Initiators and Targets. Another way to look at it is that a Bus Phase is a way of "bundling" different types of information; the bus signals that define the phase define the type of information bundle, just like a "header" defines a "packet" type in a communication system.

There are two types of phases: one type is what we will call a Connection Phase, which is any phase that takes part in establishing and holding a Connection; and the other type is what the SCSI Standard calls an ***Information Transfer Phase***. We can define the phases as follows:

- Connection Phases:
  - ***BUS FREE Phase***: Bus idle; i.e., the absence of a Connection.
  - ***ARBITRATION Phase***: A ***SCSI Device*** requesting the use of the bus.
  - ***SELECTION Phase***: A device informs another device that the start of a transaction between the two devices is requested.
  - ***RESELECTION Phase***: A device informs another device that a continuation of a transaction between the two devices is requested.

- Information Transfer Phases:
  - ***COMMAND Phase***: A command request "bundle" from the Initiator to the Target.
  - ***STATUS Phase***: A completion report from the Target to the Initiator.
  - ***DATA OUT Phase***: A block of data or parameters from the Initiator to the Target.
  - ***DATA IN Phase***: A block of data or parameters from the Target to the Initiator.
  - ***MESSAGE OUT Phase***: A ***Path Control*** request or response (see ***Message System***) from the Initiator to the Target.
  - ***MESSAGE IN Phase***: A Path Control request or response from the Target to the Initiator.

Diagram 9 shows a simple series of phases that results in data blocks being written to the Target. In the diagram, the Initiator connects to the Target, establishes (through Path Control) the ***Nexus***, issues a command, sends the data, and gets a report on the result of the operation. The Target then breaks the path and the operation is completed.



Establish a Connection:

When the bus is free, the Initiator Arbitrates to take possesion of the bus

The Initiator Selects the Target

They complete the Nexus with the IDENTIFY message

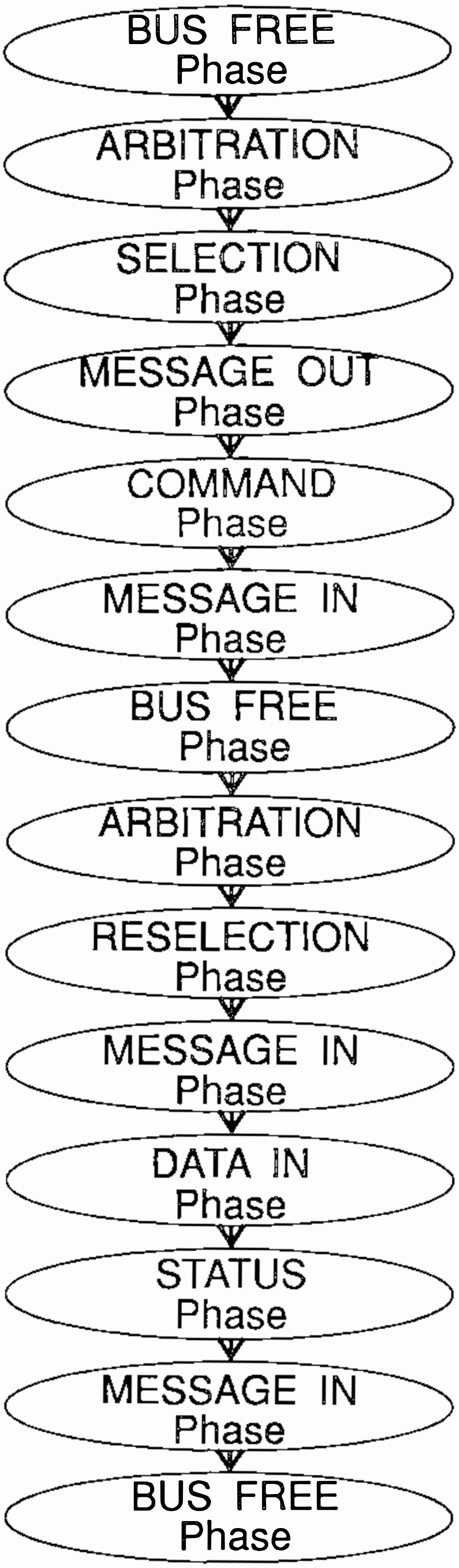
The Initiator sends the Command

Data is transferred from the Initiator to the Target

The Target sends command completion Status

The Target sends the COMMAND COMPLETE message to end the command

DIAGRAM 9: SIMPLE PHASE FLOW



Establish a Connection:

When the bus is free, the Initiator Arbitrates to take possesion of the bus

The Initiator Selects the Target

They complete the Nexus with the IDENTIFY message

The Initiator sends the Command

The Target wants to break the Connection for now and sends the DISCONNECT message

With the bus free, the Target arbitrates for the bus to Reconnect to the Initiator

The Target Reselects the Initiator

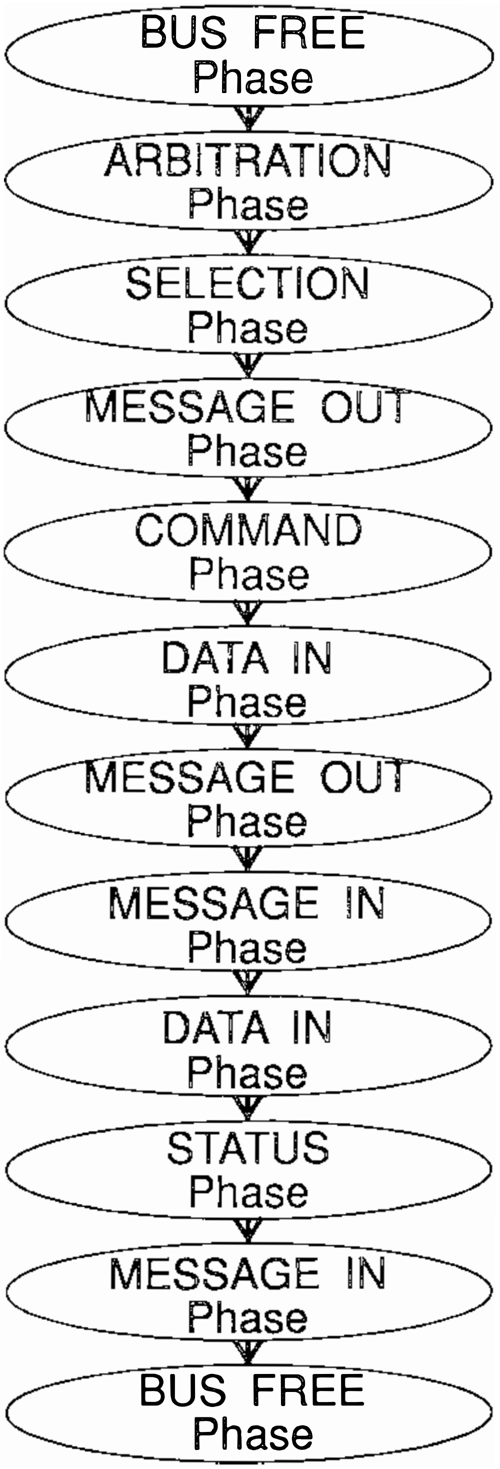
They re-establish the Nexus with the IDENTIFY message

Data is transferred from the Target to the Initiator

The Target sends command completion Status

The Target sends the COMMAND COMPLETE message to end the command

DIAGRAM 10: COMPLEX PHASE FLOW



Establish a Connection:

When the bus is free, the Initiator Arbitrates to take possesion of the bus

The Initiator Selects the Target

They complete the Nexus with the IDENTIFY message

The Initiator sends the Command

Data is transferred from the Target to the Initiator

The Initiator sends an INITIATOR DETECTED ERROR message when it detects a Parity error

The Target sends the RESTORE POINTERS message to retry the data transfer

The data transfer from the Target to the Initiator is retried

The Target sends command completion Status

The Target sends the COMMAND COMPLETE message to end the command

DIAGRAM 11: ERROR RECOVERY PHASE FLOW

Diagram 10 shows the flow for a more complex sequence. Path Control, via the ***Message System***, was used by the Target to ***Disconnect*** from the Initiator. The Target disconnected because the command request from Initiator could not be acted upon immediately (more on this later). This releases the bus so that other devices may use the bus.

Diagram 11 shows a series of phases which includes error recovery. If an error occurs during an Information Transfer phase, the Initiator and Target use Path Control to retry the phase. In this case, a data transfer in was retried. Again, see ***Message System*** for more details on Path Control and error recovery.

Who Controls the Phases? A Bus Phase is defined by the current state of the bus signals. These signals are controlled by the SCSI devices currently connected to the bus. Each device has signals which it is responsible for, depending on the role the device has taken (Initiator or Target). The Target controls all of the signals that define the Information Transfer Phases. The Connection Phases are controlled by the device that initiated the series of phases:

- BUS FREE is the phase if no device has a signal asserted. Therefore, BUS FREE is really the "absence of control" on the bus.
- ARBITRATION is controlled by the Arbitration Winner.
- SELECTION is controlled by the Initiator following an Arbitration Win.
- RESELECTION is controlled by the Target following an Arbitration Win.

What Signals Control the Phases? The Connection Phases are controlled by BSY, SEL, and I/O. The Information Transfer Phases are controlled by BSY, MSG, C/D, I/O, REQ, and ACK. Table 10 shows these signals and the phases they define.

Note that RST and ATN do not define phases; rather, they define conditions. They may also "force" a phase to occur. See ***Reset Condition*** and ***Attention Condition***.

See ***Between Phases*** for details on how to transition between Information Transfer Phases.

Other notes on the table:

- ARBITRATION as shown is identical to "Between Information Transfer Phases", but the former only occurs after BUS FREE Phase, and the latter only occurs after SELECTION Phase or RESELECTION Phase has completed.

- Reserved Phase (In) and Reserved Phase (Out) are not yet defined by any SCSI Standard. Do not use them! Uses may be found for these phases in ***SCSI-3***.

- The Invalid Phases are all associated with asserting the SEL signal while asserting REQ or ACK. After SELECTION Phase or RESELECTION Phase, the SEL signal must not be asserted during the remainder of the Connection.

TABLE 10: BUS PHASE DEFINITIONS

BSY	SEL	MSG	C/D	I/O	REQ	ACK	Phase
0	0	X	X	X	X	X	BUS FREE
1	0	0	0	0	0	0	ARBITRATION (after BUS FREE only)
0	1	X	X	0	0	0	SELECTION
1	1	X	X	0	0	0	
0	1	X	X	1	0	0	RESELECTION
1	1	X	X	1	0	0	
1	0	0	0	0	1	0	DATA OUT
1	0	0	0	0	1	1	
1	0	0	0	0	0	1	
1	0	0	0	0	0	0	
1	0	0	0	1	1	0	DATA IN
1	0	0	0	1	1	1	
1	0	0	0	1	0	1	
1	0	0	0	1	0	0	
1	0	0	1	0	1	0	COMMAND
1	0	0	1	0	1	1	
1	0	0	1	0	0	1	
1	0	0	1	0	0	0	
1	0	0	1	1	1	0	STATUS
1	0	0	1	1	1	1	
1	0	0	1	1	0	1	
1	0	0	1	1	0	0	
1	0	1	0	0	1	0	Reserved Phase (Out)
1	0	1	0	0	1	1	(DO NOT USE!)
1	0	1	0	0	0	1	
1	0	1	0	0	0	0	
1	0	1	0	1	1	0	Reserved Phase (In)
1	0	1	0	1	1	1	(DO NOT USE!)
1	0	1	0	1	0	1	
1	0	1	0	1	0	0	
1	0	1	1	0	1	0	MESSAGE OUT
1	0	1	1	0	1	1	
1	0	1	1	0	0	1	
1	0	1	1	0	0	0	
1	0	1	1	1	1	0	MESSAGE IN
1	0	1	1	1	1	1	
1	0	1	1	1	0	1	
1	0	1	1	1	0	0	
1	0	X	X	X	0	0	Between Information Transfer Phases, if a phase change is about to occur.
0	1	X	X	X	1	X	Invalid Phases!
0	1	X	X	X	X	1	(after SELECTION Phase, see text)
1	1	X	X	X	1	X	
1	1	X	X	X	X	1	

The flow diagrams on the following two pages show the possible flows of all of these phases. Diagram 12 shows the Connection Phases. Some observations from this diagram:

- SELECTION and RESELECTION normally proceed to an Information Transfer Phase. If the SCSI Device to be Selected or Reselected does not respond, a timeout occurs and the bus returns to BUS FREE.

- SELECTION is always followed by a MESSAGE OUT Phase (see ***Connection***). Some older SCSI-1 and pre-SCSI devices will skip MESSAGE OUT (they don't even support it) and go straight to COMMAND Phase.

- RESELECTION is usually followed by a MESSAGE IN Phase (see ***Connection***).

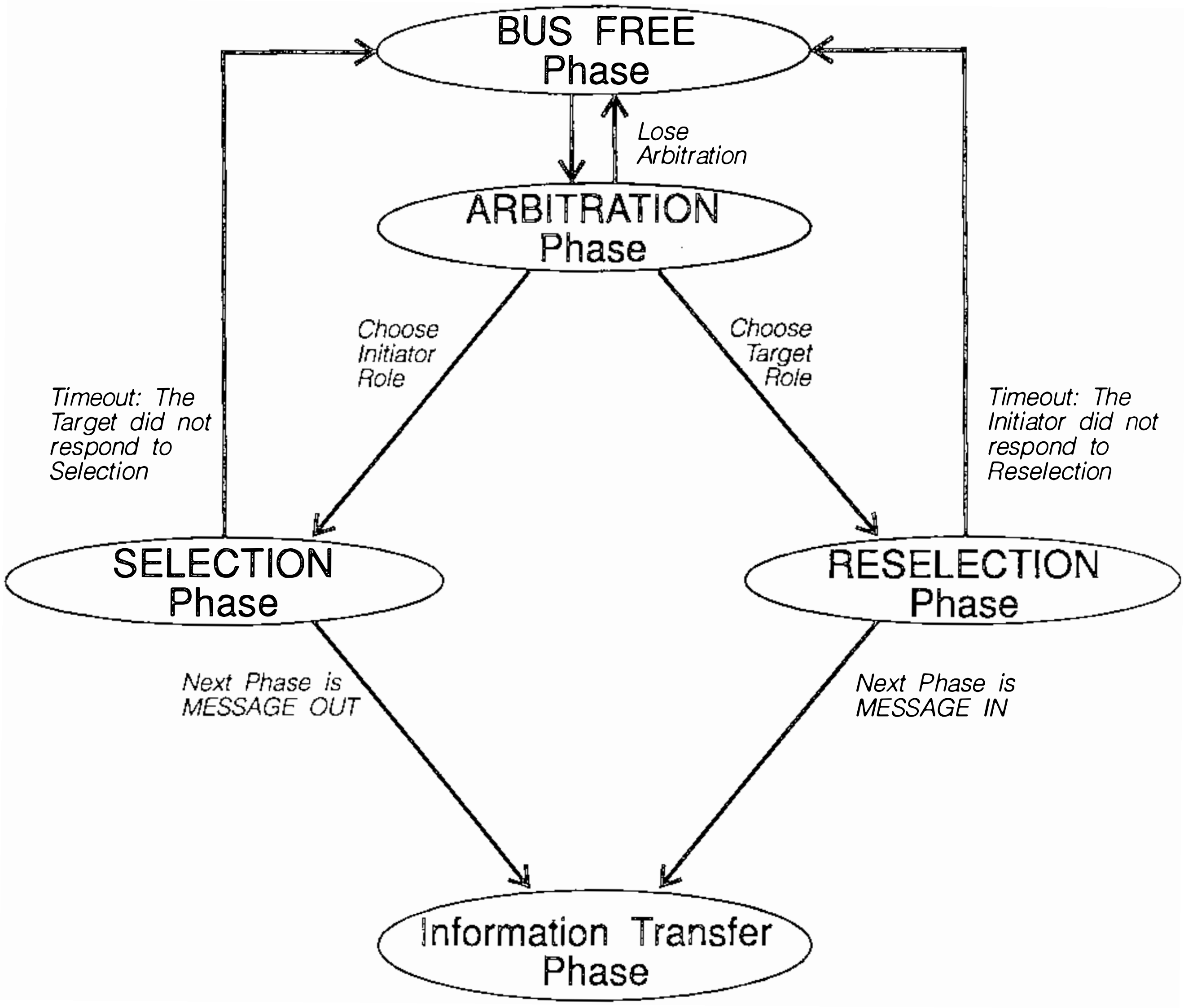

BUS FREE Phase
Lose Arbitration
ARBITRATION Phase
Choose Initiator Role
Choose Target Role
Timeout: The Target did not respond to Selection
Timeout: The Initiator did not respond to Reselection
SELECTION Phase
RESELECTION Phase
Next Phase is MESSAGE OUT
Next Phase is MESSAGE IN
Information Transfer Phase

DIAGRAM 12: CONNECTION PHASE TRANSITIONS

Diagram 13 shows the valid transitions between Information Transfer Phases. Some more observations from the diagram:

- The last two items in the observations for Connection Phases also apply here.

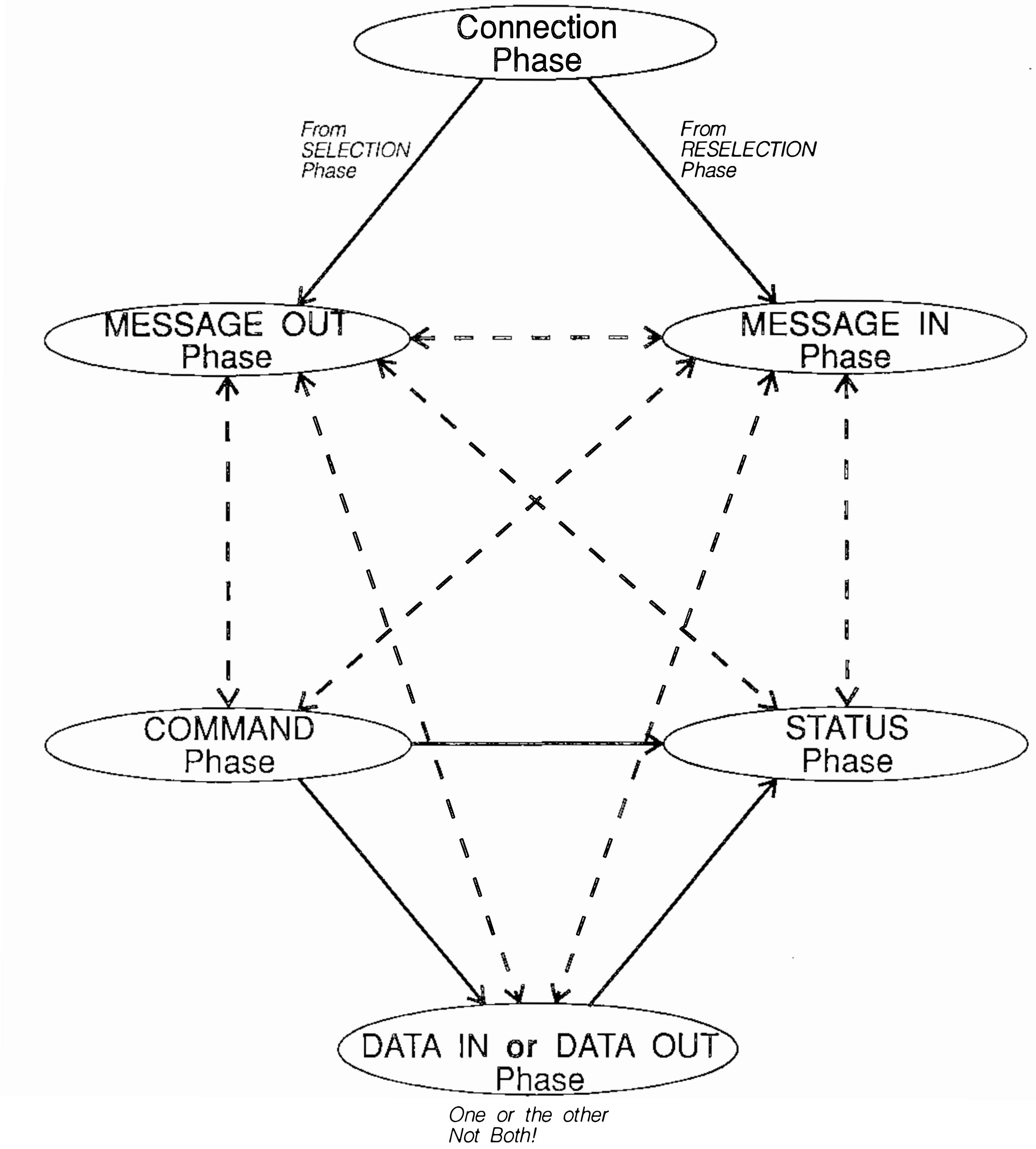
- The "most normal" sequence goes around the circle counterclockwise, starting with the Connection Phase, and going to MESSAGE OUT Phase. The sequence continues around COMMAND, DATA IN or OUT, and STATUS, finally ending up at MESSAGE IN to send the ***COMMAND COMPLETE Message***.

- COMMAND, DATA IN or DATA OUT, and STATUS may change to MESSAGE IN or MESSAGE OUT for purposes of Path Control. After the MESSAGE phase, there may be a return to the same phase or perhaps a different phase. See ***Message System*** for a complete list of phases that follow messages.

- Any Information Transfer Phase may be followed by BUS FREE; that is the only possible transition to a Connection Phase. MESSAGE IN and MESSAGE OUT are the most likely phases to see this transition (see ***BUS FREE Phase*** for a list of messages that lead to BUS FREE Phase). A transition to BUS FREE in any other case indicates a major failure at the Target.

- During a single command execution (see ***I/O Process***), there may be only a DATA IN Phase or a DATA OUT Phase. Even though this is not explicitly stated anywhere in the standard, it is implicit in the ***Pointer*** model (separate DATA IN and DATA OUT Pointers would be required, unless one is very careful). So, in general, DATA IN Phase and DATA OUT Phase may not occur in the same command!

See ***Examples*** (at the end of the Encyclopedia) and ***Message System*** for illustrations of phase transitions. We recommend looking at these flow diagrams while going over the examples.



NOTE: Any Information Transfer Phase can be followed by a BUS FREE Phase

DIAGRAM 13: INFORMATION TRANSFER PHASE TRANSITIONS

Bus Phase Signals. The Bus Phase Signals are the three signals that define a type of ***Information Transfer Phase***: the ***MSG Signal***, the ***C/D Signal***, and the ***I/O Signal***. See ***Bus Phases*** for a complete table of bus phases defined by these signals. See ***Between Phases*** for a description of how these signals may change to cause a phase change.

Bus Set Delay. *tbset = 1800 nsec*. The **maximum** time between detecting ***BUS FREE Phase*** and asserting BSY to begin ***ARBITRATION Phase***. The minimum time is the ***Bus Free Delay***. See ARBITRATION Phase.

Bus Settle Delay. *tbsd = 400 nsec*. The Bus Settle Delay is the time allowed from the change of a bus signal to the change of some other bus signal, or to the detection of a bus state. The length of the delay is derived from the total round-trip propagation delay of the longest possible ***Cable***. Examples include:

- The time between the change of the bus phase signals (***MSG, C/D, I/O Signals***) and the assertion of the ***REQ Signal***.

- The time between the transition of both ***BSY*** and ***SEL*** to false and the detection of ***BUS FREE Phase***.

Bus Timing. Table 11 shows all of the bus timing values named by the SCSI Standard. Each named value is described in this encyclopedia in its own subject entry.

TABLE 11: BUS TIMING VALUES

Timing Value	Symbol	Duration
Arbitration Delay	t_{arbd}	2400 nanoseconds MINIMUM
Assertion Period	t_{ast}	90 nanoseconds MINIMUM
Bus Clear Delay	t_{bcd}	800 nanoseconds MAXIMUM
Bus Free Delay	t_{bfd}	400 nanoseconds MINIMUM
Bus Set Delay	t_{bset}	1800 nanoseconds MAXIMUM
Bus Settle Delay	t_{bsd}	400 nanoseconds MINIMUM
Cable Skew Delay	t_{csd}	10 nanoseconds MINIMUM
Data Release Delay	t_{drd}	400 nanoseconds MINIMUM
Deskew Delay	t_{ds}	45 nanoseconds MINIMUM
Disconnection Delay	t_{dd}	200 microseconds MINIMUM
Fast Assertion Period	t_{fast}	30 nanoseconds MINIMUM
Fast Cable Skew Delay	t_{fcsd}	5 nanoseconds MINIMUM
Fast Deskew Delay	t_{fds}	20 nanoseconds MINIMUM
Fast Hold Time	t_{fht}	10 nanoseconds MINIMUM
Fast Negation Period	t_{fnp}	30 nanoseconds MINIMUM
Hold Time	t_{ht}	45 nanoseconds MINIMUM
Negation Period	t_{np}	90 nanoseconds MINIMUM
Power-On to Selection Time	t_{post}	10 seconds MAXIMUM (recommended)
Reset to Selection Time	t_{rst}	250 milliseconds MAXIMUM (recommended)
Reset Hold Time	t_{rht}	25 microseconds MINIMUM
Selection Abort Time	t_{sat}	200 microseconds MINIMUM
Selection Time-out Delay	t_{std}	250 milliseconds MAXIMUM (recommended)
Transfer Period	t_{tp}	set during **Synchronous Data Transfer Negotiation**

This page is nearly blank!

C/D Signal. The C/D signal is driven by a Target to control whether "data" or "control" information is transferred over the bus. In fact "C/D" stands for "Control/Data". This signal is one of the three ***Bus Phase Signals*** (MSG, C/D, and I/O) (see also ***Bus Phases***).

- When this signal is false (***Negated*** or ***Released***), the bus contains "data", which is simply what might be called "user data" or "host data"; i.e., data to or from a data block in the Target. The data may also be "parameter data" C/D is false during ***SELECTION Phase***, ***RESELECTION Phase***, ***DATA OUT Phase***, and ***DATA IN Phase***.

- When this signal is true (***Asserted***), the bus contains "control", which is anything that is "not data". This includes command blocks, status, and messages. C/D is true during ***MESSAGE IN Phase***, ***MESSAGE OUT Phase***, ***STATUS Phase***, and ***COMMAND Phase***.

Cables. For SCSI, a Cable is a set of N conductors that transmits the ***SCSI Bus*** signals between two or more ***SCSI Devices*** along a common path. A cable will have two or more ***Connectors*** installed on it to attach the Cable to the Devices.

Note that SCSI-2 does not define which conductor is used for which signal; it only defines the Connector pinout. In general, what happens between the Connectors of a cable is the domain of the cable manufacturer and/or the system integrator, however, some care is needed to reduce crosstalk between some signals, as will be discussed below.

SCSI-1 defines a 50-conductor/25 signal pair cable for both single ended and differential options. With the coming of ***SCSI-2*** and ***SCSI-3***, other cables have been defined so that ***Wide Data Transfer*** can be implemented. All three cables have the same characteristics, only the number of conductors changes.

- The original 50-conductor has been labeled the ***A Cable*** to differentiate it from the other two cables. The A Cable supports 8-bit transfers only.

- In SCSI-2, the 68-conductor ***B Cable*** was defined. The B Cable permits 16-bit or 32-bit Wide Data Transfers as an additional cable used in conjunction with the A Cable.

- The 68-conductor ***P Cable*** has been defined in SCSI-3. The P Cable permits 8-bit or 16-bit transfers as a single cable; the P Cable replaces the A Cable for 16-bit implementations. A "Q Cable" may also be defined as a 32-bit extension for the P Cable.

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 full B-cable and connector. 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. See the section on the P Cable for some additional details and caveats.

Cable Characteristics. Past wisdom has held that the single ended cable can be "cheaper", while the differential cable is more expensive. Recent studies have indicated otherwise. With today's higher performance single ended devices, good quality cable must be used (beware of discount specials!). A good quality cable for either single ended or differential use should have the characteristics shown in Table 12. Note that the table shows our recommendations, which either meet or exceed what is called for by the SCSI-2 Standard. But don't stop with the table, because following the table on the next page is a discussion of what we really mean.

TABLE 12: CABLE CHARACTERISTICS

Parameter	Single Ended	Differential
Maximum Total Length	6 meters (20 feet)	25 meters (85 feet)
Maximum Stub Length	10 cm (4 inches)	20 cm (8 inches)
Minimum Stub Spacing	30 cm (12 inches)	30 cm (12 inches)
Characteristic Impedance	75 ohms to 132 ohms (*)	90 ohms to 132 ohms
Cable Type	Flat ribbon or twisted pair flat or round	Twisted pair flat or round
Conductor Size	0.08042 mm^2 (28 AWG)	0.08042 mm^2 (28 AWG)
Maximum Signal Attenuation	0.095 dB per meter at 5MHz	0.095 dB per meter at 5MHz
Maximum Propagation Delay Difference Between any two signal pairs	0.20 nsec per meter	0.20 nsec per meter
Maximum DC Resistance	0.230 ohms per meter at 20°C	0.230 ohms per meter at 20°C

(*) Impedance measurement technique differs for Single Ended (see below).

Note that many of the cable characteristics are those called out in the Standard for ***Fast Data Transfer***. For reliable operation at full transfer rates for any kind of SCSI, these standards should also be met. Some additional comments should be made on the table above:

- The maximum cable length is limited by the ***ARBITRATION Phase*** timing. The cable length is shorter for the Single-Ended Interface because (historically) the drivers could not drive 25 meters of cheap ribbon cable reliably with the full complement of eight devices. With modern devices, and high-quality low-loss cable (as specified here), it is possible to drive a longer cable. There may be an effort in ***SCSI-3*** to extend the maximum length of the cable for the ***Single-Ended Interface***.

  To see how to measure cable length, see the examples at the end of this section.

- The maximum stub length and minimum stub spacing exist to ensure that the quality of the signals is not degraded. When a stub is too long or too close to another stub it can affect the local impedance of the cable, which in turn can cause signal reflections that cause discontinuities which degrade signal quality. As stubs are placed closer together, the magnitude of these discontinuities increases. When stubs are lengthened, the width of these discontinuities increases. To see how to measure stub length, see the examples at the end of this section.

- The characteristic impedance is a property of the cable. The ***Termination*** is matched relative to the impedance of the cable to ensure the best signal quality. If they are not properly matched, signal quality is degraded by signal reflections from the point of mismatch back to the driving device. Termination is matched to the cable as follows:

  For the Single-Ended Interface, sometimes the best results are achieved when the cable impedance is only 75% of the termination impedance. Sometimes the best results are achieved when the cable impedance is 100% of the termination impedance. Considerable work by several members of the ***X3T9.2 Committee*** has concluded that there is no exact solution. Results can vary based on the following factors:

  - Cable attenuation.
  - Type of Termination used.
  - ***Terminator Power*** voltage level.
  - Type of insulating material used in the Cable.
  - Low voltage level of the Single-Ended driver.
  - Type of cable (shielded twisted-pair or flat).

  For the Differential Interface, the best results are achieved when the cable impedance is 100% of the termination impedance. Differential is "different" because the balanced signal pairs cancel many of the effects seen in the Single-Ended Interface.

In practice, several ohms of mismatch does not cause a problem since the device receivers can compensate for some signal degradation. (See ***Single-Ended Interface*** and ***Differential Interface*** for a description of the receiver characteristics.)

Unfortunately for the Single-Ended Interface, the original SCSI-1 passive Termination (using the 220 Ω/330 Ω resistor network) has an impedance of 132 Ω. This is very different from the characteristic impedance of typical cables (and is another good reason to use the "preferred" Termination!). An inexpensive cable in Single-Ended mode can have a characteristic impedance of less than 60 Ω. Using such a low impedance cable can cause signal reflections of a magnitude that can cause the bus to fail.

The characteristic impedance of the cable needs to be measured in the mode that the cable will be used. For the Single-Ended Interface, ground one wire of each signal pair. For the Differential Interface, both wires of each signal pair are left open. Cable impedance is typically measured using time-domain reflectometry. The Single-Ended measurement for a typical cable will be about ⅔ the Differential measurement.

Note also that some manufacturers will specify a MINIMUM impedance and others will specify a MAXIMUM for the whole cable. For example, a cable with a MAXIMUM impedance of 80 Ω will have pairs on the outer layer with the impedance as little as 70 Ω.

Practice has shown that 80-85 Ω (measured in Single-Ended mode) shielded cable works well with the "preferred" Termination. Also, unshielded ribbon cable at 100 Ω has been found to be reliable for unshielded applications. These are only guidelines, there is no simple answer. It behooves you to test the system and find the best solution for your needs.

- The cable type is specified as twisted pair for differential to prevent crosstalk at the higher transfer speeds and longer lengths usually associated with the differential interface. If you are attempting to use the single-ended interface at high speed (e.g., for ***Fast Data Transfer***) it would be wise to use twisted pair.

A caveat on twisted pair: The electrical length, the potential for crosstalk, and characteristic impedance of any twisted pair in a cable can vary dependent on the location of the pair. The SCSI-2 standard only defines the connector positions of a cable, but not the conductor location. A round twisted pair cable contains the pairs in two or three concentric layers. Some guidelines on using this cable:

High Speed Edge-Triggered Signals: In general, the pairs closest to the center of a round cable have higher impedance than outer pairs, and therefore should be used for the most speed-critical signals (the ***REQ Signal*** and the ***ACK Signal***). The shielding provided by the middle layer will also minimize crosstalk.

Low Speed Signals with set-up time: Assign ***Data Bus Signals*** to the outer pairs, since they are much slower and enjoy a set-up time before they are examined. Also, this provides maximum separation from the REQ and ACK signals (reducing crosstalk from these signals is particularly important). The ***Terminator Power*** should also be assigned to one of these low impedance pairs on the outside of the cable, if possible.

Other Signals: All other signals are relatively insensitive to position in the cable.

Standards Efforts: The ***X3T9.2 Committee*** is working on defining conductor placement at this time; contact them for the latest working document(s).

- The conductor size can minimize signal attenuation over longer distances, and is also important in ensuring that the Terminator Power voltage is maintained at both ends of the cable. For shorter cables, the bus signals can use a smaller conductor of 30 AWG. The conductors that carry the Terminator Power should always be at least 28 AWG.

- The last three items on the list are from the Fast SCSI specification, but they are also recommended for increased reliability at "normal" rates. The Maximum Signal Attenuation minimizes the degradation of the voltage margin on the signals. The Maximum Propagation Delay Difference between pairs is important to minimize skew between signals. The Maximum DC Resistance minimizes the degradation of the Terminator Power voltage.

There are two diagrams on the following pages that illustrate how to measure cable length and stub length. There are factors that add to cable stub lengths beyond the cables connecting SCSI Devices.

Diagram 14 shows a schematic representation of cable length and stub length. Note that cable length is measured from the ***Terminator*** at one end of the system to the terminator at the other end. In other words, if there is any conductor between the last stub to the end device and the terminator on either end of the cable, that length should be included in the total cable length.

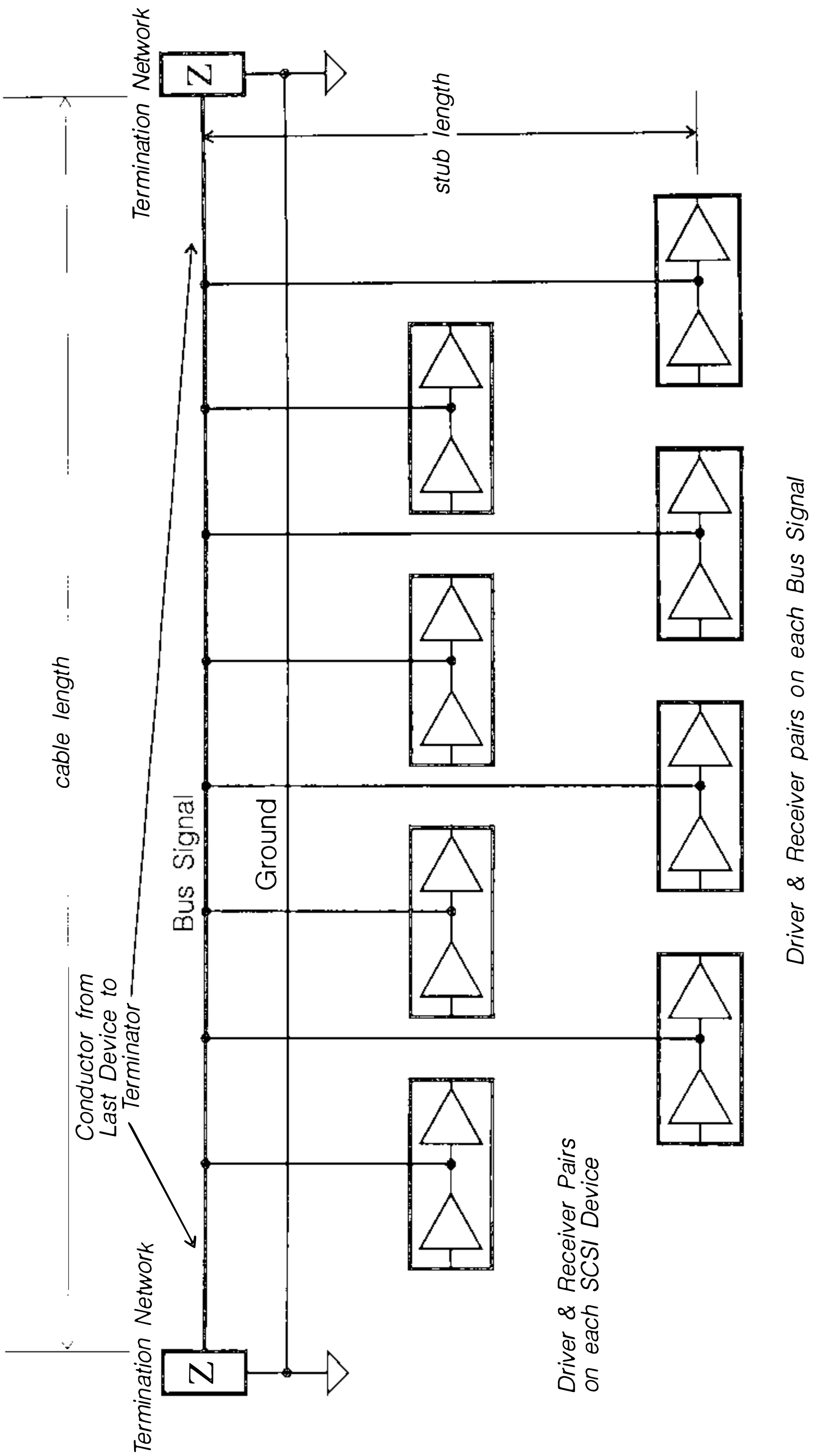
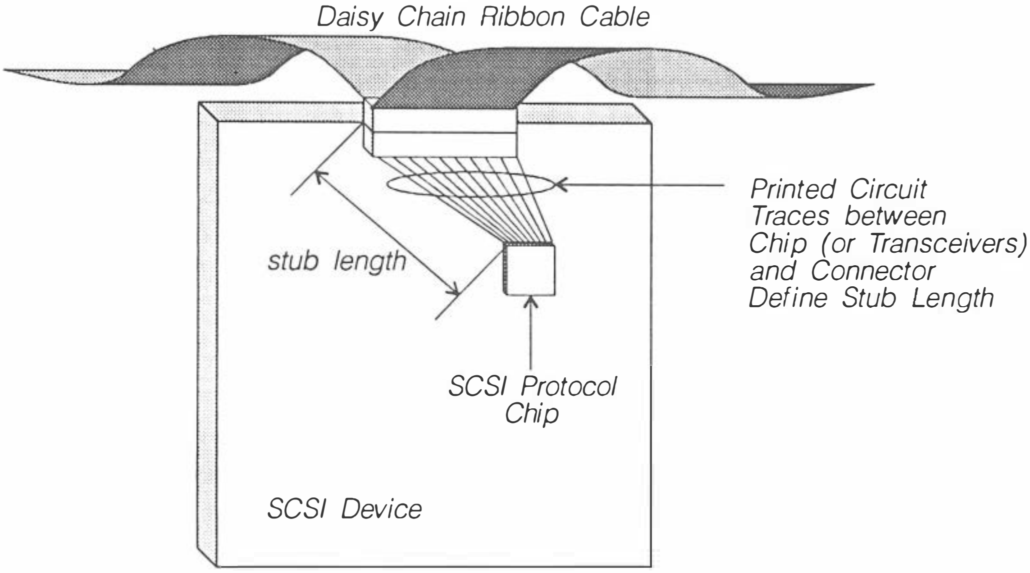

cable length
Termination Network
Conductor from Last Device to Terminator
Termination Network
Z
Bus Signal
Z
Ground
stub length
Driver & Receiver Pairs on each SCSI Device
Driver & Receiver pairs on each Bus Signal

DIAGRAM 14: CABLE AND STUB LENGTH SCHEMATIC

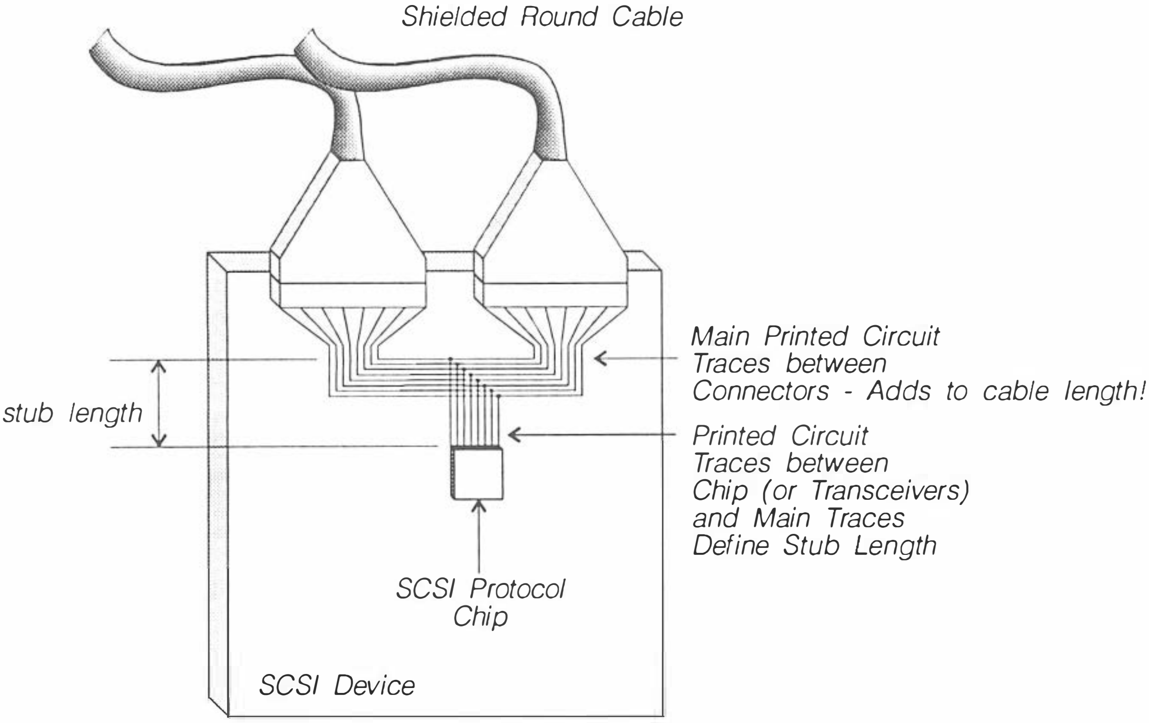
Diagram 15 shows two examples of device cable connections. The first example shows a SCSI Device with a single connector. This might be a small disk drive connected internally within a cabinet to a ***Host Adapter***. The cable is a ribbon style with a "mass-termination" connector inserted in the middle. In this example, the stub length is measured as the length of the longest printed circuit trace from the on-board connector to the SCSI Protocol ***Chip***. The cable length is measured only on the cable itself; the conductors on the board do not add to the cable length.

The second example shows a SCSI Device with two shielded connectors. Each connector attaches to a shielded round cable that leads to other SCSI devices in the chain. This might be a stand-alone device outside the host cabinet, like a tape drive shared between systems. The two connectors are bussed together by printed circuit traces; these traces add to the total cable length. The SCSI Protocol Chip is connected to these traces by stub traces; the stub length is the length of the longest stub trace.

> NOTE: When using the Differential Interface, the stub length is measured to the differential transceivers, not to the SCSI Protocol Chip. (As of this writing, there are no SCSI Protocol Chips available which integrate the differential transceivers.)


Daisy Chain Ribbon Cable
Printed Circuit Traces between Chip (or Transceivers) and Connector Define Stub Length
stub length
SCSI Protocol Chip
SCSI Device

Daisy Chain Ribbon Cable & One Connector


Shielded Round Cable
Main Printed Circuit Traces between Connectors - Adds to cable length!
stub length
Printed Circuit Traces between Chip (or Transceivers) and Main Traces Define Stub Length
SCSI Protocol Chip
SCSI Device

Shielded Cable & Two Connectors

DIAGRAM 15: CABLE AND STUB LENGTH EXAMPLES

Cable Skew Delay. *tcsd = 10 nsec.* The Cable Skew Delay is used to compensate for signal propagation speed differences within a ***Cable***. These speed differences can be caused by variations in cable impedance or loading between signal pairs. This can particularly occur in poorly constructed cable that has variations in insulation quality and wire length. On the other hand, it can **also** occur in good round cable where the inner signal pairs have different characteristics than the outer pairs. This delay is used for data transfers to ensure that setup and hold times are met at the receiving end of the cable. See ***Asynchronous Data Transfer*** and ***Synchronous Data Transfer***.

CAM (Common Access Method). A committee was formed by several suppliers of SCSI products to define a common software interface between the ***Host Adapter*** and the operating system. In 1990, the Transport/SCSI Interface Module ("XPT/SIM") was accepted by the ***X3T9.2 Committee*** as the basis for a standard. For more information on the current state of the CAM document, contact the X3T9.2 Committee.

Chips. It is impossible to describe all the chips available for SCSI in a volume such as this, since the information ages quickly as new introductions occur regularly. The SCSI protocol chip suppliers have been responsible for much of SCSI's success:

- NCR introduced the 5380, which included the drivers, receivers, and transfer logic that reduced the cost of a SCSI interface.
- Western Digital introduced the 33C93, which combined ***Bus Phases*** into single operations and began the push by all chip suppliers to reduce the time overhead involved in a ***Connection***.
- NCR's C700 introduced the concept of "Scripts", which allow the user to specify a program to the device that details how to execute a SCSI sequence.

Each new generation of chips includes more functionality and reduces overhead. The biggest problem is that microcode can seldom be transported from one generation of chips to the next, and, except for 5380 clones, not at all between different suppliers. When evaluating a new generation of chips, it is recommended that a rewrite of microcode be included in any development schedule.

CLEAR QUEUE Message. The CLEAR QUEUE message is used when ***Queuing*** is implemented. Note that the CLEAR QUEUE message must be supported in Targets that support Tagged Queuing, but it may also be supported for Untagged Queuing. You could think of CLEAR QUEUE as a gentle reset. CLEAR QUEUE is a single byte message:

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

The effect of this message is:

- ALL ***I/O Processes*** are removed from the Queue. The currently executing I/O Process is also aborted, no matter which Initiator issued it. That's **ALL** commands from ALL Initiators to ONE ***Logical Unit***.

- No ***Status*** or ***Messages*** are sent to any Initiator after this message is sent. The Target goes to ***BUS FREE Phase*** after it receives the message. All record of all processes are lost, and no Sense Data is retained.

- A ***Unit Attention Condition*** is created for each of the Initiators that had a command aborted as a result of this message. The Unit Attention Condition is not created for the Initiator that issued the message. The other Initiators will not find out that their I/O Processes were cleared until they decide to issue another command to the Target and receive CHECK CONDITION ***Status***.

  Note: The Unit Attention Condition may also be reported by ***Asynchronous Event Notification*** (AEN). AEN allows the Target to notify the Initiators directly.

- The net effect of this message is for the Target to perform the same action as if it received one ***ABORT Message*** from each Initiators to abort every I/O Process for the Logical Unit.

- This message is pretty hostile to the rest of the system. Consider this message to be just slightly less drastic than issuing a ***BUS DEVICE RESET Message***. See the ***ABORT TAG Message*** for a list of a queue-clearing messages, and some less drastic alternatives.

Summary of Use: The CLEAR QUEUE message is sent only by an Initiator at any time to immediately terminate all Active and Queued I/O Processes for all Initiators from the Queue of a Logical Unit.

COMMAND COMPLETE Message. The COMMAND COMPLETE message indicates to the Initiator that the command is all done (simple, eh?). Actually, it is more accurate to say that the message completes an ***I/O Process***. COMMAND COMPLETE is a single byte ***Message***:

Bit Byte	7	6	5	4	3	2	1	0
0	Message Code = 00 hex							

There are two things that happen at the end of a command execution:

(1) ***Status*** is sent by the Target to the Initiator to indicate how well the command execution went. This is a "command level" function and is often passed back to the Initiator's host system.

(2) A COMMAND COMPLETE message is sent to indicate that the command is no longer active. This tells the ***Path Control*** logic of the Initiator that the internal pointers and any other resources allocated to the command may be released, and that the following ***BUS FREE*** phase does not indicate an error condition. (Note: A ***LINKED COMMAND COMPLETE Message*** is also an appropriate response; see also ***Linked Commands***.)

Note that the command is completed, but only as far as the Initiator resources are concerned. The Target may not have done anything yet with the Command! In certain modes, and with certain commands, the Target may report that a command is completed before it actually executes the action. For example, a REWIND command causes a ***SCSI Device*** to rewind a tape. The device may report COMMAND COMPLETE, and then begin the rewind. This lets the Initiator go on to the next operation without holding it up until the end of the rewind. See ***Asynchronous Event Notification (AEN)***, and the other volumes of the SCSI Encyclopedia, for more.

If ***Linked Commands*** are enabled, then a different message (see ***LINKED COMMAND COMPLETE Messages***) is sent to indicate the end of a command. A sequence of linked commands making up one I/O Process is still ended with a COMMAND COMPLETE Message.

Summary of Use: The COMMAND COMPLETE message is sent only by a Target after a ***STATUS Phase*** to indicate the completion of an I/O Process.

Command Descriptor Block. The Command Descriptor Block (CDB) defines the basic function to be performed by a Target. The CDB may also contain some parameters or identifiers appropriate to the function. In this discussion we will show the common portions of all CDBs, and we will show the specific possible formats of each size CDB.

A SCSI CDB is fully specified by the following items:

- The Operation Code (or "OpCode") which specifies the function to be performed. The first byte of the CDB contains the OpCode.

- The Command Parameters, which make up the rest of the CDB not containing the OpCode, the ***Logical Unit Number (LUN)*** (see note below), and the Control Byte.

- The Control Byte, which controls ***Linked Commands***.

NOTE: The Logical Unit Number (LUN), which specifies the unit for the commanded function, is usually specified in the ***IDENTIFY Message***. In older SCSI-1 devices, the LUN was specified in the second byte of the CDB if no IDENTIFY Message was sent. SCSI-2 users, and any new implementations, should ignore bits 7, 6, and 5 of the second byte of the CDB, as these bits may be assigned to other uses in ***SCSI-3***. Initiators should always set the LUN bits to zero.

Figure 18 shows the "generic" format for **all** SCSI CDBs of any length:

Bit Byte	7	6	5	4	3	2	1	0
0	Operation Code							
1	Parameter Byte 1							
2	Parameter Byte 2							
....	....							
N	Parameter Byte N							
N+1	Control Byte							

FIGURE 18: GENERIC COMMAND DESCRIPTOR BLOCK FORMAT

All SCSI CDBs conform to the format shown in Figure 18. The Operation Code is made up of two parts, as shown in Figure 19:

Bit	7	6	5	4	3	2	1	0
	Group Code			Command Code				

FIGURE 19: OPERATION CODE FORMAT

The **Group Code** portion of the OpCode indicates the command length (as shown in Table 15), while the **Command Code** indicates the function to be performed. With a few exceptions (such as READ and WRITE) the Command Code indicates a different function in each Group; this is not consistent. It is better and safer to treat the OpCode as a single value, and extract the Group Code only when the command length must be determined.

TABLE 15: COMMAND LENGTH

Group Code				
Binary	Hex	Command Length	Number of Parameters	Control Byte Location
000XXXXX	00 thru 1F	6 bytes	4 bytes	Byte 5
001XXXXX	20 thru 3F	10 bytes	8 bytes	Byte 9
010XXXXX	40 thru 5F	10 bytes	8 bytes	Byte 9
011XXXXX	60 thru 7F	***** RESERVED BY SCSI *****		
100XXXXX	80 thru 9F	***** RESERVED BY SCSI *****		
101XXXXX	A0 thru BF	12 bytes	10 bytes	Byte 11
110XXXXX	C0 thru DF	***** Vendor Specific *****		
111XXXXX	E0 thru FF	***** Vendor Specific *****		

NOTE: The length of Groups 6 and 7 is unique to the device supplier. Current practice in most cases is to use a length of ten bytes for these Groups, but this cannot be relied upon.

The content of the Parameter Bytes is discussed after the Control Byte. The format of the Control Byte is shown in Figure 20.

Bit	7	6	5	4	3	2	1	0
	Vendor Specific		Reserved				Flag	Link

FIGURE 20: CONTROL BYTE FORMAT

The **Control Byte** is always the last byte of any CDB. The primary purpose of the Control Byte is to control the behavior of the Target during ***Linked Commands***. If you are not interested in Linked Commands, then you always set the Control Byte to zero. In detail, the fields in the Control Byte are:

Link (bit 0): When the Link bit is set to one, Linked Commands are enabled. Refer to that section for more details.

Flag (bit 1): The Flag bit controls which ***LINKED COMMAND COMPLETE Message*** is sent to report command completion when Linked Commands are enabled. It may only be set to one when the Link bit is also set. Refer to Linked Commands for more details.

Reserved (bits 2 through 5): These bits have been ***Reserved*** for future use. As with any reserved bits or bytes, don't use them! Initiators should always set them to zero.

Vendor Specific (bits 6 and 7): These two bits are now seldom used. In the early days of SASI, these bits were used for controlling some functions that were common to several six byte commands, such as read retries. Since there were no other bit locations available in the Group 0 command blocks (see below), the Vendor Specific bits were preserved in ***SCSI-1*** for these special functions. In ***SCSI-2***, the bits are still available, but now the functions are available in ten byte commands or the control has been moved to MODE SELECT. New designs should avoid using these bits if there are any other alternatives.

Figure 21, Figure 22, and Figure 23, respectively, show the typical formats for six, ten, and twelve byte Command Descriptor Blocks. The sidebar to the right of the CDB indicates what each byte might be used for. A glossary of those uses follows the figures.

Bit Byte	7	6	5	4	3	2	1	0	Command Byte Use
0	Group Code			Command Code					Operation Code
1	Logical Unit Number			Parameter Byte 1					LUN, option bits, LBA
2	Parameter Byte 2								LBA, transfer length, special
3	Parameter Byte 3								LBA, transfer length, special
4	Parameter Byte 4								LBA, transfer length, special
5	Vendor Specific		Reserved				Flag	Link	Control Byte

FIGURE 21: SIX BYTE COMMAND DESCRIPTOR BLOCK FORMAT

Bit Byte	7	6	5	4	3	2	1	0	Command Byte Use
0	Group Code			Command Code					Operation Code
1	Logical Unit Number			Parameter Byte 1					LUN, option bits
2	Parameter Byte 2								LBA, special
3	Parameter Byte 3								LBA, special
4	Parameter Byte 4								LBA, special
5	Parameter Byte 5								LBA, special
6	Parameter Byte 6								transfer length, special
7	Parameter Byte 7								transfer length, special
8	Parameter Byte 8								transfer length, special
9	Vendor Specific		Reserved				Flag	Link	Control Byte

FIGURE 22: TEN BYTE COMMAND DESCRIPTOR BLOCK FORMAT

Bit Byte	7	6	5	4	3	2	1	0	Command Byte Use
0	Group Code			Command Code					Operation Code
1	Logical Unit Number			Parameter Byte 1					LUN, option bits
2	Parameter Byte 2								LBA, special
3	Parameter Byte 3								LBA, special
4	Parameter Byte 4								LBA, special
5	Parameter Byte 5								LBA, special
6	Parameter Byte 6								transfer length, special
7	Parameter Byte 7								transfer length, special
8	Parameter Byte 8								transfer length, special
9	Parameter Byte 9								transfer length, special
10	Parameter Byte 10								always reserved
11	Vendor Specific		Reserved				Flag	Link	Control Byte

FIGURE 23: TWELVE BYTE COMMAND DESCRIPTOR BLOCK FORMAT

Summary of Parameter Byte Uses:

LUN: The second byte of any CDB contains a three bit field which specifies the ***Logical Unit Number***. Since SCSI-2 requires the use of the ***IDENTIFY Message***, this field is essentially unused. SCSI-1 Targets will recognize this field only if no IDENTIFY Message is sent. Since this field is virtually unused in SCSI-2, the ***SCSI-3*** effort may reclaim the LUN field for other uses. Initiators are well advised to use the IDENTIFY Message exclusively and set these bits to zero.

Option Bits: The remainder of the second CDB byte is typically used to control command options. These controls can be one, two, or three bit fields. In six byte commands the second CDB byte can also contain the upper bits of the LBA.

LBA: The next several bytes after the first or second CDB byte often specify the ***Logical Block Address***. This is true for any command in which a LBA is specified.

Transfer Length: The bytes toward the end of the CDB are often used to specify the transfer length of the DATA phase bytes that are to be transferred. The types of transfers lengths can include:

- Parameter List Length: Length in bytes of additional parameter data that further defines what operation the command will perform.

- Allocation Length: Length in bytes of the space that the Initiator has allocated to receive a list of data from the Target.

- Transfer Length: The number of blocks or bytes of READ or WRITE data (data that is stored on the Target at some point) to be transferred.

Special: Once you depart from the READ and WRITE commands, the parameter bytes can be used for almost anything. Some of the things that can be "special" include:

- more option bits
- vendor specific fields
- sub-codes
- disk interleave

COMMAND Phase. ***Command Descriptor Blocks (CDB)*** are passed from the Initiator to the Target during a COMMAND phase. The only purpose of a COMMAND phase is to send a six, ten, or twelve byte CDB. The COMMAND phase can actually occur in only three different general sequences:

- Following the ***Initial Connection*** to the Target by the Initiator. After ***SELECTION Phase***, the Initiator sends an ***IDENTIFY Message*** and (when ***Tagged Queuing*** is being used) a ***Queue Tag*** message. After the first or both messages are received by the Target a COMMAND phase occurs. This is by far the most typical case.

  For SCSI-1 devices, the Initial Connection might be completed without a MESSAGE OUT Phase (SELECTION Phase only). In this case, COMMAND Phase follows after SELECTION Phase.

- A less typical case occurs with ***Linked Commands***. When a Linked Command is completed, a ***LINKED COMMAND COMPLETE Message*** is sent by the Target. When that ***MESSAGE IN Phase*** is completed, the Target switches to COMMAND Phase to get the next command.

- A much less typical case can occur only when the Target ***Queues*** the ***I/O Processes*** for later execution. Some Targets will save a minimum set of data which records the Initiator that selected it, the ***Logical Unit Number (LUN)***, and (optional) Queue Tag. When the queued I/O Process is activated for execution, the Target will ***Reconnect*** to the Initiator, send an IDENTIFY message (and optional Queue Tag), and enter COMMAND phase to get the command. This scheme allows a Target to queue more I/O Processes using less internal memory. Robust Initiator designs should be prepared for this kind of behavior from a Target.

The COMMAND Phase may be followed by any of the other five ***Information Transfer Phases***:

- A ***DATA Phase*** so that the Target may transfer the data requested by the command.

- A ***MESSAGE IN Phase*** to send a ***DISCONNECT Message*** or perform some other form of ***Path Control***.

- A ***STATUS Phase*** to return status of a command that required no data transfer.

- A ***MESSAGE OUT Phase*** may occur if the Initiator created the ***Attention Condition*** during the COMMAND Phase.

These transitions are illustrated in Diagram 16.

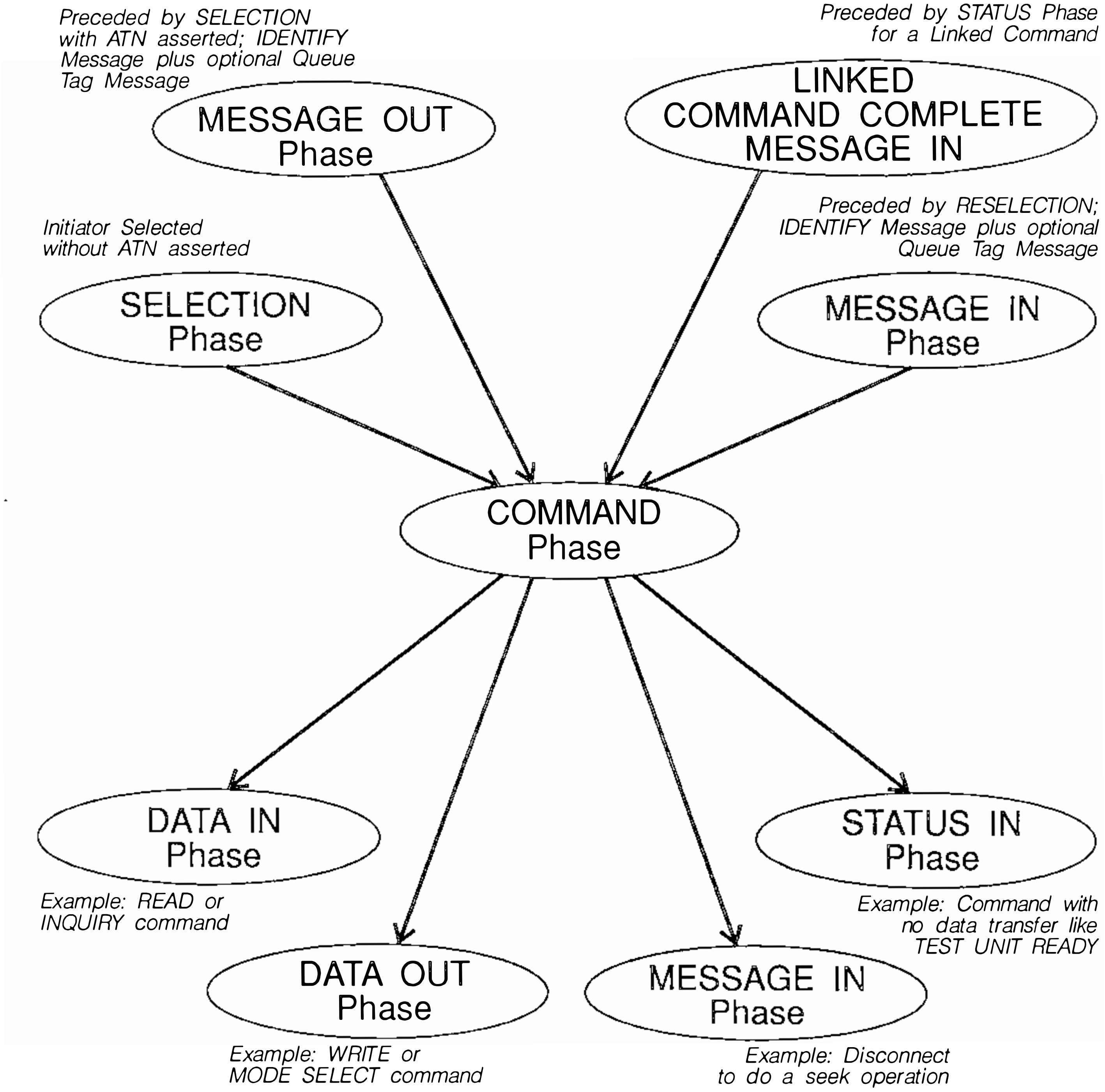

Preceded by SELECTION with ATN asserted; IDENTIFY Message plus optional Queue Tag Message
MESSAGE OUT Phase
Preceded by STATUS Phase for a Linked Command
LINKED COMMAND COMPLETE MESSAGE IN
Initiator Selected without ATN asserted
SELECTION Phase
Preceded by RESELECTION; IDENTIFY Message plus optional Queue Tag Message
MESSAGE IN Phase
COMMAND Phase
DATA IN Phase
Example: READ or INQUIRY command
STATUS IN Phase
Example: Command with no data transfer like TEST UNIT READY
DATA OUT Phase
Example: WRITE or MODE SELECT command
MESSAGE IN Phase
Example: Disconnect to do a seek operation

DIAGRAM 16: COMMAND PHASE TRANSITIONS

Command Pointer. The Command Pointer refers to the source of the ***Command Descriptor Block*** (CDB) within the Initiator. At any time, the Command Pointer refers to a particular byte within the CDB. As with all pointers, there is a ***Current Pointer*** and a ***Saved Pointer***. See ***Pointers*** for the whole story.

COMMAND TERMINATED Status. COMMAND TERMINATED status is returned whenever the ***I/O Process*** is terminated by the Initiator; in fact, it should really be called "I/O PROCESS TERMINATED Status". Oh well. The Initiator terminates a command by issuing the ***TERMINATE I/O PROCESS*** message. Targets that do not accept the TERMINATE I/O PROCESS message do not return this status.

This status is **not** returned when:

- The Target has already finished executing the command; the message got there too late to have any effect. In this case, the Target returns the status it would return if the TERMINATE I/O PROCESS message had not been sent. For example, if the Target had been executing a READ command, and all the Logical Blocks requested had already been transferred to the Initiator when the message is received, the Target returns ***GOOD*** status since no operation was stopped before it finished.

- A fault occurs during the execution of the command that is serious enough to be reported to the Initiator. In this case, ***Status*** that indicates the fault is returned, and the TERMINATE I/O PROCESS message is ignored.

In **all** other cases, the TERMINATE I/O PROCESS message is accepted and COMMAND TERMINATED status is returned, even if the Command Descriptor Block has not yet been transferred to the Target.

When COMMAND TERMINATED status is sent by the Target, a ***Contingent Allegiance*** condition is created (which must be cleared by the Initiator). The sense data created describes how and where the I/O Process was terminated:

- If the I/O Process is terminated during a data transfer (***DATA IN Phase*** or ***DATA OUT Phase***), then the sense data indicates where the data transfer was interrupted. If the data was completely transferred before the TERMINATE I/O PROCESS message is received by the Target, the sense data would indicate that all data was transferred.

- If the command has no data transfer associated with it, then the sense data just indicates that the termination occurred, and any other information that is appropriate.

- If a command has not yet been received by the Target when it receives the TERMINATE I/O PROCESS message (pretty fickle Initiator to terminate before it gets started!), then the sense data just indicates that the termination occurred.

- If the Target is ***Queuing*** I/O Processes, and the message applies to an I/O Process already in the Queue (the Initiator connected to the Target just to send the TERMINATE I/O PROCESS Message), the Target responds as if no command had been received (see above). The Target is allowed, however, to disconnect and let the I/O Process exit the queue normally before terminating it.

Condition. A SCSI Bus Condition is a state that a ***SCSI Device*** creates that is not a ***Bus Phase***. The state may be indicated by ***Control Signals*** on the bus, or it may be indicated by ***Status*** or ***Messages*** passed between two SCSI Devices. The following Conditions are defined:

- ***Attention Condition***. The Initiator creates this condition on the bus by asserting the ***ATN Signal*** while ***Connected*** to a Target. The purpose of the Attention Condition is to alert the Target to the fact that the Initiator has a ***Message*** to send. The Attention Condition is followed (and ended) by a ***MESSAGE OUT Phase***. The scope of the Condition is between the Initiator and Target currently Connected on the bus, and it exists only for the duration of that ***Connection***, or until the MESSAGE OUT Phase is completed.

- ***Contingent Allegiance Condition***. The Target creates this Condition by sending CHECK CONDITION or COMMAND TERMINATED ***Status***, or possibly by a transition to an ***Unexpected BUS FREE Phase***. The purpose of the Contingent Allegiance Condition is to create a state where Sense Data for the Initiator is maintained by the Target for a ***Logical Unit*** until the Initiator either requests it or indicates that it should be discarded. The scope of the Condition is between an Initiator and Target for a Logical Unit independent of any Connection, and ends when the Target sends or discards the Sense Data.

- ***Extended Contingent Allegiance (ECA) Condition***. The Target creates this Condition by sending the ***INITIATE RECOVERY Message***. A SCSI Device may also create this Condition by taking the Initiator role and sending the INITIATE RECOVERY Message as part of an ***Asynchronous Event Notification (AEN)***. The purpose of the ECA Condition is to cause command processing for a Logical Unit in the Target to be suspended until the Initiator can effect an error recovery procedure with the Target. The scope of the Condition is between two SCSI Devices, and a Logical Unit on the device that sent the INITIATE RECOVERY Message, independent of any particular connection, and ends when the device that received the INITIATE RECOV-

ERY sends the ***RELEASE RECOVERY Message*** or ***BUS DEVICE RESET Message*** to the other device, or a ***Hard Reset*** occurs on the other device.

- ***Reset Condition***. Any SCSI Device creates this condition by asserting the ***RST Signal*** at any time, whether or not it is Connected to another device. The purpose of the Reset Condition is to clear any currently Connected devices off the bus (usually when one is failing and hanging up the bus) and/or initialize all of the devices attached to the ***SCSI Bus***. The Reset Condition is followed by the ***BUS FREE Phase***. The scope of the Condition is between all SCSI Devices attached to the SCSI Bus until the trailing edge of the RST Signal.

- ***Unit Attention Condition***. This Condition is created within a Target whenever an internal state changes that is important enough that any affected Initiators that subsequently ***Connect*** to the Target should know about it. The state change may be caused by the ***Peripheral Device***, or by another SCSI Device. The purpose of this Condition is to provide a mechanism for reporting operational changes within the Target outside the scope of a particular command. The scope of the Condition is between a Target and all Initiators that have not had the condition reported yet, and is independent of any particular Connection.

To contrast Condition to Bus Phase: A Phase is a state of the SCSI Bus (specifically), and is used to effect a Connection or ***I/O Process*** between two SCSI Devices. A Condition exists outside the scope of any Phase and may involve more than two SCSI Devices. It may help to think of a Condition as a "Meta-State" of the SCSI Bus.

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

(1) ***SELECTION*** of a Target by the Initiator, which establishes an I_T Nexus.

(2) The transfer of an ***IDENTIFY*** message from the Initiator to the Target to identify the ***Logical Unit*** or ***Target Routine***, and establish an I_T_x Nexus.

(3) If ***Queuing*** is used, the transfer of one of the ***Queue Tag Messages*** to specify a ***Queue Tag*** to identify the command in the queue, and establish an I_T_L_Q Nexus (there are no I_T_R_Q Nexuses).

The flow to establish a connection for various types of Nexuses is shown in Diagram 17.

When the Initiator has "connected" to the Target, a Nexus (relationship) has been established between the two devices, and an ***I/O Process*** is begun. The SCSI standard calls this an ***Initial Connection***. Usually after the connect is completed, the Target requests a ***COMMAND Phase*** and the Initiator sends a ***Command Descriptor Block (CDB)*** to the Target. Or, the Initiator may send more messages in the same ***MESSAGE OUT Phase*** as the messages listed above.

Sometimes, it helps to use a word in a sentence to better define it:

> "The Initiator **connects** to a Target to begin an I/O Process."
>
> "The Initiator will now **connect** to the Target to ABORT the command issued the last time the Initiator **connected** to the Target."
>
> "An Initiator **connects** to a Target, but a Target reconnects to an Initiator."

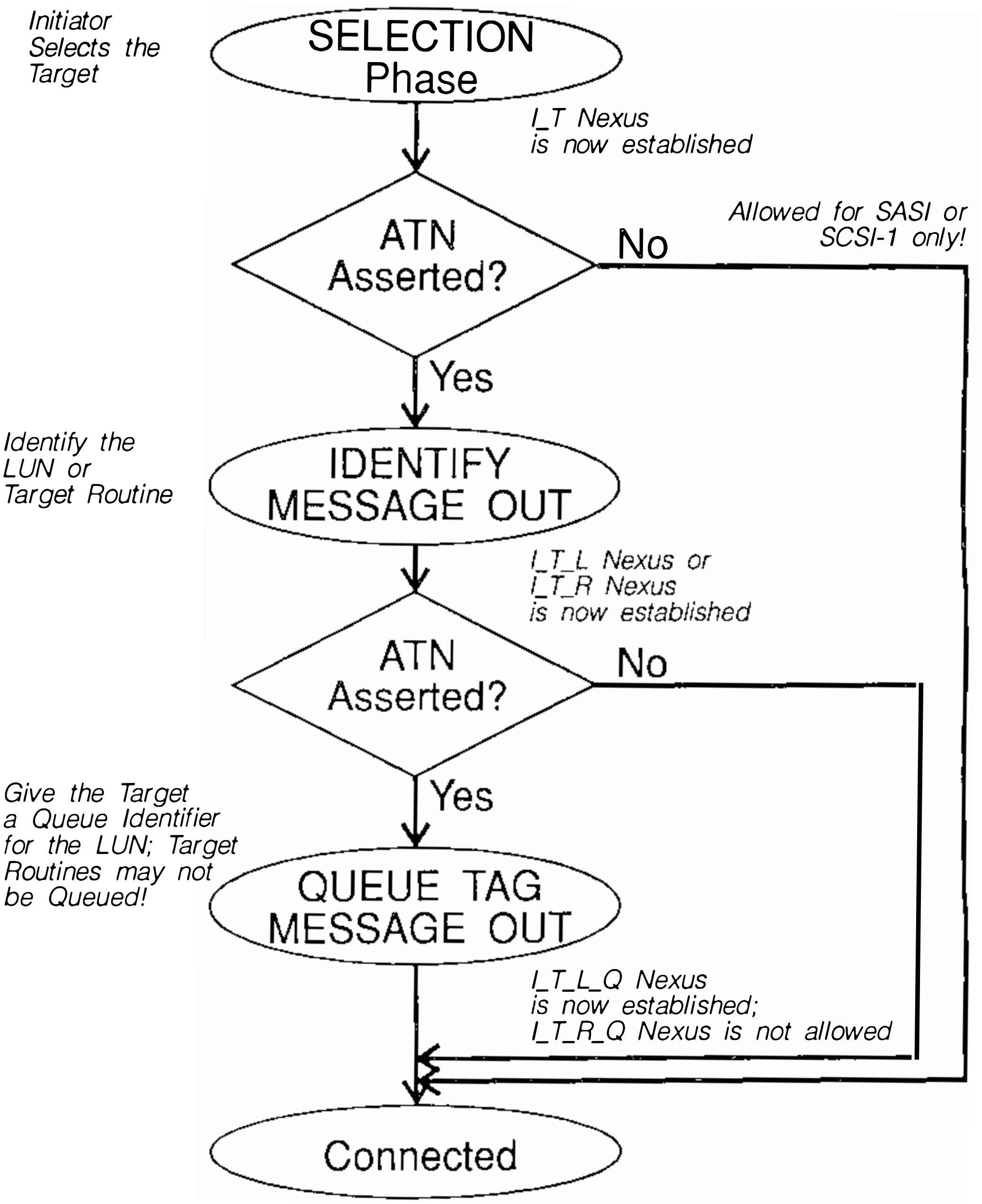

Initiator Selects the Target
SELECTION Phase
I_T Nexus is now established
ATN Asserted?
No
Allowed for SASI or SCSI-1 only!
Yes
Identify the LUN or Target Routine
IDENTIFY MESSAGE OUT
I_T_L Nexus or I_T_R Nexus is now established
ATN Asserted?
No
Yes
Give the Target a Queue Identifier for the LUN; Target Routines may not be Queued!
QUEUE TAG MESSAGE OUT
I_T_L_Q Nexus is now established; I_T_R_Q Nexus is not allowed
Connected

DIAGRAM 17: CONNECTION FLOW DIAGRAM

Connected I/O Process. Same as a ***Current I/O Process*** (see below).

Connection. A "connection" is what exists after:

- An Initiator initially ***Connects*** to a Target; or
- A Target ***Reconnects*** to an Initiator.

A connection ends when a BUS FREE phase occurs.

Connection Phases. These phases (as differentiated from ***Information Transfer Phases***) are the ***Bus Phases*** that control the connecting of a ***SCSI Device*** to the bus. These phases are:

- The ***ARBITRATION Phase***, which is used to establish Bus ownership prior to a ***Connection***.
- The ***SELECTION Phase***, which is used by Initiator to establish an ***Initial Connection*** with a Target.
- The ***RESELECTION Phase***, which is used by a Target to establish a ***Reconnection*** with an Initiator.

Connectors. Connectors attach ***Cables*** to ***SCSI Devices***. SCSI defines several connectors for this purpose, each of which has its own strengths and weaknesses. Table 16 lists the different types of connectors defined by SCSI, the ***SCSI-1*** and ***SCSI-2*** names for them, and the "common names". For exact details on each of these connectors, we must refer you to the appropriate standard, since we are loathe to recreate connector drawings or list manufacturers part numbers (we'd probably get them wrong; it took the ***X3T9.2 Committee*** years to get them right...).

TABLE 16: CONNECTOR ALTERNATIVES

SCSI-1 Connector Name	SCSI-2 Connector Name	Common Name(s)
Non-Shielded, Standard	Non-shielded Connector Alternative 2 (allowed for A Cable only)	"Header" "Mass Termination Header" "Flat Ribbon Cable Connector"
Shielded Alternative 1, Recommended	N/A	"AMP-MODU" "Shielded Header"
Shielded Alternative 2, Recommended	Shielded Connector Alternative 2 (allowed for A Cable only)	"Shielded Miniature Ribbon Connector" "Ribbon Style Printer Connector"
N/A	Non-shielded Connector Alternative 1	"High Density" "Micro-connector" "High Density Pin and Socket" "High Density Tab and Receptacle"
N/A	Shielded Connector Alternative 1	"Shielded High Density" "Shielded Micro-connector" "Shielded High Density Pin and Socket" "Shielded High Density Tab and Receptacle"

Got all that...? The "moral" is that you should be aware of which standard is being referred to when someone specifies an "Alternative X" connector.

Pros and Cons. Each of these connectors has pros and cons associated with its use. Table 17 summarizes them. Note that we use the "Common Name" from the previous table; we are not gluttons for punishment...

TABLE 17: CONNECTOR ALTERNATIVES

Common Connector Name	Advantages	Disadvantages
"Header"	1. It's cheap! 2. It's easy to use and build cables (through mass-termination).	1. It takes more circuit board area than other alternatives. 2. They are often not well keyed, even though the standard calls for it.
"AMP-MODU"	1. Uh, it's shielded...	1. It's not often used. 2. It's more expensive than other alternatives. 3. It takes more circuit board area than other alternatives.
"Shielded Ribbon"	1. It may be the lowest cost shielded alternative, since it is so common. 2. It's shielded.	1. It is too big to fit on many Host Adapter cards that present one side to the outside of the cabinet (e.g., Micro Channel). 2. It takes more circuit board area than other alternatives.
"High Density"	1. It takes less circuit board area than other alternatives. 2. It is well keyed.	1. It's more expensive than the Header.
"Shielded High Density"	1. It is small enough to fit on most Host Adapter cards that present one side to the outside of the cabinet (e.g., Micro Channel). 2. It takes less circuit board area than other alternatives.	1. It's more expensive than other alternatives. 2. Physical limitations degrade the cable characteristics (impedance and crosstalk) due to tighter spacing.

Non-Standard Connectors. By now you may notice we have an attitude problem about Connectors. I think many computer design engineers would agree that a working connector is all that matters; let the mechanical engineers do their fine job on the details of retention force and mating cycles and stuff. Unfortunately, a member of the ***X3T9.2 Committee*** has to sit through meeting after meeting slogging through these details. The SCSI-2 effort was particularly tedious, since nearly all of the connectors considered were fine products in their own right. Politics made the choice...

Through politics, and the proprietary nature of some companies' products, non-standard connectors have proliferated through the industry. Some companies have opted for connectors that lost out in the SCSI-2 selection process. Others selected connectors independently since standard connectors for what they needed did not exist yet. The two desktop leaders, Apple and IBM, both use non-standard connectors, and IBM's connector won't mate to Apple's. Fortunately, most peripheral suppliers use standard connectors.

The "good news" is that there exist adapter cables between the non-standard and standard connectors. The "bad news" is that these are often sold by the system suppliers at exorbitant prices. You should also know that some non-standard connectors, particularly the ones that have a smaller number of pins than 50, may have grounding or impedance matching problems. See ***Cables***.

Contingent Allegiance. "Say what?" Admit it: that's what you said when you first saw this term; so did most of the SCSI committee. A handy pocket dictionary gives the following definitions:

Contingent: 1. possible 2. accidental 3. dependent (on an uncertainty).

Allegiance: 1. loyalty 2. devotion (to a cause).

So, if we make up all possible combinations, let's see what makes sense:

(1) Possible Loyalty: Not something **we** would depend on.
(2) Accidental Loyalty: We prefer determinism here.
(3) Dependent Loyalty: Bolts at the first sign of trouble.
(4) Possible Devotion: Won't help you in your sick bed.
(5) Accidental Devotion: Touching, but what has it to do with SCSI?
(6) Dependent Devotion: Actually, the closest to what SCSI calls it!

Getting down to brass tacks, "Contingent Allegiance" is a name for the ***Condition*** that can exist when a fault occurs during command execution. When the fault occurs, the Target returns CHECK CONDITION ***Status*** to the Initiator. After that status is transferred, the Contingent Allegiance condition exists between the Initiator and Target for that Logical Unit where the fault occurred. The Contingent Allegiance condition may also exist after ***COMMAND TERMINATED Status*** is returned, and may optionally exist after an ***Unexpected BUS FREE Phase***.

When there exists a Contingent Allegiance condition, you could say that a state of "dependent devotion" exists. Why? Let's follow it step by step in Diagram 18.

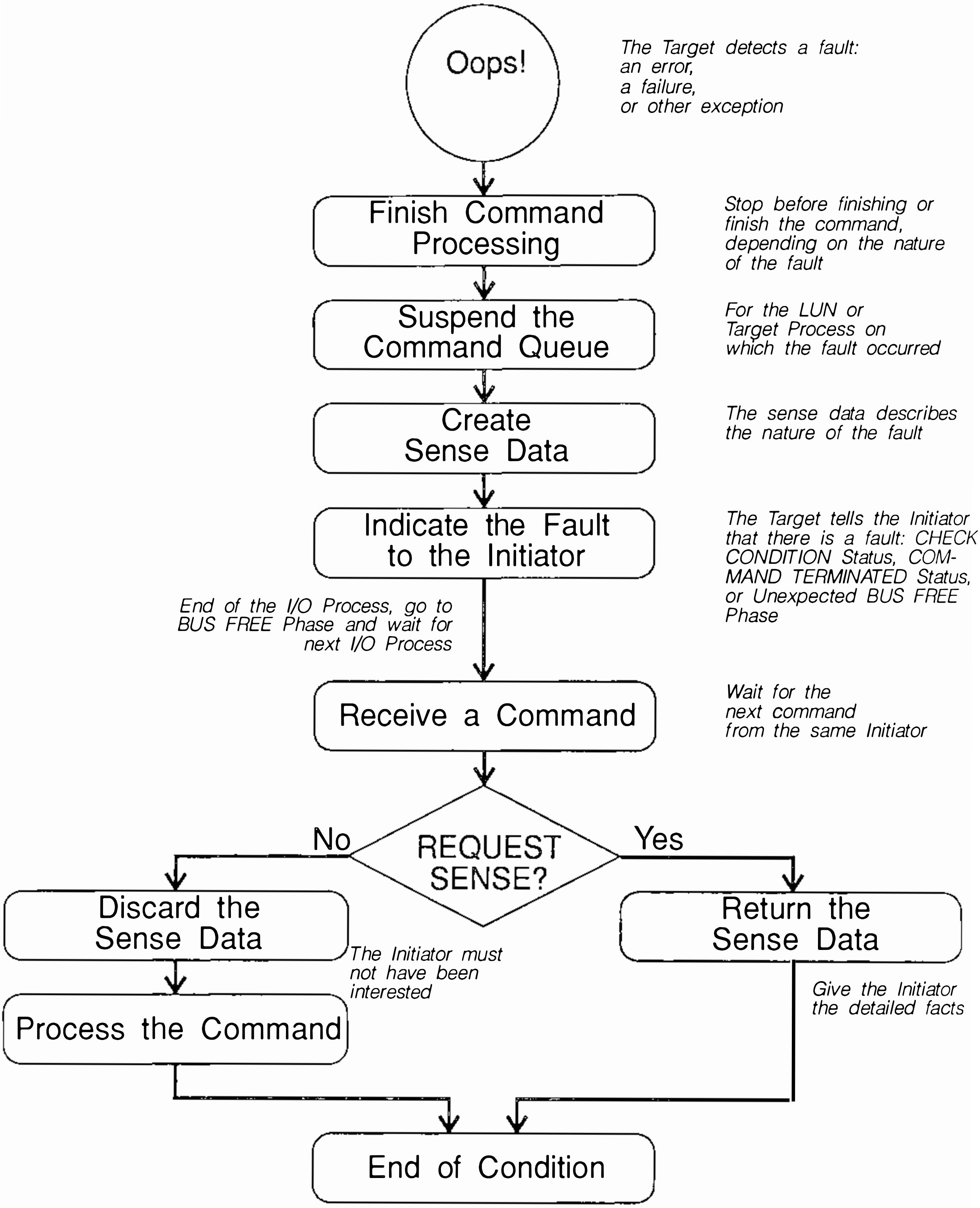

Oops!
The Target detects a fault: an error, a failure, or other exception
Finish Command Processing
Stop before finishing or finish the command, depending on the nature of the fault
Suspend the Command Queue
For the LUN or Target Process on which the fault occurred
Create Sense Data
The sense data describes the nature of the fault
Indicate the Fault to the Initiator
The Target tells the Initiator that there is a fault: CHECK CONDITION Status, COMMAND TERMINATED Status, or Unexpected BUS FREE Phase
End of the I/O Process, go to BUS FREE Phase and wait for next I/O Process
Receive a Command
Wait for the next command from the same Initiator
REQUEST SENSE?
No
Yes
Discard the Sense Data
The Initiator must not have been interested
Process the Command
Return the Sense Data
Give the Initiator the detailed facts
End of Condition

DIAGRAM 18: CONTINGENT ALLEGIANCE FLOW DIAGRAM

The steps that occur to establish a Contingent Allegiance Condition are:

(1) A fault occurs on the Target that is serious enough to report to the Initiator.

(2) The Target creates ***Sense Data*** that describes the fault.

(3) The Target changes to ***STATUS*** phase and returns CHECK CONDITION status to the Initiator.

(4) The Target changes to ***MESSAGE IN Phase***, returns a ***COMMAND COMPLETE Message***, and goes to ***BUS FREE Phase***.

Now, let's examine the state of affairs. The Target had a problem that was serious enough to bother the Initiator with, so it terminates the ***I/O Process***. Further, the Target has additional data describing the problem to the Initiator taking up space in its memory. This is where the "dependent devotion" comes in. The Target will hold onto that data for the Initiator until the Initiator comes and retrieves it (using a REQUEST SENSE command), or discards it (see below). Two other things can happen:

- If ***Queuing*** is implemented, any commands pending in the queue are suspended until the condition is cleared. This potentially ties up the other Initiators. NOTE: The Target is not precluded from accepting more commands to the Queue for the ***Logical Unit*** from other Initiators. Other Logical Units are not affected.

- If the Target has little space available for storing sense data, it may not be able to accept any more commands. In this case the Target will be returning BUSY status to the other Initiators. This can also potentially tie them up.

As you can see, it is important that the Initiator which received the CHECK CONDITION do everything it can to clear the condition. The Contingent Allegiance condition can be cleared by the Initiator in one of the following ways:

- By issuing a REQUEST SENSE command to the Target and Logical Unit or ***Target Routine*** (i.e., the I_T_x ***Nexus***) that reported the fault, and receiving the sense data.

- By issuing any other command to the I_T_x Nexus that reported the fault. When the Target sees that the command is not REQUEST SENSE, it may discard the sense data. This is not recommended; you never can be **sure** what the fault really was.

- By issuing an ***ABORT Message*** to the I_T_x Nexus that reported the condition. This also causes the Target to discard the sense data. This is also not recommended since the data describing the problem is lost.

- By issuing a ***BUS DEVICE RESET Message*** to the Target. This is a fairly serious and "system hostile" response. You would only do this if you thought the Target was in a very odd state. (But how would you know? You never got the sense data!)

- By generating a ***Reset Condition*** on the bus; i.e., asserting the ***RST Signal***. This is very hostile to the system. Not recommended.

So far, we have shown that the Contingent Allegiance condition can be created due to a fault on the Target. There are two other ways:

- If the Target deliberately causes an ***Unexpected BUS FREE Phase***, then the Target might also create sense data. Since the Target created sense data (to describe a catastrophic fault), the Contingent Allegiance condition also exists. It is recommended that an Initiator that sees an unexpected BUS FREE phase attempt to retrieve sense data from the Target to clear a possible Contingent Allegiance condition.

- The Contingent Allegiance condition can also be created when the Target returns ***COMMAND TERMINATED Status***, because sense data describing when the command was terminated is created. Recall that this status can only occur when the ***TERMINATE I/O PROCESS*** message is sent by the Initiator.

Okay, we've given you the whole story, and there are a lot of options. In general, however, we can give you a summary that can work for virtually all occasions:

(1) The Target creates the Contingent Allegiance condition, and sense data to describe why, whenever one of the following events occurs:

- CHECK CONDITION status;
- COMMAND TERMINATED status;
- optionally (on the Target's part) Unexpected BUS FREE phase.

(2) After one of the preceding events occurs, the Initiator should attempt to recover sense data by using the REQUEST SENSE command. Issuing a different command or an ABORT message to clear the condition is not recommended. Issuing a BUS DEVICE RESET or bus reset should be done only after attempting to recover sense data.

Control Byte. See ***Command Descriptor Block (CDB)***.

Controller. A Controller is (loosely) a device which takes "Host System Friendly" data and converts it into a form that can be applied to the ***Peripheral Device*** for storage or transport. A Target is usually a Controller. SCSI uses the arbitrary term Target to prevent confusion with the nebulous definition of Controller. A Target has specific functions defined by SCSI, while everyone has their own idea/impression/prejudice of what a Controller is.

Control Signals. The Control Signals comprise nine signals that directly determine the current ***Bus Phase*** and any ***Condition*** that is directly established on the bus. These signals plus the ***Data Bus Signals*** are combined to define the complete ***SCSI Bus***. The Control Signals are:

- ***ACK Signal.*** The Initiator asserts and negates ACK to complete an Information Transfer.
- ***ATN Signal.*** The Initiator asserts ATN to interrupt the Target.
- ***BSY Signal.*** BSY is asserted by a device to hold the bus.
- ***C/D Signal.*** C/D indicates whether Data or other information is currently being transferred on the bus.
- ***I/O Signal.*** I/O indicates the direction of information on the bus.
- ***MSG Signal.*** MSG indicates whether a Message or other information is currently being transferred on the bus.
- ***REQ Signal.*** The Target asserts REQ to begin an Information Transfer.
- ***RST Signal.*** RST is asserted by any device to reset the bus.
- ***SEL Signal.*** SEL is asserted by one device to select another device.

Current I/O Process. The Current I/O Process is simply the ***I/O Process*** associated with the Initiator and Target currently connected on the bus. This term is defined by the SCSI standard to differentiate it from I/O processes that may exist (as queued or active but disconnected I/O Processes), but are not currently connected to the bus. See also ***Active I/O Process***.

Data Bus Signals. The Data Bus signals carry all information transferred on the SCSI bus. Contrast with the ***Control Signals*** that manage ***Connection*** and ***Information Transfer Phases***. Since the ***SCSI Bus*** may be configured to different bus widths (8-bit, 16-bit, or 32-bit), the number of Data Bus signals that are actually used varies. Diagram 19 shows the different bus widths, the data and parity signals, and the cable names associated with them.

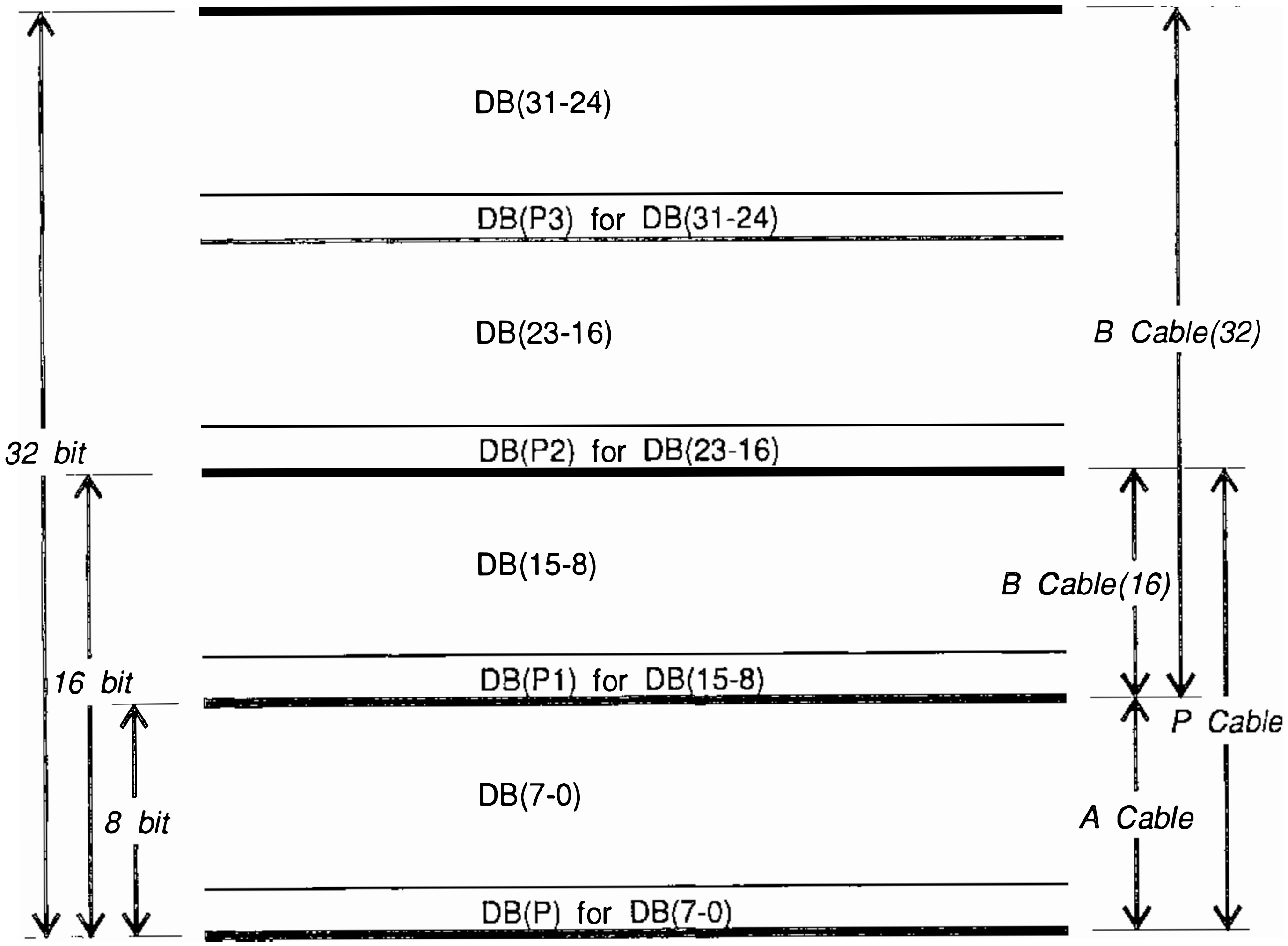


DIAGRAM 19: DATA BUS SIGNALS

DATA IN Phase. A DATA IN Phase is a ***DATA Phase*** in the direction from the Target to the Initiator.

DATA OUT Phase. A DATA OUT Phase is a ***DATA Phase*** in the direction from the Initiator to the Target.

DATA Phase. A DATA phase is defined by the SCSI standard to indicate either a ***DATA IN Phase*** or a ***DATA OUT Phase***. A DATA Phase transfers "system data" or "parameter data" between the Initiator and the Target. It would be good now to review the other types of information transferred between SCSI devices:

- Commands; i.e., the ***Command Descriptor Block (CDB)***.
- Status; i.e., the ***Status*** code.
- Messages, which are used for ***Path Control***.

The DATA Phase is used to transfer all other types of information:

- Logical Block Data; e.g., Data stored, or to be stored, on a disk, tape, or other storage medium.
- Command Parameter Data; e.g., MODE SELECT data.
- Command Response Data; e.g., REQUEST SENSE data.

The actual method for transferring data is either an ***Asynchronous Data Transfer*** or a ***Synchronous Data Transfer***. The Initiator and Target agree on the method of transfer by using ***Synchronous Data Transfer Negotiation*** and ***Wide Data Transfer Negotiation***.

Diagram 20 illustrates the possible phases that may precede and follow a DATA Phase.

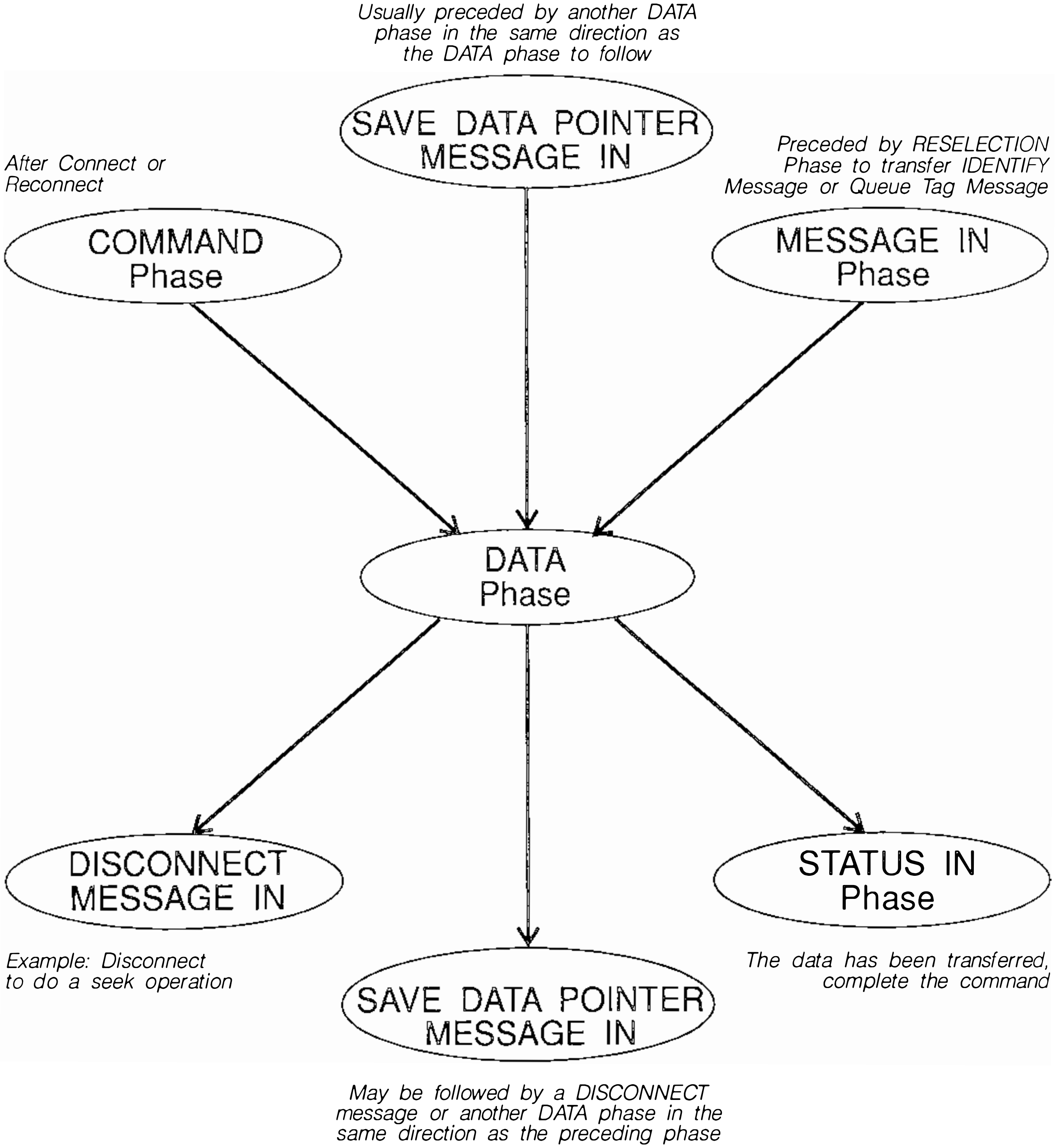

Usually preceded by another DATA phase in the same direction as the DATA phase to follow
SAVE DATA POINTER MESSAGE IN
After Connect or Reconnect
COMMAND Phase
Preceded by RESELECTION Phase to transfer IDENTIFY Message or Queue Tag Message
MESSAGE IN Phase
DATA Phase
DISCONNECT MESSAGE IN
Example: Disconnect to do a seek operation
SAVE DATA POINTER MESSAGE IN
May be followed by a DISCONNECT message or another DATA phase in the same direction as the preceding phase
STATUS IN Phase
The data has been transferred, complete the command

DIAGRAM 20: FLOW INTO AND OUT OF DATA PHASES

Data Pointer. The Data Pointer refers to the source or destination within the Initiator of the information transferred during the ***DATA Phase***. At any time, the Data Pointer refers to a particular byte of the data to be transferred. As with all pointers, there is an ***Active Pointer*** and a ***Saved Pointer***. See ***Pointers*** for the whole story.

Data Release Delay. *tdrd = 400 nsec*. The Data Release Delay is defined as the maximum time that an Initiator may drive the ***Data Bus Signals*** after it detects the transition of the ***I/O Signal*** from false to true. The Data Release Delay is also used to define the minimum time from the I/O transition before the Target may drive the data signals. See ***Between Phases***.

Deassert. See ***Negate***. We don't say "deassert" around here!

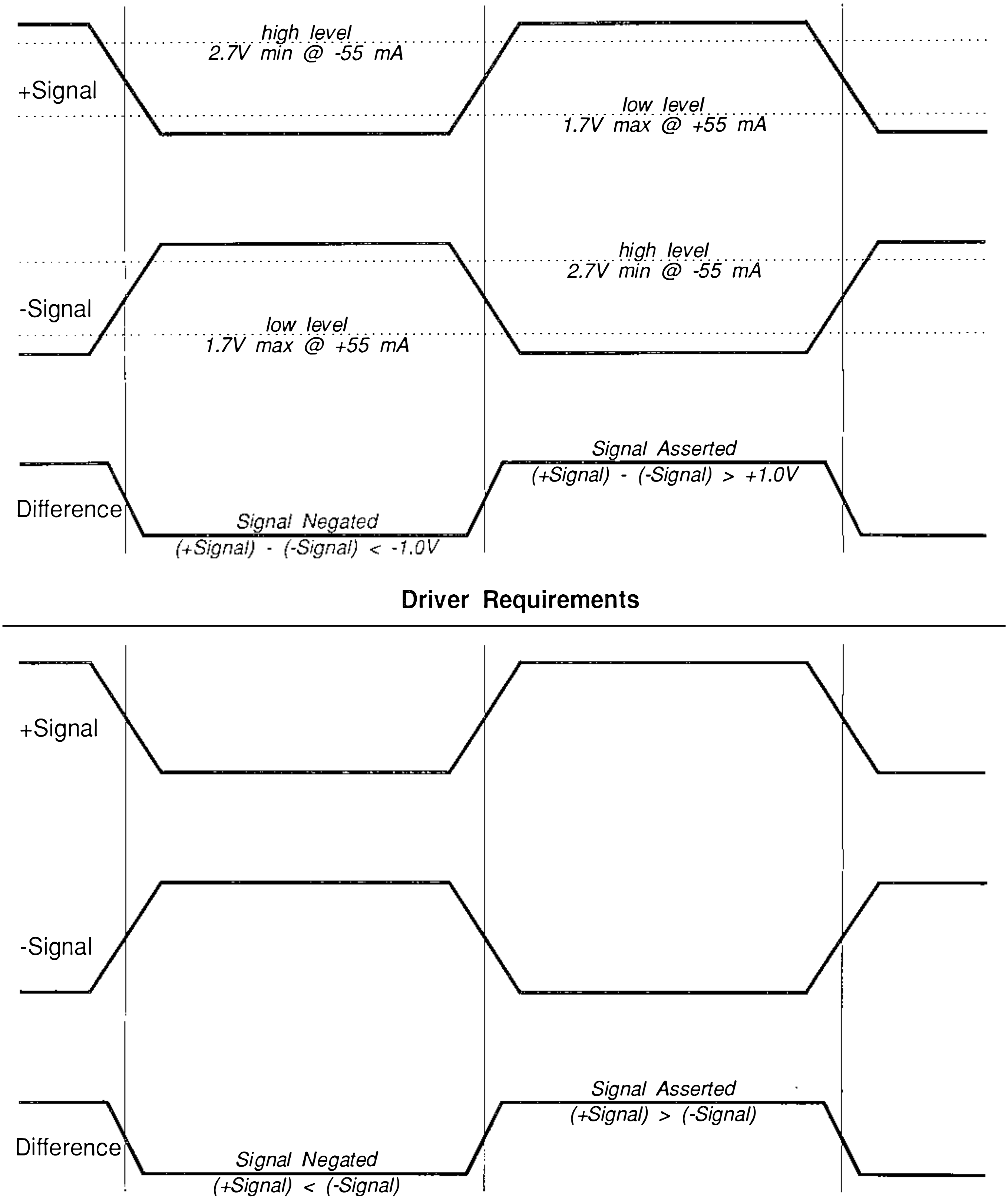
Deskew Delay. *tds = 45 nsec*. The Deskew Delay is a kind of "general purpose" delay that is used to compensate for differences between the drivers and between the receivers within a ***SCSI Device***. Each driver on a device propagates a signal at a different speed than another driver on the same device; the same is true for receivers. This delay is designed such that the differences in propagation speed have no effect in a reasonably well designed device.

The delay of 45 nsec seems large, but it's somewhat historic and still relevant for many cases. The delay was defined in the early days of ***SASI*** and ***SCSI-1***.In those days, all SCSI drivers and receivers were <u>external</u> to the protocol chip or chips (Remember the 7438 and the 74LS240? Then why are you reading this?). The first ***Chip*** with built-in drivers and receivers (the NCR 5380) came out well after the Deskew Delay was defined, which was too late to change it.

Device. See ***SCSI Device***.

Differential Interface. The Differential Interface is one of two electrical interfaces (see also ***Single-Ended 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 Differential Interface is used when signals must be sent over a longer distance.

The Differential Interface transmits the signals using "signal pairs" which are compared against each other to determine the state of the signal. The signal driver drives one wire of the pair to a voltage level greater than the level of the other wire; at least 1.0 Volt higher. The receiver detects this voltage difference and decides the state of the signal. Diagram 21 shows the operating characteristics of the driver at the top of the diagram, and shows the operating characteristics of the receiver at the bottom.



Driver Requirements

Receiver Requirements

DIAGRAM 21: DIFFERENTIAL DRIVER AND RECEIVER VOLTAGES

Other characteristics of the Differential Interface are:

- The current at any input (either +Signal or −Signal) must not be greater than 2.0 mA, or less than −2.0 mA. This must hold for all voltages on the input between −7.0 V and +12.0 V. This must also hold when the device is powered down.

- The capacitance on any input (either +Signal or −Signal) must not exceed 25 pF. Note that this is measured at the connector and includes:

  - Connector capacitance;
  - Printed circuit trace capacitance;
  - Differential transceiver chip capacitance.

- The ***Terminator*** biases each signal pair when ***Released*** as follows:

  - The −Signal is biased between 3.5 V and 2.1 V.
  - The +Signal is biased between 2.4 V and 1.4 V.

  The variation in bias voltage is due to resistor tolerances at the Terminator, and is due to variation in the ***Terminator Voltage***. Despite these variations, the bias on each signal tends to track, so that the difference between the −Signal and the +Signal remains greater than or equal to 1.0 V.

The advantages of the Differential Interface are:

- Cable length (25 meters) is longer than the Single-Ended Interface (6 meters). Use the Differential Interface when the cable must leave a cabinet for a total cable length longer than 6 meters.

- Signal quality is better than the Single-Ended Interface. The differential method is virtually immune to noise and crosstalk induced in the cable, since the noise affects both signal pairs (almost) equally. Use the Differential Interface when the environment is "hostile".

The disadvantages of the Differential Interface are:

- The Differential Interface uses more power than the Single-Ended Interface. Since each signal is actually a pair, there are twice as many signals to drive. Also, each driver of the pair must sink 25% more current than a Single-Ended driver.

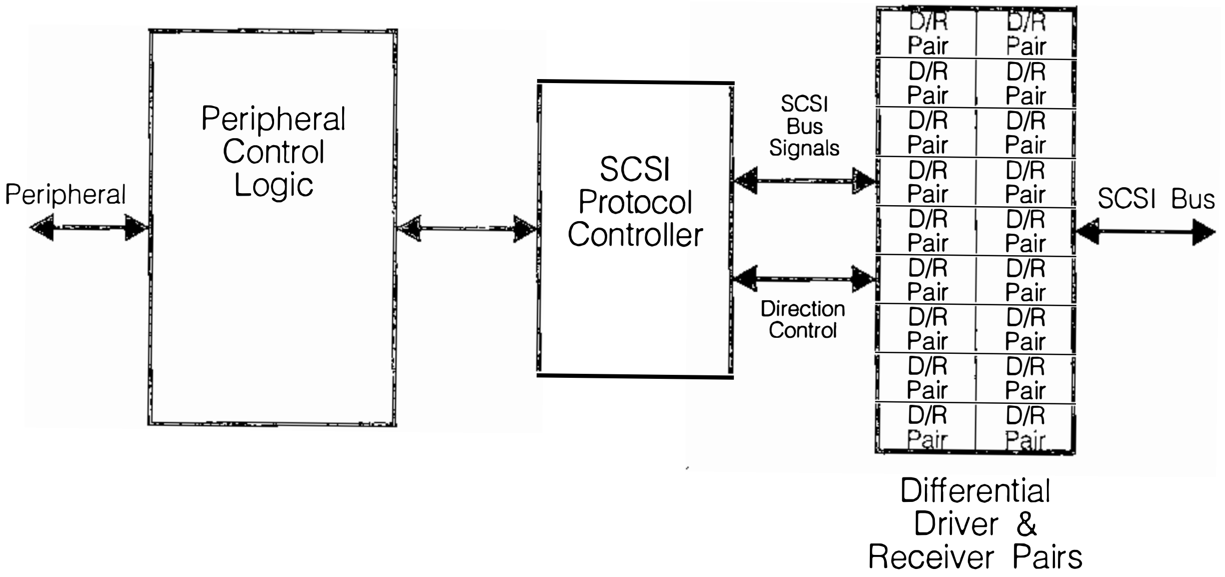
- The Differential Interface dissipates more power than Single-Ended. An 8-bit Differential driver set will dissipate up to about 4 watts, while an 8-bit Single-Ended driver set dissipates less than 1 watt.

- As of this writing, a Differential Interface costs several dollars more to implement than a Single-Ended Interface.

- A Differential Interface is more costly in terms of circuit board space as well, since the drivers and receivers are very difficult to integrate. Until recently, a single 8-pin DIP package was required for each driver/receiver pair, for a total of 18 packages for an 8-bit SCSI bus, not counting the protocol controller. This is illustrated in Diagram 22 for both Target and Initiator implementations. (The drivers and receivers for a protocol controller for a Single-Ended Interface are usually included in the device.)

  As of this writing, device manufacturers have been able to integrate three driver/receiver pairs in a single package, reducing the board space required somewhat.

Be sure to read the next section on ***DIFFSENS***.



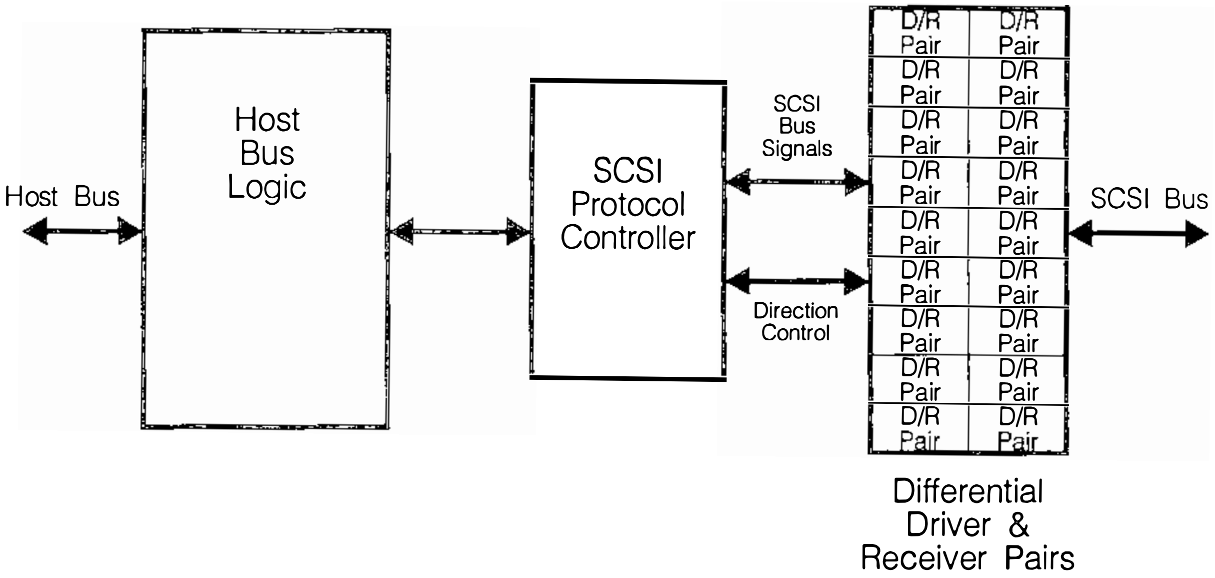


DIAGRAM 22: DIFFERENTIAL IMPLEMENTATION

DIFFSENS. AKA "Differential Sense". This signal is unique to the ***Differential Interface***. The purpose of DIFFSENS is to enable or disable the Differential drivers onto the SCSI Bus:

- When DIFFSENS is high, all Differential drivers are enabled. DIFFSENS is high whenever the bus is populated only with ***SCSI Devices*** that support the Differential Interface.

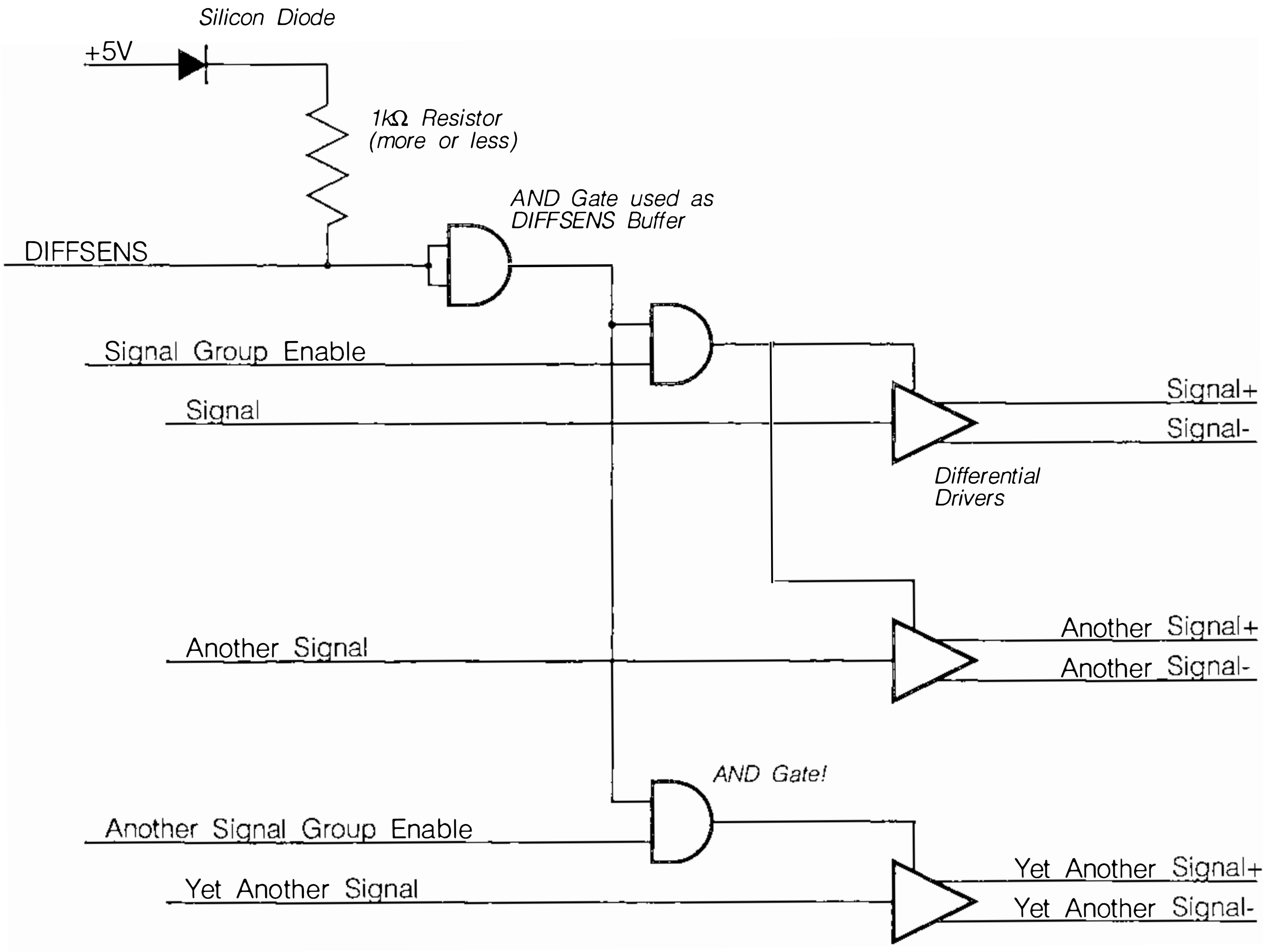
- When DIFFSENS is low, all Differential drivers on each Device on the bus are disabled. DIFFSENS is low whenever there is one or more SCSI Devices that support the ***Single-Ended Interface*** plugged into the bus. Examine the ***Connector*** pin assignments and you'll note that the pin assigned to DIFFSENS on the Differential Interface is attached to GROUND on the Single-Ended Interface.

That's nice, but what's it for? Back in the early days of ***SCSI-1***, everyone was gearing up to build devices for each interface; each controller would have one version for Single-Ended, and another for Differential. These products would therefore look fairly similar to each other. There was a concern that a well-meaning systems support person would at some point plug a Differential board into a Single-Ended bus, or vice versa.

The DIFFSENS signal was created to prevent damage to the devices when this situation occurs. If you are in an environment that only supports the Single-Ended Interface, this seems like a remote possibility. On the other hand, if you do Differential, then there is a pretty good possibility that one of your users is going to try and plug in that extra tape unit "just to see if it works". Therefore, it is particularly incumbent on the designer of a device for the Differential Interface to properly support the DIFFSENS signal.

Diagram 23 shows an example of the DIFFSENS circuit. The "pull up" resistor ensures that the DIFFSENS signal is high when no Single-Ended devices are installed on the bus. The diode protects the device power supply against back flow current from other devices bussed to the same signal. The DIFFSENS signal should be bufferred before distributing within the device.

DIAGRAM 23: DIFFSENS IMPLEMENTATION



Disconnect. "To break the ***Connection*** of". A Disconnect occurs when a SCSI device releases control of the SCSI bus, allowing it to go to ***BUS FREE Phase***. This release of control can occur in any of the following ways:

- The Target releases BSY after a ***MESSAGE Phase***, where a message is exchanged between the Initiator and Target that calls for a BUS FREE Phase after the message transfer has been completed. See ***BUS FREE Phase*** for a complete list of Messages that are followed by BUS FREE Phase.

- The Target releases BSY any other time. This indicates a catastrophic failure. This is called an ***Unexpected BUS FREE Phase***.

- The Initiator releases the ***SEL Signal*** during a ***SELECTION Phase***. The Initiator will do this as a result of a ***Selection Timeout***.

- The Target releases the ***SEL Signal*** during a ***RESELECTION Phase***. The Target will do this as a result of a ***Reselection Timeout***.

Disconnection Delay. *tdd = 200 µsec.* The Disconnection Delay is only used when the Target accepts ***DISCONNECT Messages*** from the Initiator. After the Target accepts the DISCONNECT message from the Initiator, it then disconnects from the bus by returning a DISCONNECT message and going to ***BUS FREE Phase***. After this occurs the Target must wait a Disconnection Delay before attempting to arbitrate for the bus. The timing for this sequence is illustrated under DISCONNECT message, below.

DISCONNECT Message. The DISCONNECT message is sent by either the Target or the Initiator to cause a break of the ***Connection*** between the devices. DISCONNECT is a single byte message:

Bit Byte	7	6	5	4	3	2	1	0
0	Message Code = 04 hex							

- The Target sends the DISCONNECT message to indicate to the Initiator that it intends to go to ***BUS FREE Phase*** and will ***Reconnect*** to continue the ***I/O Process*** with the Initiator at a later time. The Target goes to BUS FREE Phase as soon as possible after the trailing edge (transition from ***True*** to ***False***) of the ***ACK Signal***.

- The Initiator sends the DISCONNECT message to the Target to request that the Target disconnect from the bus. The Target then decides when (usually, as soon as possible) it will ***Disconnect*** as described in (1) above. After the Target disconnects, it is required to not participate in ***ARBITRATION Phase*** for a period of time (a ***Disconnection Delay***).

The DISCONNECT message should be used whenever it is "reasonable" for the Target to disconnect from the bus. As you might imagine, "reasonable" is in the eye of the beholder. Examples of "reasonable" can include:

- There is no information to be transferred on the bus. If the Target has nothing to "say", leaving the bus busy but with no activity, it should release the bus so that another device can use it.

- The Target can continue without interaction with the Initiator. Typically, the Target contains a RAM buffer where it can hold data before transfer to the Initiator (READ data) or after transfer from the Initiator (WRITE data). With this buffer, the Target can fill it or empty it without connection to the bus. The Target then connects to the Initiator long enough to "burst transfer" the next buffer full of data on the SCSI Bus.

- The Initiator can ask the Target to disconnect as well. If the Initiator has a higher-priority task to perform, or runs into a resource contention problem, it can request the Target to disconnect from the bus. The Initiator then can attempt to ***Connect*** to the bus to perform the new task.

Summary of Use: The DISCONNECT message is sent by a Target to indicate that it will go to BUS FREE Phase and reconnect later. The DISCONNECT message is sent by an Initiator to request that the Target issue a DISCONNECT message and go to BUS FREE Phase and reconnect later.

This page is nearly blank!

ECA.
Error Recovery.
Etiquette.
Extended Contingent Allegiance (ECA) Condition
Extended Messages.

ECA. See ***Extended Contingent Allegiance***.

Error Recovery. There are lots of ways to recover from errors in SCSI, depending on what you are doing. Let's summarize them here:

- ***BUS FREE Phase*** and ***ARBITRATION Phase*** have no errors from which to recover.

- ***SELECTION Phase*** and ***RESELECTION Phase*** handle errors by going to ***BUS FREE Phase***.

- The ***Message System*** (***MESSAGE IN Phase*** and ***MESSAGE OUT Phase***) handles its own recovery.

Error Recovery for each of the above are covered within those topics. There is one more:

- All of the other ***Information Transfer Phases*** (***DATA IN Phase***, ***DATA OUT Phase***, ***COMMAND Phase***, and ***STATUS Phase***) use the ***Path Control*** portion of the Message System to handle error recovery.

We will discuss this type of Error Recovery here. Before tackling this subject, we suggest first reading ***Message System*** and ***Pointers***.

We should first define "Error" for our purposes. An Error is any failure to maintain the integrity of the data between a data source within the sending ***SCSI Device*** and the data destination within the receiving SCSI Device. Diagram 24 shows the data path between two devices and where errors can occur and be detected.

In this diagram, we assume that for all types of information transferred, there is some form of "safe" Local Storage within the device. This is assuredly true for ***SCSI Commands*** and ***Status*** (they may even originate from read-only memory). It may not be true for Data, which may pass straight through to or from its ultimate destination or source (e.g., the storage medium or Host memory). Even in this case, the model still applies; at some point the data is either safe, or beyond the control of the device.

The point of all this is to illustrate <u>when</u> SCSI Error Recovery is necessary. When data leaves its safe place at the source, it is in danger of being corrupted, and therefore the destination device should be able to detect when corruption occurs. When data is safe at the destination, the SCSI Device has it and doesn't need to get it again from the source device.

In summary, it is appropriate to use SCSI Error Recovery when data gets corrupted between the safe data source and the safe data destination, whether or not the corruption actually occurs on the SCSI Bus.

DIAGRAM 24: DATA PATHS AND ERROR SOURCES

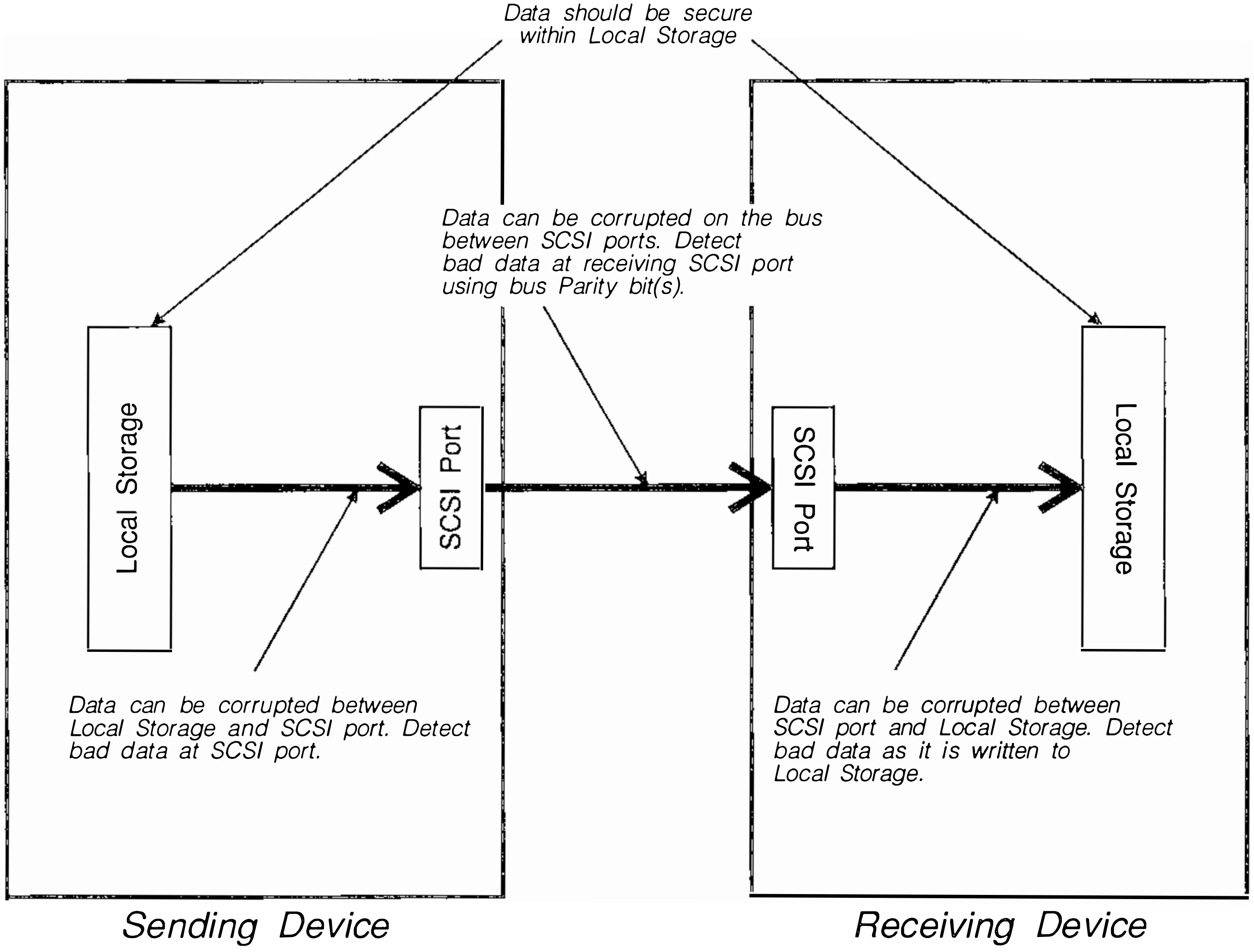

Data should be secure within Local Storage
Data can be corrupted on the bus between SCSI ports. Detect bad data at receiving SCSI port using bus Parity bit(s).
Local Storage
SCSI Port
SCSI Port
Local Storage
Data can be corrupted between Local Storage and SCSI port. Detect bad data at SCSI port.
Data can be corrupted between SCSI port and Local Storage. Detect bad data as it is written to Local Storage.
Sending Device
Receiving Device

Diagram 25 shows what happens when a Target detects an error that requires a retry. During an ***Information Transfer Phase*** (Command, Status, or Data), one of the following errors occurs:

- SCSI ***Parity*** Error during DATA OUT Phase or COMMAND Phase.

- An error is detected within the Target between the SCSI Port and the "Safe" data source or destination.

When the error is detected, the Target changes to MESSAGE IN Phase from the original Phase in which the error occurred. The Target sends (only) the ***RESTORE POINTERS Message*** during the MESSAGE IN Phase. If the Initiator accepts the Message, the Target returns to the original Phase and repeats the transfer. See ***Pointers*** to understand exactly where the repeated transfer begins.

If the Initiator does not accept the RESTORE POINTERS Message, and does not give the Target any other guidance in response (see Message System), then the only thing for the Target to do is go to an ***Unexpected BUS FREE Phase***.

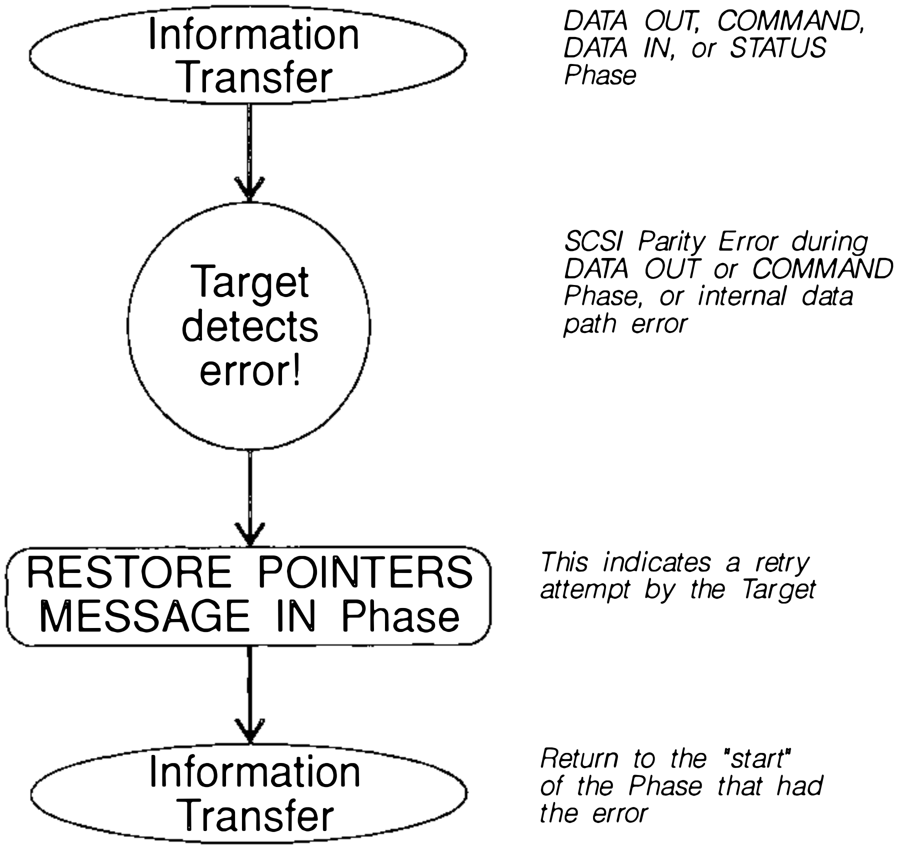

Information Transfer
DATA OUT, COMMAND, DATA IN, or STATUS Phase
Target detects error!
SCSI Parity Error during DATA OUT or COMMAND Phase, or internal data path error
RESTORE POINTERS MESSAGE IN Phase
This indicates a retry attempt by the Target
Information Transfer
Return to the "start" of the Phase that had the error

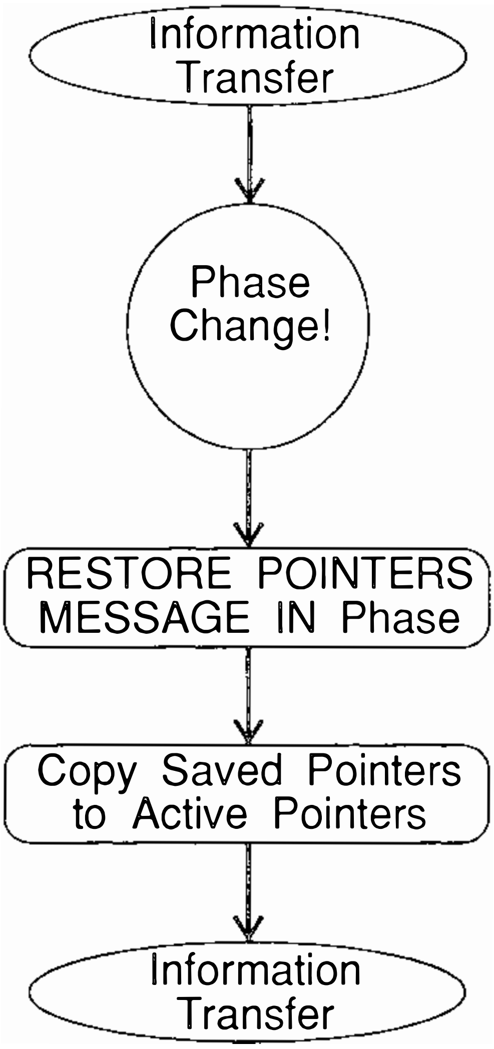
DIAGRAM 25: TARGET DETECTED ERROR

Diagram 26 shows the Initiator response to a Target request for retry. During the Information Transfer, the Initiator sees an unexpected phase change. At this point, the Initiator has no idea why there was a phase change.

The Target changes to MESSAGE IN Phase, so the Initiator dutifully receives the Message from the Target, which is the RESTORE POINTERS Message. As described under ***Pointers***, the Initiator accepts the Message and copies the ***Saved Pointers*** to the ***Active Pointers***. The Target then returns back to the original Information Transfer Phase to repeat the transfer.

The Initiator may choose to reject the RESTORE POINTERS Message if it feels that a retry is inappropriate or impossible. An ***ABORT Message*** or ***ABORT TAG Message*** from the Initiator would then be the best thing to do. (If you are using ***Tagged Queuing***, remember that the ABORT Message affects more than the current I/O Process.)

If the Target does not return to the original Phase, the Initiator may have to assume the Target is crazy and ABORT the ***I/O Process***.


Information Transfer
Phase Change!
RESTORE POINTERS MESSAGE IN Phase
Copy Saved Pointers to Active Pointers
Information Transfer

DATA OUT, COMMAND, DATA IN, or STATUS Phase

Suddenly, the Target changes Phase

This indicates a retry attempt by the Target

Return to the "start" of the Phase that had the error

The Target should then return to the prior Information Transfer Phase, and resend the same data

DIAGRAM 26: INIT. RESPONSE TO TARGET DETECTED ERROR

Diagram 27 shows what happens when an Initiator detects an error that requires a retry. During an Information Transfer Phase (Command, Status, or Data), one of the following errors occurs:

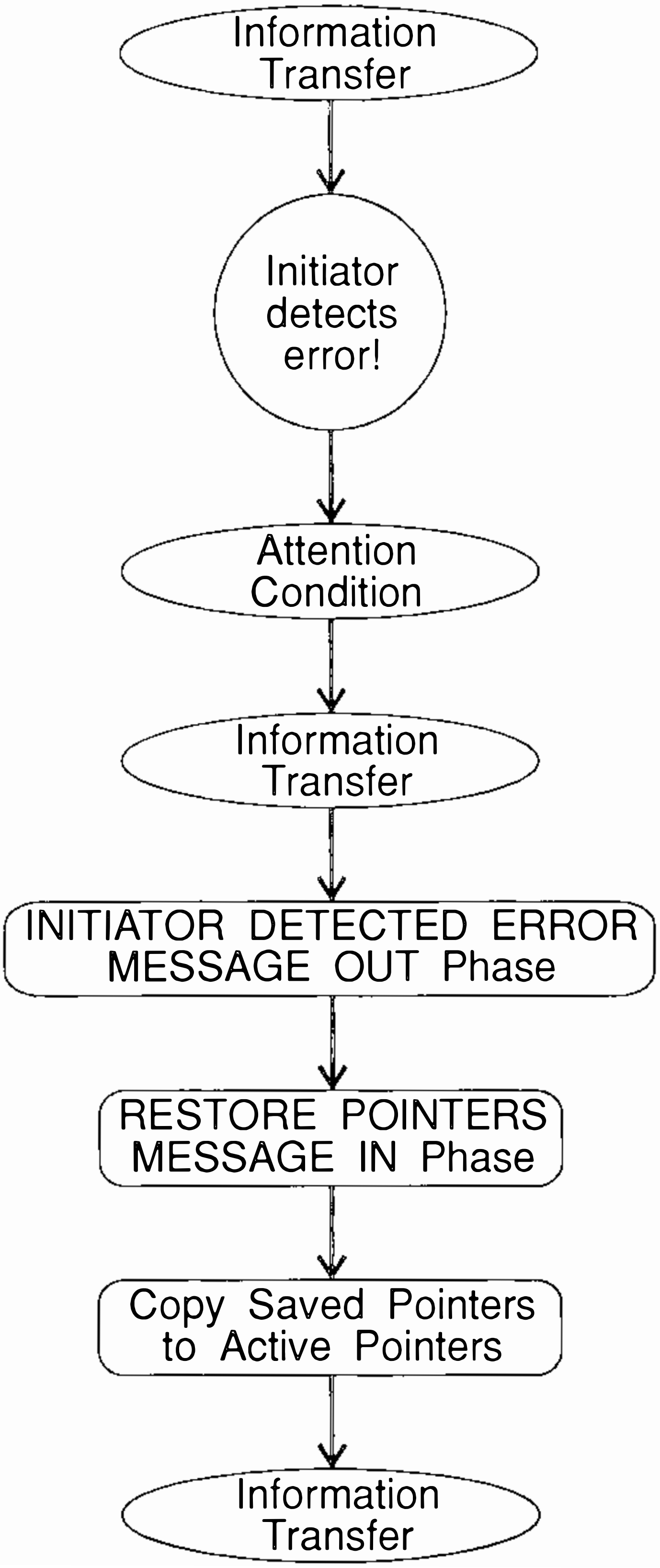
- SCSI Parity Error during DATA IN Phase or STATUS Phase.
- An error is detected within the Initiator between the SCSI Port and the "Safe" data source or destination.

When the error is detected, the Initiator creates the ***Attention Condition***, after which the Initiator continues transferring in the original Phase until the Target responds to the Attention Condition. The Target acknowledges the Attention Condition by changing to MESSAGE OUT Phase immediately following the original Phase in which the error occurred. The Initiator sends (only) the ***INITIATOR DETECTED ERROR Message*** during the MESSAGE OUT Phase.

When the Target accepts the Message from the Initiator, the Target changes to MESSAGE IN Phase and sends (only) the RESTORE POINTERS Message. If the Initiator accepts the Message, the Target returns to the original Phase and repeats the transfer with the same data as the first transfer attempt. See ***Pointers*** to understand exactly where the repeated transfer begins.

If the Target does not accept the INITIATOR DETECTED ERROR Message (by rejecting it), the Initiator should probably then issue the ***ABORT Message*** or ***ABORT TAG Message*** to end the ***I/O Process***.

If the Target does not respond to the Attention Condition, or does not return to the original Phase, the Initiator may have to assume the Target is crazy and ABORT the I/O Process.


Information Transfer
Initiator detects error!
Attention Condition
Information Transfer
INITIATOR DETECTED ERROR MESSAGE OUT Phase
RESTORE POINTERS MESSAGE IN Phase
Copy Saved Pointers to Active Pointers
Information Transfer

DATA OUT, COMMAND, DATA IN, or STATUS Phase

SCSI Parity Error during DATA IN or STATUS Phase, or internal data path error

Created by the Initiator so it can send a Message to the Target

Continue in the Phase that had the error until the Target recognizes the Attention Condition

This indicates a retry request by the Initiator

This indicates a retry attempt by the Target

Return to the "start" of the Phase that had the error

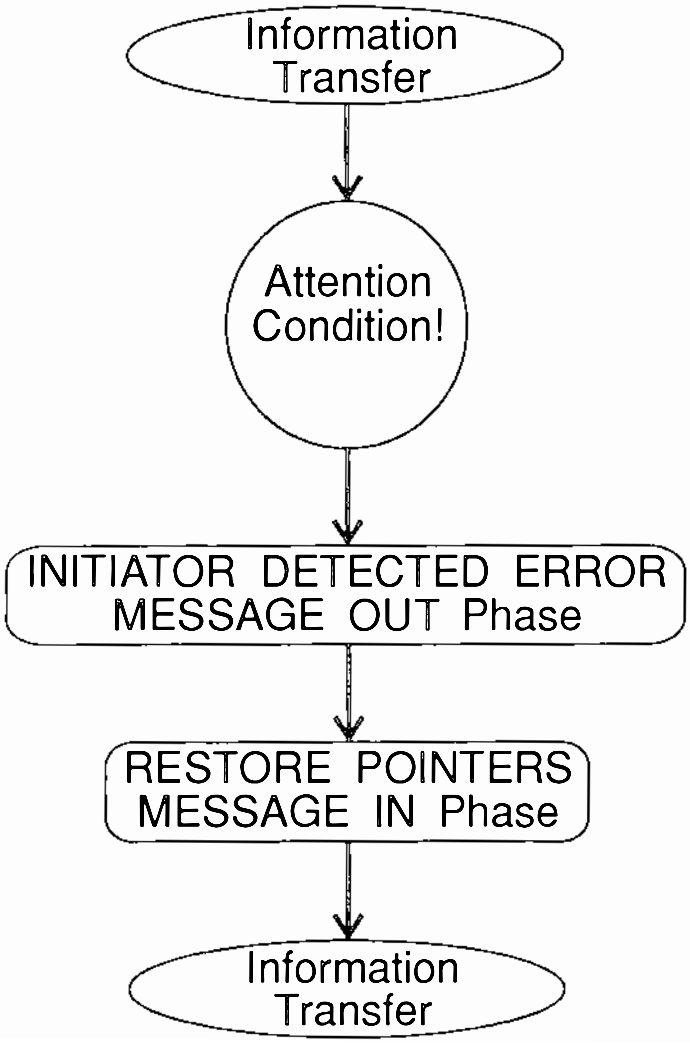
The Target should return to the prior Information Transfer Phase. The Initiator resends the same data.

DIAGRAM 27: INITIATOR DETECTED ERROR

Diagram 28 shows the Target response to an Initiator request for retry. During the Information Transfer, the Target sees the Initiator create the Attention Condition. At this point, the Target has no idea why.

The Target changes to MESSAGE OUT Phase, and the Initiator dutifully sends the INITIATOR DETECTED ERROR Message. The Target accepts the message, changes to MESSAGE IN Phase, and the Initiator receives the Message from the Target, which is the RESTORE POINTERS Message. As described under Pointers, the Initiator accepts the Message and copies the ***Saved Pointers*** to the ***Active Pointers***. The Target then returns back to the original Information Transfer Phase to repeat the transfer.

The Initiator should not reject the RESTORE POINTERS Message; if it feels that a retry is inappropriate or impossible, it should not have sent the INITIATOR DETECTED ERROR Message. If the Initiator does not accept the RESTORE POINTERS Message, and does not give the Target any other guidance in response (see Message System), then the only thing for the Target to do is go to an Unexpected BUS FREE Phase.


Information Transfer
Attention Condition!
INITIATOR DETECTED ERROR MESSAGE OUT Phase
RESTORE POINTERS MESSAGE IN Phase
Information Transfer

DATA OUT, COMMAND, DATA IN, or STATUS Phase

The Initiator tries to get the Target's "attention" to send a Message

This indicates a retry request by the Initiator

This indicates a retry attempt by the Target

Return to the "start" of the Phase that had the error

DIAGRAM 28: TARGET RESPONSE TO INIT. DETECTED ERROR

Etiquette. You are developing a SCSI product. You spends months in design; you spend months in the lab verifying the design. You ship your first samples to customers. Three days later, a customer calls and complains, "Your lousy widget doesn't work!" You find out what the failure mode is and with whose product they connected to your prototype. You call the vendor of that product and describe the failure mode.

And they reply: "Oh. We didn't implement it that way."

It won't do any good to point out your interpretation of the standard. It won't do any good to point out that they failed to implement a mandatory feature. Your only option (since they got their product out first) is to change yours to conform.

Anyone who has any experience implementing or just using SCSI is too familiar with this scenario. A classic example occurred in recent years where devices at both ends of the cable (Initiators and Targets) failed to implement ***Synchronous Data Transfer Negotiation*** correctly. Initiators couldn't handle a Target that wanted to start a negotiation, and some Targets were incapable of starting a negotiation. These situations could easily cause a system lockup, yet many products were shipped this way. Products that conformed to that standard could not operate properly with non-conforming devices.

What has happened before cannot be repaired unless a vendor decides to make the effort, but perhaps the future can be saved from some repeat occurrences. This section, then, is my attempt at being the "Miss Manners" for SCSI (I know I'm going to take a lot of ribbing for this one). Maybe if enough designers read this section, the world will be a better place...

SCSI Etiquette #1: "Just say NO to deviations from the standard!" From our experience, this is easier said than done. Just do your best to convince your prospect that compliance is desirable (even if it does ruin your salesman's appetite over dinner!).

SCSI Etiquette #2: Don't use a ***Vendor Specific*** method to implement a function when a standard method can do the same job. You may have a "more efficient" method for doing the function because of your particular product architecture. In the long run, your customers will be happier to have the standard method (Note: It is OK to provide the Vendor Specific method as an addition for those customers who want to be non-standard).

SCSI Etiquette #3: CCS (Common Command Set) is dead. It's Dead. Dead Dead Dead. Don't cripple your design by following the CCS document, which is loaded with inconsistencies and contradictions. Use the full features of ***SCSI-2***, which is a superset of CCS.

SCSI Etiquette #4: Use standard ***Connectors*** on the devices you manufacture or specify. Sure, there are some VERY LARGE COMPANIES that use non-standard Connectors, but you don't have to. First, there are lots of adapter cables between those non-standard connectors and the standard ones. You'll always be able to connect to those products. Second, you'll be able to connect to all of the other nice products that use standard connectors.

SCSI Etiquette #5: And make sure the ***SCSI Address*** selector is accessible and easy-to-use too! (The first manufacturers who used thumbwheel switches earned many thanks from their customers.)

SCSI Etiquette #6: Speaking of Connectors, put those ***Terminators*** where people can get to them. Sometimes, that tape drive or scanner just has to go on that shelf over there, or worse, has to be moved. Don't make the poor system integrator disassemble the whole box just to change a ***Terminator Power*** jumper or move a resistor pack.

SCSI Etiquette #7: If you are designing a Target, and you think you may need to use the ***IGNORE WIDE RESIDUE Message***, look again. In almost every case, you should be able to size your data blocks to be a multiple of the bus width. For example, if you are designing a 16-bit Target, don't make the INQUIRY Command data block 17 bytes long! Pad it to 18 bytes and save yourself some trouble. In fact, pad it to 20 bytes to allow for a 32-bit upgrade. Even if you are designing an 8-bit Target, a little foresight now can save you much grief later. The only time you should need IGNORE WIDE RESIDUE is when you have an oddball block size, or you are trying to be backward compatible.

SCSI Etiquette #7A: Okay, okay. If you must use the IGNORE WIDE RESIDUE Message, do so only on the last ***DATA IN Phase*** of the command. If the data is broken up into several ***Connections***, ***Disconnect*** only on bus-width boundaries.

SCSI Etiquette #8: A ***Bus Phase*** is not established until the ***REQ Signal*** is asserted. If you try to anticipate Bus Phases, re-qualify them after REQ is asserted.

SCSI Etiquette #9: For most normal systems, there is no real reason to implement ***Asynchronous Event Notification (AEN)***. If you think you might want to use it, think REALLY REALLY hard about simpler alternatives. If you must use AEN, know your reasons (and there are good reasons sometimes!), and only use it on Initiators that have already spoken to you. In fact use it only on Initiators that have enabled AEN via the MODE SELECT command. Don't go looking for them, because searching for them is more trouble than it's worth.

SCSI Etiquette #10: The same for AEN goes for the ***Extended Contingent Allegiance (ECA) Condition***. On the other hand, a well-designed Initiator should receive the ***INITIATE RECOVERY Message*** successfully, even if its only response is the ***RELEASE RECOVERY Message***.

SCSI Etiquette #11: Stay away from the ***RST Signal***! It's such a hostile thing to do, because when you assert the RST Signal, everybody has to pick up the pieces. Before trying to reset a device using RST, try to issue the ***BUS DEVICE RESET Message***, which limits its effect to one device.

SCSI Etiquette #12: Targets, please respond to the ***ATN Signal*** in a reasonable manner. Most of the response table at the end of the ***Attention Condition*** is right on, but we can think of a couple of adjustments. First, service ATN during a ***DATA Phase*** on a ***Logical Block*** boundary. If it's a short data transfer, it's OK to wait until the end of the transfer. It's also OK to wait for the end of a ***COMMAND Phase*** to service ATN. NO ONE needs to be able to recognize which Data or Command byte had a problem.

SCSI Etiquette #13: Use quality ***Cables***. Assign the conductors according to the recommendation of the ***X3T9.2 Committee***. Install the cables properly. And mind your stub length and spacing!

SCSI Etiquette #14: Use the ***P Cable*** for ***Wide Data Transfer***. Forget the ***B Cable***.

SCSI Etiquette #15: Initiators, don't try to use the ***Saved Pointers*** as an indicator of the number of bytes actually transferred in a Bus Phase. One of two things will happen: either it won't work because the Target ***Disconnected*** at the end of the DATA Phase and ***Reconnected*** to send ***Status***; or, the Target will have to use extra ***SAVE DATA POINTERS Messages*** to make it work for you. This just screws up performance. Find another way to do it, like special hardware, or forget it. BUT, a well designed Initiator will never overstep the memory allocated to it by the Host, so keep that in mind.

SCSI Etiquette #16: Always Disconnect if you have nothing to transfer on the SCSI Bus at the moment. Good times to Disconnect include during a disk seek, a tape positioning, or a data buffer empty or full condition (i.e., when the SCSI Bus is much faster than the peripheral data rate). Caution: Disconnecting too often can hurt performance. Be sure that your dead time between transfers is longer than the overhead of Disconnecting and Reconnecting.

SCSI Etiquette #17: Initiators, be ready for any Bus Phase! An Initiator should be able to properly respond correctly to any Bus Phase during any particular Connection or Reconnection. We know of Initiators that can only transfer a ***Command Descriptor Block (CDB)*** one time. Let's say, during an ***Initial Connection***, the Target takes the Messages and Command in from the Initiator and decides to ***Queue*** the Command for later and Disconnect. However, the Target has no room to store the Command bytes and discards them. Later, the Target Reconnects to the Initiator and then goes to COMMAND Phase to re-fetch the CDB. An Initiator should be able to handle this.

SCSI Etiquette #18: Retry Once. In other words, perform ***Error Recovery*** once during any Connection. This includes retries within the ***Message System*** and other Bus Phases using the ***RESTORE POINTERS Message***. If the bus is so bad that a second retry is called for (even for a different Phase), it is better to just give up. Giving up means going to an ***Unexpected BUS FREE Phase***.

SCSI Etiquette #19: As indicated in the introduction to this section, ***Synchronous Data Transfer Negotiation (SDTN)*** is everybody's responsibility! For Initiators, SDTN should be done as part your ***Initialization*** procedure. For Targets, SDTN should be done after anything that causes a ***Hard Reset***. This goes for ***Wide Data Transfer Negotiation (WDTN)*** as well.

SCSI Etiquette #20: Having said #19, we have to admit there are certain rude SCSI Devices out there that don't know the rules regarding SDTN. Admittedly, many were built that way by demand of customers, but didja have to sell them to everybody else? Sigh. The sad fact is that you will probably have to deal with these non-standard devices. Devices will have to have a jumper or something to disable SDTN. (Frankly, most Initiators available at publication time can't handle a Target originating SDTN).

SCSI Etiquette #21: Initiators have a bigger problem with SDTN. If a Target cannot originate SDTN, it can get "out of sync" (pun intended) with the Initiator. For example, if the power cycles on a Target, the Initiator won't know it until it starts a new I/O Process with the Target. When the Target attempts to return Sense Data describing a ***Unit Attention Condition***, it will use ***Asynchronous Data Transfer***, but the Initiator uses ***Synchronous Data Transfer***. Big problem. The Initiator can protect itself from this by always originating SDTN when issuing REQUEST SENSE or INQUIRY Commands. Note that this is not a performance hit; these commands are just not used enough to matter.

SCSI Etiquette #22: If you are going to supply ***Terminator Power***, do it well or don't do it at all! Get something out on the cable that is as high as possible without exceeding 5.25 Volts.

SCSI Etiquette #23: Do yourself and your users a favor and use the "preferred" ***Terminator*** for the ***Single-Ended Interface***. You'll have a much cleaner bus, and it also reduces the importance of having a high Terminator Power voltage. If you can only put a "preferred" Terminator at one end of the Cable (with the "old-style" at the other end), it is still a great improvement.

SCSI Etiquette #24: Just set the ***Reserved*** bits and fields to zero. This goes for Initiators and Targets! Initiators, when receiving something with Reserved bits and/or fields, mask them to zero before evaluating. Saves a lot of trouble later...

SCSI Etiquette #25: This should go without saying, but don't invent any ***Vendor Specific*** Messages. Chances are really good that the function you want either exists as a standard Message, or should really be defined using a ***Command Descriptor Block (CDB)***.

SCSI Etiquette #26: For you ASIC (Application Specific Integrated Circuit) designers out there, design the Single-Ended drivers for the ***REQ Signal*** and the ***ACK Signal*** so that they are actively ***Negated***. This ensures that these signals are driven to their maximum voltage level, improving the noise margin. Note that the drivers are "tri-state", so that they are only driven at the appropriate times.

SCSI Etiquette #27: Another one for ASIC designers is to center the Single-Ended receiver threshold voltage, and to increase the receiver hysteresis from 200 mV to 400 mV minimum. This increases the tolerance of the receiver to noisy signals.

SCSI Etiquette #28: Initiators, if you assert the ***ATN Signal***, leave it asserted until the Target responds with the ***MESSAGE OUT Phase***. If you have nothing to say at that time, send the ***NOP Message***. Taking away the ATN Signal early is a little like taking away an interrupt signal to your microprocessor before it can respond; it responds, but nothing is there! If you are careful, you can probably make things work, but it's safer to do this right at the outset.

SCSI Etiquette #29: Can't figure something out? Need a feature that doesn't seem to be there? Don't fake it! Call on the ***X3T9.2 Committee*** to help. They just might be able to show you how to use a tool that already exists, or they just might add the feature you need.

Extended Contingent Allegiance (ECA) Condition. Big words! Actually, this is just a prolonged ("Extended") version of the ***Contingent Allegiance Condition***. If that's no help, go read it (again), or recall that the Contingent Allegiance Condition is the state that exists between an Initiator and a Target-plus-***Logical Unit*** whenever a failure/error/exception occurs during an ***I/O Process***. The Contingent Allegiance Condition ends when the Initiator recovers (or discards) the Sense Data from the Target. After the Sense Data is transferred, the Target is free to continue normal processing of other commands.

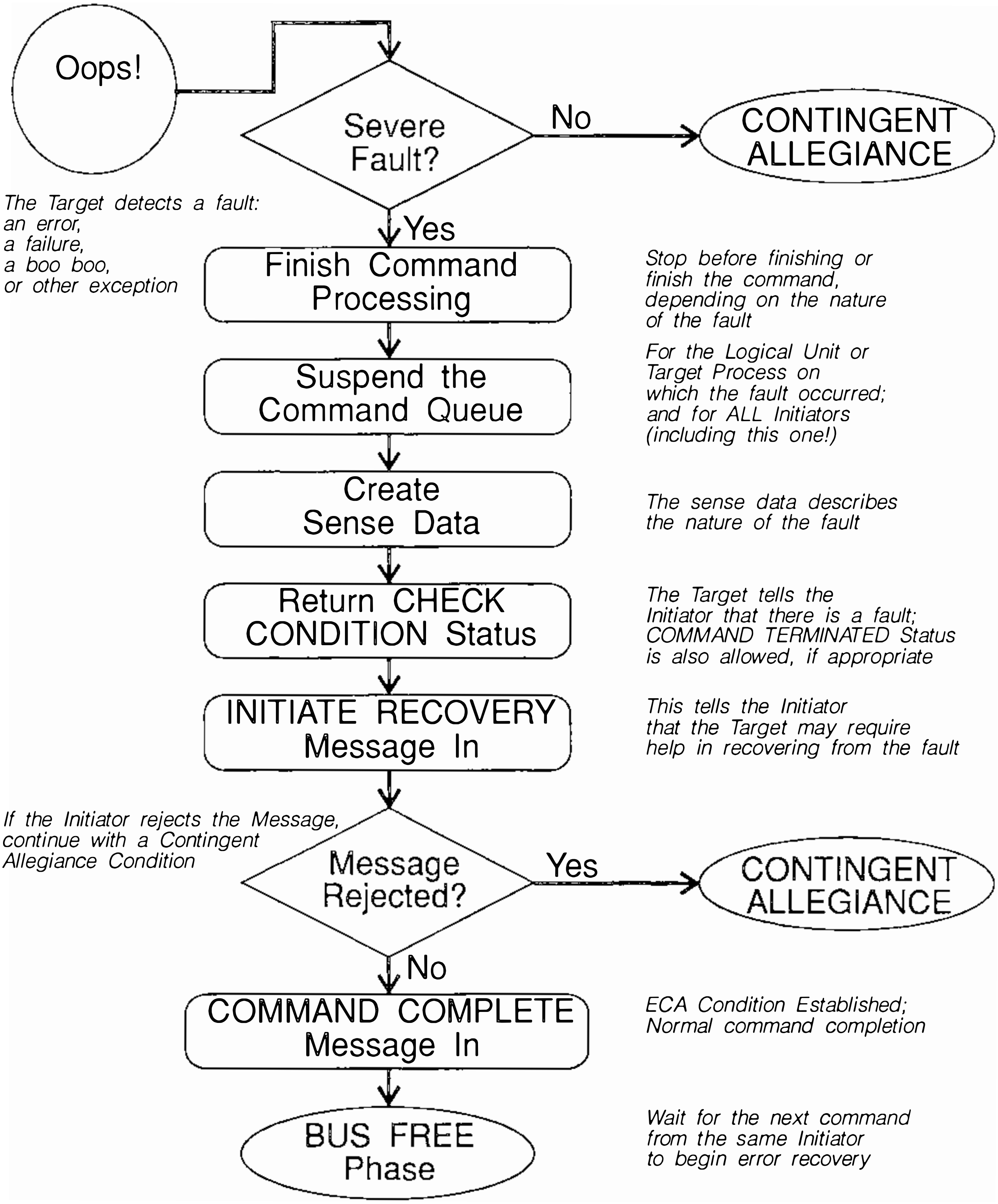
This is fine for many situations (in this writer's opinion, nearly all situations), but in some cases it may not be a good idea to allow the Target to continue normal processing after the failure. The problem with Contingent Allegiance is that the Initiator may not be able to make the decision about whether to continue until after it has examined the Sense Data. After the Sense Data is examined, the Initiator may need to issue some special ***SCSI Commands*** or ***Messages*** to recover the Target from the failure. After the recovery, the Initiator would then release the Target for further processing.

The previous paragraph describes what Extended Contingent Allegiance achieves. To prevent fatigue, I'll be referring to it as **ECA** from now on...

There is a catch to all this: The Target decides to begin the ECA Condition, not the Initiator. As a result, the Target must have some idea about which failures/errors/exceptions requires the Initiator's help to perform recovery. Diagram 29 shows how the Target begins the ECA Condition. It is almost the same as a normal Contingent Allegiance Condition, except that the Target returns the ***INITIATE RECOVERY Message*** prior to returning the ***COMMAND COMPLETE Message*** at the end of the command.

Once the ECA Condition is established, the Target waits for the Initiator to perform one or more recovery ***I/O Processes*** on it. (We say "I/O Process" since the recovery can include both command and Message oriented operations.) While this is going on, the Target will accept I/O Processes for the Logical Unit with the active ECA Condition only from the Initiator with which it established the ECA Condition. These I/O Processes are issued to the I_T_x ***Nexus***, even if the fault occurred during an I/O process with an I_T_L_Q Nexus.

All other Initiators are given BUSY ***Status*** until the ECA Condition is ended. Further, the ***Queue*** for that Logical Unit is suspended for the duration; no I/O Processes are activated from the Queue until the ECA Condition has ended.


Oops!
Severe Fault?
No
CONTINGENT ALLEGIANCE
Yes
Finish Command Processing
Suspend the Command Queue
Create Sense Data
Return CHECK CONDITION Status
INITIATE RECOVERY Message In
Message Rejected?
Yes
CONTINGENT ALLEGIANCE
No
COMMAND COMPLETE Message In
BUS FREE Phase

The Target detects a fault:
an error,
a failure,
a boo boo,
or other exception

Stop before finishing or finish the command, depending on the nature of the fault

For the Logical Unit or Target Process on which the fault occurred; and for ALL Initiators (including this one!)

The sense data describes the nature of the fault

The Target tells the Initiator that there is a fault; COMMAND TERMINATED Status is also allowed, if appropriate

This tells the Initiator that the Target may require help in recovering from the fault

If the Initiator rejects the Message, continue with a Contingent Allegiance Condition

ECA Condition Established; Normal command completion

Wait for the next command from the same Initiator to begin error recovery

Note: Until the ECA Condition is cleared, all other Initiators will receive BUSY Status from the Target for this Logical Unit!

DIAGRAM 29: TARGET START OF ECA CONDITION

Diagram 30 shows how the Initiator handles the ECA Condition. The Initiator sees that the INITIATE RECOVERY Message has been sent by the Target. Its next step is to find out what the problem is by issuing a REQUEST SENSE Command to recover the Sense Data. By examining the Sense Data, the Initiator can determine whether any recovery is needed at all.

If recovery is needed, the Initiator issues one or more I/O Processes to perform that recovery. While the exact sequence is very device-specific, the I/O Processes might include:

- ***ABORT, ABORT TAG, or CLEAR QUEUE Message***
- Medium Positioning Commands (like REWIND)
- Medium Replacement Commands (like LOAD/UNLOAD)
- RECOVER BUFFERED DATA Command
- LOG Commands
- MODE Commands
- READ or WRITE retries
- DIAGNOSTIC Commands

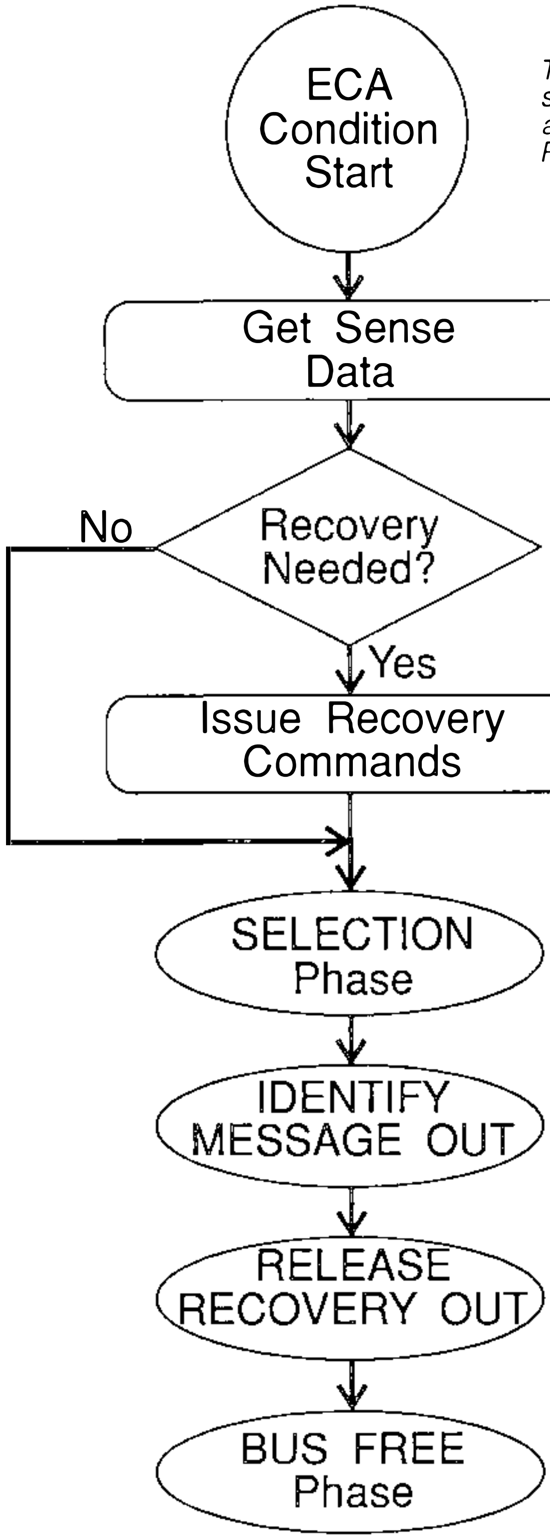
One example might be a tape device that has a large buffer. The tape device receives telemetry from another device to its buffer. If a fault occurs that precludes the data from being recorded on the tape, it is wise to retrieve that data to a safe place for recovery so that the data is not lost. In this case, the Target creates the ECA Condition. After the REQUEST SENSE Command, the Initiator issues the RECOVER BUFFERED DATA Command to retrieve the data. It does whatever else it needs to do to complete the recovery and then releases the ECA Condition.

It should be noted that the recovery commands issued must be "non-queued"; they must be issued without a ***Queue Tag Message***. The Target will reject any queued commands from the Initiator (see below).

To end the ECA Condition, the Initiator issues a special I/O Process. After ***SELECTION Phase***, the Initiator sends the ***IDENTIFY Message*** out, followed in the same phase by the ***RELEASE RECOVERY Message*** out. After receiving the Messages, Target goes to ***BUS FREE Phase***, ending the ECA Condition.

The Initiator may also end the ECA Condition before it starts by responding to the INITIATE RECOVERY Message with the ***MESSAGE REJECT Message***. Presumably, the Target then falls back to a Contingent Allegiance Condition.

The ECA Condition may also be ended by a ***BUS DEVICE RESET Message*** from any Initiator, or a ***Hard Reset Condition***.



The Target detects a fault suitable for the ECA Condition and returns the INITIATE RECOVERY Message

The Initiator issues the REQUEST SENSE Command to recover the Sense Data

The Initiator decides whether or not any recovery is really needed

The Initiator issues one or more I/O Processes to handle the recovery on the Target

Selecting the Target with the ECA Condition

Sent for the LUN with the ECA Condition during the first MESSAGE OUT Phase after Selection

Sent after the IDENTIFY during the same MESSAGE OUT Phase

End of ECA Condition

DIAGRAM 30: INITIATOR HANDLING OF ECA CONDITION

Diagram 31 shows how the ECA Condition is ended at the Target. The Target goes into a "limited" command processing mode on that Logical Unit (note that other Logical Units are not bound by the ECA Condition):

- If an I/O Process is received from any Initiator other than the Initiator that has the active ECA Condition, the Target returns BUSY Status.

- If an I/O Process is received from the Initiator that has the ECA Condition, and that I/O Process has a ***Queue Tag***, then the Target returns QUEUE FULL Status.

- If the initial ***MESSAGE OUT Phase*** of the I/O Process ends with a RELEASE RECOVERY Message, end the ECA Condition, and go to BUS FREE Phase.

- Otherwise, execute the recovery I/O Process.

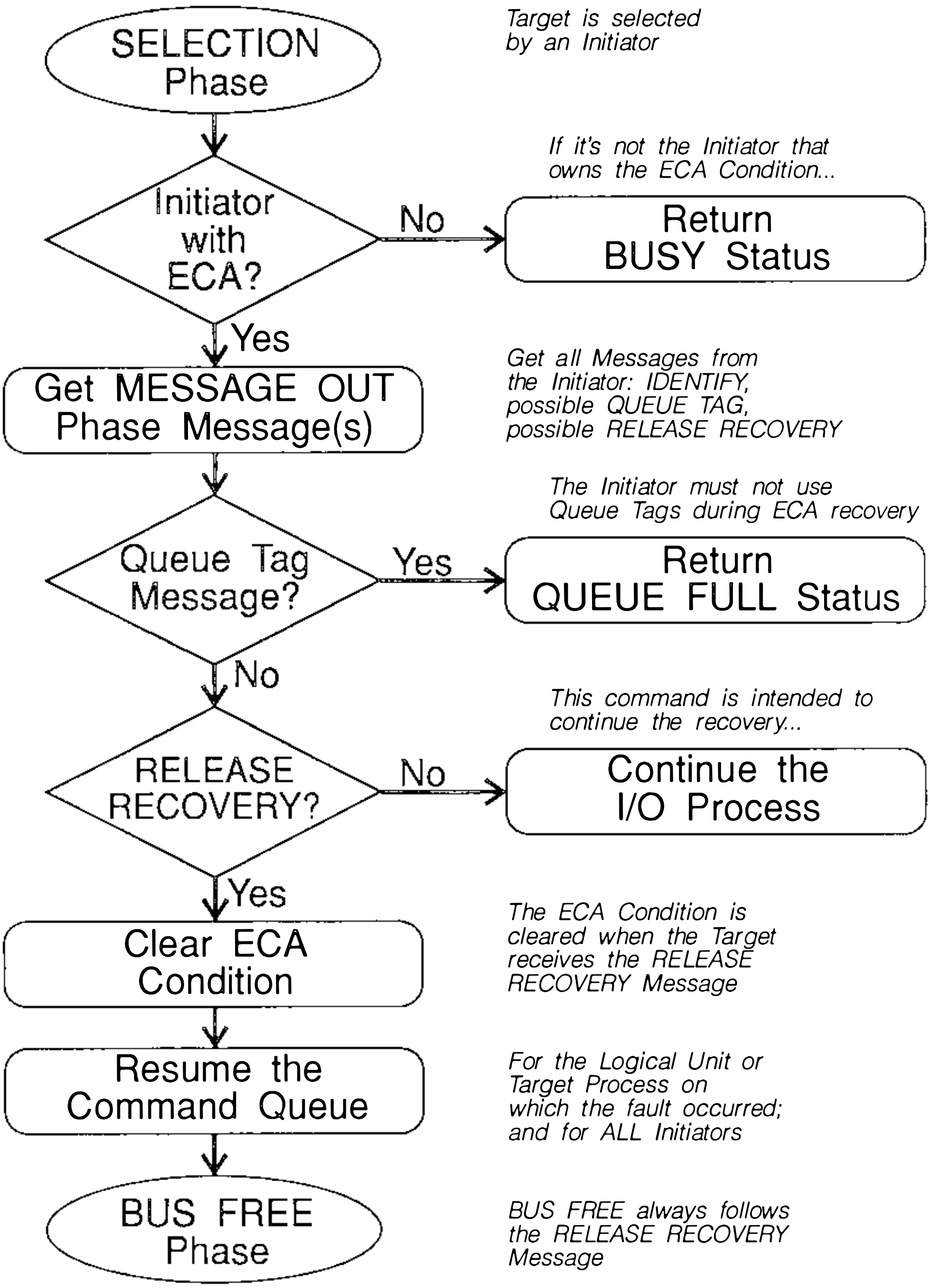
A SCSI Device may begin an ECA Condition via ***Asynchronous Event Notification (AEN)***. We suggest you read that section if you are interested. To begin the ECA Condition via AEN, the "Initiating Device" sends the INITIATE RECOVERY Message at the end of the initial MESSAGE OUT Phase, before the ***COMMAND Phase***.

...and now the question:

Why? Frankly, ECA has limited use for traditional "low-end" systems, such as an embedded SCSI disk drive. A well designed Target of this type will perform its own recovery, and will be well behaved enough to preserve diagnostic information through most fault conditions. Also, ECA is only needed in systems with more than one Initiator. With one Initiator, the "lockout" of other Initiators by the ECA Condition is not necessary.

ECA finds a home in fault tolerant systems where data integrity and recovery is a system function. In a small desktop system, data recovery is often (sadly) left to the user; "Abort, Retry, Fail". ECA will probably find its way into such devices as disk drive arrays.

Currently, ECA is most commonly used for some Sequential Access Devices (Tape Drives) to allow the Initiator to recover data in the buffer, reposition the tape to a "safe" place, or to replace the tape, before allowing other Initiators to access the device. Even in this case, there must be agreement on which error conditions in the Target can trigger an ECA Condition.



Target is selected by an Initiator

If it's not the Initiator that owns the ECA Condition...

Get all Messages from the Initiator: IDENTIFY, possible QUEUE TAG, possible RELEASE RECOVERY

The Initiator must not use Queue Tags during ECA recovery

This command is intended to continue the recovery...

The ECA Condition is cleared when the Target receives the RELEASE RECOVERY Message

For the Logical Unit or Target Process on which the fault occurred; and for ALL Initiators

BUS FREE always follows the RELEASE RECOVERY Message

DIAGRAM 31: TARGET END OF ECA CONDITION

Extended Messages. Extended messages are simply ***Messages*** that are longer than one or two bytes. Think of them as "Long Messages" if it helps. Extended messages are of varied lengths, though each specific message has a fixed length. Extended messages have a common byte form shown in Figure 24.

Bit Byte	7	6	5	4	3	2	1	0
0	Message Code = 01h							
1	Additional Message Length = mm							
2	Extended Message Code = xx							
3	Parameter Byte 1							
...	.....							
mm+1	Parameter Byte mm-1							

FIGURE 24: EXTENDED MESSAGE GENERIC FORMAT

The first byte is a code (01 hex) that indicates an Extended message. This is an indication that the rest of the message should be interpreted in the standard Extended format. The second byte gives the length of the message, which is the number of bytes in the message starting with the third byte; in other words, the number of remaining bytes in the message. This will always be at least 01 hex, since the extended code itself must be sent.

The third byte is the extended message code. This indicates what action the message is expected to cause, and provides a context for interpreting the parameter bytes. As with other messages, think of the extended messages as "commands" for path control. Table 19 is a complete list of all Extended messages currently defined.

TABLE 19: EXTENDED MESSAGE CODES AND LENGTHS

Additional Message Length	Extended Message Code	Total Message Length	Message Name
05 hex bytes	00 hex	7 bytes	MODIFY DATA POINTER
03 hex bytes	01 hex	5 bytes	SYNCHRONOUS DATA TRANSFER REQUEST
02 hex bytes	03 hex	4 bytes	WIDE DATA TRANSFER REQUEST
not defined	02 hex	not defined	not used
not defined	04 hex - 7F hex	not defined	Reserved
Vendor Defined	80 hex - FF hex	Vendor Defined	Available for Vendor Unique Messages

See each individual message topic for a description of the parameters for the messages.

This page is nearly blank!

False.
Fast Assertion Period.
Fast Cable Skew Delay.
Fast Data Transfer.
Fast Deskew Delay.
Fast Hold Time.
Fast Negation Period.
Forced Perfect Termination (FPT).

False. "Not ***True***", of course! In SCSI, a signal that is ***Negated*** or ***Released*** is in the False state. The definition of Negated and Released is specific to the Interface type, either ***Single-Ended*** or ***Differential***. The following are equivalent, from a logical point of view, in SCSI:

- "0"
- Zero
- False
- Negated
- Released

Fast Assertion Period. *tfast = 30 nsec.* The Fast Assertion Period is the minimum specification of the ***Assertion Period*** when using ***Fast Data Transfer***.

Fast Cable Skew Delay. *tfcsd = 5 nsec.* The Fast Cable Skew Delay is the minimum specification of the ***Cable Skew Delay*** when using ***Fast Data Transfer***.

Fast Data Transfer. Fast Data Transfer is just a variation of ***Synchronous Data Transfer*** that allows a higher data transfer rate. A normal Synchronous Data Transfer has a minimum ***Transfer Period*** of 200 nsec; the maximum transfer rate is 5 Mega-transfers per second. When using Fast Data Transfer timing, the minimum Transfer Period drops down to 100 nsec; the maximum transfer rate is then 10 Mega-transfers per second.

Fast Data Transfer timing is enabled during ***Synchronous Data Transfer Negotiation (SDTN)***. When two devices agree to a Transfer Period less than 200 nsec, Fast Data Transfer is enabled.

Table 20 shows how the timing changes between normal and Fast Data Transfer.

TABLE 20: FAST DATA TRANSFER TIMING

Timing Parameter	Normal Sync Data Transfer (Transfer Period ≥ 200 nsec)	Fast Data Transfer (Transfer Period < 200 nsec)
SCSI Defined Timing		
Assertion Period	90 nsec minimum	30 nsec minimum
Cable Skew Delay	10 nsec minimum	5 nsec minimum
Deskew Delay	45 nsec minimum	20 nsec minimum
Hold Time	45 nsec minimum	10 nsec minimum
Negation Period	90 nsec minimum	30 nsec minimum
Derived Timing		
Data Valid to Rising Edge of REQ or ACK (Deskew Delay + Cable Skew Delay)	55 nsec minimum	25 nsec minimum
Rising Edge of REQ or ACK to Data Invalid (Deskew Delay + Cable Skew Delay + Hold Time)	100 nsec minimum	35 nsec minimum

Figure 25 and Figure 26 show Synchronous Data Transfer with Fast Data Transfer timing applied. See ***Synchronous Data Transfer*** for a description of the events (and the names!) in the timing diagrams.

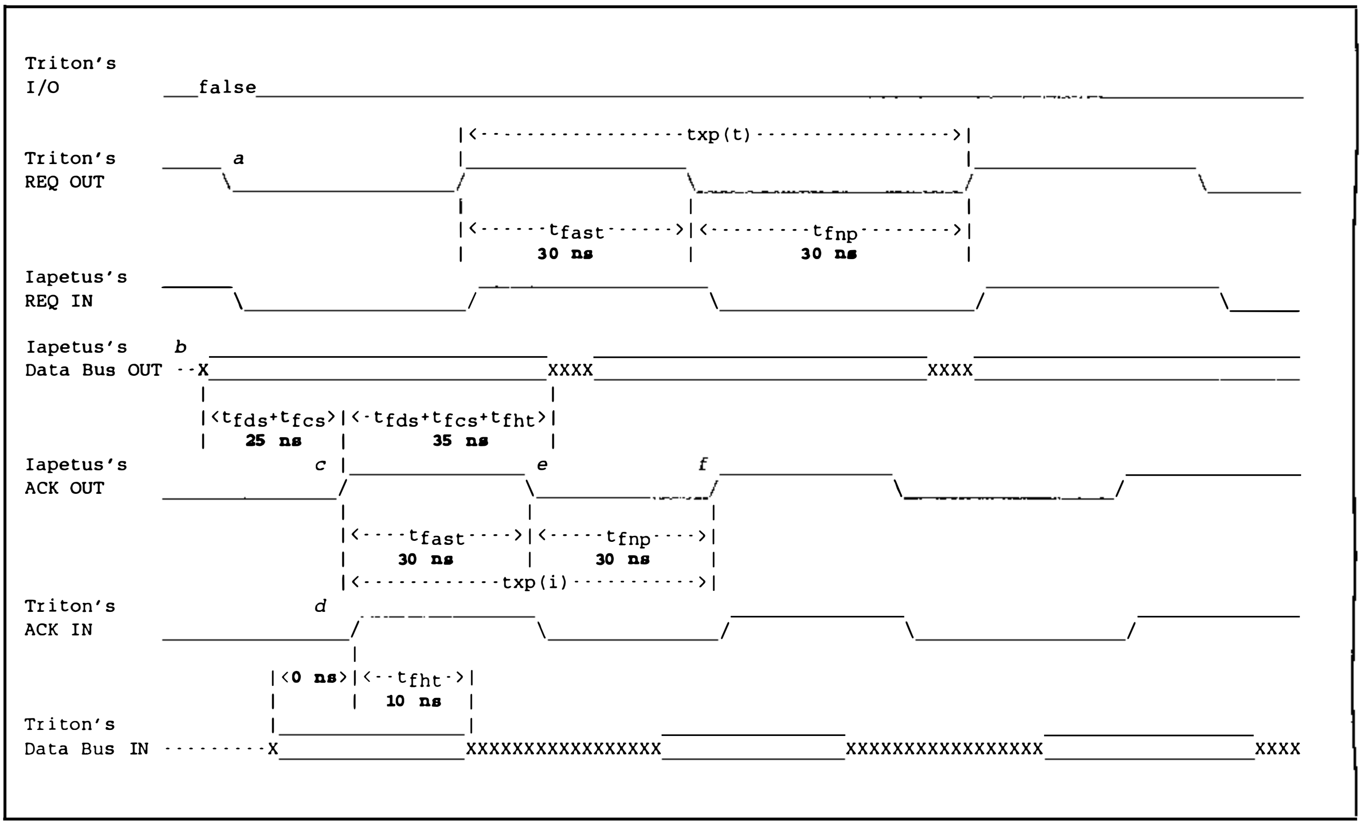


FIGURE 25: FAST DATA TRANSFER OUT - INITIATOR TO TARGET

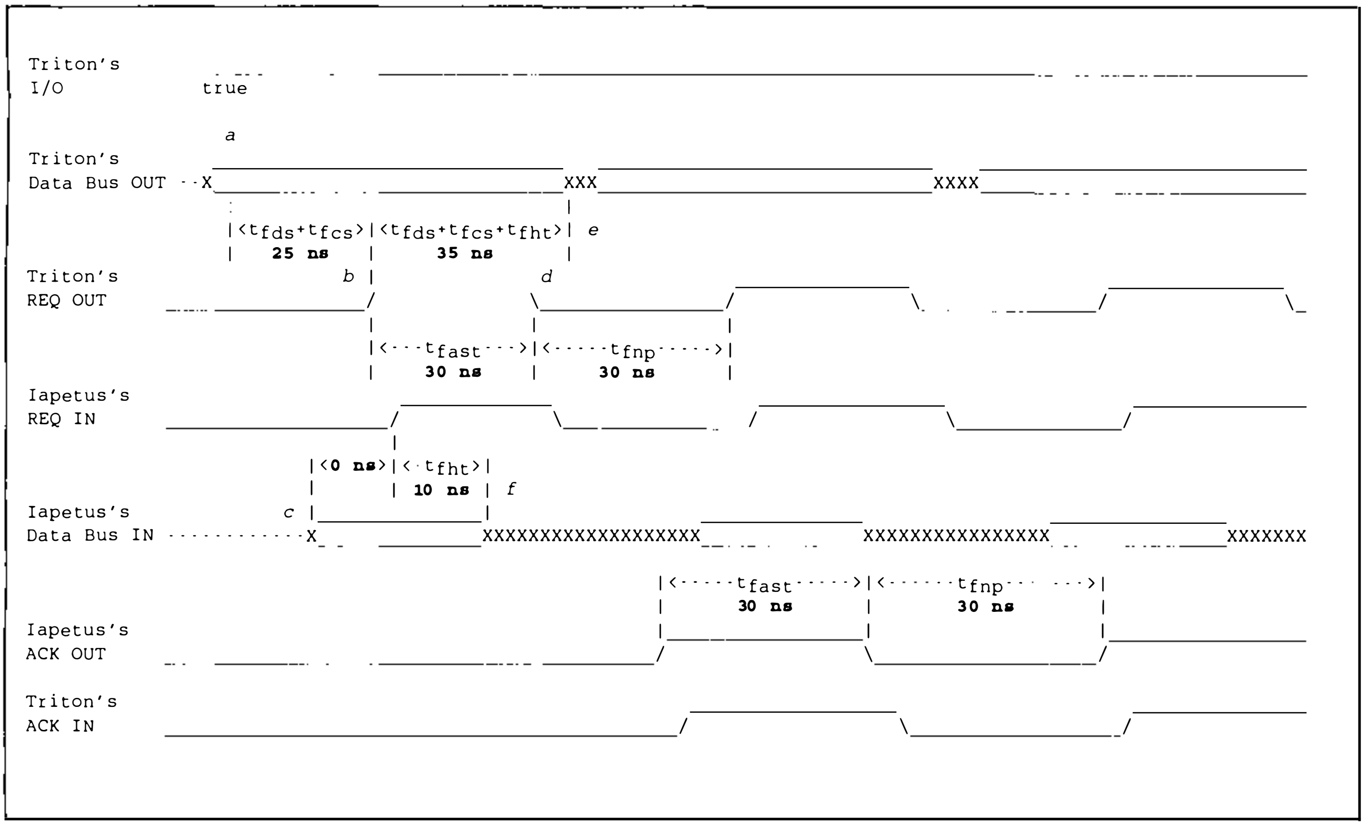


FIGURE 26: FAST DATA TRANSFER IN - TARGET TO INITIATOR

TABLE 21: TIMING VALUES USED DURING FAST DATA TRANSFER

Symbol	Timing Name	MIN or MAX?	Time
t_{fast}	Fast Assertion Period	MINIMUM	30 nsec
t_{fds}	Fast Deskew Delay	MINIMUM	20 nsec
t_{fcs}	Fast Cable Skew Delay	MAX/MIN	5 nsec
t_{fht}	Fast Hold Time	MINIMUM	10 nsec
t_{fnp}	Fast Negation Period	MINIMUM	30 nsec

Fast Deskew Delay. *tfds = 20 nsec.* The Fast Deskew Delay is the minimum specification of the ***Deskew Delay*** when using ***Fast Data Transfer***.

Fast Hold Time. *tfht = 10 nsec.* The Fast Hold Time is the minimum specification of the ***Hold Time*** when using ***Fast Data Transfer***.

Fast Negation Period. *tfnp = 30 nsec.* The Fast Negation Period is the minimum specification of the ***Negation Period*** when using ***Fast Data Transfer***.

Forced Perfect Termination (FPT). FPT is a ***Termination*** method recently proposed to the ***X3T9.2 Committee***. This terminator attempts to achieve good signal quality by first forcing a termination mismatch with the ***Cable***, and then providing additional circuitry (switching diodes) to cancel the effects of the mismatch. The intent is to eliminate the need to closely match the Cable to the Termination.

The committee is considering FPT for inclusion in ***SCSI-3***. While it looks promising, as of this writing it has been shown to be sensitive to low ***Terminator Power*** levels, so proceed with caution. Hybrids with the "preferred" Single-Ended Terminator are being examined to reduce this sensitivity. Also, there may be problems when using FPT with a device whose drivers implement ***Active Pull-ups***. Contact X3T9.2 for the latest scoop.

Hard Reset.
HEAD OF QUEUE TAG Message.
Hold Time.
Host Adapter.

Hard Reset. A Hard Reset is one of two possible responses that a ***SCSI Device*** can make to a ***Reset Condition***; the other response is ***Soft Reset***, appropriately enough. A device must respond to the Reset Condition with either a Hard Reset OR a Soft Reset; it may not mix parts of each. Note that the only appropriate response to a ***BUS DEVICE RESET Message*** is a Hard Reset.

A Hard Reset is analogous to a "Power-On Reset": the device goes back to an initialized state and remembers no previous history that was not held in non-volatile storage. Let's get specific:

- All ***I/O Processes*** that are executing or ***Queued*** are cleared and discarded. No ***Status*** or Sense Data is created; i.e., no ***Contingent Allegiance Condition*** is created. Forget them, they're gone.

- All Reservations are cleared and discarded. The same rules as for I/O Processes apply here; i.e., forget them. We cover Reservations in the other volumes of this Encyclopedia.

- Any other pending conditions are cleared:

  - ***Attention Condition***
  - Contingent Allegiance Conditions
  - ***Extended Contingent Allegiance Conditions (ECA)***
  - ***Unit Attention Conditions***

- All pointers currently maintained by Initiators are discarded.

- Any bus transactions that were ongoing at the instant of the reset are discarded.

- All operating modes are returned to their initial conditions:

  - The data transfer mode becomes 8-bit ***Asynchronous Data Transfer*** until a new ***Synchronous Data Transfer Negotiation (SDTN)*** and (possibly) a new ***Wide Data Transfer Negotiation (WDTN)*** can be performed.

  - All modes specified by a MODE SELECT Command (see next volume!) are returned to their saved or default state, as appropriate.

  - Any other modes unique to the device return to their default state.

- The ***SCSI ID*** or ***SCSI Address*** does not change. This value should be hardwired or stored in a non-volatile manner in the device.

After the device does everything listed above, if it is principally an Initiator, it does nothing more, except of course to start issuing the appropriate commands to re-initialize and continue the operation of the system.

If the device is principally a Target, then it creates the Unit Attention Condition for all other SCSI Devices. This is done to alert the other devices to the fact that the Target has been reset.

HEAD OF QUEUE TAG Message. The HEAD OF QUEUE TAG Message is used by an Initiator to get an ***I/O Process*** executed as soon as possible when a ***Queue*** is implemented by the Target. HEAD OF QUEUE TAG is a two byte message:

Bit Byte	7	6	5	4	3	2	1	0
0	Message Code = 21 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 front of the Queue, ahead of any other I/O Processes that currently may be queued. It does <u>not</u> cause any I/O Process that is currently being executed to be aborted. That command must be completed (one way or another, see next paragraph) before the new I/O Process is executed. If no I/O Process is currently active for that ***Logical Unit***, then the new I/O Process is executed immediately.

If the new I/O Process must be started absolutely immediately, the Initiator has the option of issuing a ***CLEAR QUEUE Message***, and then ***Connecting*** again to issue the new I/O Process.

Summary of Use: The HEAD OF QUEUE TAG Message is sent only by an Initiator to a Target to cause the I/O Process to be placed at the front of the queue.

Hold Time. *tht = 45 nsec*. That's what it is. Hold Time. During ***Synchronous Data Transfer***, the Hold Time is used by the sending device to provide data hold time for the receiving device from the leading edge of the ***REQ Signal*** or the ***ACK Signal***. See Synchronous Data Transfer.

Host Adapter. A Host Adapter is that portion of a Host System that performs all or part of the Initiator function. The Host Adapter works with the Host to manage Commands, Data Transfer, and returned Status. The Host Adapter may also assume the ***Target*** role to transfer data between Host Adapters on the SCSI Bus (as a Processor Device), or to receive ***Asynchronous Event Notification (AEN)***. Two examples of Host Adapter design are shown in Diagram 32.

A simple Host Adapter may contain only a SCSI protocol control ***Chip***, which is then manipulated by a driver in the Host System. This means the Host is responsible for ***Path Control*** details. The simple Host Adapter interfaces to the Host System in much the same way as a simple controller interface, such as the disk controller in a PC.

A more complex Host Adapter includes an auxiliary processor which manipulates the SCSI Chip, and also contains logic for direct memory access (DMA) as a Bus Master. The Bus Master/DMA logic allows the complex Host Adapter to directly handle the Path Control function of the Initiator. With a complex Host Adapter, the Host asks for an ***I/O Process*** to be executed, and the Host Adapter takes care of all of the details. The complex Host Adapter behaves more like an I/O channel in a mainframe or minicomputer. A complex Host Adapter usually (not always!) gives higher performance than a simple Host Adapter.

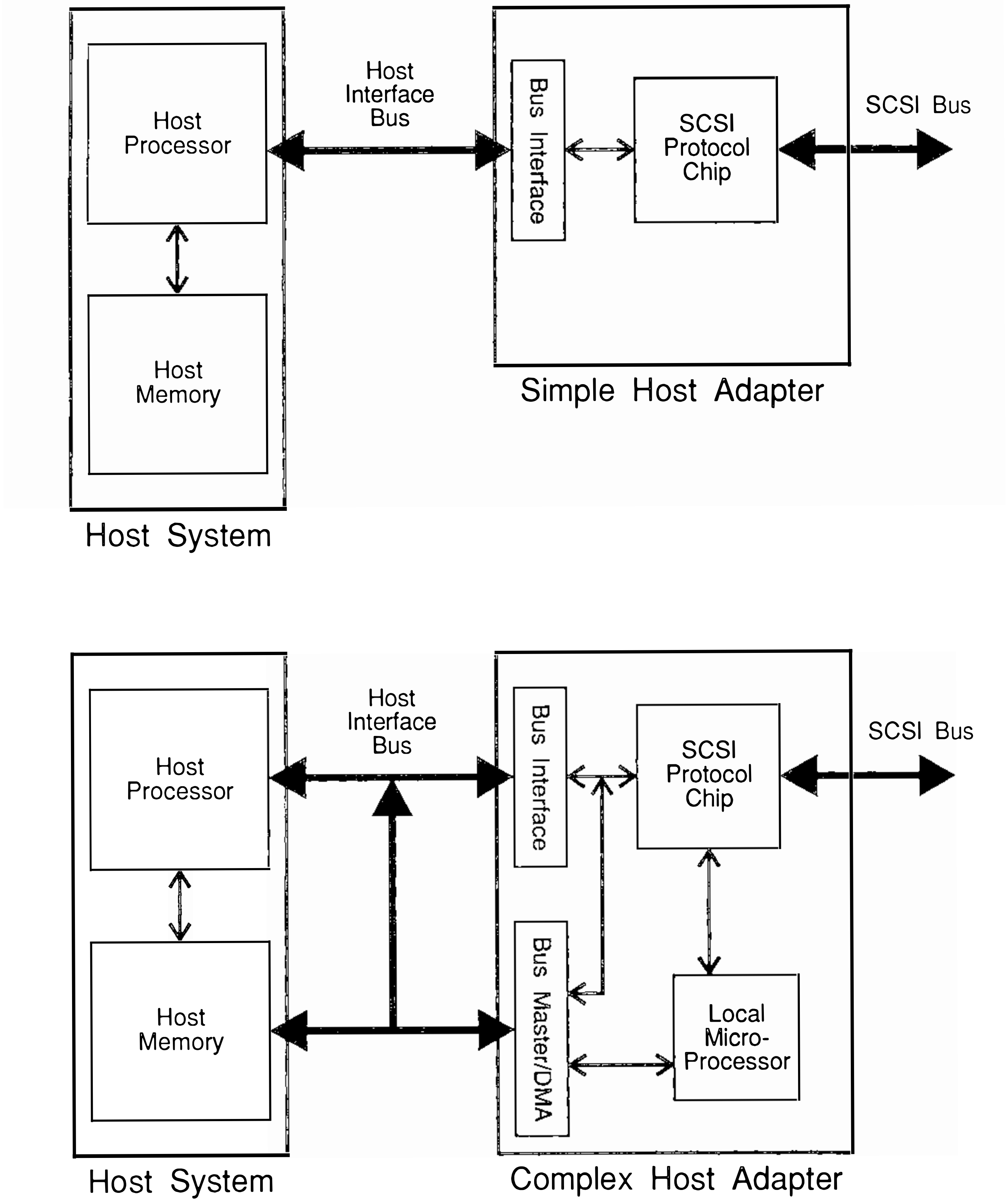


DIAGRAM 32: HOST ADAPTER DESIGN EXAMPLES

To perform Path Control on a complex Host Adapter, many designs use a Host Adapter Control Block, such as the one shown in Table 23. The Host passes a pointer to the Host Adapter that tells it where to find the Control Block. The Host Adapter then uses its Bus Master/DMA Capability to read the Control Block and begin the requested ***I/O Process***. The Control Block will often include the fields shown:

- The ***Command Descriptor Block (CDB)***, just as it would be sent to the Target. The Host address of the first byte of the CDB in the Control Block can be used as the Saved Command Pointer value.

- ***Pointer*** to the Data Area. This is the Host pointer to the start of the Host Data Area in Host memory. The Host Adapter may use this as the Saved Data Pointer during a ***DATA Phase***.

- The Data Area Size tells the Host Adapter where the limit of the Data Area is during a DATA Phase. This allows the Host Adapter to prevent an overrun of Host memory if the Target tries to transfer too much data.

- The Link Pointer is the Host address to the next Control Block to be used when using ***Linked Commands***. The next Control Block contains all of the information to be used during the next Linked Command. If the CDB does not contain a Linked Command, then this field is not used and should be set to zero.

- The ***Status*** code, just as it is received from the Target. The Host address of the Status byte in the Control Block can be used as the Saved Status Pointer value.

- The Target ***SCSI Address***, the ***Logical Unit Number (LUN)***, and (optionally) the ***Queue Tag*** give the Host Adapter the information it needs to establish the ***Nexus*** with the Target.

- The CDB Length is used by the Host Adapter to know how many bytes should be transferred during the ***COMMAND Phase***. While the first byte of the CDB usually defines the length of the CDB, the ***Vendor Specific*** Command Codes are not of standard length.

TABLE 23: HOST ADAPTER CONTROL BLOCK EXAMPLE

Byte Number	Field Description
00-11	Command Descriptor Block (CDB)
12-15	Pointer to Data Area
16-19	Data Area Size
20-23	Link Pointer to Next Control Block
24	Status Code
25	Target SCSI Address
26	Logical Unit Number
27	Queue Tag
28	CDB Length

As of this writing, complex Host Adapters are coming down in price to the point that it makes sense to use the complex Host Adapter in all but the most cost sensitive or size sensitive applications.

This page is nearly blank!

I/O Process. OK, we all know what an I/O process is: "Well, now I am going to do some input or output with my peripheral." Well, close enough....

On the other hand, maybe not. SCSI has specifically defined the term I/O process for a reason. Let's see what SCSI means by I/O Process, what it means to us, and how it relates to the traditional concept of I/O process.

To SCSI, an I/O Process has the following characteristics:

(1) It begins with an ***Initial Connection*** (an Initiator Selects a Target; kind of like "Boy meets Girl").

(2) It is described by a ***Nexus*** which defines the relationship between the Initiator and Target.

(3) It may involve no ***SCSI Commands***, one command, or more than one command if they are ***Linked Commands***.

(4) It may have a ***DATA Phase***, but only for certain SCSI Commands.

(5) It may involve no ***Reconnection***, or it may have one or more Reconnections.

(6) It may have a ***STATUS Phase***, with or without a SCSI Command.

(7) It ends with the ***BUS FREE Phase*** that occurs after one of the following ending events:

- The Target sends the ***COMMAND COMPLETE Message***; the normal way to end the I/O Process.
- ***ABORT Message***; an abnormal exit requested by the Initiator.
- ***ABORT TAG Message***; another abnormal exit requested by the Initiator.
- ***BUS DEVICE RESET Message***; a re-initialization by the Initiator.
- ***CLEAR QUEUE Message***; another abnormal exit by the Initiator.
- ***Hard Reset***; by the Initiator or another ***SCSI Device***.
- ***RELEASE RECOVERY Message***; see ***Extended Contingent Allegiance.***
- Unexpected ***BUS FREE Phase***; an abnormal exit by the Target.

You can see all of this illustrated in Diagram 33.

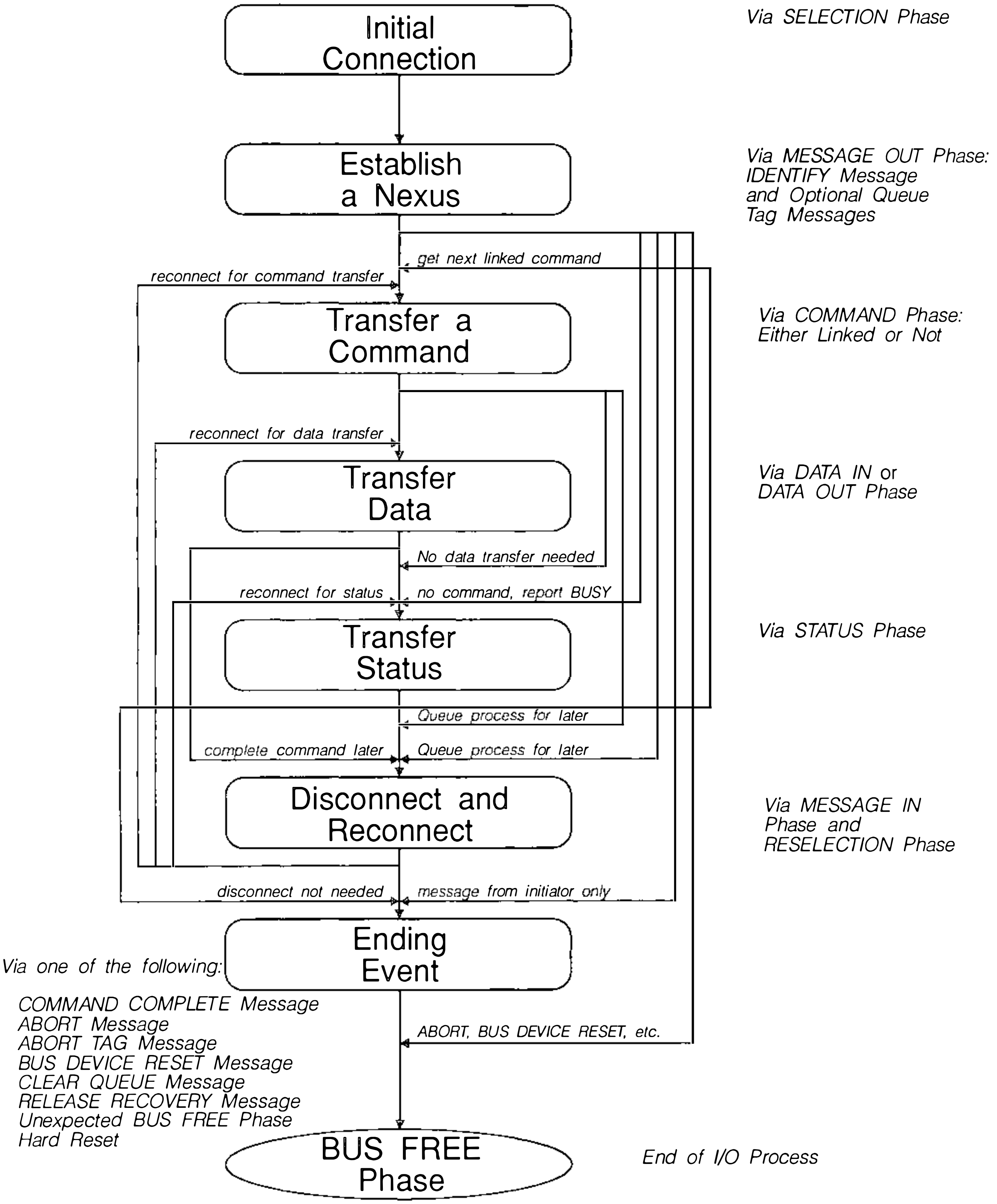


DIAGRAM 33: FLOW OF EVENTS DURING AN I/O PROCESS

Some examples always help too:

Example #1: A command from an Initiator to a Target with no data transfer (Diagram 34):

(1) An Initiator connects to a Target by Arbitrating for the bus and Selecting the Target.

(2) A Nexus is established via the ***IDENTIFY Message***, sent during the ***MESSAGE OUT Phase***. The Nexus established in this case is called an "I_T_L Nexus": The Nexus is described by an Initiator ***SCSI Bus ID*** (I_), a Target SCSI Bus ID (_T_), and a ***Logical Unit Number (LUN)*** (_L).

(3) A ***Command Descriptor Block (CDB)*** is sent during a ***COMMAND Phase*** from the Initiator to the Target. The CDB contains the SCSI Command for the Target to execute.

(4) The Target returns "GOOD" ***Status*** during the STATUS Phase.

(5) The Target returns the COMMAND COMPLETE Message during ***MESSAGE IN Phase***. In this case, the Target establishes the "Ending Event".

(6) BUS FREE Phase completes the I/O Process.

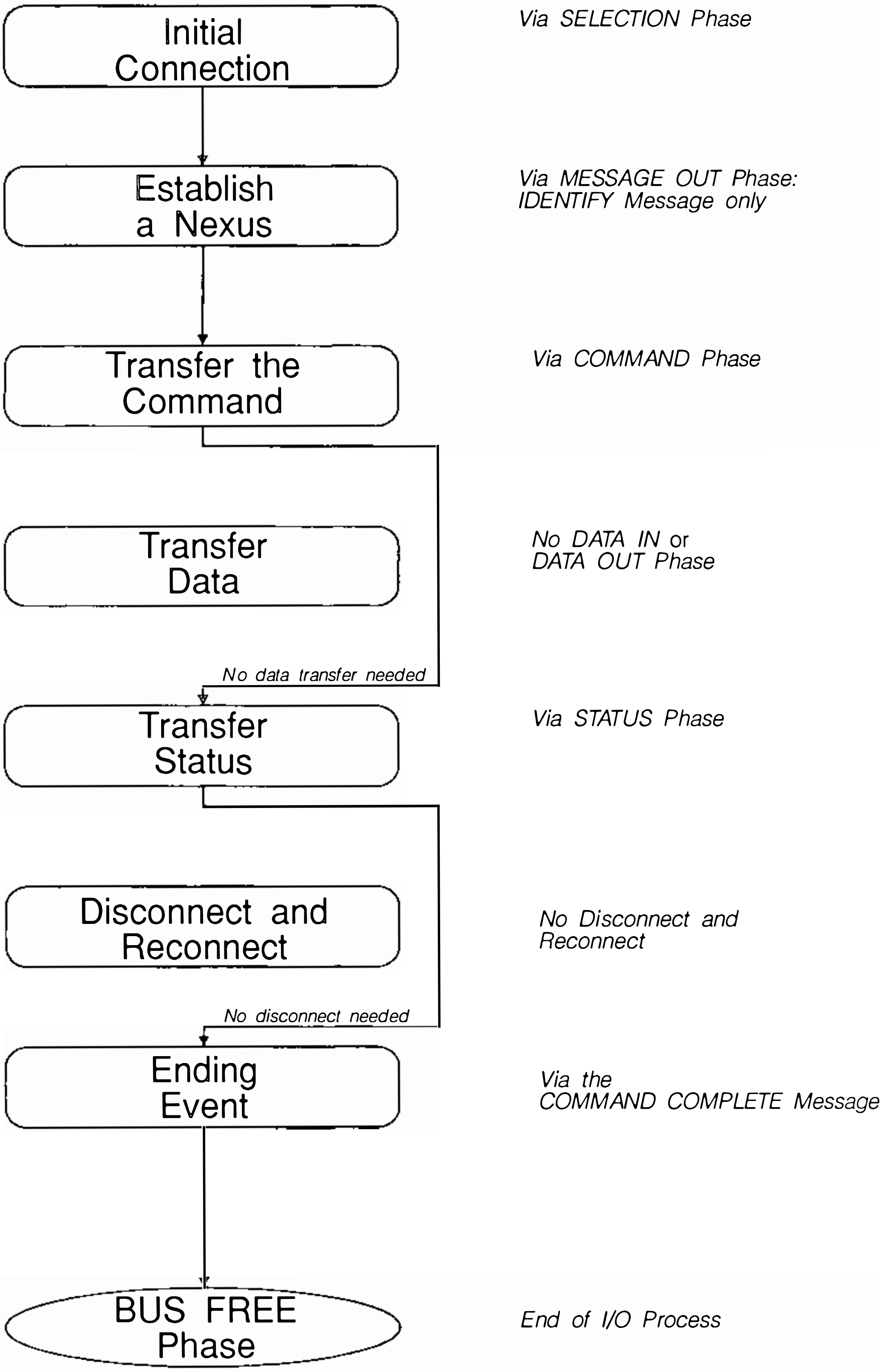


DIAGRAM 34: I/O PROCESS FLOW EXAMPLE #1

Example #2: A special message from an Initiator to a Target is sent with no SCSI Command (Diagram 35):

(1) An Initiator connects to a Target by Arbitrating for control of the bus and Selecting the Target. This establishes an "I_T Nexus", described only by the Initiator and Target SCSI Bus IDs. This is a sufficient Nexus for the events to follow.

(2) A ***BUS DEVICE RESET Message*** is sent from the Initiator to the Target to cause the Target to clear all activity. In this case, the Initiator establishes the "Ending Event". Note that no COMMAND, DATA, or STATUS Phases occur in this I/O Process.

(3) BUS FREE Phase completes the I/O Process.

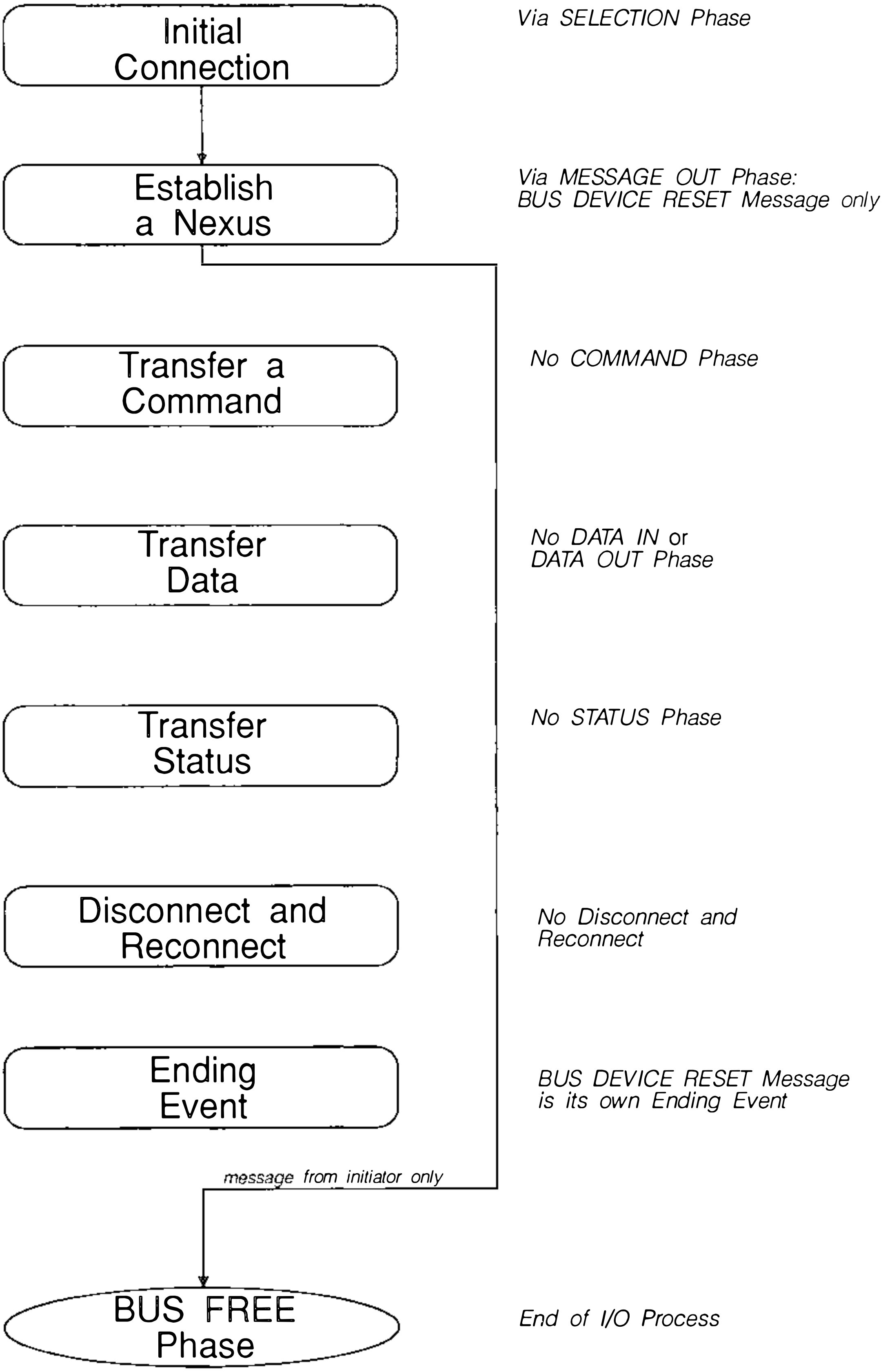


DIAGRAM 35: I/O PROCESS FLOW EXAMPLE #2

Example #3: The Target ***Queues*** the I/O Process from the Initiator and ***Disconnects*** before executing it (Diagram 36):

(1) An Initiator connects to a Target by Arbitrating for control of the bus and Selecting the Target.

(2) A Nexus is established via the IDENTIFY Message and the ***SIMPLE QUEUE TAG Message***. The Nexus established in this case is called an "I_T_L_Q Nexus": The Nexus is described by an Initiator SCSI Bus ID (I_), a Target SCSI Bus ID (_T_), a Logical Unit Number (LUN) (_L_), and a ***Queue Tag*** (_Q).

(3) A CDB containing a SCSI Command is sent during COMMAND Phase from the Initiator to the Target for the Target to execute.

(4) The Target Disconnects from the Initiator by sending a ***DISCONNECT Message*** during ***MESSAGE IN Phase***. A BUS FREE Phase follows the DISCONNECT Message, but this does not end the I/O Process. Also, note that the Target may decide to queue before transferring the SCSI Command (i.e., skip the COMMAND Phase) if it does not have enough storage space for the CDB.

(5) Later, the Target Reconnects to the Initiator via the ***RESELECTION Phase***. During Reconnection, the Target re-establishes the Nexus that was established during the ***Initial Connection***. During a MESSAGE IN Phase, the Target sends an IDENTIFY Message which establishes the LUN, and then sends a SIMPLE QUEUE TAG Message that establishes the original Queue Tag value.

(6) The SCSI Command sent by the Initiator requested the Target to return some data to the Initiator. The Target switches to ***DATA IN Phase*** to send the data to the Initiator.

(7) The Target returns Status and then the COMMAND COMPLETE Message as in Example #1.

(8) BUS FREE Phase completes the I/O Process.

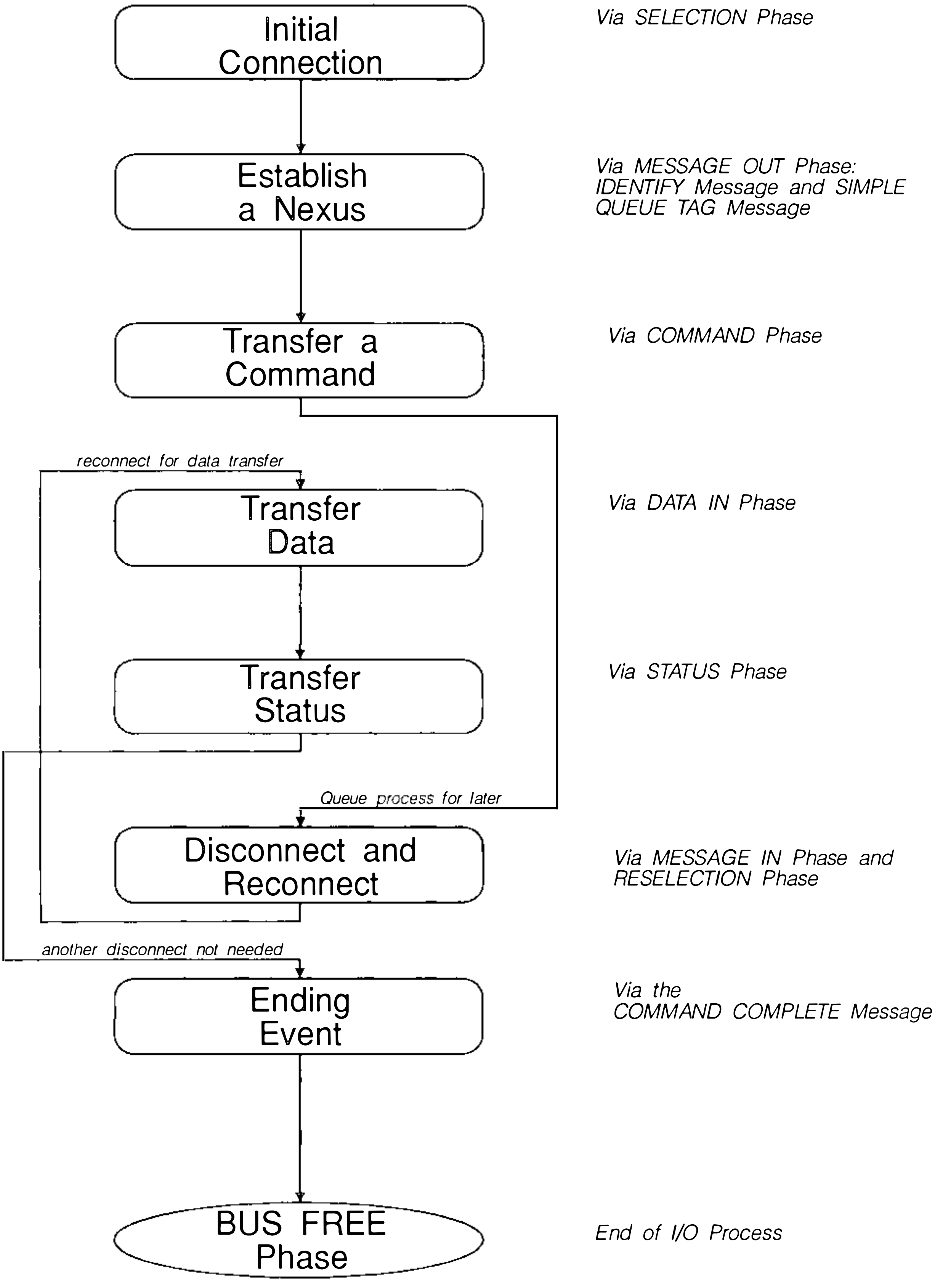


DIAGRAM 36: I/O PROCESS FLOW EXAMPLE #3

Example #4: An Initiator sends two SCSI commands in one I/O Process. The first CDB is a ***Linked Command*** which is queued by the Target. The second CDB is sent after the completion of the first command (Diagram 37):

(1) An Initiator Connects to a Target.

(2) An "I_T_L_Q Nexus" is established via the IDENTIFY Message and the ***ORDERED QUEUE TAG Message***.

(3) A CDB containing the first SCSI Command is sent during COMMAND Phase from the Initiator to the Target for the Target to execute. The LINK Bit is set to one in the CDB to indicate that this command is Linked to a following command.

(4) The Target Disconnects from the Initiator by sending a ***DISCONNECT Message*** during ***MESSAGE IN Phase***. A BUS FREE Phase follows the DISCONNECT Message, but this does not end the I/O Process.

(5) Later, the Target Reconnects to the Initiator via the ***RESELECTION Phase***. During Reconnection, the Target re-establishes the Nexus that was established during the ***Initial Connection***. During a MESSAGE IN Phase, the Target sends an IDENTIFY Message which establishes the LUN, and then sends a SIMPLE QUEUE TAG Message (not an ORDERED QUEUE TAG Message) that establishes the original Queue Tag value.

(6) Since this is a Linked Command, the Target returns "INTERMEDIATE/GOOD" Status during the STATUS Phase.

(7) Also, since this is a Linked Command, the Target returns the ***LINKED COMMAND COMPLETE Message*** during the MESSAGE IN Phase. This completes only the first SCSI Command, but does not complete the I/O Process.

(8) Immediately after the MESSAGE IN Phase, a CDB containing the second SCSI Command is sent during the second COMMAND Phase from the Initiator to the Target for the Target to execute. Since this command is not Linked to a following command, the LINK Bit is set to zero in the CDB.

(9) The second SCSI Command sent by the Initiator requests the Target to receive some data from the Initiator. The Target switches to ***DATA OUT Phase*** to get the data to the Initiator.

(10) The Target returns Status and then the COMMAND COMPLETE Message as in Example #1.

(11) BUS FREE Phase completes the I/O Process.

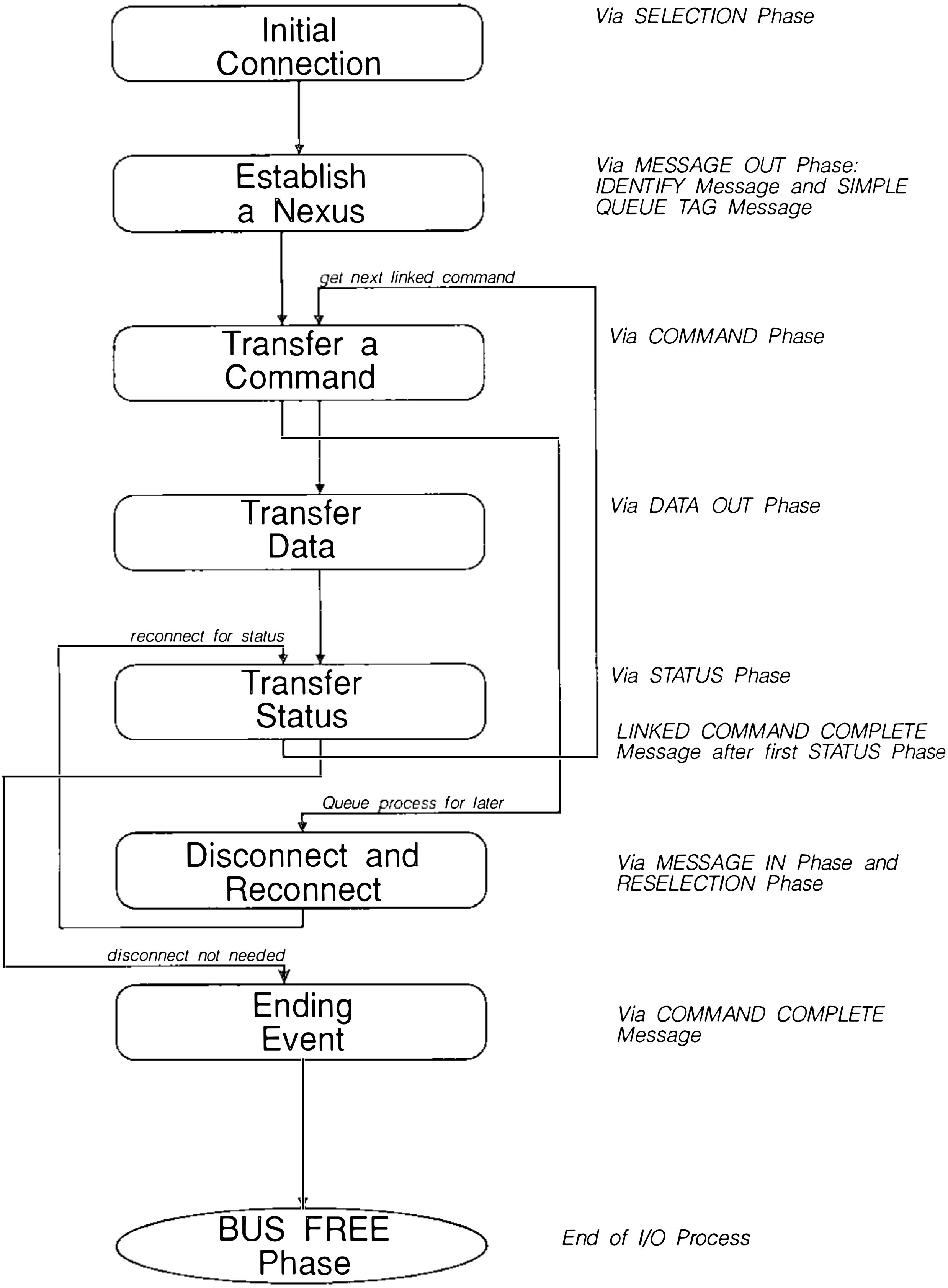


DIAGRAM 37: I/O PROCESS FLOW EXAMPLE #4

Why define an I/O Process? As you can see, an I/O Process is more involved than the "read a few sectors" type of I/O most of us are used to. Going back to Diagram 33, we see that a SCSI I/O Process can be any of the following:

- A SCSI Command with no data transfer.
- A SCSI Command that returns data to the Initiator.
- A SCSI Command that takes data from the Initiator.
- Any of the above broken up into one or more Reconnections.
- Any of the above queued for later execution by the Target.
- More than one SCSI Command Linked together into a single I/O Process.
- A "BUSY" Status-only transfer from the Target telling the Initiator to try the SCSI Command at a later time.
- A ***Path Control*** Message-only transfer from the Initiator, like BUS DEVICE RESET or ABORT, with no SCSI Command.
- Any of the above interrupted by a ***Hard Reset*** or transition to an ***Unexpected BUS FREE Phase***.

This list gives an insight to how SCSI gets things done. Some of the things on the list are just like a "traditional I/O". Many of the other tasks (such as Path Control) were performed more directly in older interfaces. In SCSI, almost everything is accomplished by a bus transaction that looks a lot like a "traditional I/O". This made it necessary in SCSI to have a broad definition of an I/O Process.

I/O Signal. The I/O signal is driven by the Target to control the direction of all transactions on the SCSI bus. I/O is one of the three ***Bus Phase Signals*** (See also ***Bus Phases***). Note that bus direction in SCSI is always defined relative to the Initiator:

- When this signal is false (negated or released), the bus direction is OUT from the Initiator. I/O is false during ***SELECTION Phase***, ***DATA OUT Phase***, ***MESSAGE OUT Phase***, and ***COMMAND Phase***.
- When the signal is true (asserted), the bus direction is IN to the Initiator. I/O is true during ***RESELECTION Phase***, ***DATA IN Phase***, ***MESSAGE IN Phase***, and ***STATUS Phase***.

IDENTIFY Message. The IDENTIFY message is used by one device to identify the ***Logical Unit*** or ***Target Routine*** to another. The IDENTIFY message is the first message sent by an Initiator at the beginning of an ***Initial Connection*** for most ***I/O Processes***. It is also the first message sent by a Target at the beginning of a ***Reconnection***. After the IDENTIFY message is transferred, a ***Nexus*** is established; either an I_T_L nexus or an I_T_R nexus.

Note: In practice, Target Routines are used only for a special feature of the INQUIRY Command. Except for that, you may never encounter the need to support Target Routines.

Bit Byte	7	6	5	4	3	2	1	0
0	Identify	DiscPriv	LUNTAR	Reserved	Reserved	LUNTRN		

The IDENTIFY Message is unusual in that it has several bits defined within it. The Identify bit determines that this is an IDENTIFY message:

Identify = 0: If the Identify bit is zero, then this is not an IDENTIFY message. Look somewhere else.

Identify = 1: If the Identify bit is one, then this is an IDENTIFY message. You may proceed.

The DiscPriv bit can only be set by the Initiator. When the Target sends the IDENTIFY message, it always clears this bit to zero.

DiscPriv = 0: When this bit is cleared to zero by the Initiator, the Target is not permitted to ***Disconnect*** from the bus. The Initiator might clear this bit to zero when it requires an immediate response, or cannot tolerate a disconnect for any other reason.

DiscPriv = 1: When this bit is set to one by the Initiator, the Target is permitted to Disconnect from the bus.

For simplicity, the Initiator should always set the DiscPriv bit to one, since the Targets usually disconnect only when it's appropriate. Really. In practice, modern SCSI Initiators can accept a disconnect at any time. Therefore, the Initiator should generally not clear this bit to zero.

A case where clearing DiscPriv to zero would be better is in a system where there is only one Target and one Initiator. In this case DiscPriv should be zero because the extra Disconnects are unnecessary and add extra overhead time.

The LUNTAR bit determines whether a Logical Unit or a Target Routine is being identified:

LUNTAR = 0: When LUNTAR is zero, a Logical Unit is being identified. This is the typical case, as noted before.

LUNTAR = 1: When LUNTAR is one, a Target Routine is being identified. This is a rare case, as noted before.

The LUNTRN field contains the ***Logical Unit Number (LUN)*** or the ***Target Routine Number (TRN)***, as indicated by the LUNTAR bit. This is what is really being identified. (The number of bits in this field may be extended in the ***SCSI-3*** Standard.)

Because it is such an important message (or, because it has been around for a long time...), IDENTIFY has some special rules:

- If the IDENTIFY message is received by a Target during an Initial Connection with any of the Reserved bits set to one, or if the Target does not support Target routines and LUNTAR is one, it may respond in one of two ways:
  - Go to ***MESSAGE IN Phase*** and send a ***MESSAGE REJECT Message*** immediately after receiving the IDENTIFY message. This method is "architecturally correct"; an unacceptable message is rejected in kind. This is preferred, but not always possible.
  - Go to ***STATUS Phase*** and return CHECK CONDITION ***Status***. This establishes ***Contingent Allegiance Condition*** with the Initiator. The Target creates SENSE DATA that indicates that the message was an ILLEGAL REQUEST with INVALID BITS IN THE IDENTIFY MESSAGE (SCSI caps, not mine). This option must exist because many SCSI Protocol ***Chips*** do not test the validity of the IDENTIFY message until after the ***COMMAND Phase*** is completed. (This is done to enhance performance.) While not the optimum solution, it is acceptable under the standard, and is worth it to get the performance enhancement in most cases.

- If the IDENTIFY message is received by an Initiator during a Reconnection with any of the Reserved bits set to one, or if the DiscPriv bit is set, the Initiator may respond any way it chooses, including:

  - Creating the ***Attention Condition*** and sending the MESSAGE REJECT message.

  - Sending the ***ABORT Message***, the ***ABORT TAG Message***, or ***BUS DEVICE RESET Message***.

- The Initiator may send the IDENTIFY message more than once during an ***I/O Process***, but it may not change the identified Logical Unit or Target Routine. In other words, it may only change the state of the DiscPriv bit. We recommend using the Disconnect control features of the MODE SELECT command instead, for lower overhead and simpler design. If the Target receives a second IDENTIFY Message during an I/O Process that changes anything other than the DiscPriv bit, it transitions to an ***Unexpected BUS FREE Phase***; naughty Initiator.

- When the Initiator receives the IDENTIFY message after a ***Reconnect*** (plus the ***SIMPLE QUEUE TAG Message***, if reconnecting for an I_T_L_Q ***Nexus***), it has the same effect as a ***RESTORE POINTERS Message***.

Summary of Use: The IDENTIFY message may be transferred in the following situations:

- By the Initiator immediately after SELECTION Phase.

- By the Target immediately after RESELECTION Phase.

The Initiator is allowed to send the IDENTIFY message at other times to control the Target's disconnect privilege, but there are more efficient ways to control disconnect via the MODE SELECT command, so please avoid using the IDENTIFY message this way.

IGNORE WIDE RESIDUE Message. There are many types of residue that you do not have to be told to ignore! When using ***Wide Data Transfer***, you need a little guidance. This message is sent from the Target to the Initiator immediately after a ***DATA IN Phase*** in which Wide Data Transfer was used. It is not sent from the Initiator to the Target after a ***DATA OUT Phase*** (Why? See below).

Why IGNORE WIDE RESIDUE? The Target may have to return data of odd or indeterminate length during a 16-bit or 32-bit transfer:

- The Target is a tape drive with variable length blocks, and the block length is an odd number of bytes. In this case, IGNORE WIDE RESIDUE is sent to inform the Initiator of the exact end of the block.

- The Initiator has allocated an odd number of bytes for the data returned by a command. (This might happen with older Initiator software). In this case, IGNORE WIDE RESIDUE is sent to ensure that the Initiator does not register a buffer overflow error (see ***Path Control*** and ***Host Adapter***).

- When doing 32-bit transfers, there are certain commands which may require a number of bytes that is not divisible by 4, such as 10 bytes. In this case, IGNORE WIDE RESIDUE is sent to inform the Initiator of the exact transfer length.

We can think of other ways that really don't make sense, like a disk drive with 513 byte sectors, but this writer hasn't seen them. The fact is, any Target designed for Wide Data Transfer should make every effort to transfer a number of bytes such that this message isn't necessary (see ***Etiquette***).

Why is IGNORE WIDE RESIDUE not sent after a DATA OUT Phase? Because:

- If a byte transfer length is specified in the command, the Target knows the exact byte count from the Transfer Length in the CDB, so no message is needed.

- If a block transfer length is specified in the command, the Target knows the block length, so no message is needed.

At last, the message format:

Bit Byte	7	6	5	4	3	2	1	0
0	Message Code = 23 hex							
1	Ignore Code							

The second byte of the message is the Ignore Code, which tells the Initiator what to ignore. Table 26 shows the meaning of the Ignore Codes for 32-bit and 16-bit transfers.

TABLE 26: IGNORE WIDE RESIDUE IGNORE CODES

Ignore Code	Meaning for 32-bit Transfers	Meaning for 16-bit Transfers
00 hex	Reserved - not used	Reserved - not used
01 hex	Discard the data bits on DB(31-24) (Discard the upper byte)	Discard the data bits on DB(15-8) (Discard the upper byte)
02 hex	Discard the data bits on DB(31-16) (Discard the upper two bytes)	Reserved - not used
03 hex	Discard the data bits on DB(31-8) (Discard the upper three bytes)	Reserved - not used
04-FF hex	Reserved - not used	Reserved - not used

Summary of Use: The IGNORE WIDE RESIDUE message is sent only by a Target after a DATA IN Phase in which Wide Data Transfer was used, and the transfer length in bytes was not a multiple in bytes of the transfer width.

Information Transfer Phases. The Information Transfer Phases are the types of ***Bus Phases*** in which some type of real information is transferred on the bus (imagine that?). Contrast to the ***Connection Phases***. An Information Transfer Phase is any phase in which the ***REQ Signal*** and ***ACK Signal*** control the flow of the phase. The phases are:

- ***COMMAND Phase***: Used to transfer ***Command Descriptor Blocks (CDBs)*** from the Initiator to the Target.
- ***STATUS Phase***: Used to transfer ***Status*** from the Target to the Initiator.
- ***DATA OUT Phase***: Used to transfer data from the Initiator to the Target.
- ***DATA IN Phase*** Used to transfer data from the Target to the Initiator.
- ***MESSAGE OUT Phase***: Used to send a ***Message*** from the Initiator to the Target.
- ***MESSAGE IN Phase***: Used to send a ***Message*** from the Target to the Initiator.

Initial Connection. An Initial Connection is the ***Connection*** that begins an ***I/O Process***. The Initial Connection begins with a ***SELECTION Phase*** and continues with an ***IDENTIFY Message*** and possibly a ***Queue Tag Message*** in a ***MESSAGE OUT Phase***. This results in a ***Nexus*** being established between the Initiator and Target.

Initialization. Initialization is performed by an Initiator or a Target after a Power-up reset or a ***Hard Reset***. Since it is somewhat unique to each Device Type, and involves specific command sequences, this subject is fully covered in the other volumes of the SCSI Encyclopedia.

INITIATE RECOVERY Message. The INITIATE RECOVERY message is sent by a ***SCSI Device*** to begin an ***Extended Contingent Allegiance (ECA)*** condition. The message is normally sent by a Target, but if a device that is normally a Target uses ***Asynchronous Event Notification (AEN)***, it may also send this message when it takes the Initiator role in order to send the AEN data.

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

See Extended Contingent Allegiance (ECA) for complete details on how to use this message.

Summary of Use: The INITIATE RECOVERY message is sent by a SCSI Device to begin an ECA condition.

Initiator. From the dictionary: "One who initiates...". Well, that works! On the ***SCSI Bus***, the Initiator begins a bus transaction (which is called an ***I/O Process***).

In general, the Initiator is a role assumed by any ***SCSI Device*** to issue a ***SCSI Command*** to another SCSI Device that can assume the ***Target*** role. In traditional terms, an Initiator is usually the primary role assumed by a ***Host Adapter***. A peripheral ***Controller*** may also assume the Initiator role to perform, for example, an ***Asynchronous Event Notification (AEN)*** or the COPY Command.

Diagram 38 shows the Initiator role in a system, and also shows which signals on the SCSI Bus the Initiator drives and receives. The following bullets give a flavor of the Initiator's contribution to the execution of an I/O Process:

- Only an Initiator can begin an I/O Process. An I/O Process is begun when a SCSI Device takes the Initiator role and ***Selects*** a Target.

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

- The Initiator may attempt to get the Target's attention by (appropriately enough) creating the ***Attention Condition***. This is the only way to modify the Target's behavior regarding Bus Phases.

- If the Target cannot be reasoned with by the Attention Condition and subsequent ***MESSAGE OUT Phase***, the only other option available to the Initiator is to create the ***Reset Condition***.

- The Initiator keeps ***Pointers*** for each pending I/O Process that maintain the current state of Command, ***Status***, and Data transfer. Pending I/O Processes include both ***Active I/O Processes***, <u>and</u> I/O Processes waiting in a ***Queue***.

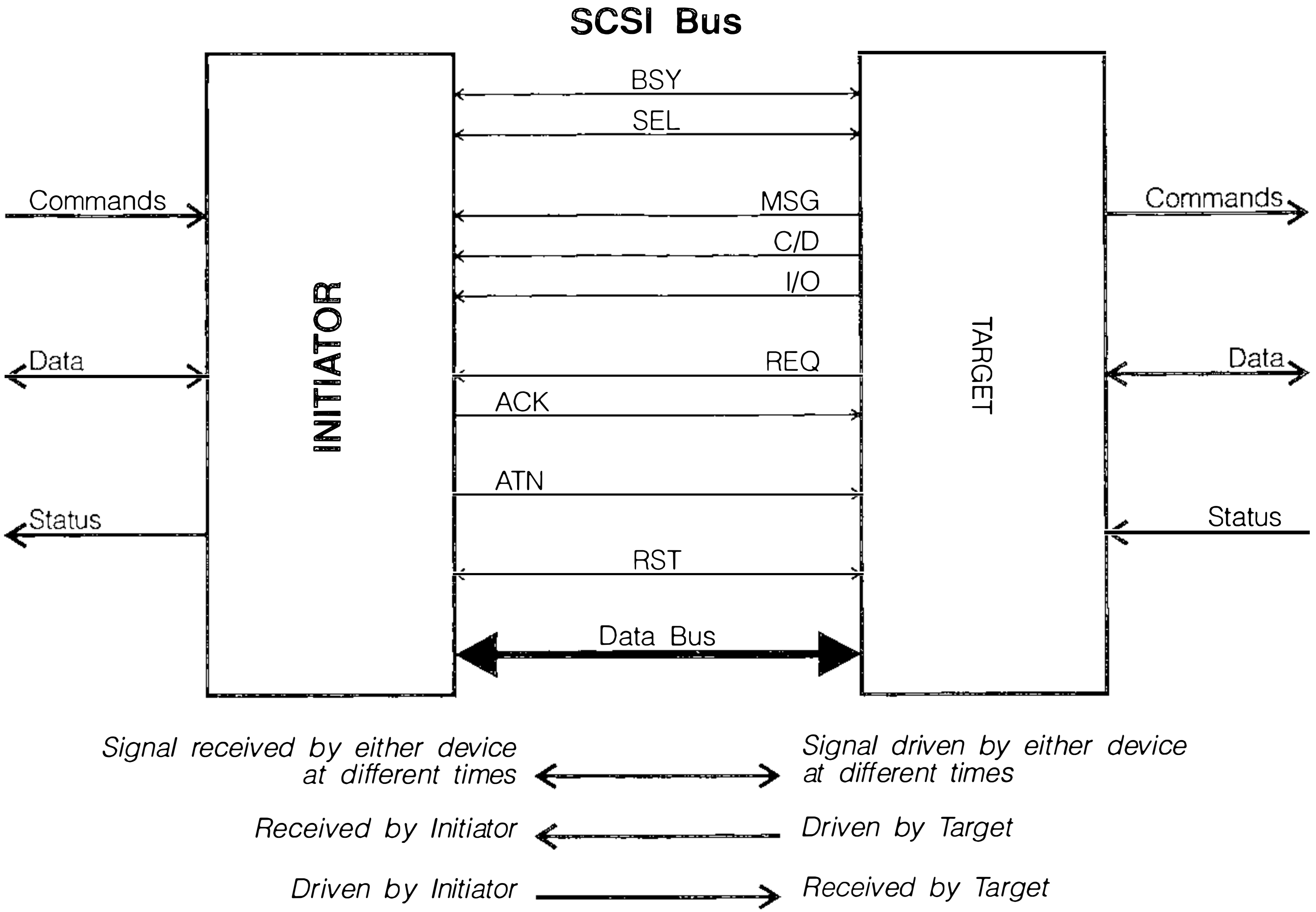


DIAGRAM 38: INITIATOR ROLE FOR SCSI DEVICES

INITIATOR DETECTED ERROR Message. The INITIATOR DETECTED ERROR message is sent by the Initiator to inform the Target that an error has been detected by the Initiator (either on the bus or somewhere within the Initiator). The intent of the Initiator is to prompt the Target to perform an ***Error Recovery*** procedure, such as the Target sending a ***RESTORE POINTERS Message***.

Bit Byte	7	6	5	4	3	2	1	0
0	Message Code = 05 hex							

Some examples of when the Initiator would send this message:

- After detecting a ***Parity*** error during an inbound ***Information Transfer Phase***, either a ***STATUS Phase*** or a ***DATA IN Phase***.

- After detecting an internal error during an outbound Information Transfer Phase, either a ***COMMAND Phase*** or a ***DATA OUT Phase***.

Note that it is <u>not</u> appropriate to send this message as a result of an error detected during ***MESSAGE IN Phase*** or ***MESSAGE OUT Phase***. These phases have special error handling. See ***Message System***.

The Initiator sends the INITIATOR DETECTED ERROR message by creating the ***Attention Condition*** and waiting for the Target to respond with MESSAGE OUT Phase.

Summary of Use: The INITIATOR DETECTED ERROR message is sent only by the Initiator after it detects an error during a DATA IN, DATA OUT, COMMAND, or STATUS Phase.

LINKED COMMAND COMPLETE Messages.
Linked Commands.
Logical Block.
Logical Block Address (LBA).
Logical Unit.
Logical Unit Number (LUN).

LINKED COMMAND COMPLETE Messages. These messages are used to indicate that one command of a "sequence" of Linked commands has been completed successfully. There are two different single byte messages for this:

- LINKED COMMAND COMPLETE:

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

- LINKED COMMAND COMPLETE (WITH FLAG):

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

The LINKED COMMAND COMPLETE messages are used to allow more than one command to be executed in a single ***I/O Process***. The Target sends one of these messages after ***STATUS Phase*** to tell the Initiator to discard the ***Pointers*** in use at present, and load the Pointers for the next command of the I/O Process.

Summary of Use: The LINKED COMMAND COMPLETE messages are sent only by a Target after a STATUS Phase to begin the next Linked Command of an I/O Process.

Okay, but what's a FLAG? Probably, the best thing to do is to get the whole story by reading the next section on ***Linked Commands***.

Linked Commands. The SCSI-2 standard defines Linked commands as a single ***I/O Process*** to a single ***Nexus*** in which two or more ***Command Descriptor Blocks (CDBs)*** are executed consecutively. Not much help, is it? Let's try an example....

In the following example (see Table 31), the Initiator (let's call him Ian this time) wants to read some sectors from three different areas of the disk (our Target, Tanya). Normally, three read commands would be issued individually, and we show this in the left hand column. In the right hand column, we show the same sequence with Linked commands.

TABLE 31: LINKED COMMAND COMPARISON

Unlinked Commands	Linked Commands
(U1): Ian prepares the first command to send to Tanya. (U2): Ian Arbitrates for the Bus and Selects Tanya. (U3): Ian sends the Command Descriptor Block (CDB) for the first READ command to Tanya. (U4): Tanya returns sector data to Ian. (U5): Tanya sends Status and a COMMAND COMPLETE Message to Ian and goes to BUS FREE Phase.	(L1): Ian prepares all three commands to send to Tanya. (L2): Ian Arbitrates for the Bus and Selects Tanya. (L3): Ian sends the CDB for the first READ command to Tanya. (L4): Tanya returns sector data to Ian. (L5): Tanya sends Status and a LINKED COMMAND COMPLETE Message to Ian.
(U6): Ian processes the completion Status and releases the data to the operating system. (U7): Ian prepares the second command to send to Tanya.(*) (U8): Ian Arbitrates for the Bus and Selects Tanya.	(L6): Ian processes the completion Status and may or may not release the data to the operating system at this time.
(U9): Ian sends the CDB for the second READ command to Tanya. (U10): Tanya returns sector data to Ian. (U11): Tanya sends Status and a COMMAND COMPLETE Message to Ian and goes to BUS FREE Phase.	(L7): Ian sends the CDB for the second READ command to Tanya. (L8): Tanya returns sector data to Ian. (L9): Tanya sends Status and a completion message to Ian.
(U12): Ian processes the completion Status and releases the data to the operating system. (U13): Ian prepares the third command to send to Tanya.(*) (U14): Ian Arbitrates for the Bus and Selects Tanya.	(L10): Ian processes the completion Status and may or may not release the data to the operating system at this time.
(U15): Ian sends the CDB for the third READ command to Tanya. (U16): Tanya returns sector data to Ian. (U17): Tanya sends Status and a COMMAND COMPLETE Message to Ian and goes to BUS FREE Phase. (U18): Ian processes the completion Status and releases the data to the operating system.	(L11): Ian sends the CDB for the third READ command to Tanya. (L12): Tanya returns sector data to Ian. (L13): Tanya sends Status and a COMMAND COMPLETE Message to Ian. (L14): Ian processes the completion Status and releases the data from this command (or all three commands if the data was not released before) to the operating system.

(*) These steps could also occur as part of step U1.

As we can see from Table 31, when Ian used Linked Commands, he saved the overhead of Arbitrating and ***Connecting*** to Tanya for each command. This gives a modest performance improvement.

More importantly, he was able to execute three commands in an unbroken manner. When Tanya accepted the first Linked Command, he gained exclusive access to Tanya for the duration of the Link. As long as no errors or faults occur, Ian can keep sending commands to Tanya as long as he sets the Link bit. No other Initiators can have commands executed in between Ian's commands. This amounts to a "poor-man's reservation" of Tanya by Ian. In practice, there are "friendlier" ways to get a reservation (see the other Volumes of the SCSI Encyclopedia), but this is one way to do a short term, low-overhead, quickie reservation.

An Initlator begins a sequence of Linked Commands by setting the Link bit to one in the ***Control Byte*** of the CDB. As long as the Initiator keeps the Link bit set in each CDB, the sequence of Linked Commands continues. The Linkage ends when the Initiator sends a CDB with the Link bit cleared to zero, or an "error" occurs.

Diagram 39 shows how a Target processes Linked Commands. The Target executes the Command. If the execution resulted in one of the following ***Status*** codes, the link is broken and the Target returns the ***COMMAND COMPLETE Message*** to end the I/O Process:

- CHECK CONDITION
- BUSY
- RESERVATION CONFLICT
- COMMAND TERMINATED
- QUEUE FULL

If the Status is not on the above list, the Target next examines the CDB to see if the Link bit is set to one:

- If the Link bit is zero, things end normally (GOOD Status or CONDITION MET Status, COMMAND COMPLETE Message, and ***BUS FREE Phase***.

- If the Link bit is one, the Target returns Status that indicates the completion of a Linked Command; either INTERMEDIATE Status or INTERMEDIATE CONDITION MET Status.

The Target then examines the Flag bit in the CDB:

- If the Flag bit is zero, the Target returns the ***LINKED COMMAND COMPLETE Message*** in the MESSAGE IN Phase following the STATUS Phase.

- If the Flag bit is one, the Target returns the ***LINKED COMMAND COMPLETE (WITH FLAG) Message*** in the MESSAGE IN Phase following the STATUS Phase.

In either case, the next Phase is <u>not</u> BUS FREE Phase. The next Phase is COMMAND Phase to get the next CDB in the Linked Command sequence.

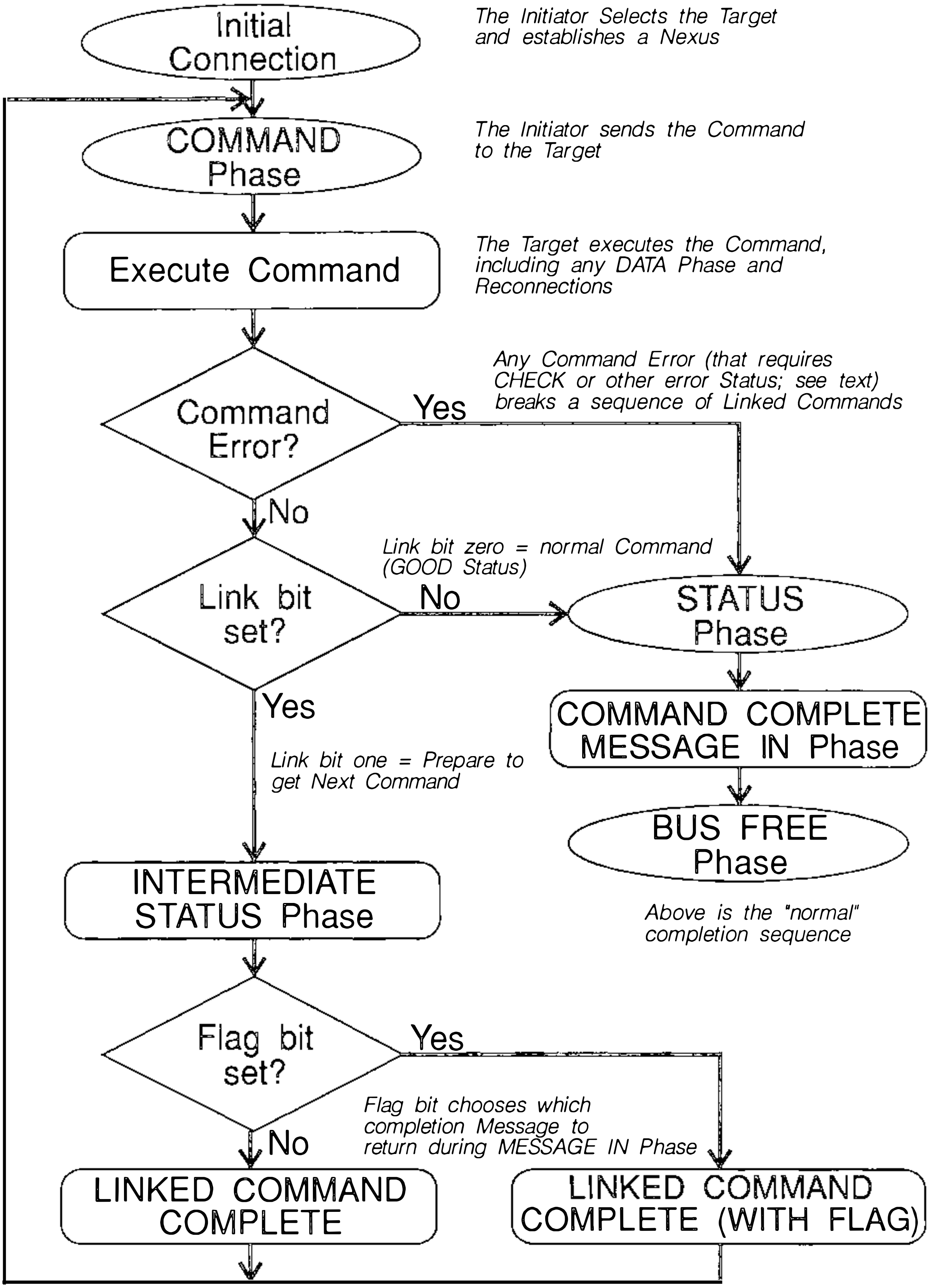

Initial Connection
The Initiator Selects the Target and establishes a Nexus
COMMAND Phase
The Initiator sends the Command to the Target
Execute Command
The Target executes the Command, including any DATA Phase and Reconnections
Command Error?
Yes
Any Command Error (that requires CHECK or other error Status; see text) breaks a sequence of Linked Commands
No
Link bit set?
Link bit zero = normal Command (GOOD Status)
No
STATUS Phase
COMMAND COMPLETE MESSAGE IN Phase
BUS FREE Phase
Above is the "normal" completion sequence
Yes
Link bit one = Prepare to get Next Command
INTERMEDIATE STATUS Phase
Flag bit set?
Yes
Flag bit chooses which completion Message to return during MESSAGE IN Phase
No
LINKED COMMAND COMPLETE
LINKED COMMAND COMPLETE (WITH FLAG)

DIAGRAM 39: LINKED COMMAND PROCESSING FOR TARGETS

Diagram 40 shows how Initiators process the end of a Linked Command. After the STATUS Phase (note that the Initiator doesn't particularly care what the Status is to process a Linked Command), the Initiator receives a Message during the MESSAGE IN Phase. Normally, this message is one of the following three:

- COMMAND COMPLETE
- LINKED COMMAND COMPLETE
- LINKED COMMAND COMPLETE (WITH FLAG)

If it's not one of these three, it should be rejected as described in the ***Message System***.

If the Message is COMMAND COMPLETE, the Command is done, any Linkage is broken, and the Nexus no longer exists. The old ***Pointers*** are discarded. Completion of the Command should be reported to the Initiator's Host.

If the Message is one of the two LINKED COMMAND COMPLETE Messages, the Initiator discards the old Pointers and gets a new set of Pointers from the Host (by some prior arrangement; see ***Host Adapter***).

If the Message was LINKED COMMAND COMPLETE (WITH FLAG), the Initiator should also let its Host know that a Flagged Command was completed. (This is often done via a special interrupt.)

The next Phase after MESSAGE IN Phase should be COMMAND Phase so that the Target can receive the new Command. If it isn't, the Initiator should probably issue the ***ABORT Message***, ***ABORT TAG Message***, or ***TERMINATE I/O PROCESS Message*** to break the linkage.

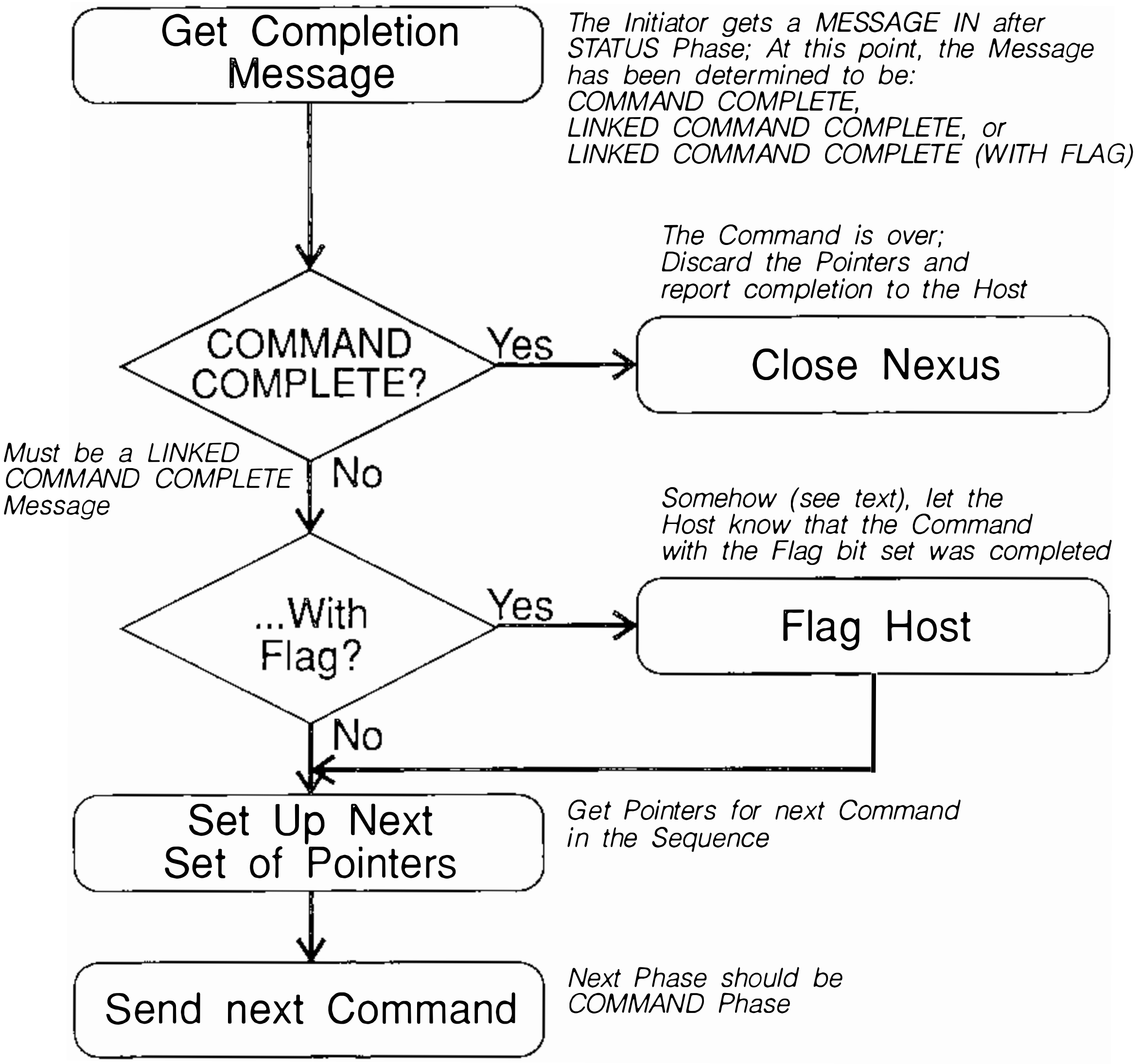

Get Completion Message
The Initiator gets a MESSAGE IN after STATUS Phase; At this point, the Message has been determined to be:
COMMAND COMPLETE,
LINKED COMMAND COMPLETE, or
LINKED COMMAND COMPLETE (WITH FLAG)
COMMAND COMPLETE?
Yes
The Command is over; Discard the Pointers and report completion to the Host
Close Nexus
No
Must be a LINKED COMMAND COMPLETE Message
...With Flag?
Yes
Somehow (see text), let the Host know that the Command with the Flag bit set was completed
Flag Host
No
Set Up Next Set of Pointers
Get Pointers for next Command in the Sequence
Send next Command
Next Phase should be COMMAND Phase

DIAGRAM 40: INITIATOR PROCESS AT END OF LINKED COMMAND

Finally, we ask the musical question...

What are Linked Commands for? In practice, there is really no reason to use Linked Commands. This may sound surprising at first, because Linking commands together seems like an easy way to give a sequence of commands some priority. But consider what it does to other Initiators on the bus. Their commands are put on hold for the duration of the link, and it also ties up the bus.

Another bad effect of Linked Commands is that it defeats the performance advantages of a command ***Queue***. By linking the commands together, the commands are forced into sequential order. This defeats the Target's ability to rearrange the order of command execution for maximum efficiency.

The better way to do it is to use the RESERVE and RELEASE commands. All SCSI Device Types support these commands. Like Linked Commands, an Initiator may lock out the commands from other Initiators for the duration of the Reservation. The RESERVE Command has the additional ability to reserve a portion of the medium, allowing other Initiators access to the rest of the ***Logical Unit***. The reordering of commands in a Queue may also continue. Reservations are explained in the other Volumes of the SCSI Encyclopedia.

The only "practical" application of Linked Commands is for certain special commands that can pass results to a subsequent command. For example, Linked Commands are used to pass the result of a SEARCH Command to a subsequent READ command on a Direct Access Device. We will cover this application in Volume II of the SCSI Encyclopedia. Initiators should be aware that we have also seen ***Vendor Unique*** commands that use this application.

Logical Block. The Logical Block is the basic unit of data storage within a ***SCSI Device***. As we will see in later Volumes of this Encyclopedia, all data associated with normal READ and WRITE Commands is stored as a Logical Block:

- On a Direct Access Device (disk drive), a Logical Block is often a physical disk sector, but it could also be made up of several physical disk sectors, or be a portion of a single physical sector. A Logical Block is fixed in size for the whole ***Logical Unit***, or for a portion (or "extent") of the unit.

  The Initiator is able to determine the Logical Block size (or sizes) within a Direct Access Device Logical Unit by certain device specific commands (which we also won't get into here).

- On a Sequential Access Device (tape drive), a Logical Block is usually the native physical block size of the tape format in use (e.g., QIC or 9-track), but the variations noted for disks could also apply (rarely). A Logical Block could be fixed in size for the whole ***Logical Unit***, or for a portion (or "extent") of the unit. A Logical Block may also be variable in size on a block-by-block basis.

  If the Logical Block size is fixed, then the Initiator is able to determine the Logical Block size (or sizes) within a Logical Unit by certain device specific commands (which we also won't get into here). If the Logical Block size is variable, read Volume III of the SCSI Encyclopedia!

Some devices, such Processor or Communication Devices, do not use Logical Blocks, since they transfer information packets of variable byte length.

Logical Block Address (LBA). The Logical Block Address specifies a ***Logical Block*** of data within a device. When you send a command with an LBA to a device, the device uses the LBA to select the first block of data that it will operate on. The treatment of Logical Blocks and LBAs is somewhat unique for each device type (and we will cover them in detail in later Volumes of this Encyclopedia), but there are some common characteristics:

- If there are any Logical Blocks, there is always an LBA #0.

- Logical Blocks are assigned LBAs, starting with #0, incrementing by one until there are no more Logical Blocks. In other words, Logical Blocks are addressed as if they were a contiguous set of fixed or variable memory blocks.

- LBAs are not necessarily assigned to perfectly corresponding physical sectors or blocks. With a Direct Access (disk) Device, mapping Logical Blocks directly to corresponding physical sectors is often done to improve performance, but sometimes a Logical Block Address may point at a sector on a different track of the device. This allows Logical Blocks to be reassigned if defects render the original sector or block unusable. A Sequential Access (tape) Device may avoid defective physical blocks by "skipping" over them.

Logical Unit. A Logical Unit is the basic addressable unit on a Target. A Target may have up to eight Logical Units. Diagram 41 shows the model of a Target and its Logical Units.

Each Logical Unit represents some form of physical ***Peripheral Device***. The relationship of a Logical Unit to Peripheral Device may be one of the following:

- The Logical Unit directly references a single Peripheral Device. An example of this is an embedded SCSI disk or tape drive.
- The Logical Unit references multiple Peripheral Devices. An example of this is a disk drive array where each Logical Unit is composed of several disk drives.
- The Logical Unit references a portion of a Peripheral Device. An example would be a very large optical disk drive partitioned into several Logical Units.
- The Logical Unit references multiple portions of several Peripheral Devices. An example of this is a disk drive array where each Logical Unit is composed of portions of several disk drives.

We are not passing final judgement on how possible each of the relationships listed above are, but in the humble opinion of this writer, they are in order of most common to least common...

The concept of a Logical Unit was to allow the above mappings to occur within a consistent structure. No matter what the physical configuration, a Logical Unit is a Logical Unit, and all Logical Units of a given device type are accessed the same way.

It is possible that a Target can have more than Device Type; e.g., a combination disk and tape ***Controller***. In this case, the disk would be implemented as one Logical Unit, and the tape as another Logical Unit.

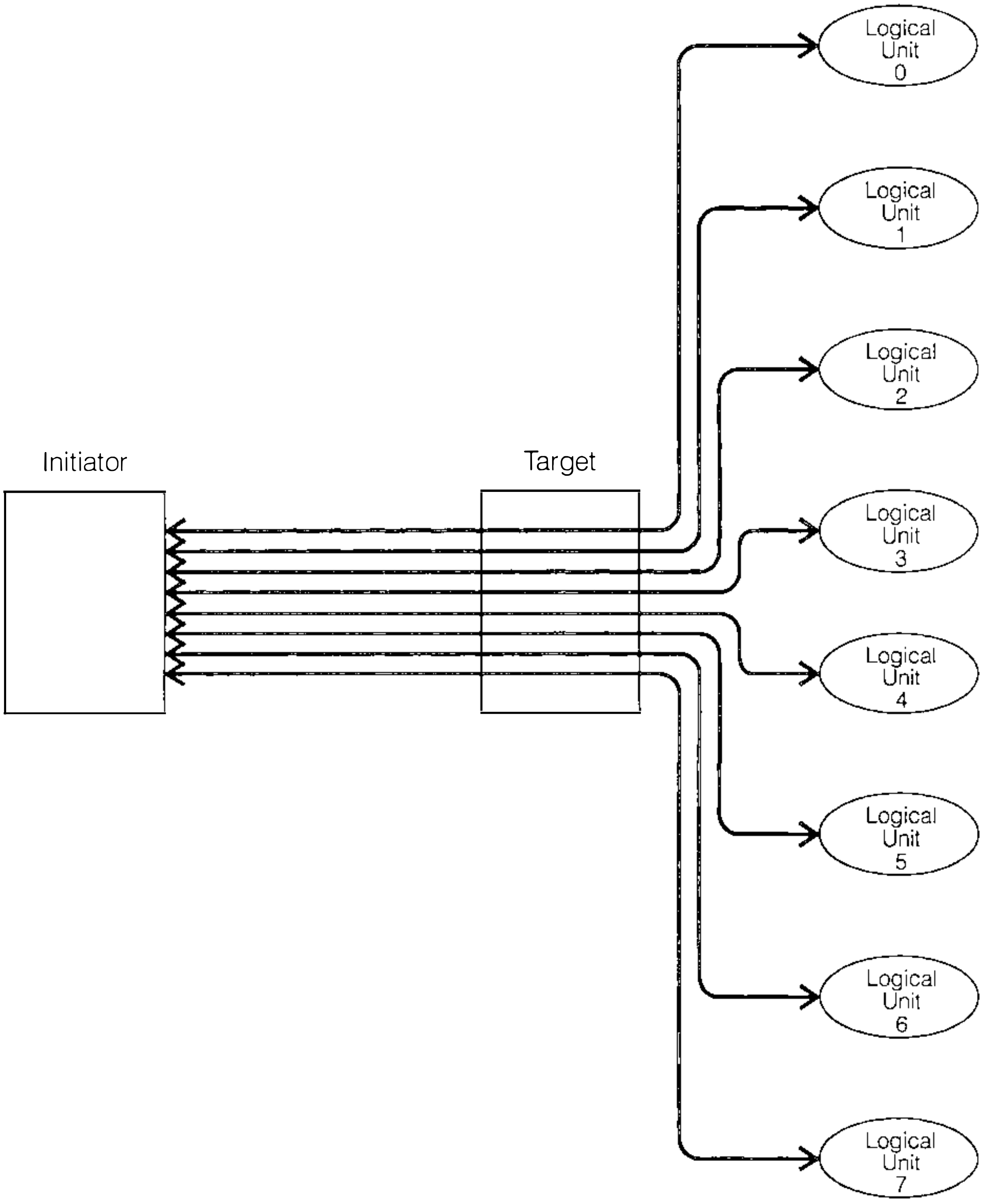


DIAGRAM 41: TARGET AND LOGICAL UNITS

Logical Unit Number (LUN). The Logical Unit Number (or LUN, some people pronounce it "loon", we prefer "ell-you-enn") is the address of the ***Logical Unit*** within a Target. As described and shown above, each Target may have up to eight Logical Units. The Logical Units on a Target may be allocated LUNs arbitrarily, except that there must be a LUN #0; a Target with a single Logical Unit will always assign it to LUN #0.

The LUN is used by the Initiator and Target to specify which Logical Unit is being referred to. The ***IDENTIFY Message*** is used to exchange the LUN between the Initiator and Target. (Older ***SCSI-1*** devices that did not support the IDENTIFY Message specified the LUN in the ***Command Descriptor Block (CDB)***.) The exchange of LUN is part of establishing a ***Nexus*** between two devices.

Message.
MESSAGE IN Phase.
MESSAGE OUT Phase.
MESSAGE PARITY ERROR Message.
MESSAGE REJECT Message.
Message System.
MODIFY DATA POINTER Message.
MSG Signal.

Message. A Message is a unit of information sent:

- from an Initiator to a Target during a ***MESSAGE OUT Phase***; or,
- from a Target to an Initiator during a ***MESSAGE IN Phase***.

A Message may be one or more bytes in length. A Message is used for:

- ***Path Control***, which manages other ***Bus Phases***.
- ***Message System***, which manages the Message transfers.

MESSAGE IN Phase. The MESSAGE IN Phase is used to transfer messages from the Target to the Initiator. See ***Message System*** for a description of how message transfers are handled in general.

Diagram 42 and Diagram 43 show how Initiators and Targets handle the MESSAGE IN Phase:

(1) The Target sets the ***Bus Phase*** to MESSAGE IN Phase and asserts the ***REQ Phase***.

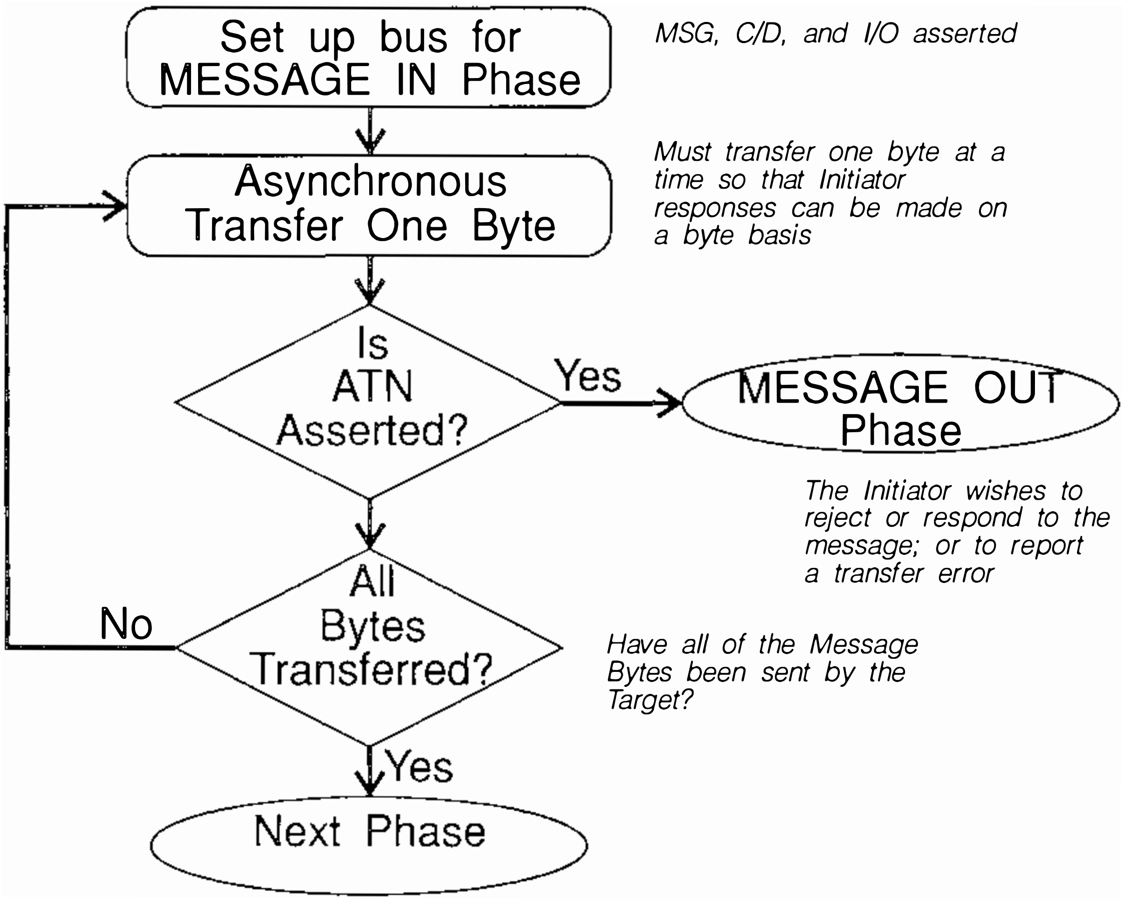
(2) The Initiator takes the message byte by asserting the ***ACK Signal***. The Target follows this by negating the REQ Signal.

(3) The Initiator then checks to see if a ***Parity*** error occurred. If it did, it asserts the ***ATN Signal***. (see ***Attention Condition***).

(4) If a complete message has now been received by the Initiator, it processes the message. If the message requires a response, or if it must be rejected, the Initiator asserts the ATN Signal.

(5) The Initiator negates the ACK Signal to complete the byte transfer (see also ***Asynchronous Data Transfer***).

(6) If the ATN Signal was asserted, the Target switches to ***MESSAGE OUT Phase*** for response or error handling. Otherwise, transfers continue until the Target has no more messages to send.



MSG, C/D, and I/O asserted

Must transfer one byte at a time so that Initiator responses can be made on a byte basis

The Initiator wishes to reject or respond to the message; or to report a transfer error

Have all of the Message Bytes been sent by the Target?

Transfer Phase or BUS FREE Phase

DIAGRAM 42: MESSAGE IN FLOW DIAGRAM FOR TARGETS

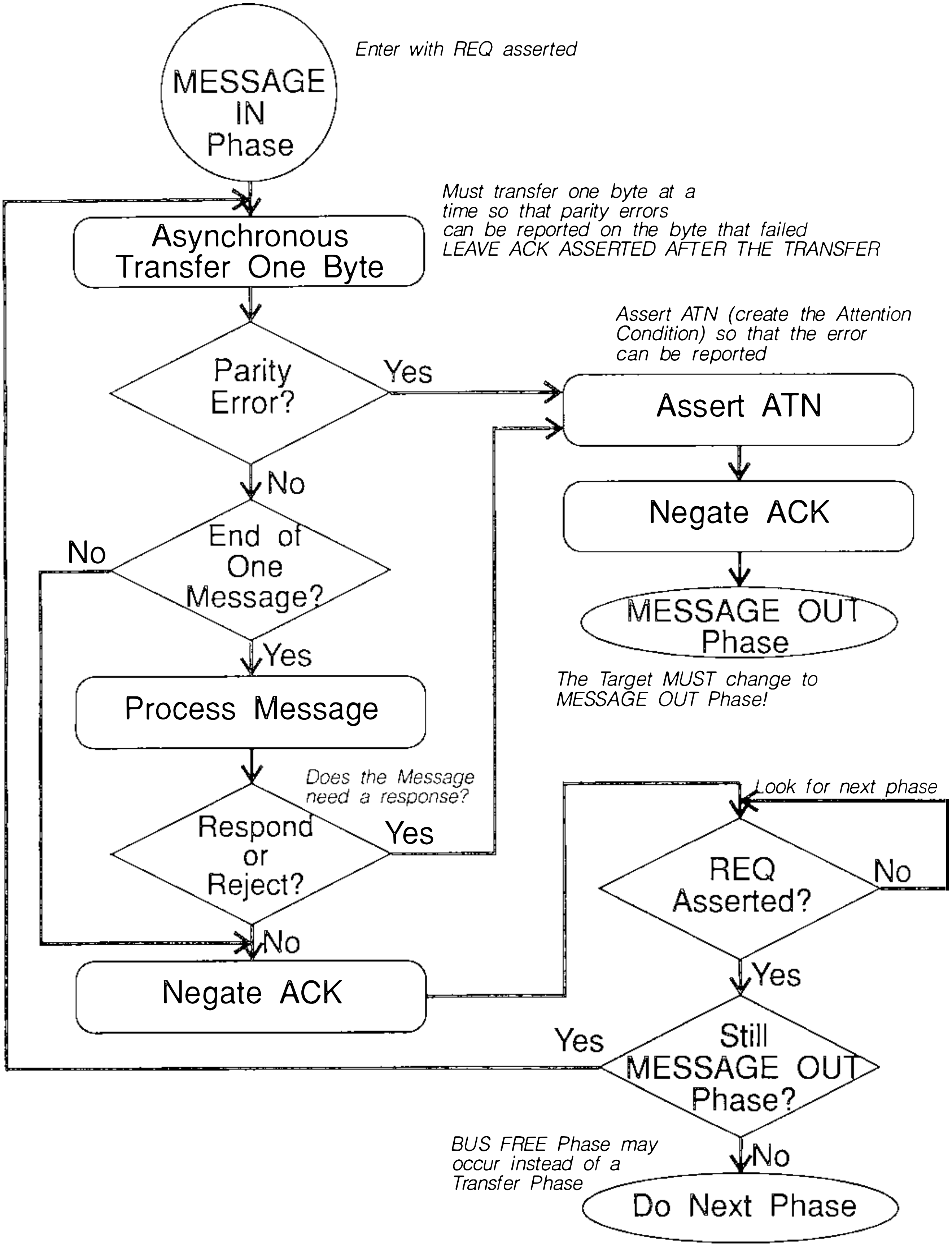

MESSAGE IN Phase
Enter with REQ asserted
Must transfer one byte at a time so that parity errors can be reported on the byte that failed
LEAVE ACK ASSERTED AFTER THE TRANSFER
Asynchronous Transfer One Byte
Parity Error?
Yes
No
Assert ATN (create the Attention Condition) so that the error can be reported
Assert ATN
Negate ACK
MESSAGE OUT Phase
The Target MUST change to MESSAGE OUT Phase!
End of One Message?
No
Yes
Process Message
Does the Message need a response?
Respond or Reject?
Yes
No
Negate ACK
Look for next phase
REQ Asserted?
No
Yes
Still MESSAGE OUT Phase?
Yes
No
BUS FREE Phase may occur instead of a Transfer Phase
Do Next Phase

DIAGRAM 43: MESSAGE IN FLOW DIAGRAM FOR INITIATORS

This page is nearly blank!

MESSAGE OUT Phase. The MESSAGE OUT Phase is used to transfer messages from the Initiator to the Target. See ***Message System*** for a description of how message transfers are handled in general.

Diagram 44 and Diagram 45 show how Initiators and Targets handle the MESSAGE OUT Phase:

(1) The Initiator requests a MESSAGE OUT Phase to send one or more ***Messages*** by creating the ***Attention Condition*** during a ***Bus Phase***.

(2) The Target sets the Bus Phase to MESSAGE OUT Phase and asserts the ***REQ Phase***.

(3) The Initiator takes the message byte by asserting the ***ACK Signal***. The Target follows this by negating the REQ Signal, and the Initiator negates the ACK Signal.

(4) The Target then checks to see if a ***Parity*** error occurred. If it did, it continues the transfer until the ***ATN Signal*** goes false, discarding the bytes.

(5) The Initiator continues transferring until all message bytes have been transferred except one. The Initiator then negates the ***ATN Signal*** after the REQ Signal for the final byte transfer has been asserted.

(6) When the Target completes a byte transfer and sees the ATN Signal has been negated, it knows the transfer is complete. If there was no Parity error, it may then change to another phase.

(7) If there was a Parity error, after the Target has completed transferring and discarding the message bytes, it asserts the REQ Signal while staying in the MESSAGE OUT Phase. The Initiator sees the REQ Signal and asserts the ***ATN Signal*** in response. The message(s) are re-transmitted as in #3 above. Any of the messages transferred before the Parity error that the Target processed should be ignored by the Target during the retry. Don't process them twice!

(8) If the Target must respond to or reject a particular message, it switches to ***MESSAGE IN Phase*** to send the response, even if the ATN Signal is still asserted.

It should be noted that the Initiator is allowed to negate the ATN Signal (subject to the rules described in the ***Attention Condition***) prior to the MESSAGE OUT Phase. This might occur if the Initiator decides to send a Message, and then conditions change and the Message is no longer necessary. Even though it is allowed, it is better if the Initiator leaves ATN asserted until it is serviced by the Target. The Initiator can then send the ***NOP Message*** to clear the Attention Condition.

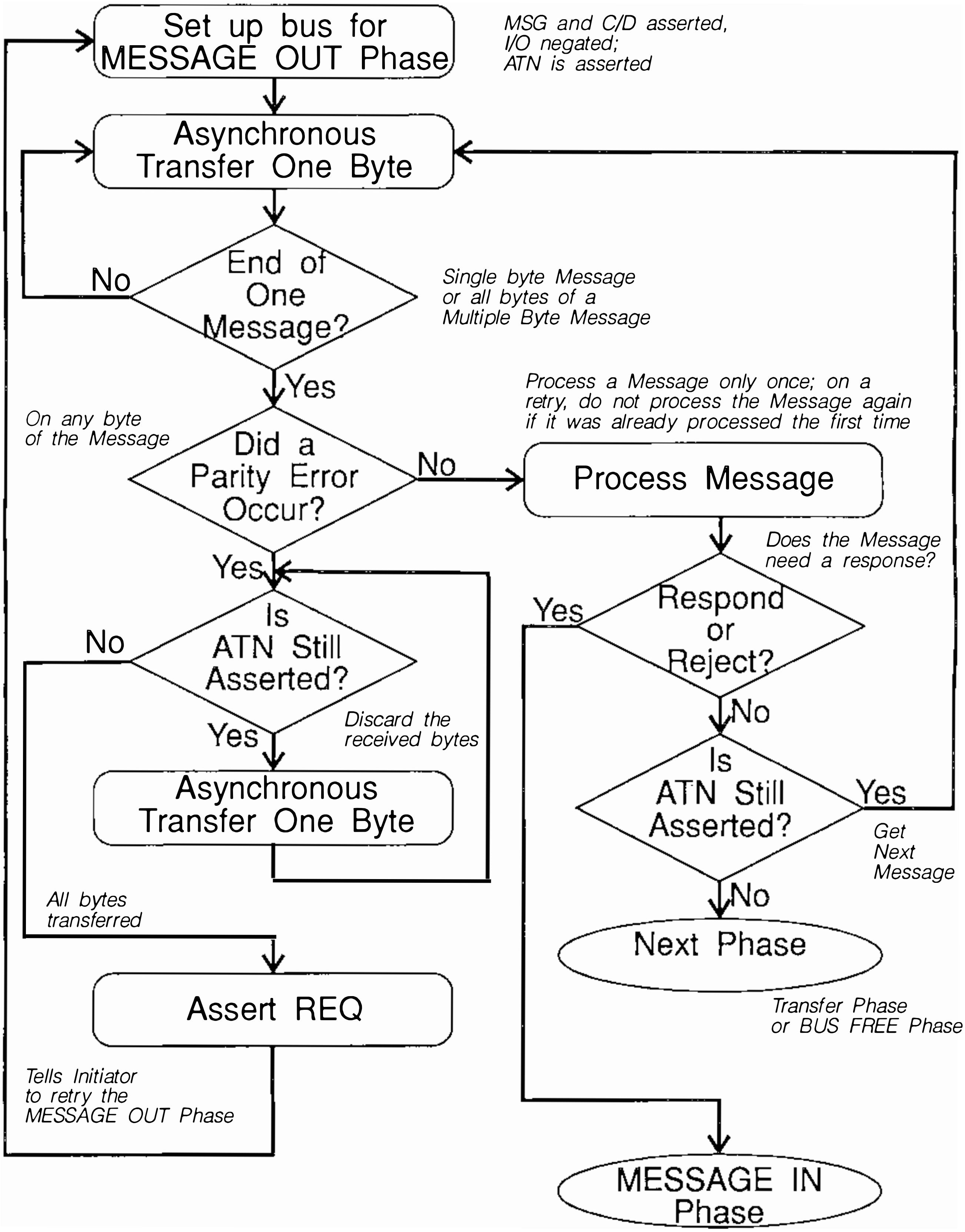

Set up bus for MESSAGE OUT Phase
MSG and C/D asserted, I/O negated; ATN is asserted
Asynchronous Transfer One Byte
End of One Message?
No
Single byte Message or all bytes of a Multiple Byte Message
Yes
Process a Message only once; on a retry, do not process the Message again if it was already processed the first time
On any byte of the Message
Did a Parity Error Occur?
No
Process Message
Does the Message need a response?
Yes
Respond or Reject?
Yes
No
Is ATN Still Asserted?
No
Is ATN Still Asserted?
Yes
Yes
Get Next Message
Discard the received bytes
Asynchronous Transfer One Byte
No
Next Phase
All bytes transferred
Transfer Phase or BUS FREE Phase
Assert REQ
Tells Initiator to retry the MESSAGE OUT Phase
MESSAGE IN Phase

DIAGRAM 44: MESSAGE OUT FLOW DIAGRAM FOR TARGETS

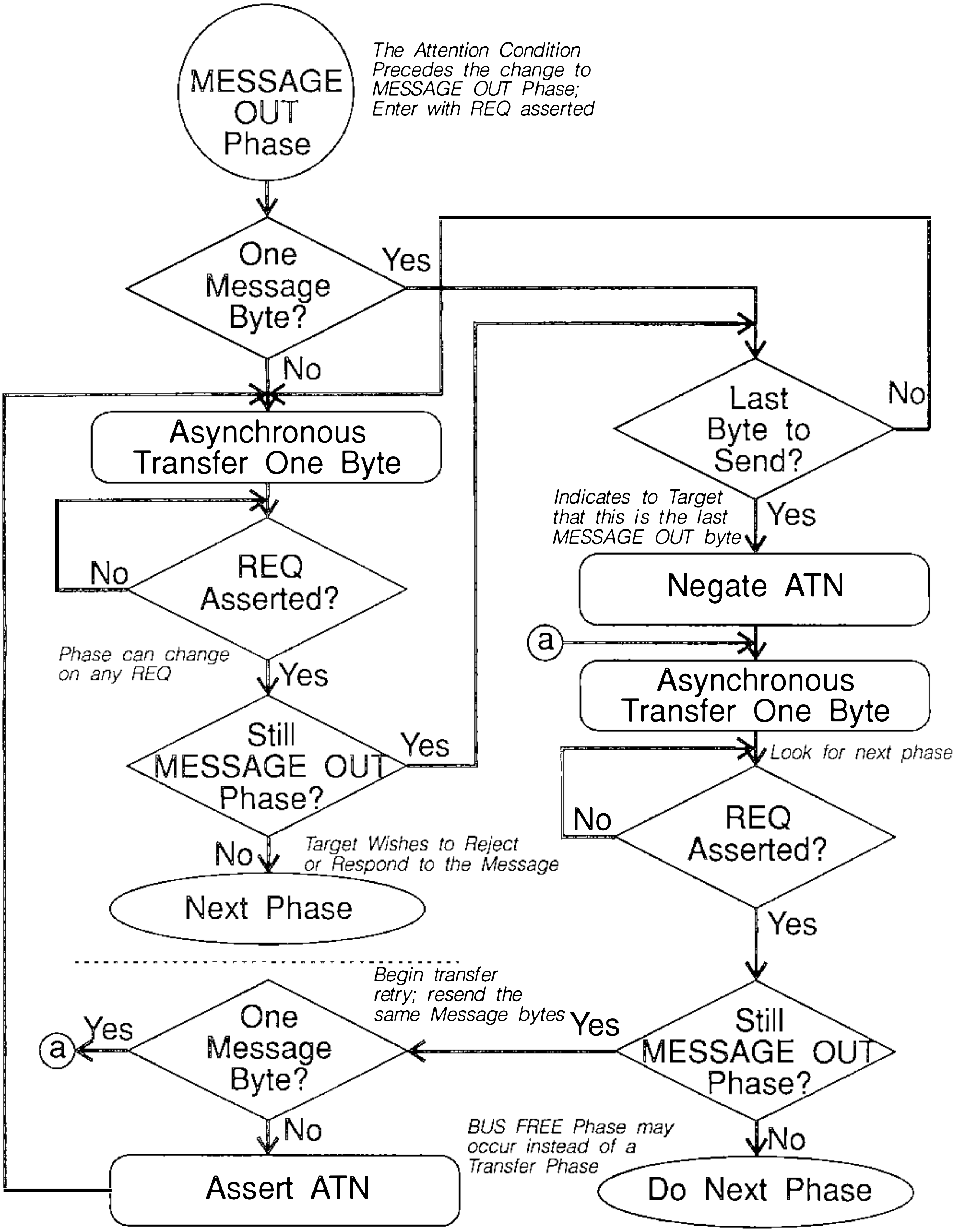

MESSAGE OUT Phase
The Attention Condition Precedes the change to MESSAGE OUT Phase; Enter with REQ asserted
One Message Byte?
Yes
No
Asynchronous Transfer One Byte
No
REQ Asserted?
Phase can change on any REQ
Yes
Still MESSAGE OUT Phase?
Yes
No
Target Wishes to Reject or Respond to the Message
Next Phase
Last Byte to Send?
No
Indicates to Target that this is the last MESSAGE OUT byte
Yes
Negate ATN
a
Asynchronous Transfer One Byte
Look for next phase
No
REQ Asserted?
Yes
Begin transfer retry; resend the same Message bytes
Yes
Still MESSAGE OUT Phase?
Yes
a
One Message Byte?
No
Assert ATN
BUS FREE Phase may occur instead of a Transfer Phase
No
Do Next Phase

DIAGRAM 45: MESSAGE OUT FLOW DIAGRAM FOR INITIATORS

MESSAGE PARITY ERROR Message. This is a message that is sent by an Initiator in response to a ***MESSAGE IN Phase*** byte that had a ***Parity*** error in it. MESSAGE PARITY ERROR is a single byte message:

Bit Byte	7	6	5	4	3	2	1	0
0	Message Code = 09 hex							

This message is **only** sent from an Initiator to a Target. This message is similar to the ***INITIATOR DETECTED ERROR Message***, which is used to report errors in other ***Information Transfer Phases***. If the Target detects a parity error, there is a different protocol used. See ***Message System*** for a description of parity error recovery for message phases.

Summary of Use: The MESSAGE PARITY ERROR message is sent only by an Initiator to inform the Target that a parity error occurred in the transfer of a byte in a MESSAGE IN Phase.

MESSAGE REJECT Message. The MESSAGE REJECT Message is sent by an Initiator or a Target to the other ***SCSI Device*** when that device sends a message that is not appropriate for the current situation. This is a single byte message:

Bit Byte	7	6	5	4	3	2	1	0
0	Message Code = 07 hex							

Situations where a message might not be appropriate are:

- The message cannot be understood by the receiving device. In this case, the receiving device probably has not implemented the message it received. An example of this is a device that has not implemented ***Synchronous Data Transfer***. It has no reason, then, to implement the ***SYNCHRONOUS DATA TRANSFER REQUEST Message***, and would reject that message if it was ever received.

- A ***Reserved*** bit or field is set in the message. In this case, the receiving device understands the basic message, but an option bit is set, or a field is set to an unexpected value. In this case the message should be rejected since the receiving device may not perform the intended action if it continues. An example is when a Target receives an ***IDENTIFY Message*** with the

LUNTAR bit set. If the Target does not support ***Target Routines***, it should reject the IDENTIFY Message (there is an alternate response for this case described under IDENTIFY Message, just fyi...).

- The message received is just plain off-the-wall and out-of-the-blue. An example would be for a Target to receive a ***MESSAGE PARITY ERROR Message*** when no ***MESSAGE IN Phase*** preceded it (see ***Message System*** for why this is bad). In this case, it is likely that either the Initiator is a little bit nuts, or two bits were in error during the transfer (***Parity*** did not detect a fault).

- The message received cannot be acted on at this time. This is very implementation dependent. For example, if a Target sends the ***DISCONNECT Message*** and the Initiator just cannot ***Disconnect*** right now, it may reject the message. Likewise, if the Initiator sends the DISCONNECT Message to the Target, the Target may reject it if a Disconnect is not possible right now.

- The message received is just not allowed for that device role. If an Initiator or a Target sends a message not defined for the role, the receiver should reject it and may have to assume the sender is broken. An example is an Initiator that sends a ***COMMAND COMPLETE Message*** (which is not allowed for Initiators). See ***Path Control*** for a table that summarizes the allowed transfer direction for each message.

The Target sends the MESSAGE REJECT Message by changing to MESSAGE IN Phase immediately after receiving the offending message during the preceding MESSAGE OUT Phase. It must be done immediately so that the Initiator knows which message was offensive. This does imply that the Target must interpret the Message for "appropriateness" at this time, even if it doesn't do the actual processing until later. If the ***ATN Signal*** remains asserted after the MESSAGE REJECT Message is sent, then the Target must return to MESSAGE OUT Phase and hear what else the Initiator has to say.

The Initiator sends the MESSAGE REJECT Message by creating the ***Attention Condition*** before completing the handshake for the last byte of the offending message. This means that the Initiator must be interpreting messages while receiving them; the ATN Signal must be asserted before the ***ACK Signal*** is negated for the last message byte. The Target must then respond with a MESSAGE OUT Phase so that the MESSAGE REJECT Message may be sent.

There is more discussion of the MESSAGE REJECT Message in ***Message System***. Table 34 lists all messages and when it is and isn't appropriate to reject them. Note that it is always appropriate to reject an unimplemented or garbled message.

Summary of Use: The MESSAGE REJECT message is sent by either an Initiator or a Target to tell the other device that the last message sent was not understood or not appropriate.

Table 34: MESSAGE REJECT Usage

Message Name	A MESSAGE REJECT is or is not an appropriate response when:
ABORT	MESSAGE REJECT is never an appropriate response.
ABORT TAG	May be rejected only if it is not preceded by a Queue Tag Message in the same Connection.
BUS DEVICE RESET	MESSAGE REJECT is never an appropriate response.
CLEAR QUEUE	May be rejected only if it is not preceded by a Queue Tag Message in the same Connection.
COMMAND COMPLETE	May be rejected if it does not follow a STATUS Phase.
DISCONNECT (Out)	May be rejected if the Target cannot disconnect now.
DISCONNECT (In)	May be rejected if the Initiator cannot disconnect now.
HEAD OF QUEUE TAG	May be rejected if Queuing is not supported, or if it is sent any time other than after the IDENTIFY Message during an Initial Connection.
IDENTIFY (Out)	May be rejected if LUNTAR is set and Target Routines are not supported.
IDENTIFY (In)	MESSAGE REJECT is never an appropriate response; ABORT is the appropriate response to an IDENTIFY Message for a non-existent Nexus, or with the DiscPriv bit or other Reserved bit set.
IGNORE WIDE RESIDUE	May be rejected if it is received at any time other than after a Wide Data Transfer in a DATA IN Phase.
INITIATE RECOVERY (Out)	May be rejected if an Extended Contingent Allegiance Condition (ECA) is not desired by the Target.
INITIATE RECOVERY (In)	May be rejected if an ECA Condition is not desired by the Initiator.
INITIATOR DETECTED ERROR	May be rejected if received in response to a MESSAGE IN Phase; the MESSAGE PARITY ERROR Message is the appropriate response to a MESSAGE IN Phase error.
LINKED COMMAND COMPLETE	May be rejected if it does not follow a STATUS Phase.
LINKED COMMAND COMPLETE (WITH FLAG)	May be rejected if it does not follow a STATUS Phase.
MESSAGE PARITY ERROR	May be rejected if it does not follow a MESSAGE IN Phase.
MESSAGE REJECT	MESSAGE REJECT is never an appropriate response!
MODIFY DATA POINTER	May be rejected if it exceeds the Host buffer limits, or if it follows STATUS Phase, or if it precedes the COMMAND Phase.
NO OPERATION	MESSAGE REJECT is never an appropriate response.
ORDERED QUEUE TAG	May be rejected if Queuing is not supported, or if it is sent any time other than after the IDENTIFY Message during an Initial Connection.
RELEASE RECOVERY	MESSAGE REJECT is never an appropriate response.
RESTORE POINTER	MESSAGE REJECT is never an appropriate response.
SAVE DATA POINTER	MESSAGE REJECT is never an appropriate response.
SIMPLE QUEUE TAG (Out)	May be rejected if Queuing is not supported, or if it is sent any time other than after the IDENTIFY Message during an Initial Connection.
SIMPLE QUEUE TAG (In)	May be rejected if it is sent any time other than after the IDENTIFY Message during a Reconnection. If the Queue Tag and Logical Unit Number (LUN) sent during the Reconnection specify a non-existent Nexus, the appropriate response is the ABORT TAG or ABORT Message.
SYNCHRONOUS DATA TRANSFER REQUEST	May be rejected if Synchronous Data Transfer is not supported.
TERMINATE I/O PROCESS	MESSAGE REJECT is never an appropriate response.
WIDE DATA TRANSFER REQUEST	May be rejected if Wide Data Transfer is not supported.

Message System. Welcome to the most confusing subject in SCSI! Our job is to make sure you enjoy your stay...

The Message System carries the Messages that communicate ***Path Control*** information between an Initiator and a Target. The Message System uses the ***MESSAGE OUT Phase*** to transfer messages from the Initiator to the Target. The ***MESSAGE IN Phase*** is used to transfer messages from the Target to the Initiator. The Message System is used to manage the flow of all of the other ***Bus Phases*** (see also ***Error Recovery***). An interesting (and infuriating) feature of the Message System is that it uses the MESSAGE IN and MESSAGE OUT Phases to manage themselves.

Diagram 46 shows a top-level representation of the Message System. Because the Diagram shows all possibilities, it may be difficult to follow. We will clarify it in the examples that follow. Flow moves between the following states:

- MESSAGE IN Phase: The Target enters the Message System here when it wants to begin an exchange of messages. This Phase is also used by the Target to respond to a message from the Initiator.

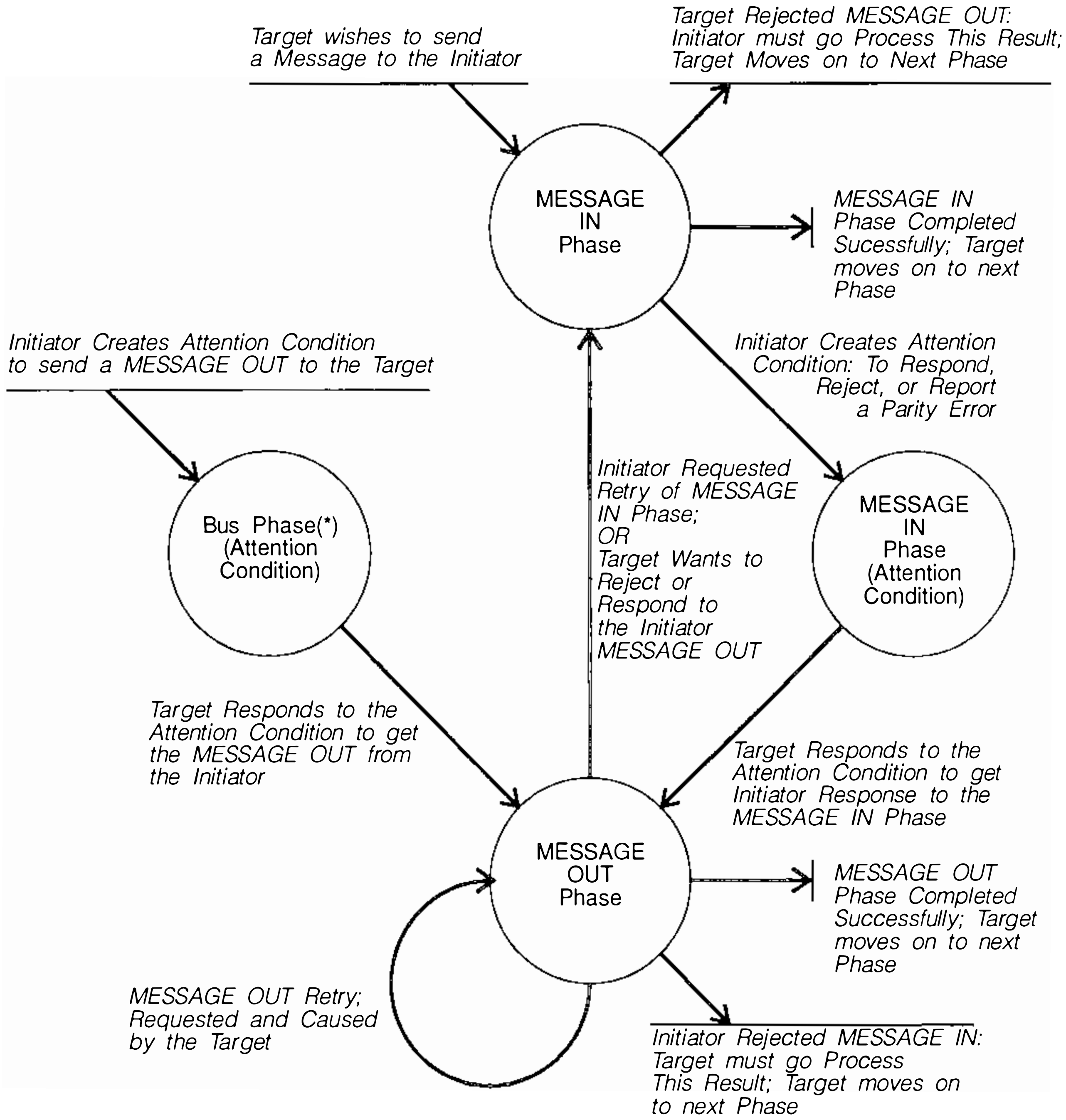
- ***Information Transfer Phase*** (Attention Condition): The Initiator enters the Message System here when it wants to begin an exchange of messages. The Initiator creates the ***Attention Condition*** during any of the following ***Bus Phases***: ***SELECTION Phase***, ***COMMAND Phase***, ***STATUS Phase***, ***DATA IN Phase***, and ***DATA OUT Phase***. This tells the Target that the Initiator wants to send a message.

- MESSAGE OUT Phase: The Target acknowledges the Attention Condition by changing to the MESSAGE OUT Phase. Recovery from errors detected during MESSAGE OUT Phase is handled by a direct retry of the Phase. This Phase is also used by the Initiator to respond to a message from the Target.

- MESSAGE IN Phase (Attention Condition): The Initiator enters this state when it wants to respond to a message from the Target. The Initiator creates the Attention Condition during MESSAGE IN Phase in response to a Target message. The response could also be a rejection or a report of an error.

Detail of these states is shown in the sections on MESSAGE IN Phase and MESSAGE OUT Phase.

We will be using portions of the diagram on the following page to develop examples of message responses and error recovery retries. Refer to the sections on ***MESSAGE IN Phase*** and ***MESSAGE OUT Phase*** for the details of those phases.


Target wishes to send a Message to the Initiator
Target Rejected MESSAGE OUT: Initiator must go Process This Result; Target Moves on to Next Phase
MESSAGE IN Phase
MESSAGE IN Phase Completed Sucessfully; Target moves on to next Phase
Initiator Creates Attention Condition to send a MESSAGE OUT to the Target
Initiator Creates Attention Condition: To Respond, Reject, or Report a Parity Error
Initiator Requested Retry of MESSAGE IN Phase; OR Target Wants to Reject or Respond to the Initiator MESSAGE OUT
Bus Phase(*) (Attention Condition)
MESSAGE IN Phase (Attention Condition)
Target Responds to the Attention Condition to get the MESSAGE OUT from the Initiator
Target Responds to the Attention Condition to get Initiator Response to the MESSAGE IN Phase
MESSAGE OUT Phase
MESSAGE OUT Phase Completed Successfully; Target moves on to next Phase
MESSAGE OUT Retry; Requested and Caused by the Target
Initiator Rejected MESSAGE IN: Target must go Process This Result; Target moves on to next Phase

(*)Bus Phases: SELECTION, RESELECTION, COMMAND, STATUS, DATA IN, or DATA OUT Phase

DIAGRAM 46: MESSAGE SYSTEM TOP LEVEL DIAGRAM

Simple Examples See Diagram 47.

(1) The Target sends a message to the Initiator:

(a) The Target enters the MESSAGE IN Phase.

(b) The Target uses an ***Asynchronous Data Transfer*** to send the message to the Initiator.

(c) The Initiator accepts the message from the Target.

(d) The Target goes to another Phase.

(2) The Initiator sends a message to the Target:

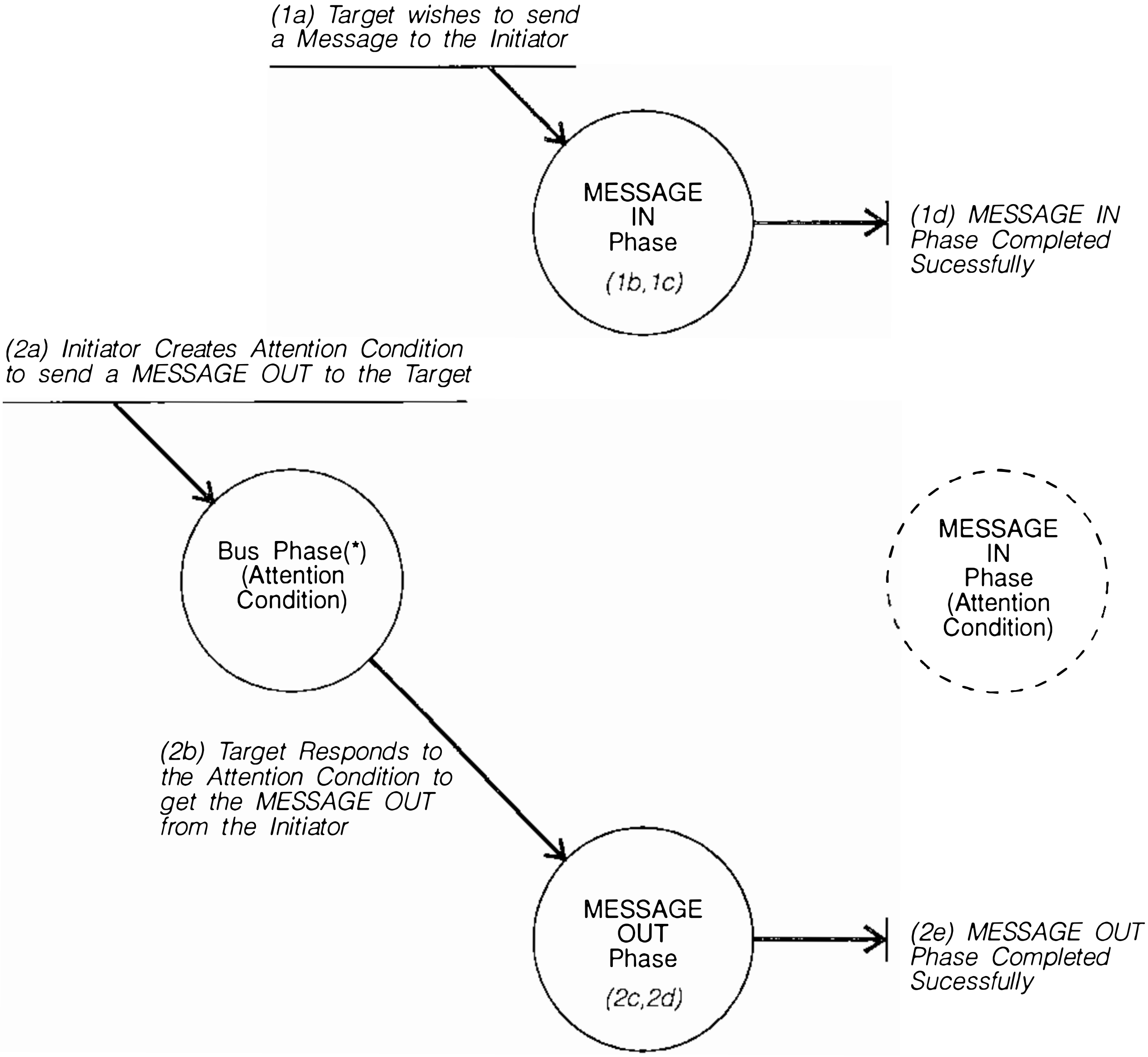
(a) The Initiator creates the Attention Condition in an appropriate phase.

(b) The Target recognizes the Attention Condition and changes to MESSAGE OUT Phase.

(c) The Initiator uses an ***Asynchronous Data Transfer*** to send the message to the Target. The Initiator negates the ***ATN Signal*** before sending the last message byte to indicate that it has sent all of the bytes it intended.

(d) The Target accepts the message from the Initiator.

(e) The Target goes to another Phase.


(1a) Target wishes to send a Message to the Initiator
MESSAGE IN Phase
(1b,1c)
(1d) MESSAGE IN Phase Completed Sucessfully
(2a) Initiator Creates Attention Condition to send a MESSAGE OUT to the Target
Bus Phase(*) (Attention Condition)
MESSAGE IN Phase (Attention Condition)
(2b) Target Responds to the Attention Condition to get the MESSAGE OUT from the Initiator
MESSAGE OUT Phase
(2c,2d)
(2e) MESSAGE OUT Phase Completed Sucessfully

(*)Bus Phases: SELECTION, RESELECTION, COMMAND, STATUS, DATA IN, or DATA OUT Phase

DIAGRAM 47: SIMPLE MESSAGE IN AND OUT PHASES

Messages and Responses (Not So Simple Examples...)

(3) The Target sends a message to the Initiator, and the Initiator rejects it (Diagram 48):

(a) The Target enters the MESSAGE IN Phase.

(b) The Target uses an ***Asynchronous Data Transfer*** to send the message to the Initiator.

(c) The Initiator decides that it cannot accept the message (see ***MESSAGE REJECT Message***), so it creates the Attention Condition prior to negating the ***ACK Signal***.

(d) The Target sees the ***ATN Signal*** after the message transfer and changes to MESSAGE OUT Phase.

(e) The Initiator uses an ***Asynchronous Data Transfer*** to send the MESSAGE REJECT Message to the Target. This tells the Target that the Initiator could not accept the message. It does not necessarily indicate an error; specifically, the Target need not retry sending the same message again. The Initiator will reject it again. The Initiator negates the ***ATN Signal*** before sending the message byte to indicate that it is the only message it intended to send.

(f) The Target accepts the message from the Initiator.

(g) The Target goes to another Phase. Depending on the message that was rejected, the Target may have to continue with a different course of action. For example, the Target may have decided to ***Disconnect*** from the bus. If its ***DISCONNECT Message*** was rejected, then the Target would have to continue with the data transfer instead.

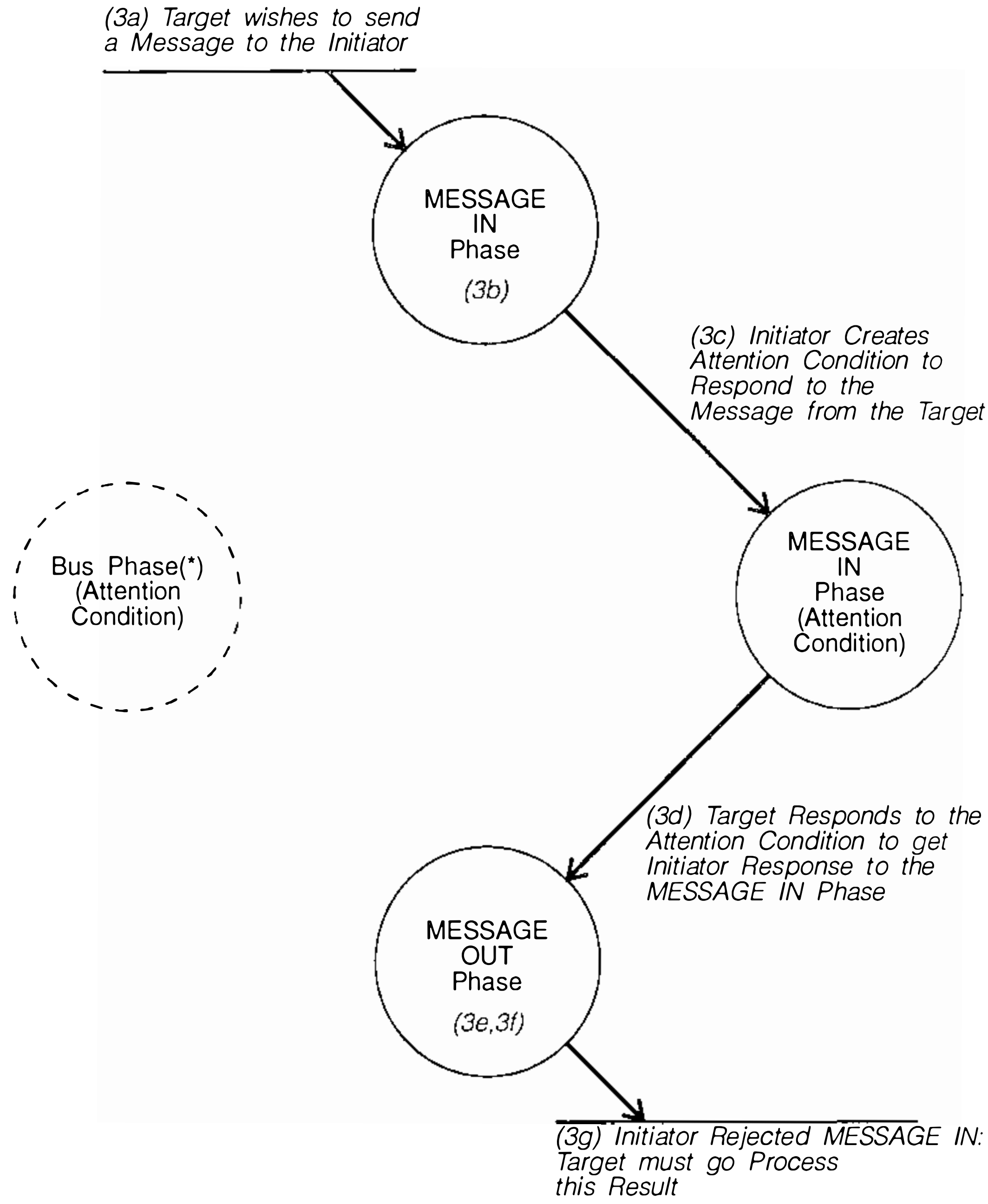

(3a) Target wishes to send a Message to the Initiator
MESSAGE IN Phase
(3b)
(3c) Initiator Creates Attention Condition to Respond to the Message from the Target
Bus Phase(*) (Attention Condition)
MESSAGE IN Phase (Attention Condition)
(3d) Target Responds to the Attention Condition to get Initiator Response to the MESSAGE IN Phase
MESSAGE OUT Phase
(3e,3f)
(3g) Initiator Rejected MESSAGE IN: Target must go Process this Result

DIAGRAM 48: TARGET MESSAGE IN WITH INITIATOR REJECT

(4) The Initiator sends a message to the Target, and the Target rejects it (Diagram 49):

(a) The Initiator creates the Attention Condition.

(b) The Target recognizes the Attention Condition and changes to MESSAGE OUT Phase.

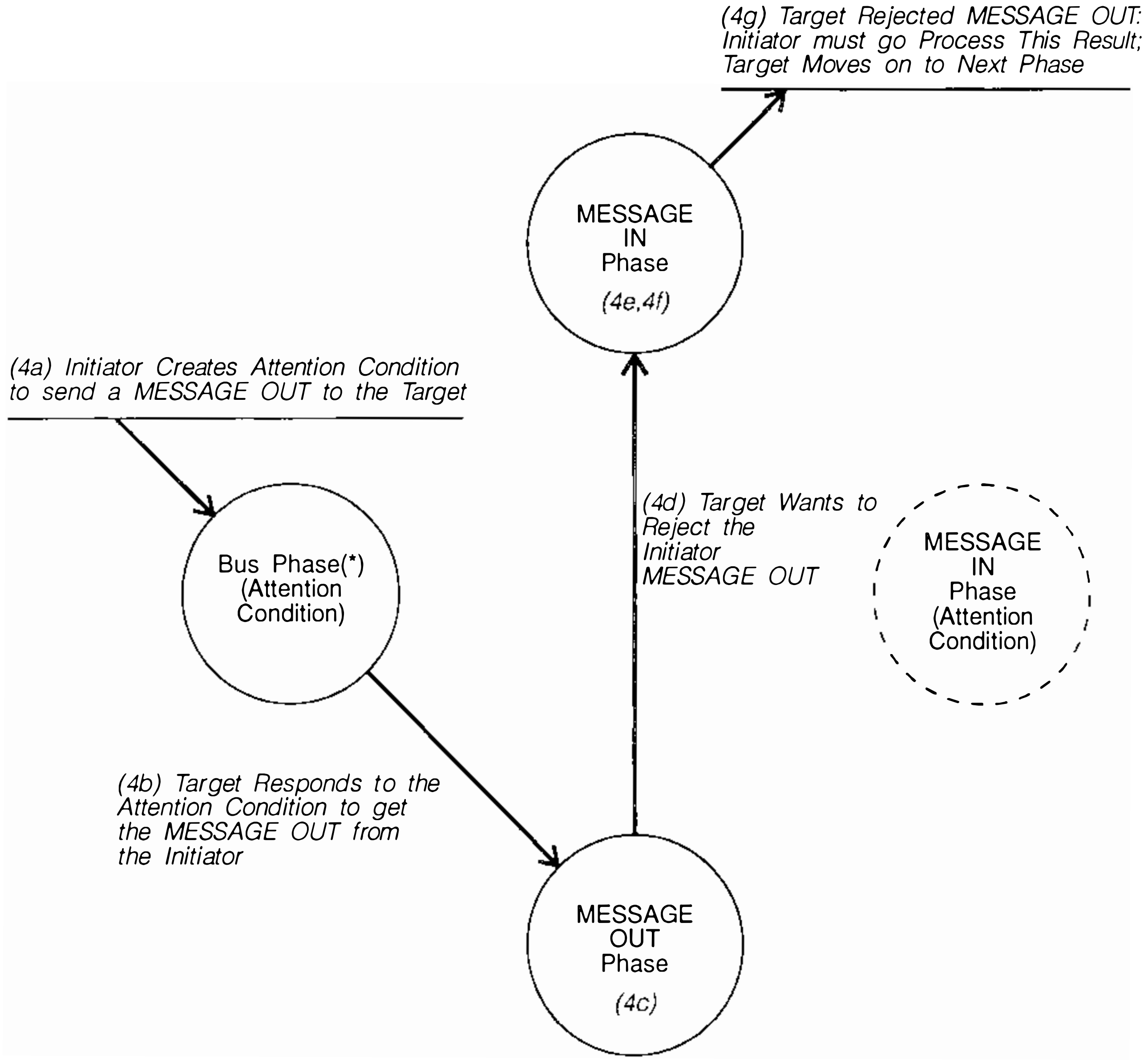
(c) The Initiator uses an ***Asynchronous Data Transfer*** to send the message to the Target. The Initiator negates the ***ATN Signal*** before sending the last message byte to indicate that it has sent all of the bytes it intended.

(d) The Target decides that it cannot accept the message (see ***MESSAGE REJECT Message***), so it changes to MESSAGE IN Phase.

(e) The Target uses an ***Asynchronous Data Transfer*** to send the MESSAGE REJECT Message to the Initiator. This tells the Initiator that the Target could not accept the message. It does <u>not</u> necessarily indicate an error; specifically, the Initiator need not retry sending the same message again. The Target will reject it again.

(f) The Initiator accepts the message from the Target.

(g) The Target goes to another Phase. Depending on the message that was rejected, the Initiator may have to continue with a different course of action. For instance, the Initiator may have decided to end the ***I/O Process*** early. If the ***TERMINATE I/O PROCESS Message*** was rejected, then the Initiator would probably issue an ***ABORT Message***.


(4g) Target Rejected MESSAGE OUT: Initiator must go Process This Result; Target Moves on to Next Phase
MESSAGE IN Phase (4e,4f)
(4a) Initiator Creates Attention Condition to send a MESSAGE OUT to the Target
Bus Phase(*) (Attention Condition)
(4d) Target Wants to Reject the Initiator MESSAGE OUT
MESSAGE IN Phase (Attention Condition)
(4b) Target Responds to the Attention Condition to get the MESSAGE OUT from the Initiator
MESSAGE OUT Phase (4c)

(*)Bus Phases: SELECTION, RESELECTION, COMMAND, STATUS, DATA IN, or DATA OUT Phase

DIAGRAM 49: INITIATOR MESSAGE OUT WITH TARGET REJECT

Note how similar the following examples are to the previous two examples. You will see that rejecting the message and a response to the message are virtually the same. These examples illustrate how ***Synchronous Data Transfer Negotiation (SDTN)*** and ***Wide Data Transfer Negotiation (WDTN)*** work.

(5) The Target sends a message to the Initiator, and the Initiator responds to it (Diagram 50):

(a) The Target enters the MESSAGE IN Phase.

(b) The Target uses an ***Asynchronous Data Transfer*** to send the message to the Initiator.

(c) The Initiator decides that it can accept the message (e.g., see ***Synchronous Data Transfer Negotiation***), but it must send a response, so it creates the Attention Condition prior to negating the ***ACK Signal*** for the last byte of the Message.

(d) The Target sees the ***ATN Signal*** after the message transfer and changes to MESSAGE OUT Phase.

(e) The Initiator uses an ***Asynchronous Data Transfer*** to send the response to the Target. The Initiator negates the ***ATN Signal*** before sending the last message byte to indicate that it has sent all of the bytes it intended.

(f) The Target accepts the response message from the Initiator.

(g) The Target goes to another Phase. Depending on the response message, the Initiator and the Target may have to continue with a different course of action. For instance, the Target may have decided to begin the Synchronous Data Transfer Negotiation mentioned above. The Initiator may respond in a way that allows a ***Synchronous Data Transfer***; or, it may have responded that it could not do what the Target requested, and the net result is that Asynchronous Data Transfer must be used.

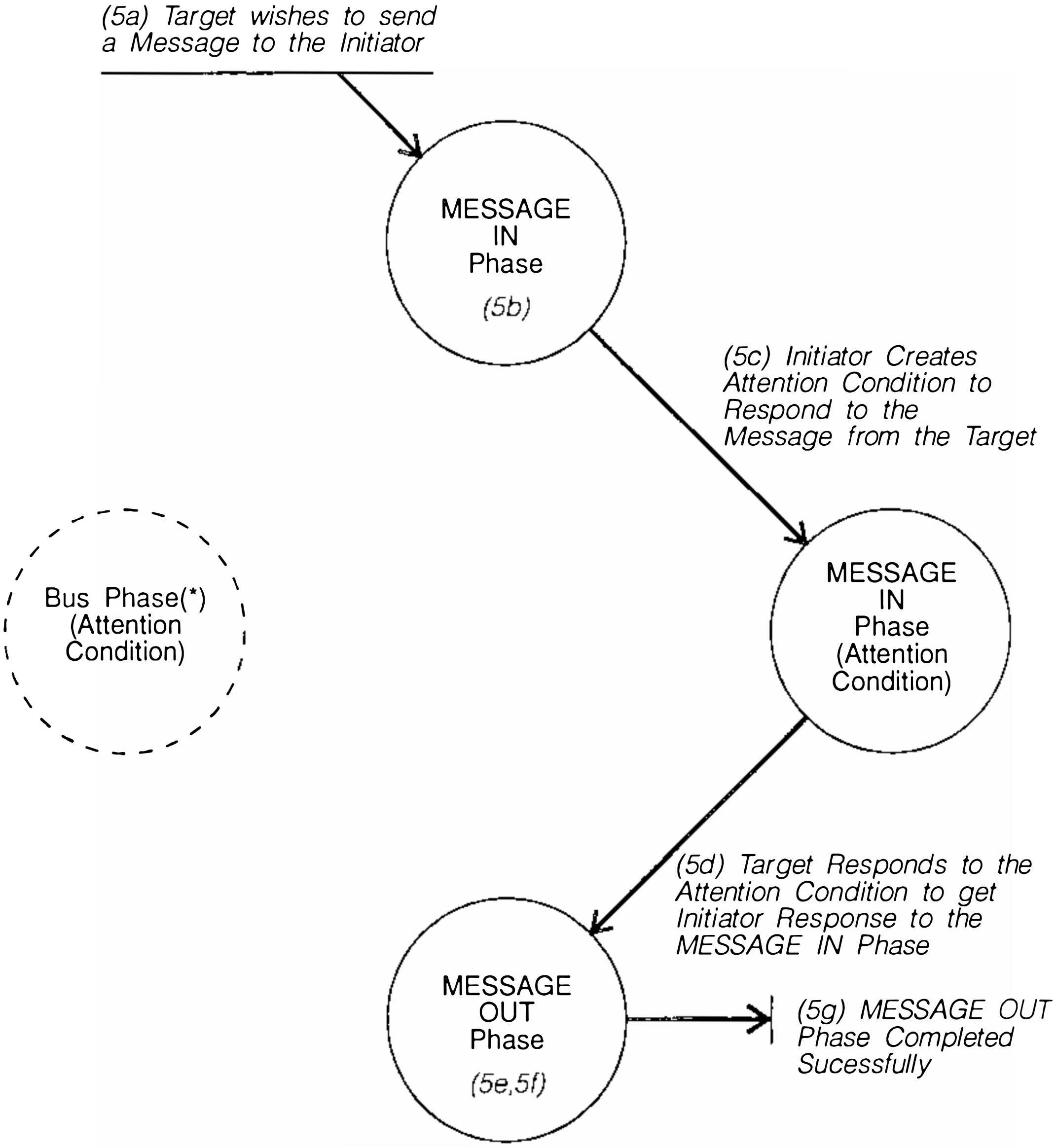

(5a) Target wishes to send a Message to the Initiator
MESSAGE IN Phase
(5b)
(5c) Initiator Creates Attention Condition to Respond to the Message from the Target
Bus Phase(*) (Attention Condition)
MESSAGE IN Phase (Attention Condition)
(5d) Target Responds to the Attention Condition to get Initiator Response to the MESSAGE IN Phase
MESSAGE OUT Phase
(5e,5f)
(5g) MESSAGE OUT Phase Completed Sucessfully

DIAGRAM 50: TARGET MESSAGE IN WITH INITIATOR RESPONSE

(6) The Initiator sends a message to the Target, and the Target responds to it (Diagram 51):

(a) The Initiator creates the Attention Condition.

(b) The Target recognizes the Attention Condition and changes to MESSAGE OUT Phase.

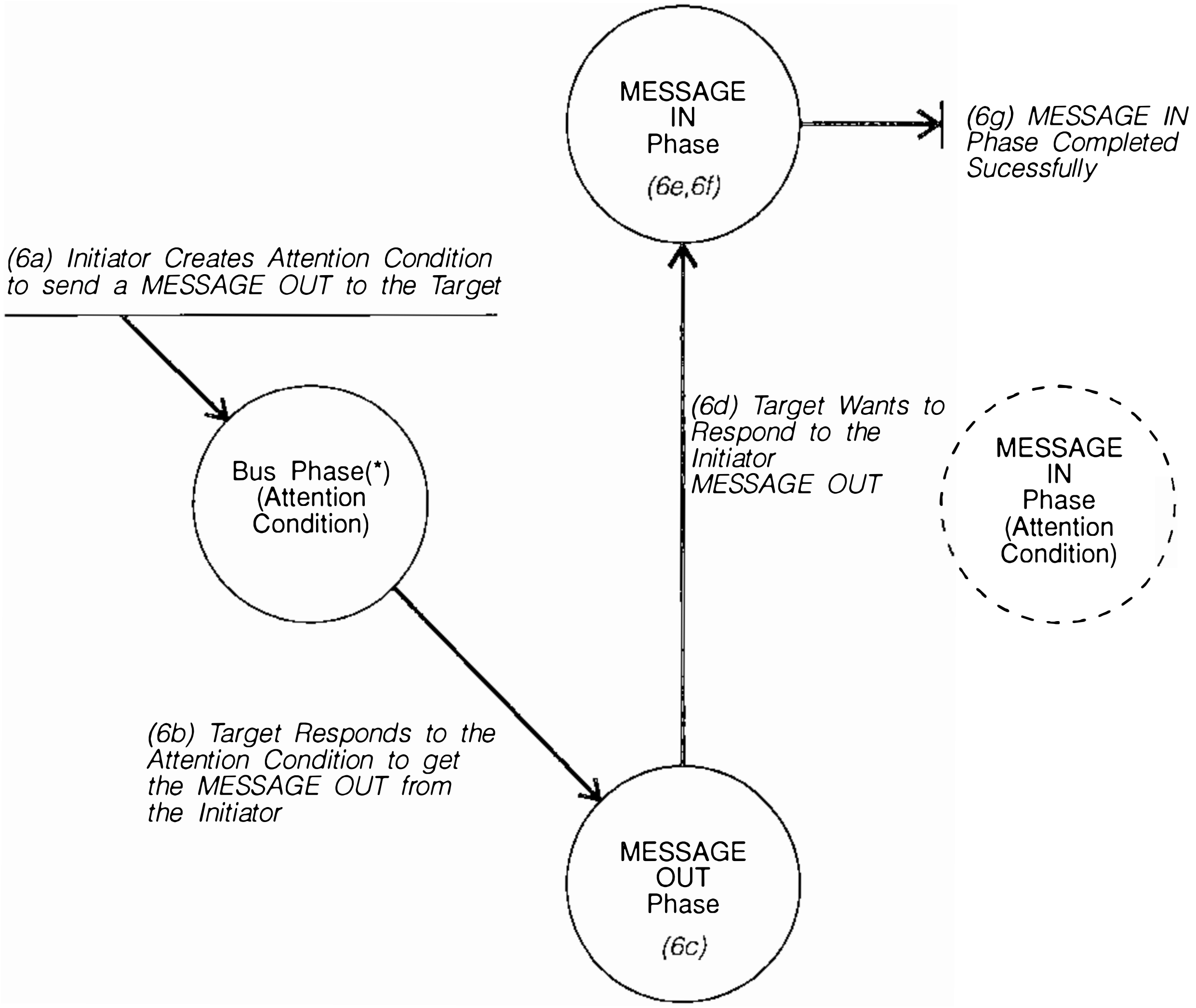
(c) The Initiator uses an ***Asynchronous Data Transfer*** to send the message to the Target. The Initiator negates the ***ATN Signal*** before sending the last message byte to indicate that it has sent all of the bytes it intended.

(d) The Target decides that it can accept the message, but the message demands a response, so it changes to MESSAGE IN Phase.

(e) The Target uses an ***Asynchronous Data Transfer*** to send the response to the Initiator.

(f) The Initiator accepts the response message from the Target.

(g) The Target goes to another Phase. Depending on the response message, the Initiator and the Target may have to continue with a different course of action. For instance, the Initiator may have detected an error during a data transfer, and so it sent the ***INITIATOR DETECTED ERROR Message*** to the Target. The Target may have responded with the ***RESTORE POINTERS Message***, or it could also respond with the ***DISCONNECT Message***.


MESSAGE
IN
Phase
(6e,6f)
(6g) MESSAGE IN Phase Completed Sucessfully
(6a) Initiator Creates Attention Condition to send a MESSAGE OUT to the Target
Bus Phase(*)
(Attention Condition)
(6d) Target Wants to Respond to the Initiator MESSAGE OUT
MESSAGE
IN
Phase
(Attention Condition)
(6b) Target Responds to the Attention Condition to get the MESSAGE OUT from the Initiator
MESSAGE
OUT
Phase
(6c)

(*)Bus Phases: SELECTION, RESELECTION, COMMAND, STATUS, DATA IN, or DATA OUT Phase

DIAGRAM 51: INITIATOR MESSAGE OUT - TARGET RESPONSE

Message and Error Recovery (Hard Examples) In these examples, an error recovery occurs in the message handling. These show how the Message System handles its own errors.

(7) The Target sends a message to the Initiator, and the Initiator detects a ***Parity*** error (Diagram 52):

(a) The Target enters the MESSAGE IN Phase.

(b) The Target uses an ***Asynchronous Data Transfer*** to send the message to the Initiator.

(c) The Initiator detects a Parity error during the transfer. As a result, it cannot accept the message byte just transferred. The Initiator creates the ***Attention Condition*** prior to negating the ***ACK Signal*** for the last byte of the message.

(d) The Target sees the ***ATN Signal*** after the message transfer and changes to MESSAGE OUT Phase.

(e) The Initiator uses an ***Asynchronous Data Transfer*** to send the ***MESSAGE PARITY ERROR Message*** to the Target. The Initiator negates the ***ATN Signal*** before sending the message byte to indicate that the message is the only one it intended to send.

(f) The Target accepts the message from the Initiator.

(g) The Target returns to the MESSAGE IN Phase to retry the message transfer. The retry begins with the message that contained the byte with the parity error. For example, let's say that the Target was sending three messages, which we'll call X, Y, and Z. Message X is a single byte message, Message Y is a two byte message, and Message Z is a four byte message. If the Parity error happened during:

- Message X: Send Message X again, then Y and Z.
- Message Y, first byte: Send Message Y again, then Z.
- Message Y, second byte: Send Message Y again, then Z.
- Message Z, first byte: Send Message Z only again.
- Message Z, second byte: Send Message Z only again.
- Message Z, third byte: Send Message Z only again.
- Message Z, fourth byte: Send Message Z only again.

A somewhat realistic instance of the above example would be after a ***RESELECTION Phase***. The Target then sends an ***IDENTIFY Message*** (X), followed by a ***SIMPLE QUEUE TAG Message*** (Y) (that completes the ***Reconnection***), and then a ***WIDE DATA TRANSFER REQUEST Message*** (Z) during the same MESSAGE IN Phase.

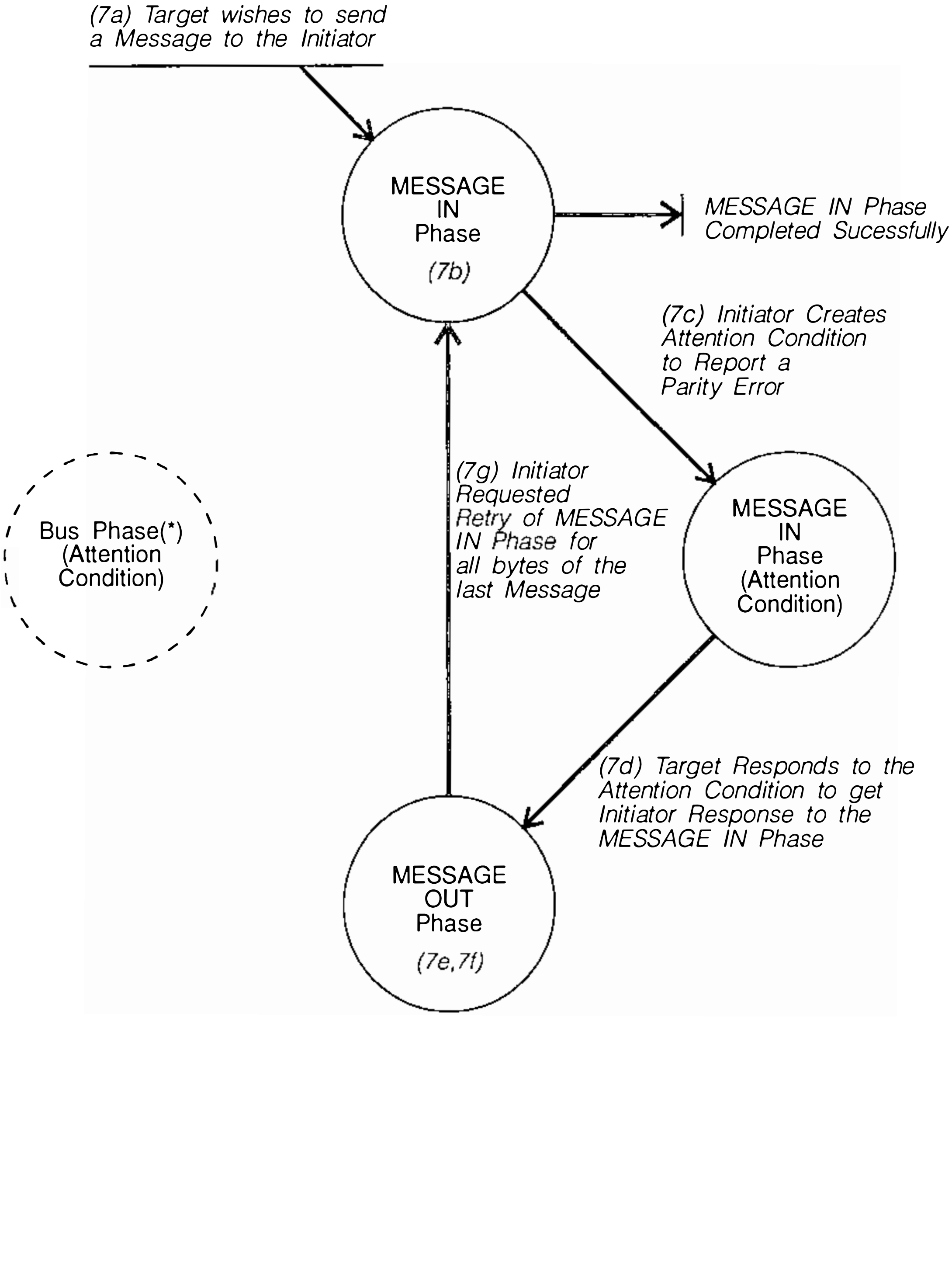

(7a) Target wishes to send a Message to the Initiator
MESSAGE IN Phase (7b)
MESSAGE IN Phase Completed Sucessfully
(7c) Initiator Creates Attention Condition to Report a Parity Error
Bus Phase(*) (Attention Condition)
(7g) Initiator Requested Retry of MESSAGE IN Phase for all bytes of the last Message
MESSAGE IN Phase (Attention Condition)
(7d) Target Responds to the Attention Condition to get Initiator Response to the MESSAGE IN Phase
MESSAGE OUT Phase (7e,7f)

DIAGRAM 52: TARGET MESSAGE IN WITH ERROR RECOVERY

(8) The Initiator sends a message to the Target, and the Target detects a ***Parity*** error (Diagram 53):

(a) The Initiator creates the Attention Condition.

(b) The Target recognizes the Attention Condition and changes to MESSAGE OUT Phase.

(c) The Initiator uses an ***Asynchronous Data Transfer*** to send the message to the Target.

(d) The Target detects a ***Parity*** error during the MESSAGE OUT Phase. After it detects the error, it continues the MESSAGE OUT Phase, but it discards the byte with the Parity error and all bytes received after the Parity error; it does not interpret them as messages. It continues receiving and discarding bytes until the ***ATN Signal*** is negated by the Initiator.

(e) The Target stays in the MESSAGE OUT Phase and asserts the ***REQ Signal***. This indicates to the Initiator that the Target desires to retry the entire MESSAGE OUT Phase.

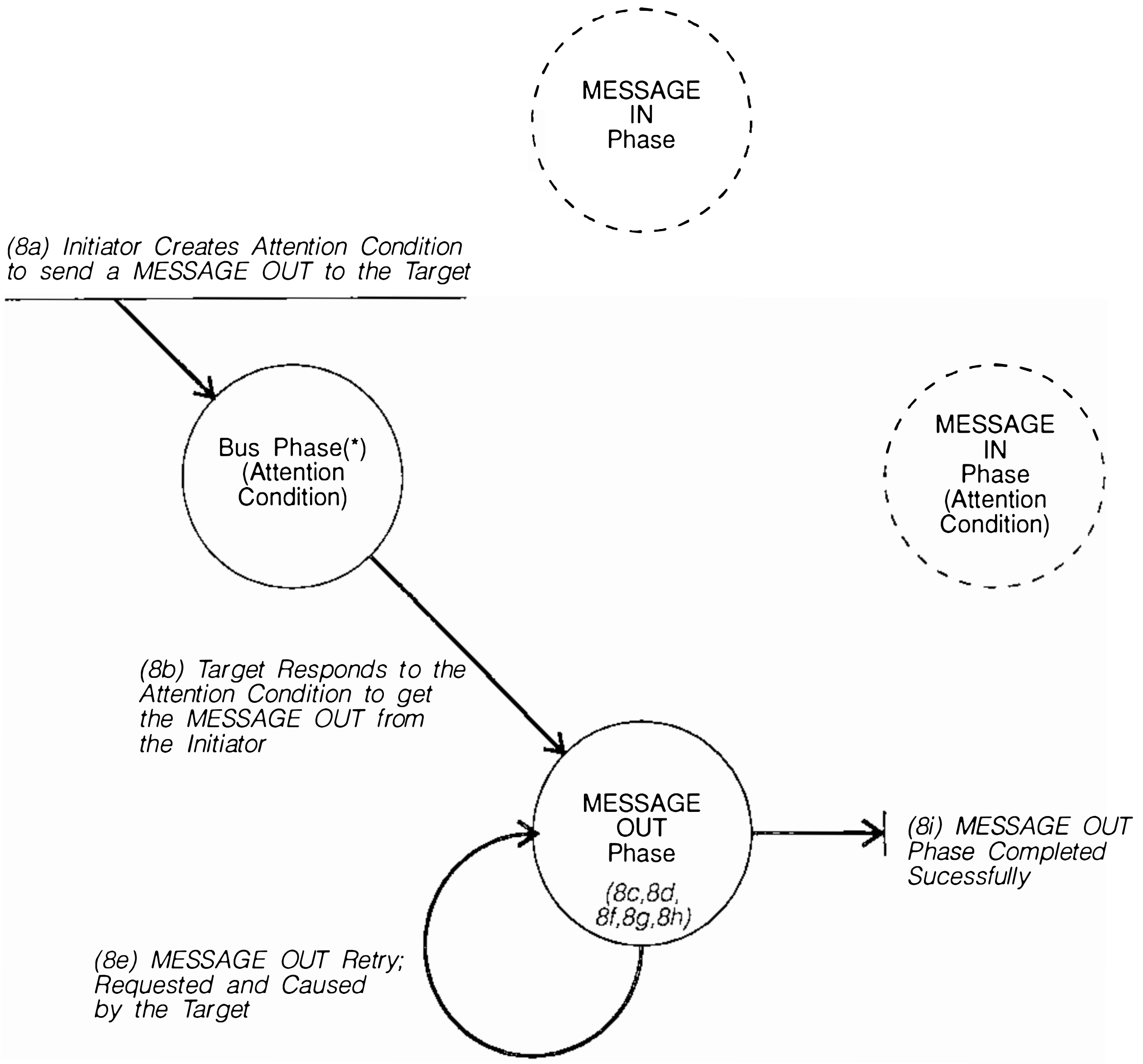
(f) The Initiator sees the REQ Signal asserted for a MESSAGE OUT Phase. Since the Initiator just completed sending message(s), it knows that this must be a retry. If there is more than one message byte to transfer, the Initiator asserts the ATN Signal.

(g) The Initiator uses an ***Asynchronous Data Transfer*** to re-send the message(s) to the Target. The Initiator negates the ***ATN Signal*** before sending the last message byte to indicate that it has sent all of the bytes it intended.

(h) The Target accepts the message(s) from the Initiator.

(i) The Target goes to another Phase. Note that because of the possibility of a retry request from the Target, the Initiator may not discard the message(s) sent until a change to some other ***Bus Phase*** occurs.

Note that for a MESSAGE OUT retry, all message bytes are sent again, while for a MESSAGE IN retry only the messages not yet successfully sent are sent again. We can only note that the difference is because the Initiator can tell the Target which byte failed during a MESSAGE IN Phase, but the Target cannot tell the Initiator which byte failed during a MESSAGE OUT Phase.


MESSAGE IN Phase
(8a) Initiator Creates Attention Condition to send a MESSAGE OUT to the Target
Bus Phase(*) (Attention Condition)
MESSAGE IN Phase (Attention Condition)
(8b) Target Responds to the Attention Condition to get the MESSAGE OUT from the Initiator
MESSAGE OUT Phase
(8c,8d, 8f,8g,8h)
(8i) MESSAGE OUT Phase Completed Sucessfully
(8e) MESSAGE OUT Retry; Requested and Caused by the Target

(*)Bus Phases: SELECTION, RESELECTION, COMMAND, STATUS, DATA IN, or DATA OUT Phase

Diagram 53: Init. MESSAGE OUT with Error Recovery

Composite Examples In these examples, both error recoveries and responses occur in the message handling. These show how these protocols work together in the Message System.

(9) The Initiator sends a message to the Target, the Target detects a ***Parity*** error, and then decides to reject the message (Diagram 54):

(a) The Initiator creates the Attention Condition.

(b) The Target recognizes the Attention Condition and changes to MESSAGE OUT Phase.

(c) The Initiator uses an ***Asynchronous Data Transfer*** to send the message to the Target.

(d) The Target detects a ***Parity*** error during the MESSAGE OUT Phase. After it detects the error, it continues the MESSAGE OUT Phase, but it discards the byte with the Parity error and all bytes received after the Parity error; it does not interpret them as messages. It continues receiving and discarding bytes until the ***ATN Signal*** is negated by the Initiator.

(e) The Target stays in the MESSAGE OUT Phase and asserts the ***REQ Signal***. This indicates to the Initiator that the Target desires to retry the entire MESSAGE OUT Phase.

(f) The Initiator sees the REQ Signal asserted for a MESSAGE OUT Phase. Since the Initiator just completed sending message(s), it knows that this must be a retry. If there is more than one message byte to transfer, the Initiator asserts the ATN Signal.

(g) The Initiator uses an ***Asynchronous Data Transfer*** to re-send the message(s) to the Target. The Initiator negates the ***ATN Signal*** before sending the last message byte to indicate that it has sent all of the bytes it intended.

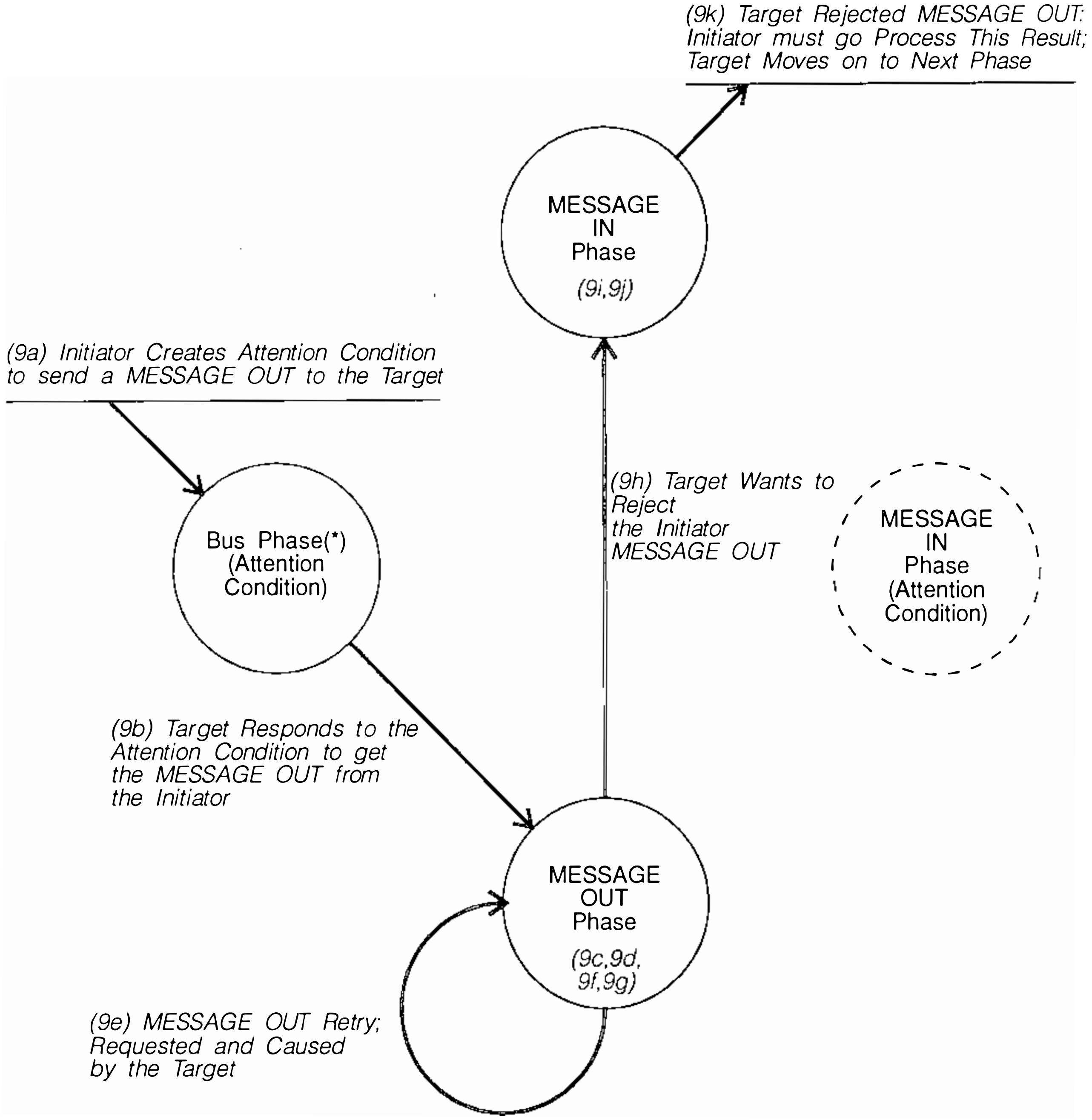
(h) The Target decides that it cannot accept the last message (after all that trouble!), so it changes to MESSAGE IN Phase.

(i) The Target uses an ***Asynchronous Data Transfer*** to send the MESSAGE REJECT Message to the Initiator. This tells the Initiator that the Target could not accept the message.

(j) The Initiator accepts the message from the Target.

(k) The Target goes to another Phase.

Note how the protocol flowed from one function to another. The fact that a retry occurred had nothing to do with the later reject. In fact, a retry during the process of rejecting the message would not affect the net result (see next example).


(9k) Target Rejected MESSAGE OUT: Initiator must go Process This Result; Target Moves on to Next Phase
MESSAGE IN Phase (9i,9j)
(9a) Initiator Creates Attention Condition to send a MESSAGE OUT to the Target
(9h) Target Wants to Reject the Initiator MESSAGE OUT
Bus Phase(*) (Attention Condition)
MESSAGE IN Phase (Attention Condition)
(9b) Target Responds to the Attention Condition to get the MESSAGE OUT from the Initiator
MESSAGE OUT Phase (9c,9d, 9f,9g)
(9e) MESSAGE OUT Retry; Requested and Caused by the Target

(*)Bus Phases: SELECTION, RESELECTION, COMMAND, STATUS, DATA IN, or DATA OUT Phase

DIAGRAM 54: MESSAGE OUT WITH RETRY AND REJECT

We have "unfolded" the basic diagram to make the following example clearer.

(10) The Target sends a message to the Initiator, the Initiator detects a ***Parity*** error, the MESSAGE IN is retried, and the Initiator rejects the message, but not before the Target also detects a Parity error and performs a MESSAGE OUT retry (pretty sick bus, Diagram 55):

(a) The Target enters the MESSAGE IN Phase.

(b) The Target uses an ***Asynchronous Data Transfer*** to send the message to the Initiator.

(c) The Initiator detects a Parity error during the transfer. As a result, it cannot accept the message byte just transferred. The Initiator creates the ***Attention Condition*** prior to negating the ***ACK Signal***.

(d) The Target sees the ***ATN Signal*** after the message transfer and changes to MESSAGE OUT Phase.

(e) The Initiator uses an ***Asynchronous Data Transfer*** to send the ***MESSAGE PARITY ERROR Message*** to the Target. The Initiator negates the ***ATN Signal*** before sending the message byte.

(f) The Target accepts the message from the Initiator.

(g) The Target returns to the MESSAGE IN Phase to retry the message transfer. The retry begins with the first byte of the message that contained the byte with the Parity error.

(h) The Initiator decides that it cannot accept the message so it creates the Attention Condition prior to negating the ***ACK Signal***.

(i) The Target sees the ***ATN Signal*** after the message transfer and changes to MESSAGE OUT Phase.

(j) The Initiator uses an ***Asynchronous Data Transfer*** to send the MESSAGE REJECT Message to the Target. The Initiator negates the ***ATN Signal*** before sending the message byte.

(k) The Target detects a ***Parity*** error during the MESSAGE OUT Phase. After it detects the error, it continues the MESSAGE OUT Phase, but it discards the byte with the Parity error and all bytes received after the Parity error; it does <u>not</u> interpret them as messages. It continues receiving and discarding bytes until the ***ATN Signal*** is negated by the Initiator.

(Text continued after diagram)

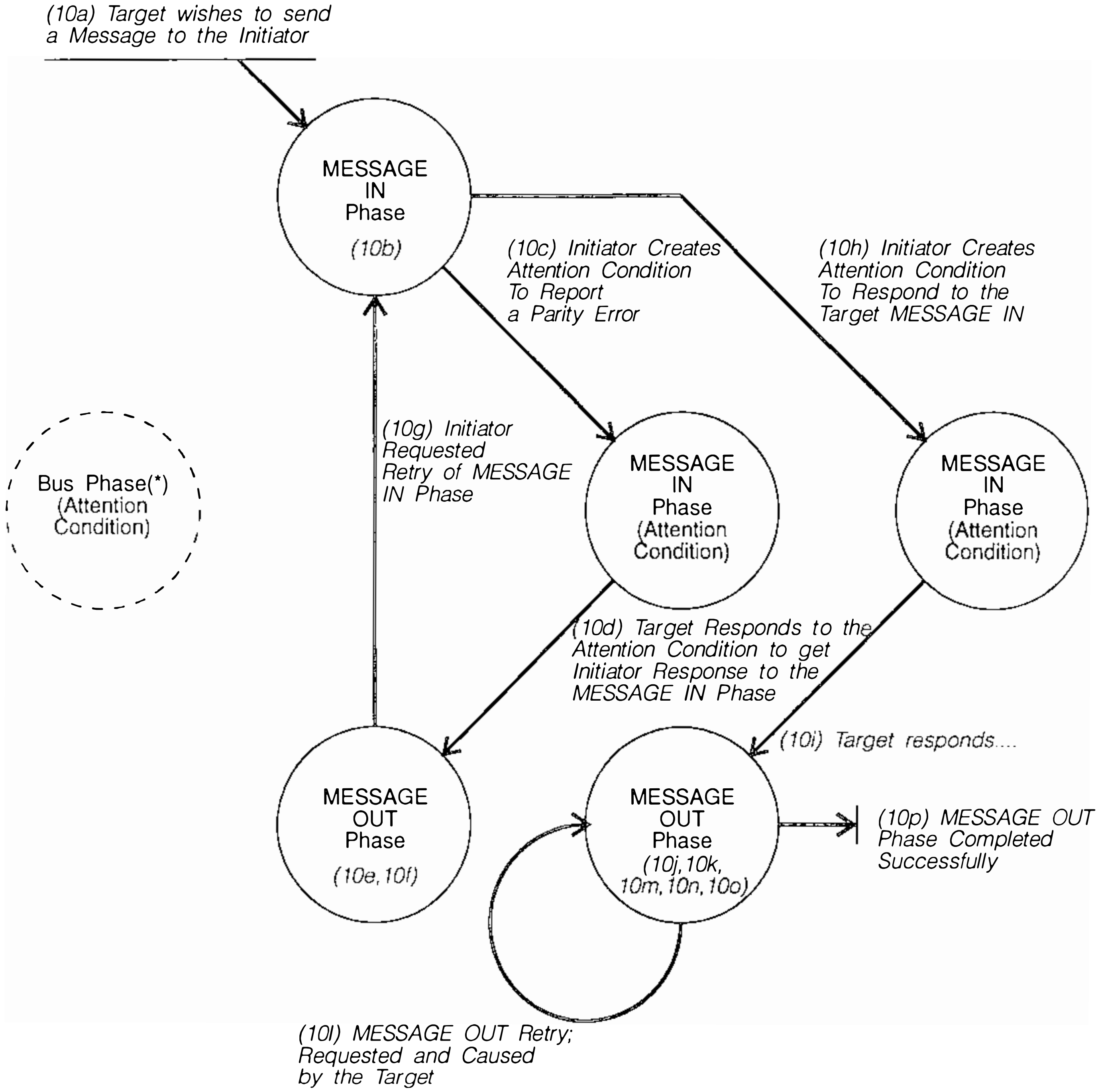

(10a) Target wishes to send a Message to the Initiator
MESSAGE IN Phase (10b)
(10c) Initiator Creates Attention Condition To Report a Parity Error
(10h) Initiator Creates Attention Condition To Respond to the Target MESSAGE IN
Bus Phase(*) (Attention Condition)
(10g) Initiator Requested Retry of MESSAGE IN Phase
MESSAGE IN Phase (Attention Condition)
MESSAGE IN Phase (Attention Condition)
(10d) Target Responds to the Attention Condition to get Initiator Response to the MESSAGE IN Phase
(10i) Target responds....
MESSAGE OUT Phase (10e, 10f)
MESSAGE OUT Phase (10j, 10k, 10m, 10n, 10o)
(10p) MESSAGE OUT Phase Completed Successfully
(10l) MESSAGE OUT Retry; Requested and Caused by the Target

DIAGRAM 55: MESSAGE IN WITH TWO RETRIES & A RESPONSE

(l) The Target stays in the MESSAGE OUT Phase and asserts the ***REQ Signal***. This indicates to the Initiator that the Target desires to retry the entire MESSAGE OUT Phase.

(m) The Initiator sees the REQ Signal asserted for a MESSAGE OUT Phase. Since the Initiator just completed sending message(s), it knows that this must be a retry. If there is more than one message byte to transfer, the Initiator asserts the ATN Signal.

(n) The Initiator uses an ***Asynchronous Data Transfer*** to re-send the message(s) to the Target. The Initiator negates the ***ATN Signal*** before sending the last message byte to indicate that it has sent all of the bytes it intended.

(o) The Target accepts the message from the Initiator.

(p) The Target goes to another Phase.

Summary:

- The Target goes to the ***MESSAGE IN Phase*** when it wishes to send a message.
- The Initiator creates the ***Attention Condition*** when it wishes to send a message. It must then wait for the Target to recognize the ***ATN Signal*** and change to ***MESSAGE OUT Phase***.
- The Initiator also uses the Attention Condition when it wishes to respond to a Target MESSAGE IN, reject the message, or report a ***Parity*** error.
- The Target uses the MESSAGE IN Phase to respond to an Initiator MESSAGE OUT or reject the message.
- The Target repeats the MESSAGE OUT Phase when it detects a Parity error.

When is a Message Appropriate? Table 35 shows what ***Bus Phases*** can precede each message. Note that any MESSAGE IN Phase transfer can be preceded by a MESSAGE OUT transfer for error recovery. Also, many of the messages listed may be preceded by another message in the same phase, and this is not indicated. This will help in determining when a Message is appropriate.

Table 36 shows what phases can follow each message. This will help the Initiator predict Target behavior. The table does not show that any of these messages can be followed by a MESSAGE IN Phase or a MESSAGE OUT Phase when a rejection or response to the message is called for, or a retry. Ultimately, any phase may follow any of these messages; the table shows the realistic ones.

TABLE 35: MESSAGE TRANSITIONS FROM PREVIOUS PHASE

Message Name	Phase or Phases that can Occur Before the MESSAGE IN Phase or MESSAGE OUT Phase in which the Message is Sent
ABORT	Any Phase except RESELECTION Phase
ABORT TAG	Any Phase except RESELECTION Phase
BUS DEVICE RESET	Any Phase except RESELECTION Phase
CLEAR QUEUE	Any Phase except RESELECTION Phase
COMMAND COMPLETE	STATUS Phase only
DISCONNECT (Out)	COMMAND, MESSAGE IN, DATA IN, or DATA OUT Phase
DISCONNECT (In)	COMMAND, DATA IN, or DATA OUT Phase
HEAD OF QUEUE TAG	SELECTION Phase only
IDENTIFY (Out)	SELECTION Phase only; also allowed after COMMAND, DATA IN, and DATA OUT Phase, but this is not encouraged!
IDENTIFY (In)	RESELECTION Phase only
IGNORE WIDE RESIDUE	DATA IN Phase only
INITIATE RECOVERY (Out)	SELECTION Phase only
INITIATE RECOVERY (In)	STATUS Phase only
INITIATOR DETECTED ERROR	COMMAND, STATUS, DATA IN, or DATA OUT Phase
LINKED COMMAND COMPLETE	STATUS Phase only
LINKED COMMAND COMPLETE (WITH FLAG)	STATUS Phase only
MESSAGE PARITY ERROR	MESSAGE IN Phase only
MESSAGE REJECT	MESSAGE IN or MESSAGE OUT Phase
MODIFY DATA POINTER	COMMAND, DATA IN or DATA OUT Phase
NO OPERATION	Any Phase except RESELECTION Phase
ORDERED QUEUE TAG	SELECTION Phase only
RELEASE RECOVERY	SELECTION Phase only
RESTORE POINTERS	COMMAND, STATUS, DATA IN, or DATA OUT Phase
SAVE DATA POINTER	COMMAND, STATUS, DATA IN, or DATA OUT Phase
SIMPLE QUEUE TAG (Out)	SELECTION Phase only
SIMPLE QUEUE TAG (In)	RESELECTION Phase only
SYNCHRONOUS DATA TRANSFER REQUEST	COMMAND, MESSAGE IN, or MESSAGE OUT Phase
TERMINATE I/O PROCESS	Any Phase except RESELECTION Phase
WIDE DATA TRANSFER REQUEST	COMMAND, MESSAGE IN, or MESSAGE OUT Phase

TABLE 36: MESSAGE TRANSITIONS TO NEXT PHASE

Message Name	Next Phase that can Occur After the Message has been Transferred
ABORT	BUS FREE Phase
ABORT TAG	BUS FREE Phase
BUS DEVICE RESET	BUS FREE Phase
CLEAR QUEUE	BUS FREE Phase
COMMAND COMPLETE	BUS FREE Phase
DISCONNECT (Out)	DISCONNECT MESSAGE IN or Previous Phase
DISCONNECT (In)	BUS FREE Phase
HEAD OF QUEUE TAG	COMMAND PHASE, MESSAGE IN (e.g., DISCONNECT), or more MESSAGE OUT
IDENTIFY (Out)	COMMAND PHASE, MESSAGE IN (e.g., DISCONNECT), or more MESSAGE OUT
IDENTIFY (In)	More MESSAGE IN or Any Other Information Transfer Phase
IGNORE WIDE RESIDUE	STATUS Phase, more MESSAGE IN, or more DATA IN Phase
INITIATE RECOVERY (Out)	COMMAND PHASE or more MESSAGE OUT
INITIATE RECOVERY (In)	COMMAND COMPLETE MESSAGE IN
INITIATOR DETECTED ERROR	Last Phase or MESSAGE IN
LINKED COMMAND COMPLETE	COMMAND PHASE or MESSAGE IN (e.g., DISCONNECT)
LINKED COMMAND COMPLETE (WITH FLAG)	COMMAND PHASE or MESSAGE IN (e.g., DISCONNECT)
MESSAGE PARITY ERROR	MESSAGE IN to retry message, other Information Transfer Phase to give up trying to send the message
MESSAGE REJECT	Retry of previous MESSAGE IN or MESSAGE OUT, any other Information Transfer Phase, or BUS FREE Phase
MODIFY DATA POINTER	Previous Phase (DATA IN or DATA OUT), or more MESSAGE IN
NO OPERATION	Previous Phase, or other Information Transfer Phase
ORDERED QUEUE TAG	COMMAND PHASE or more MESSAGE OUT
RELEASE RECOVERY	BUS FREE Phase
RESTORE POINTER	Previous Phase
SAVE DATA POINTER	Previous Phase or more MESSAGE IN
SIMPLE QUEUE TAG (Out)	COMMAND PHASE or more MESSAGE OUT
SIMPLE QUEUE TAG (In)	Any Information Transfer Phase
SYNCHRONOUS DATA TRANSFER REQUEST	An S.D.T.R. in response, or any Information Transfer Phase
TERMINATE I/O PROCESS	STATUS Phase or Previous Phase
WIDE DATA TRANSFER REQUEST	A W.D.T.R. in response, or any Information Transfer Phase

MODIFY DATA POINTER Message. The MODIFY DATA POINTER is used by a Target to change the value of the ***Active Data Pointer***. No other ***Pointers*** are affected. See also ***Path Control***. This is a multiple byte message:

Bit Byte	7	6	5	4	3	2	1	0
0	Message Code = 01 hex							
1	Additional Message Length = 05 hex							
2	Extended Message Code = 00 hex							
3	Data Pointer Offset (MSByte)							
4	Data Pointer Offset							
5	Data Pointer Offset							
6	Data Pointer Offset (LSByte)							

This message is most often used to allow the Target to have "random-access" into the Initiator's data buffer. This could be useful in the following situations:

- A Target can sense the position of the read/write head relative to a disk. When a READ command comes in, it sees that the next physical sector that can be read is the second sector requested. The Target goes ahead and begins reading the second sector. After the sector is read from the medium by the Target, it uses the MODIFY DATA POINTER Message to change the Active Data Pointer to point at the place in the Initiator's data buffer where the sector is to be transferred (see below).

- The Target is a caching disk controller. It happens to have the second sector of the sectors requested in cache, but the first sector must be fetched from the disk. The Target uses the MODIFY DATA POINTER Message to change the Active Data Pointer to point at the buffer location for the second sector. While the transfer to the Initiator is going, it transfers the first sector into its cache. There is an example of this in the section on ***Pointers***.

- A Target can sense the position of the read/write head relative to a disk. When a WRITE command comes in, it sees that it could begin the write at the second sector rather than the first sector requested. It uses the MODIFY DATA POINTER Message to change the Initiator Active Data Pointer to begin the transfer with the second sector, which will be the first sector physically accessible. Note: This is a rare case since the Target cannot be sure that the transfer with the Initiator will be completed in time to write it on the disk.

Diagram 56 shows a typical example of the relationship between disk sectors and buffer locations. Note that sectors are not necessarily contiguous as shown here.

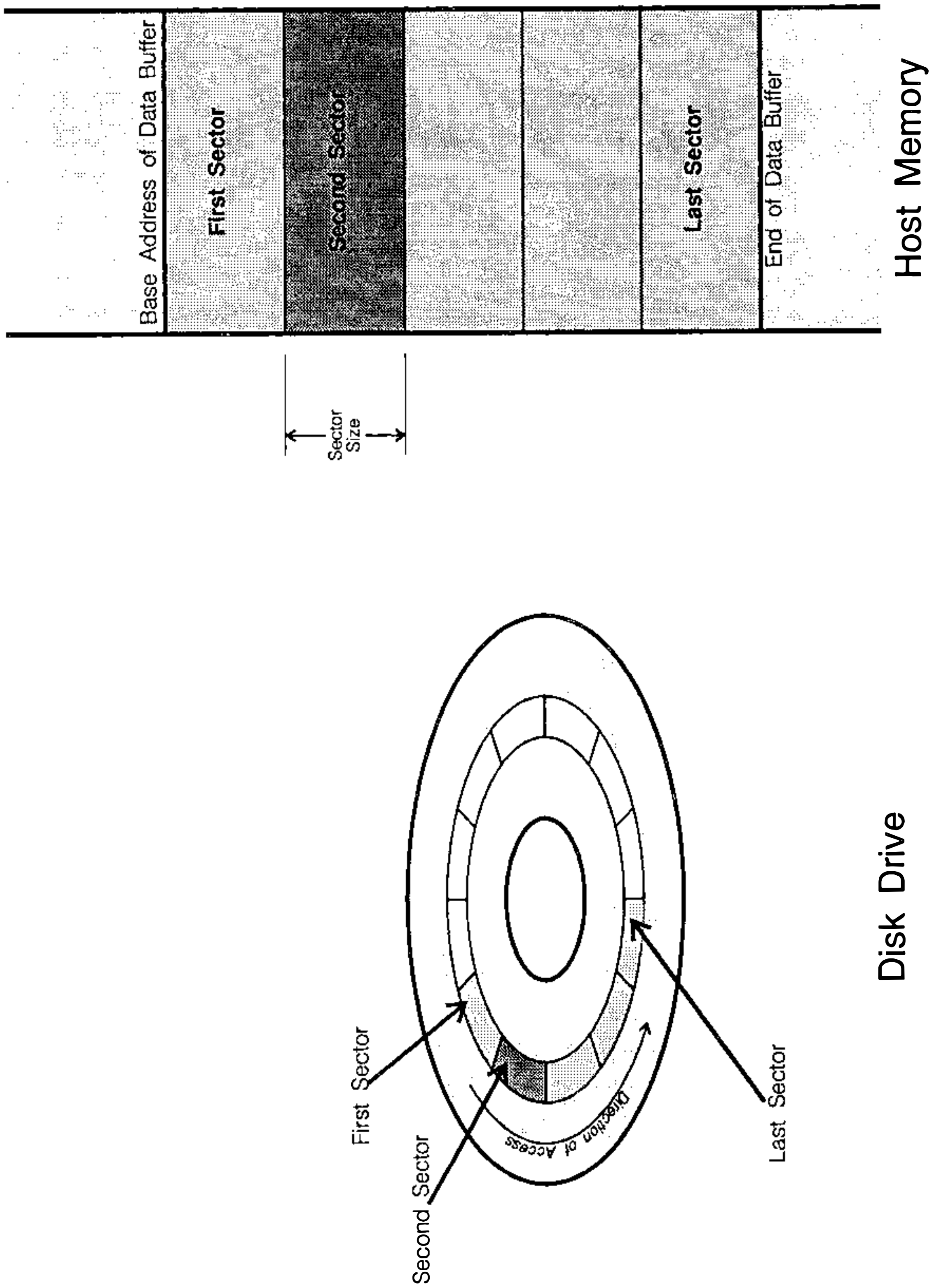


DIAGRAM 56: DISK SECTORS AND HOST MEMORY

Since MODIFY DATA POINTER is a multiple byte message, there is a field which contains variable information to be specified (see ***Extended Messages***). This field contains the Data Pointer Offset. This is a two's complement number, meaning that a positive or negative offset can be specified. For example:

- 00 00 00 00 = no offset; not illegal, but not recommended....
- 00 00 00 01 = add 1 to the Active Data Pointer
- 00 00 02 00 = add 512 to the Active Data Pointer
- FF FF FF FF = subtract 1 from the Active Data Pointer
- FF FF FE 00 = subtract 512 from the Active Data Pointer

An Initiator can expect to see a MODIFY DATA POINTER Message before a ***DATA Phase*** or between two DATA Phases. The message is sent by the Target during a ***MESSAGE IN Phase***.

An important point. This feature gives the Target a **lot** of power to really mess things up! For instance, it could add or subtract too much from the Active Data Pointer such that the Pointer references data outside of the allocated Host memory. It behooves the Target to check itself, and (most importantly) it also behooves the Initiator to carefully check all MODIFY DATA POINTER messages to ensure that this doesn't occur!

Another important point. There needs to be some agreement between a Target and an Initiator that it is okay for the Target to use this message. ***SCSI-3*** may define a standard way to enable the Target to enable the use of this message. You might think that the Target could use the ***MESSAGE REJECT Message*** to disable the message; i.e., if the Initiator rejects the MODIFY DATA POINTER, the Target disables further use of it. The catch is that this could be the one time the Initiator needed to reject the message, but it could still handle it later.

Yet another important point. An Initiator that implements the MODIFY DATA POINTER Message should execute it quickly enough to realize the performance improvement. If you can't execute it quickly, don't implement it at all.

Summary of Use: The MODIFY DATA POINTER Message is sent only by a Target to change the value of the Active Data Pointer for purposes of randomly accessing the Initiator/Host data buffer.

MSG Signal. The MSG signal is driven by the Target to control whether "message system" or "data or command" information is transferred over the bus. This signal is one of the three ***Bus Phase Signals*** (MSG, C/D, and I/O) (see also ***Bus Phases*** and ***Message System***).

- When this signal is false (negated or release), the bus contains "data or command", which is anything that is not Message System information; for example, "***Logical Block*** data", "commands", or "status". MSG is false during ***SELECTION Phase***, ***RESELECTION Phase***, ***STATUS Phase***, ***COMMAND Phase***, ***DATA OUT Phase***, and ***DATA IN Phase***.

- When this signal is true (asserted), the bus contains Message System information, which is anything that is used for ***Path Control***. MSG is true during ***MESSAGE IN Phase*** and ***MESSAGE OUT Phase***.

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 57 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.

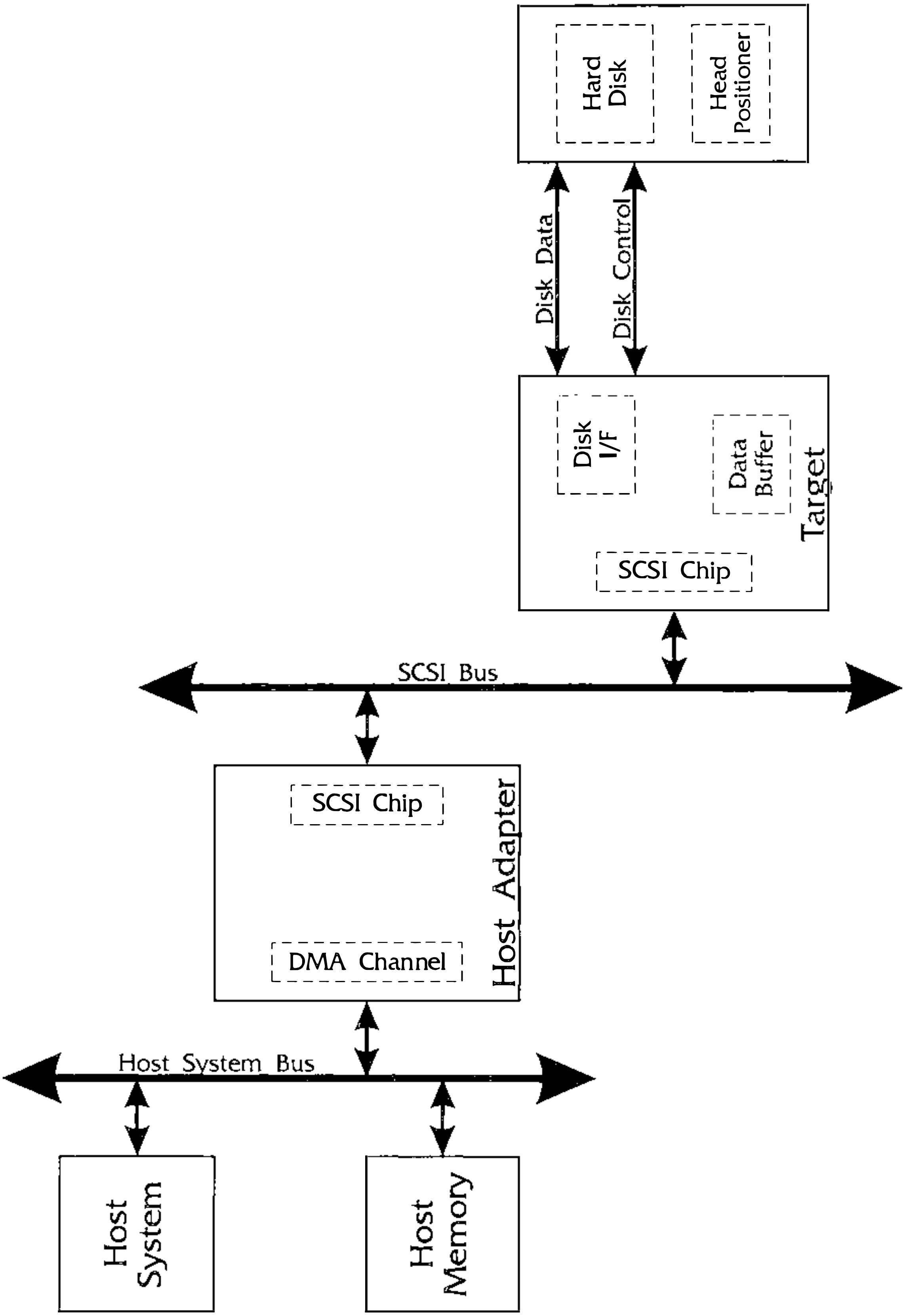


DIAGRAM 57: DISK EXAMPLE SYSTEM ARCHITECTURE

The file read request must filter down through the different layers of the Host System, as shown in Diagram 58. 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.

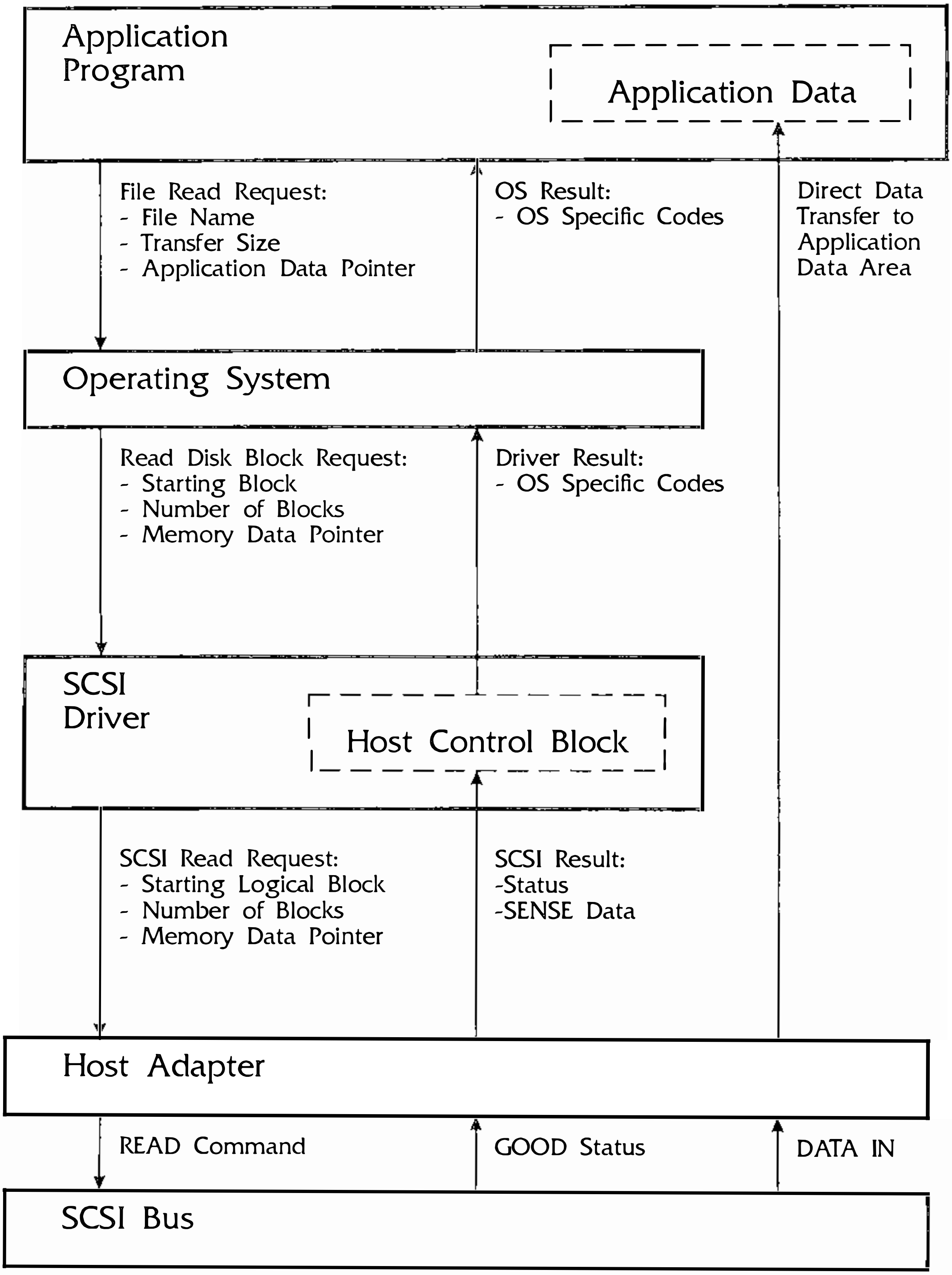


DIAGRAM 58: DISK READ EXAMPLE SOFTWARE LEVELS

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 38 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 38: 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 57.

Like the read request, the file write request must filter down through the different layers of the Host System, as shown in Diagram 59. 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.

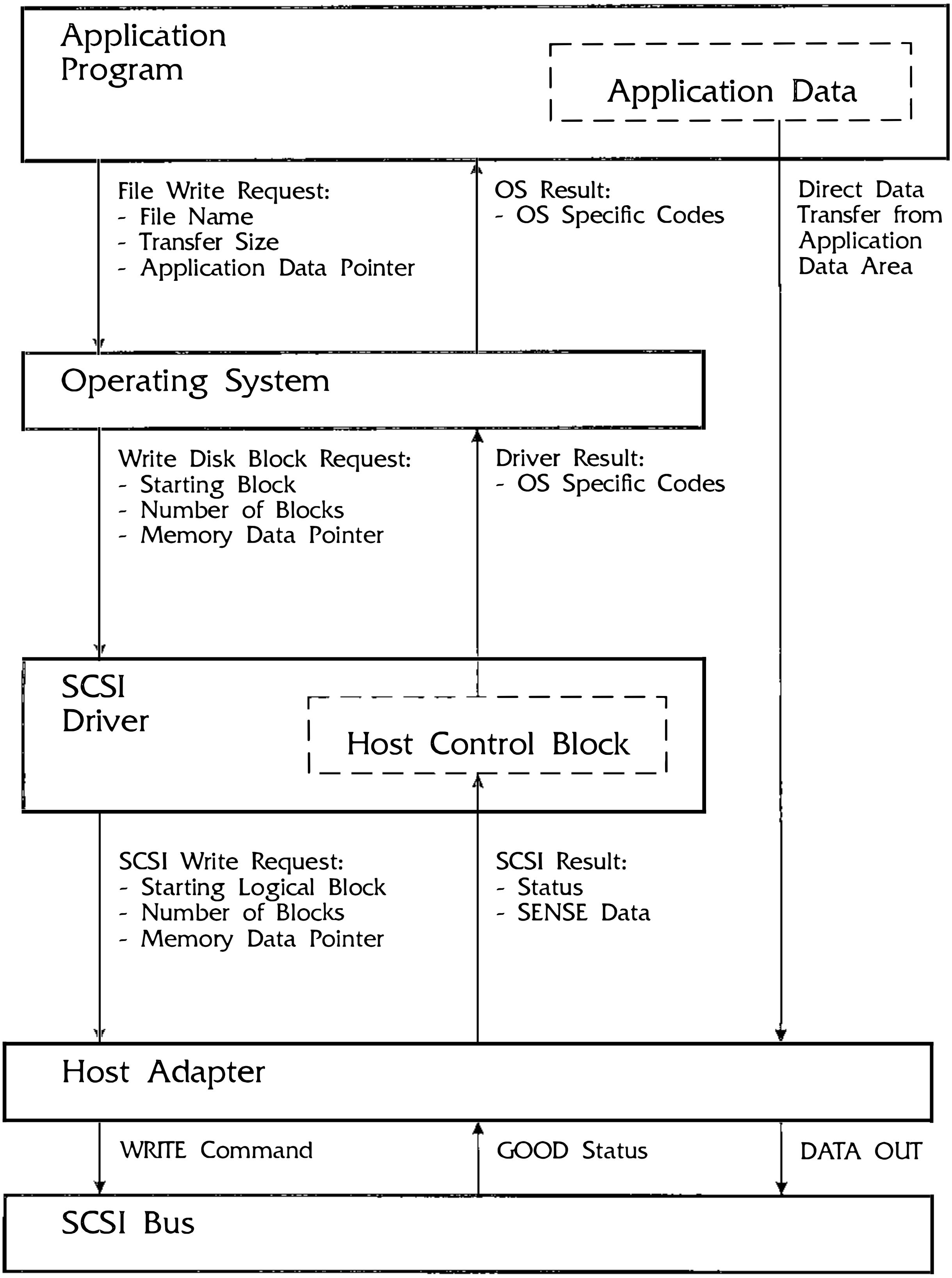


DIAGRAM 59: DISK WRITE EXAMPLE SOFTWARE LAYERS

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 39 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 39: 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 60 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.

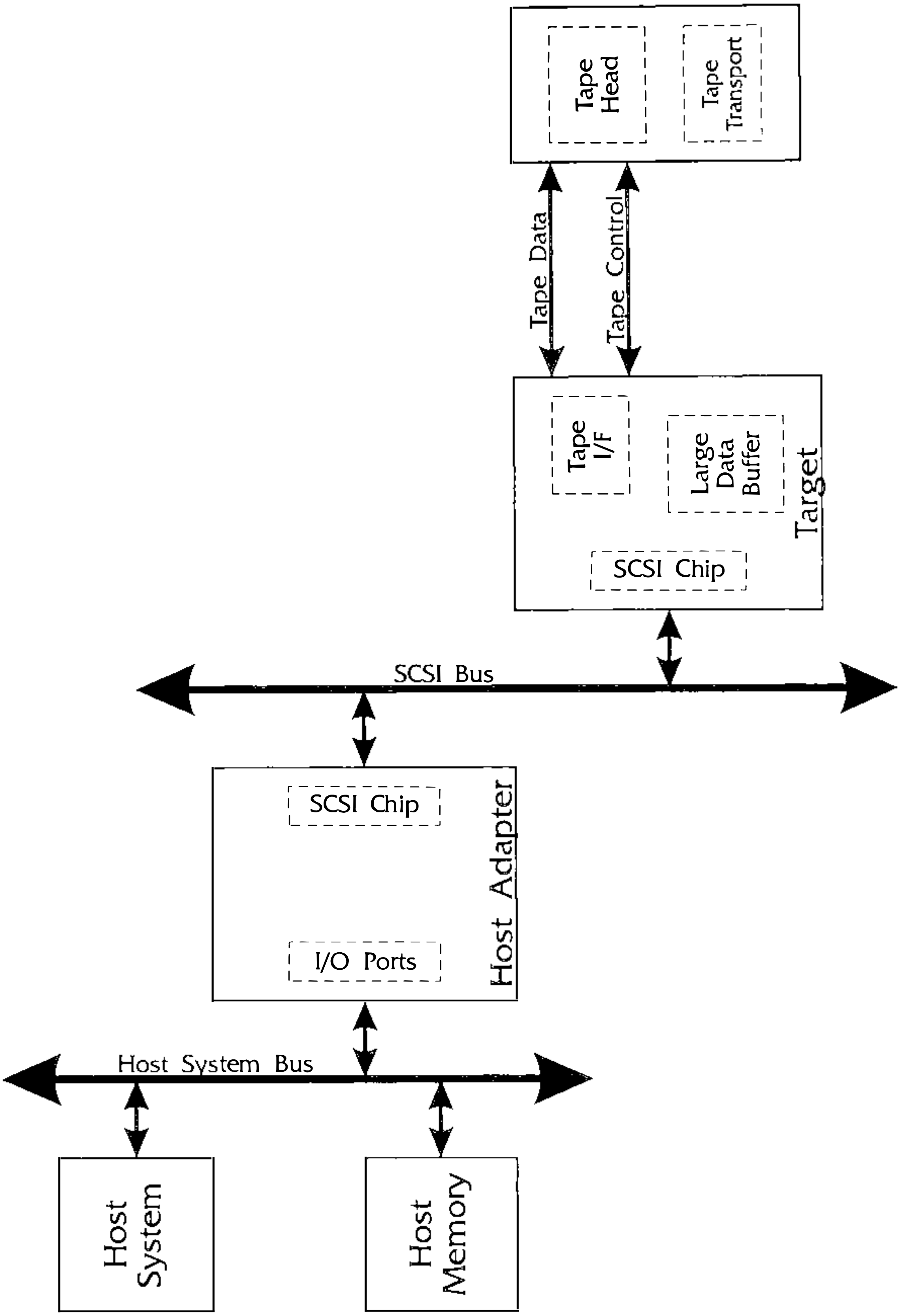


DIAGRAM 60: TAPE EXAMPLE SYSTEM ARCHITECTURE

The tape restore request must filter down through the different layers of the Host System, as shown in Diagram 61. 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.

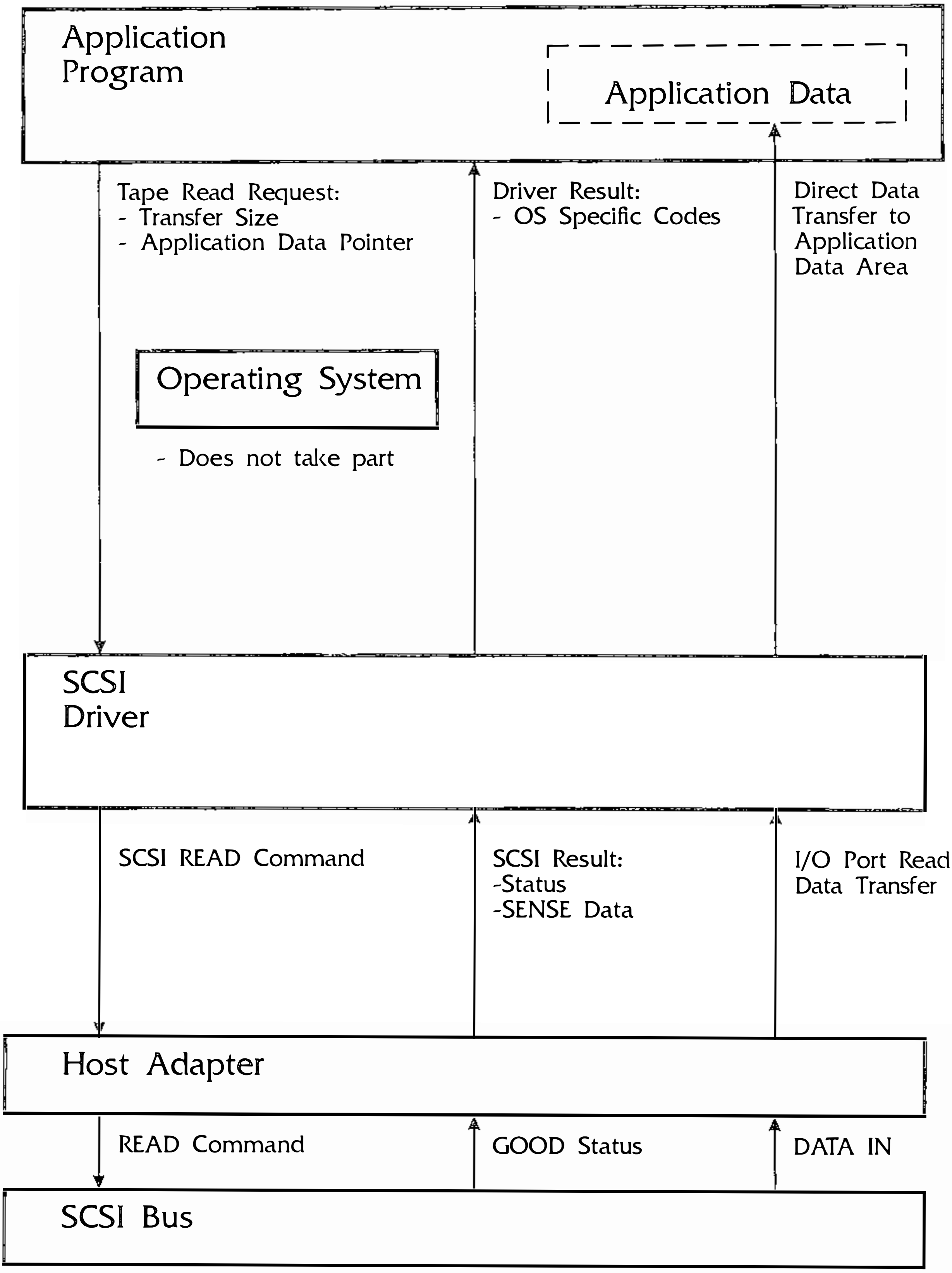


DIAGRAM 61: TAPE READ EXAMPLE SOFTWARE LEVELS

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 40 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 40: 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 60 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 62. 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.

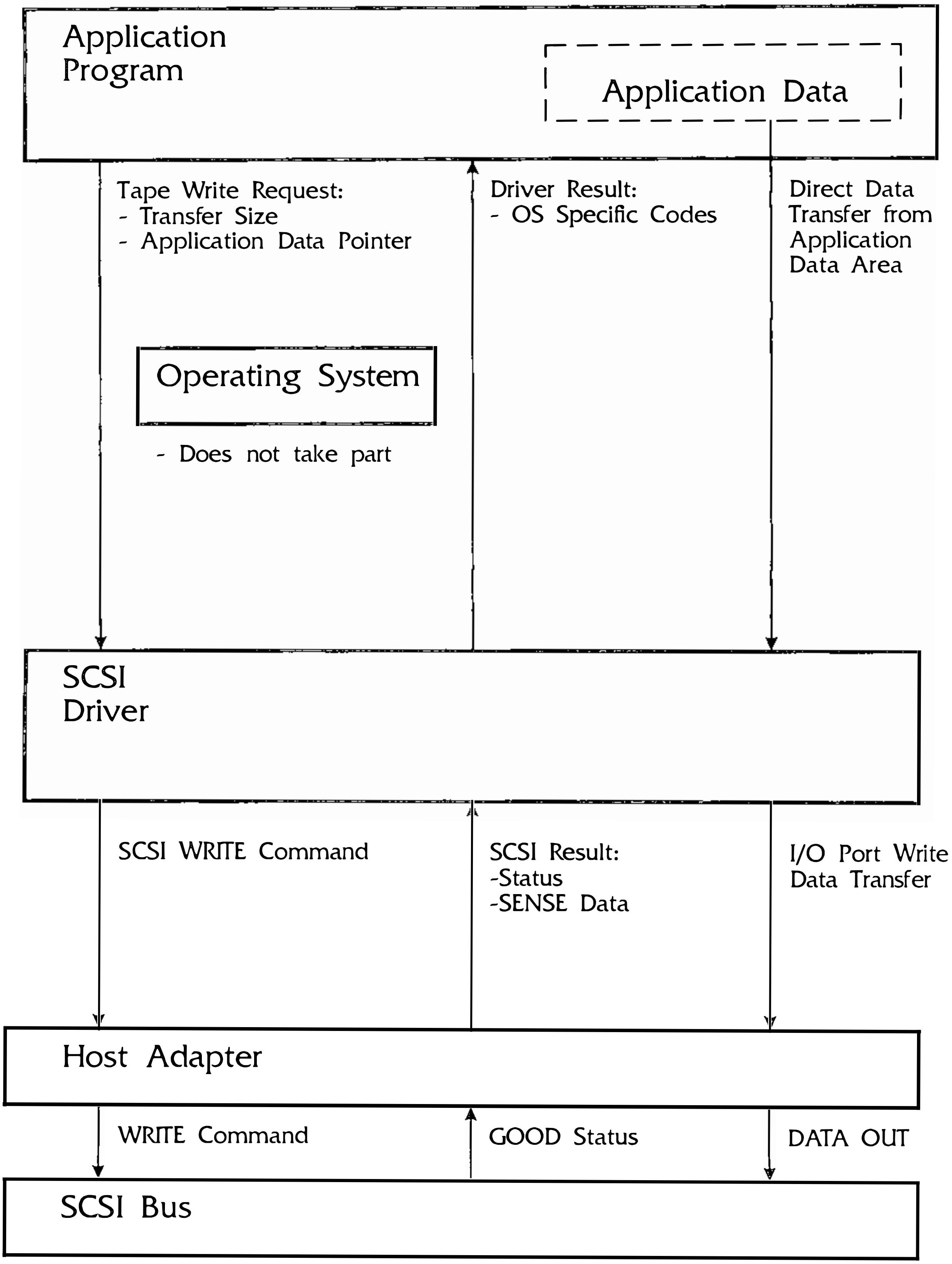


DIAGRAM 62: TAPE WRITE EXAMPLE SOFTWARE LEVELS

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 41 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 41: 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 1-879936-11-9

90000

9 781879 936119

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 (A–M)*